IMPLEMENTATION AND VALIDATION OF A-CONNECT IN SPEECH RECOGNITION APPLICATIONS

JOSÉ DAVID AMAYA HERNÁNDEZ
KAREN DAYANNA LEÓN SUÁREZ

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTY OF PHYSICAL-MECHANICAL ENGINEERING
SCHOOL OF ELECTRICAL,
ELECTRONIC AND TELECOMMUNICATIONS ENGINEERING
BUCARAMANGA
2022

IMPLEMENTATION AND VALIDATION OF A-CONNECT IN SPEECH RECOGNITION APPLICATIONS

JOSÉ DAVID AMAYA HERNÁNDEZ
KAREN DAYANNA LEÓN SUÁREZ

Bachelor degree thesis to qualify for the title of
Electronic Engineer

Director
Elkim Felipe Roa Fuentes,
Doctor of Philosophy.

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTY OF PHYSICAL-MECHANICAL ENGINEERING
SCHOOL OF ELECTRICAL,
ELECTRONIC AND TELECOMMUNICATIONS ENGINEERING
BUCARAMANGA
2022

Acknowledgment

We would like to thank all the people who supported and guided us throughout our career and the development of this project. Special thanks to professors Luis Rueda and Elkim Roa who were supervising and supporting the work done.

CONTENTS

pág.

INTRODUCTION 14

1. OBJECTIVES 17

1.1. GENERAL OBJECTIVE 17

1.2. SPECIFIC OBJECTIVES 17

2. A-CONNECT METHODOLOGY 18

3. SPEECH RECOGNITION 20

3.1. APPLICATIONS 20

3.1.1. Keyword spotting 20

3.1.2. Automatic Speech Recognition 21

3.2. MODELS USED IN SPEECH RECOGNITION APPLICATIONS 21

3.3. DATASETS USED IN SPEECH RECOGNITION APPLICATIONS 23

3.3.1. LibriSpeech 23

3.3.2. TED-LIUM 23

3.3.3. Google's Speech Commands dataset 23

3.3.4. Spoken Digits 24

3.4. FEATURE EXTRACTION 24

3.4.1. Audio cutting/padding 25

3.4.2. (Optional) Data augmentation 25

3.4.3. Short-time Fourier transform (STFT) 26

3.4.4. log-Mel scale spectrogram 27

3.4.5. MFCCs 28

3.4.6. (Optional) Standardization 28
3.5. MODELS USED IN THIS WORK 28
3.5.1. FastGRNN, LSTM, and GRU 29
3.5.2. CRNN 30

4. LIBRARY DEVELOPMENT 33
4.1. GENERAL PROCEDURE 33
4.2. FastGRNN 34
4.3. LSTM 36
4.4. GRU 38
4.5. HYPERPARAMETERS 40

5. RESULTS 42
5.1. SPOKEN DIGITS DATASET 42
5.1.1. Hyperparameters 42
5.1.2. Monte Carlo experiments 43
5.2. GOOGLE'S SPEECH COMMANDS DATASET 44
5.2.1. Hyperparameters 44
5.2.2. Comparison of the base model and base implemented version 45
5.2.3. Monte Carlo experiments 46

6. FPGA IMPLEMENTATION 51
6.1. FIXED-POINT INTEGER ARITHMETIC 51
6.2. QUANTIZATION APPROACH 52
6.3. MODULES 53
6.4. DEMONSTRATION 59
6.5. RESULTS 60
6.5.1. Power consumption 64
6.5.2. FPGA Monte Carlo experiments 65

6.5.3. Implementation details 66

7. FURTHER CONTRIBUTIONS 69

8. CONCLUSION 70

BIBLIOGRAPHY 71

LIST OF FIGURES

pág.

Figure 2.1. Neural network architecture for A-Connect 18

Figure 3.1. Feature extraction (Steps 3 to 5) 25
Figure 3.2. Mel-Scale Filter Bank. Taken from 27
Figure 3.3. Basic architecture of an RNN Model. 29
Figure 3.4. CRNN Model. Taken from 30
Figure 3.5. 2-Layer RNN. 32

Figure 4.1. FastGRNN Cell 34
Figure 4.2. LSTM Cell 38
Figure 4.3. GRU Cell 40

Figure 5.1. Results of Spoken Digits Dataset for: a. LSTM-S BaseNN and A-Connect 70 % b. GRU-S BaseNN and A-Connect 70 % c. CRNN-S BaseNN and A-Connect 70 % 44
Figure 5.2. Results for: FastGRNN 100 units A-Connect 70 % and FastGRNN 400 units A-Connect 70 % 47
Figure 5.3. Results for Google Speech Commands Dataset: a. LSTM-S BaseNN and A-Connect 70 % b. LSTM-M BaseNN and A-Connect 70 % c. LSTM-L BaseNN and A-Connect 70 % d. LSTM-S A-Connect 70 % and LSTM-L A-Connect 70 % 48
Figure 5.4. Results for Google Speech Commands Dataset: a. GRU-S BaseNN and A-Connect 70 % b. GRU-M BaseNN and A-Connect 70 % c. GRU-L BaseNN and A-Connect 70 % d. GRU-S A-Connect 70 % and GRU-L A-Connect 70 % 49

Figure 5.5. Results for Google Speech Commands Dataset: a. CRNN-S BaseNN and A-Connect 70 % b. CRNN-M BaseNN and A-Connect 70 % c. CRNN-L BaseNN and A-Connect 50 % 50

Figure 6.1. NN controller module (Top module) 54

Figure 6.2. Example of signals for a single inference 55

Figure 6.3. NN module 55

Figure 6.4. DataPath of vecMatMult module 57

Figure 6.5. DataPath of vecVecFunc module 58

Figure 6.6. DataPath of vecFunc module 58

Figure 6.7. FPGA demo web interface 60

Figure 6.8. FPGA current measurement scheme 64

Figure 6.9. Current draw on idle state 64

Figure 6.10. Current draw while performing inference 65

Figure 6.11. Results for SC - BaseNN and A-Connect 70 % 66

LIST OF TABLES

pág.

Table 3.1. Speech Recognition models 22

Table 3.2. Google Speech Dataset splits. Taken from 24

Table 3.3. Comparison of different versions of the models found in the papers 32

Table 4.1. Cell hyperparameters 41

Table 5.1. Spoken Digits Results 43

Table 5.2. Accuracy Comparison 46

Table 5.3. FastGRNN Results 46

Table 5.4. LSTM Results 48

Table 5.5. GRU Results 49

Table 5.6. CRNN Results 50

Table 6.1. Implemented models hyperparameters 61

Table 6.1. Implemented models hyperparameters 62

Table 6.2. Accuracy comparison 63

Table 6.3. FPGA implementation results 63

Table 6.4. FPGA Monte Carlo experiments results 66

Table 6.5. Resource utilization for L model 67

Table 6.6. Resource utilization for SC model 68

RESUMEN

TÍTULO: IMPLEMENTACIÓN Y VALIDACIÓN DE A-CONNECT EN APLICACIONES DE RECONOCIMIENTO DE VOZ *

AUTORES: JOSÉ DAVID AMAYA HERNÁNDEZ, KAREN DAYANNA LEÓN SUÁREZ **

PALABRAS CLAVE: REDES NEURONALES RECURRENTES, REDES NEURONALES PROFUNDAS, DETECCIÓN DE PALABRAS CLAVE.

DESCRIPCIÓN:

Los sistemas de reconocimiento de voz permiten interactuar con sistemas utilizando nuestra voz, útil para aplicaciones como atención al cliente automatizada, asistentes de voz, etc. El desarrollo de estos sistemas se ha beneficiado de los avances en Deep Learning haciéndolos más fiables y precisos. Sin embargo, el despliegue de estos sistemas suele requerir una gran cantidad de recursos de hardware y potencia para conseguir un buen rendimiento. Los aceleradores de redes neuronales analógicas son una posible solución, ya que proporcionan un rendimiento rápido con bajo consumo de energía a costa de precisión, ya que son susceptibles a variabilidad estocástica. Una solución a este problema es A-Connect, una metodología de entrenamiento que aumenta la resiliencia en precisión de las redes neuronales analógicas a la variabilidad estocástica. Se ha desarrollado una librería A-Connect en un proyecto anterior con implementaciones para capas totalmente conectadas y convolucionales. En este trabajo se extiende a diferentes tipos de RNNs utilizadas en aplicaciones de reconocimiento de voz como: FastGRNN, LSTM y GRU. Presentamos resultados utilizando las capas implementadas en diferentes modelos entrenados y probados en los datasets Spoken Digits y Speech Commands. Obtenemos con A-Connect un mejor rendimiento cuando se aplica error comparado con el modelo base, por ejemplo, con un modelo LSTM-S, logrando un 68,25 % de precisión en el dataset Speech Commands utilizando A-Connect al 70 %, lo que supone un 22,78 % más que el modelo base. Además, presentamos la implementación en FPGA del modelo GRU. Una versión grande que alcanza el 94,78 % en el dataset Speech Commands, y una versión pequeña entrena-

---

* Trabajo de grado

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones. Director: Elkim Felipe Roa Fuentes, Doctor of Philosophy.

da con A-Connect 70 % que alcanza una precisión del 72,19 % en el dataset Speech Commands, un 27,54 % más que el modelo base. También proporcionamos una demostración que ofrece una interfaz web permitiendo al usuario grabar un clip de audio y realizar inferencia en FPGA.

ABSTRACT

TITLE: IMPLEMENTATION AND VALIDATION OF A-CONNECT IN SPEECH RECOGNITION APPLICATIONS [*]

AUTHORS: JOSÉ DAVID AMAYA HERNÁNDEZ, KAREN DAYANNA LEÓN SUÁREZ [**]

KEYWORDS: RECURRENT NEURAL NETWORKS, DEEP NEURAL NETWORKS, KEYWORD SPOTTING.

DESCRIPTION:

Speech recognition systems allow us to interface with systems using our voice, useful for applications such as automated customer service, voice assistants, etc. The development of these systems has benefited from advances in Deep Learning making them more reliable and precise. However, the deployment of these systems requires a high amount of hardware resources and power to achieve good performance. Analog neural network accelerators are a possible solution as they provide fast performance with low power consumption at the cost of lower accuracy because they are susceptible to stochastic variability. A solution to this problem is A-Connect, a training methodology that increases analog neural network accuracy resilience to stochastic variability. An A-Connect library has already been developed in a previous undergraduate project containing implementations for fully connected and convolutional layers. In this work, the library is extended to different kinds of recurrent neural networks (RNNs) used in speech recognition applications such as: Fast, Accurate, Stable, and Tiny Gated Recurrent Neural Network (FastGRNN), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU). We present results using the implemented layers on different models trained and tested on the Spoken Digits and Speech Commands datasets. A-Connect shows better performance when error is applied compared to the base model, for example, for the LSTM-S model, achieving up to 68.25 % accuracy on Speech Commands dataset using A-Connect 70 %, which is 22.78 % higher than the base model. In addition, we present the FPGA implementation of a GRU model. A large version which achieves 94.78 % on Speech Commands dataset, and a small version trained with A-

---

[*] Bachelor Thesis

[**] Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones. Director: Elkim Felipe Roa Fuentes, Doctor of Philosophy.

Connect 70 % which achieves an accuracy of 72.19 % on Speech Commands dataset, 27.54 % higher than the base model. We also provide a demo that offers a web interface that allows the user to record an audio clip and perform inference on FPGA.

INTRODUCTION

Speech recognition systems recognize elements of speech by analysis of the acoustic signal and allow a program to do things such as detect the presence of specific words in a given audio signal or process the human speech into a written format. These systems are particularly useful in enabling computer interaction via voice, resulting in their widespread adoption by major companies in applications such as automated customer service, voice control, voice assistants, etc.

The development of speech recognition systems has benefited from advances in deep neural networks (DNNs) and Recurrent Neural Networks (RNNs) in particular, given their ability to process sequential data [1][2][3][4]. DNNs and RNNs require significant resources, and thereby the use of neural network accelerators. DNN digital accelerators with Von Neumann architecture are commonly used and are prone to problems such as high data latency and high energy cost. Recent works have tried to combine computation-in-memory (CIM) architectures with analog processing/memory technologies that act as synapses of the neural network to achieve higher energy efficiency, however, analog accelerators are more susceptible to hardware non-idealities compared to their digital counterparts.

The problems mentioned above converge in A-Connect, a training methodology de-

---

[1] Yanzhang He et al. *Streaming End-to-end Speech Recognition For Mobile Devices*. 2018. DOI: 10.48550/ARXIV.1811.06621.

[2] Dario Amodei et al. *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*. 2015. DOI: 10.48550/ARXIV.1512.02595.

[3] Yundong Zhang, Naveen Suda, Liangzhen Lai y Vikas Chandra. *Hello Edge: Keyword Spotting on Microcontrollers*. 2017. DOI: 10.48550/ARXIV.1711.07128.

[4] Aditya Kusupati et al. *FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network*. 2019. DOI: 10.48550/ARXIV.1901.02358.

veloped in a Ph.D. thesis from the Integrated Systems Research Group - OnChip[5], to mitigate the analog computation stochastic variability in neural networks. A-Connect increases accuracy resilience to stochastic variability because it considers different types of neural network parameter corruption during training. An A-Connect library [6] has already been developed in a previous undergraduate project [7] using existing APIs such as Keras [8] and TensorFlow [9], but currently, this library only has the implementation for fully connected and convolutional layers.

To extend A-Connect to more types of neural networks, this project makes the following contribution:

- Introduce multiple kinds of RNN layers with A-Connect to the library with stochastic variability mitigation: FastGRNN, LSTM, GRU.
- Update some auxiliary functions in the library that allow the evaluation of the performance of a neural network model in a noisy environment to support the newly implemented layers.

This document is divided as follows: Section 2 presents the A-Connect methodology used in the present work; Section 3 describes speech recognition, along with the applications, models, datasets, feature extraction used in this work, and a description of the models used; Section 4 describes the implementation of the A-Connect methodology in each of the cells used; Section 5 shows the results obtained after

---

[5] Luis Rueda y Elkim Roa. “A-Connect: Enabling Imprecise Analog Computation”. (unpublished work).

[6] J. Amaya et al. *Library for A-Connect*. 2022.

[7] Edward Albán Silva Jiménez, Ricardo Matheo Vergel Sanabria, Luis Rueda y Elkim Roa. “Library Development For A-Connect”. previous undergraduate project. 2021.

[8] François Chollet. *Deep learning API Keras*. [Online] Available: https://keras.io/. 2015.

[9] Google brain team. *Machine learning platform TensorFlow*. 2015.

the implementation; Section 6 describes the implementation and validation of the A-Connect methodology through FPGA acceleration; finally, Section 8 presents the conclusions of the work.

1. OBJECTIVES

1.1. GENERAL OBJECTIVE

- To implement the A-Connect methodology in speech-recognition applications.

1.2. SPECIFIC OBJECTIVES

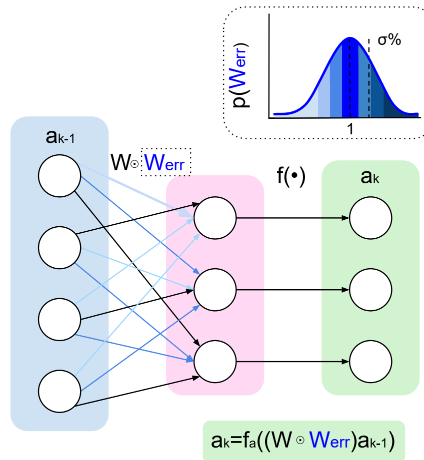
- To extend the A-Connect library for Speech-Recognition applications.
- To test the efficacy of the A-Connect methodology in deep neural networks with several well-studied datasets (e.g., Speech Accent Archive, Google Speech Commands Dataset, Synthetic Speech Commands Dataset, Common Voice, etc.)
- To validate A-Connect through FPGA acceleration.

2. A-CONNECT METHODOLOGY

A-Connect is a training methodology that seeks to mitigate the effects of stochastic variability on the accuracy of analog DNN accelerators by taking into account different types of neural network parameter corruption during training.

To achieve this, in training, during the forward propagation, A-Connect generates a batch of error matrices from a normal distribution for the parameters of the layer (weights and biases), and then multiplies them element-wise before calculating the output, as shown in Figure 2.1 [7] for a fully connected layer. Where a_{k-1} are the input activations, W is the matrix of parameters, W_{err} is the error matrix, f_a is the activation function and a_k are the output activations of the layer. By applying a variation to the parameters of each example of a batch, we are modeling the variation experienced by the parameters in an analog implementation of a neural network, resulting in a regularized model.

Figure 2.1. Neural network architecture for A-Connect



In previous works, the methodology was implemented in fully connected and convolutional layers, supporting the mitigation of stochastic variability and parameter

quantization to different levels during training. The layers were tested with models and datasets used in image classification tasks, models such as AlexNet, VGG-16, LeNet-5, and ResNet20, and datasets such as MNIST and CIFAR-10 [10] [7].

Among the results presented in previous works, in terms of stochastic variability mitigation, for a simulation error of 70 % on the parameters, they presented an accuracy of 80.0 % on the CIFAR-10 dataset with Alexnet using A-Connect at 70 %, which is 14.7 % higher than that of the base model, and an accuracy of 88.6 % on the same dataset with VGG-16 using A-Connect at 70 %, which is 74 % higher than that of the base model [7]. In terms of parameter quantization, for the ResNet20 tested on the CIFAR-10 dataset, they presented an accuracy of 91.3 % on the baseline, 88.7 % accuracy using parameters quantized at 2-bits, and 90.8 % accuracy quantizing at 8-bits [10].

[10] Andrés Felipe Centeno Ochoa, Luis Alejandro Hernández, Luis Rueda y Elkim Roa. “Parameters quantization with A-Connect”. previous undergraduate project. 2022.

3. SPEECH RECOGNITION

In this section, we mention the applications, models, and datasets most commonly used in speech recognition, the criteria of selection for the models used in this work, and the feature extraction process applied to audio signals containing speech.
Speech recognition is the process of recognizing elements of speech by analysis of the acoustic signal [11] allowing a program to detect the presence of words in an audio signal or create a transcription of it for later processing, thus enabling applications such as automated customer service, voice control, voice assistants, etc.

3.1. APPLICATIONS

We identified two main applications of speech recognition: Keyword Spotting (KWS) and Automatic Speech Recognition (ASR). Multiple other applications fall inside those two, for example, wake-word detection being a case of Keyword Spotting; or Speaker dependant speech recognition being a case of Automatic Speech Recognition. In this work, we only focus on the main applications described below.

3.1.1. Keyword spotting This is the process of detecting when a single word is spoken, from a limited set of words, with as few false positives as possible from background noise or unrelated speech. [12]. This application is part of voice assistant devices such as Amazon Echo or Google Home, which work by transmitting the spoken audio to the cloud. These devices don't have always-on speech recognition as

---

[11] *speech recognition*. DOI: 10.1093/oi/authority.20110803100522540.

[12] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018. DOI: 10.48550/ARXIV.1804.03209.

it would not be energy-efficient, could cause network congestion, and cause privacy concerns, so a keyword spotting system is deployed on the voice assistant device. This system continuously hears for a keyword (such as .Alexa.or .Ok Google"), which then wakes the device and activates the full-scale speech recognition. These keyword spotting systems should have low power consumption, high accuracy, and low latency for the best user experience [3].

3.1.2. Automatic Speech Recognition This process enables a program to process human speech into a written format [13]. The input to an ASR system is either a fixed length or continuous stream of recorded speech, and the output is a text transcription of the recording. These systems are present in dictation systems, voice assistants, automated voice-enabled customer service, etc.

3.2. MODELS USED IN SPEECH RECOGNITION APPLICATIONS

To select the models used for testing in this work, we first performed a search of popular models used in the mentioned applications. We then compare computing resources, total training time, and the dataset used to decide whether the implementation is feasible or not, given that the computing resources we have available are limited.

As shown in Table 3.1, the models used in the ASR application have a very high training time even when using a lot of computing resources, also, some of the datasets used are not freely available or are not available at all to the public.

---

[13] IBM Cloud Education. “Speech Recognition”. En: (2020).

Table 3.1. Speech Recognition models

App.[a]	Model name	Dataset used	Training time	Compute resources
ASR	Deepspeech 2	LibriSpeech, WSJ, Switchboard, Fisher	3-5 days	16 GPUs[c]
	RNN-Transducer	VS (Voice Search)[b], IME[b]	3-4 days	8x Google Colab TPUs
	Conformer	LibriSpeech	6-7 days	4x Nvidia RTX 2080 Ti GPUs
KWS	FastGRNN	Google speech commands	0.63 hours	1x P40 GPU
	LSTM	Google speech commands	4.7 hours[d]	Intel Xeon CPU (Kaggle)
	GRU	Google speech commands	4 hours[d]	Intel Xeon CPU (Kaggle)
	CRNN	Google speech commands	5 hours[d]	Intel Xeon CPU (Kaggle)
	KWT1	Google speech commands	9 hours	-

[a]Application

[b]Not publicly available.

[c]GPU model not specified

[d]Tested by ourselves.

3.3. DATASETS USED IN SPEECH RECOGNITION APPLICATIONS

There exists a variety of datasets used to train speech recognition models, some of which are not available to the public. The following are some of the datasets used in ASR and KWS applications. We omit datasets that are not freely available.

3.3.1. LibriSpeech This is a corpus of size 304.47 GiB used to train ASR models, with approximately 1000 hours of read English speech at a sampling rate of 16 kHz with data derived from audiobooks from the LibriVox project [14].

3.3.2. TED-LIUM This is a corpus of English-language TED talks with transcriptions used to train ASR models. It has three versions that differ in the amount of data and the format of the recordings. Version 1 has a size of 39.23 GiB and contains about 118 hours of speech recorded at a 16kHz sample rate. [15]

3.3.3. Google's Speech Commands dataset This is an audio dataset of spoken words whose main objective is to provide a way to build and test models that detect when a single word is uttered out of a set of ten target words (KWS). Two versions of the dataset have been relased, each having two variants of 12 or 35 classes [12]. Version 0.0.2 of the dataset with 12 classes has a size of 8.17 GiB, it is conformed of recordings of a duration of around 1 second. Ten classes are identified with the words: 'Yes', 'No', 'Up', 'Down', 'Left', 'Right', 'On', 'Off', 'Stop', and 'Go' and the two

---

[14] Vassil Panayotov, Guoguo Chen, Daniel Povey y Sanjeev Khudanpur. "Librispeech: an ASR corpus based on public domain audio books". En: *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE. 2015, págs. 5206-5210.

[15] Anthony Rousseau, Paul Deléglise y Yannick Estève. "TED-LIUM: an Automatic Speech Recognition dedicated corpus". En: *Conference on Language Resources and Evaluation (LREC)*. 2012, págs. 125-129.

additional classes are 'silence' and 'unknown' [16].

Table 3.2. Google Speech Dataset splits. Taken from

Split	Examples
test	4,890
train	85,511
validation	10,102

3.3.4. Spoken Digits This is a freely available audio dataset of spoken digits, similar to MNIST but in audio, of size 45.68 MiB consisting of 2,500 recordings of digits spoken by five speakers (50 of each digit per speaker) in WAV files at a sample rate of 8 kHz. Can be used to train KWS models. [17]

3.4. FEATURE EXTRACTION

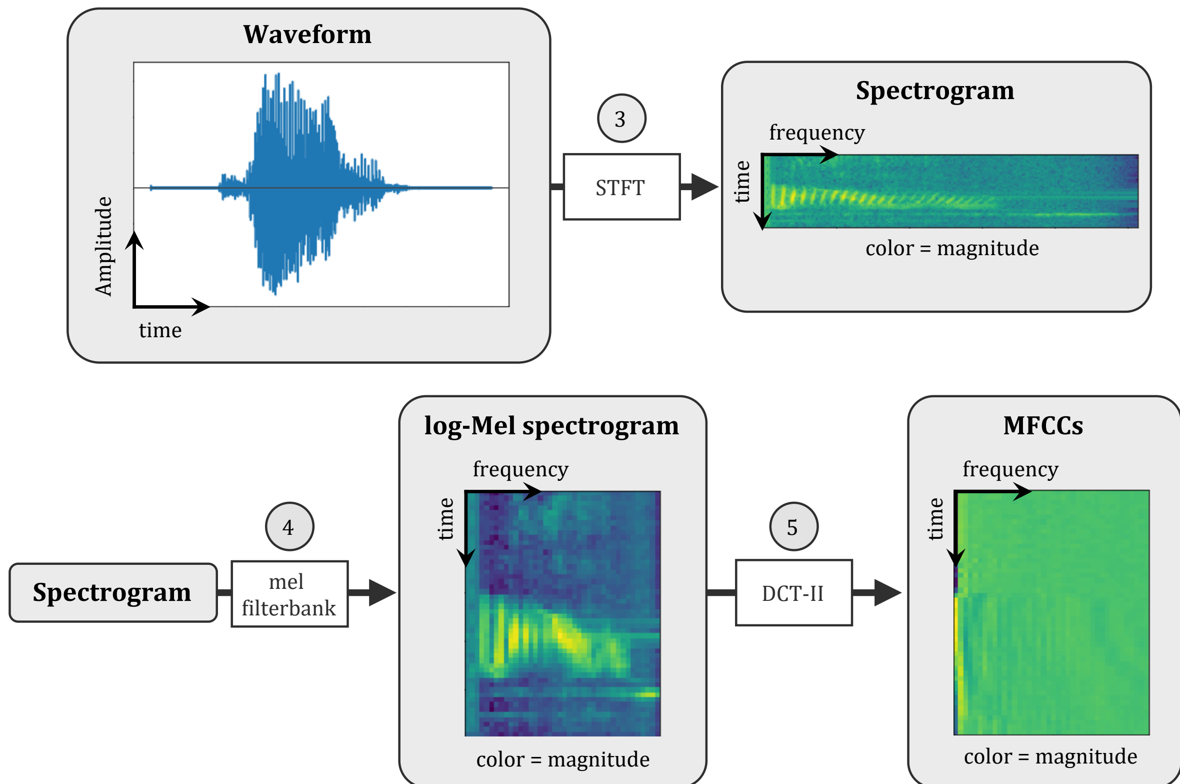
Each example in a dataset is a raw waveform, which is a vector of samples representing the amplitude of the audio signal at a specific time, however, most speech recognition models don't take raw waveforms as inputs, so the waveforms go through a preprocessing pipeline to obtain the features used as input data for the models, this process is known as feature extraction. In this work, we use two types of features, log-Mel spectrogram and Mel-frequency cepstrum coefficients (MFCCs). The steps of the preprocessing pipeline are described below (Figure 3.1).

---

[16] P. Warden. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". En: *ArXiv e-prints* (abr. de 2018). arXiv: `1804.03209 [cs.CL]`.

[17] Zohar Jackson. *Spoken Digit*. 2016.

Figure 3.1. Feature extraction (Steps 3 to 5)



3.4.1. Audio cutting/padding To ensure that every audio example has the same duration, we either discard the extra samples at the end or pad with zeroes at the end so that the duration is of 1 second (16000 samples for a 16kHz sample rate)

3.4.2. (Optional) Data augmentation Data augmentation is a way to artificially boost the range and number of training examples by transforming existing examples to create new examples [18]. We perform data augmentation as described in the paper [3]. We apply a random time shift of up to 100ms and add background noise to around 80 % of the examples with a random volume of up to 10 %, the audio used

---

[18] Google. *Machine Learning Glossary*.

for the background noise is taken from the examples provided in Google's Speech Commands dataset [16].

3.4.3. Short-time Fourier transform (STFT) To convert the time signal into a spectrogram, we apply the Discrete-time STFT. It consists of first dividing the signal into overlapping frames/windows (this is called framing), applying the Discrete Fourier Transform (DFT) to each of the frames, and then taking the magnitude squared of the resulting values [19].

The framing is done by taking a portion of the waveform of a specified duration (called window size) every specified amount of seconds (called stride). The window size and stride are both parameters of this step.

After framing, we end up with T frames of S samples each. T is equal to the duration of the example divided by the stride, and S is equal to the window size (in seconds) multiplied by the sample rate. We then apply the DFT to every frame using Equation (1). [19]

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

where $2 \times K$ is the length or number of points of the DFT ($2 \times K$ is a parameter of this step, usually 512 ($K = 256$)), and $h(n)$ is an S samples long analysis window (for example the hamming window described in Equation (2) [20]). The analysis window is applied to attenuate discontinuities at the window edges.

$$h(n) = 0.54 - 0.46 \cos\left(\frac{2\pi(n-1)}{N-1}\right) \tag{2}$$

---

[19] James Lyons. *Mel Frequency Cepstral Coefficient (MFCC) tutorial*.

[20] Steve Young et al. "The HTK book". En: *Cambridge university engineering department* 3.175 (2002), pág. 12.

Finally, we calculate the magnitude squared of each value $x = |x|^2$ to get the spectrogram which has the shape $T \times K$.

3.4.4. log-Mel scale spectrogram A representation of the spectrogram in the Mel scale approximates the behavior of the auditory system [21]. For an input of shape $T \times K$ each time step goes through a filterbank with F triangular filters (as shown in Figure 3.2 [20]) equally spaced along the mel-scale described in Equation (3) [20]. (F is a parameter, the number of filters, which is also the number of features per time step). Generally, the triangular filters are spread over the whole frequency range from 0 up to half the sample rate (Nyquist frequency).

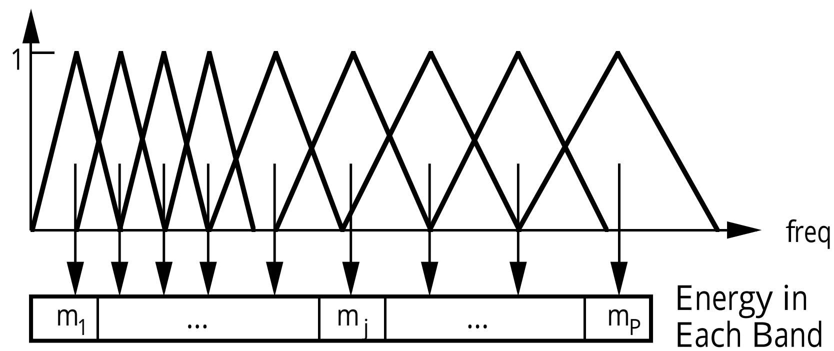


Figure 3.2. Mel-Scale Filter Bank. Taken from

$$\text{mel}(f) = 2595 * \text{log}_{10}\left(1 + \frac{f}{700}\right) \tag{3}$$

After that, we apply the $\log$ function to all of the resulting values. The output shape is $T \times F$ where F is the number of features. If the feature type required is the log-Mel spectrogram, the following preprocessing step is skipped.

---

[21] Xuedong Huang, Alex Acero, Hsiao-Wuen Hon y Raj Reddy. “Spoken Language Processing: A Guide to Theory, Algorithm and System Development”. En: 2001.

3.4.5. MFCCs We take the log-mel scale spectrogram (Of shape $T \times F$) and apply the discrete cosine transform of type 2 (DCT-II) described in Equation (4) (taken from [20]) to every time step of the spectrogram (every row) (The term c_1 is also multiplied by a factor of $1/\sqrt{2}$) (In the equation, $N = F$)

$$c_i = \sqrt{\frac{2}{N}} \sum_{j=1}^{N} m_j \cos\left(\frac{\pi i}{N}(j - 0.5)\right) for\ i = 1, \dots N \tag{4}$$

The output shape after applying the DCT is the same as the input ($T \times F$), but generally, high-frequency coefficients are discarded, resulting in a shape of ($T \times F'$) $F' < F$, where F' is the number of features or number of coefficients. F' is generally a number between 10 and 13.

3.4.6. (Optional) Standardization For each set of features x (Of shape $T \times F$) we perform standardization using Equation (5). Where μ and σ are the mean and standard deviation of x respectively.

$$x_{std} = (x - \mu)/\sigma \tag{5}$$

3.5. MODELS USED IN THIS WORK

We have available two different platforms to run the training of the models in this work: Google Colab, which offers different GPUs on the free tier depending on availability (most of the times, it provides 1x Tesla T4), and Kaggle, which offers 1x Tesla P100 GPU.

Based on the training times alone, testing the ASR models with the resources we have available is not feasible, also datasets used to train ASR models are generally tens of GiBs, and the storage available on the available platforms is limited, so we limit ourselves to working with the KWS models.

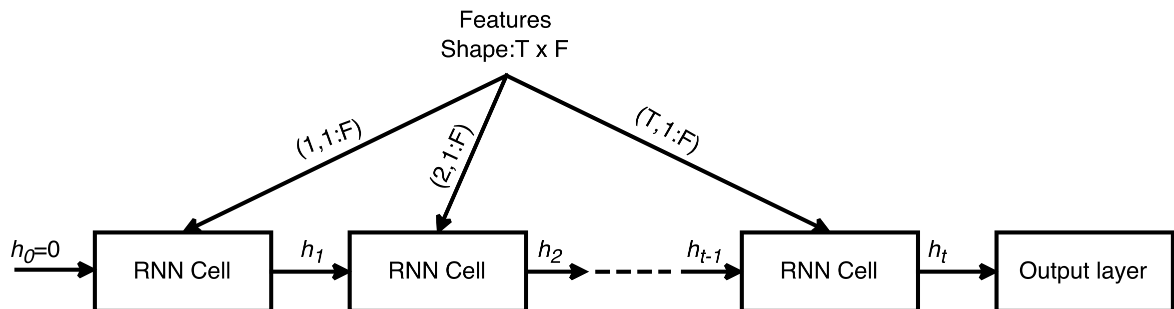
3.5.1. FastGRNN, LSTM, and GRU The architecture of the FastGRNN, LSTM, and GRU models is described in Figure 3.3 [3], where there is a recurrent layer using an RNN cell (which is either one of the FastGRNN, LSTM, or GRU cells) followed by a fully connected layer which is the output layer.

What an RNN layer does is loop through the input data using an RNN cell. For an input X of shape $T \times F$, on each time step t the RNN cell takes the input vector at that time step $X_{t,1:F}$ and the previous output (also called hidden state) of the cell h_{t-1}, and calculates the output at that time step h_t. On the first step $t = 1$, there's no previous output, so it is assumed to be 0 (in some cases it can also be a random value) $h_0 = \vec{0}$. The output of the RNN layer is the output of the RNN cell for the last time step h_T.

The size of each model is controlled with a single hyperparameter called *units*, which is the output size of the recurrent cell used in the layer. The output size of the output layer (Fully connected layer) is the number of classes of the dataset.

For the FastGRNN model, there's an additional layer before the RNN layer, called the Normalization layer, this layer doesn't have trainable parameters. It performs standardization on the input data during training based on the parameters mean (μ) and variance using equation $input_{std} = (input - mean)/\sqrt{var}$. ($\mu$) an ($var$) are set respectively to the mean and variance of all the examples in the training set.

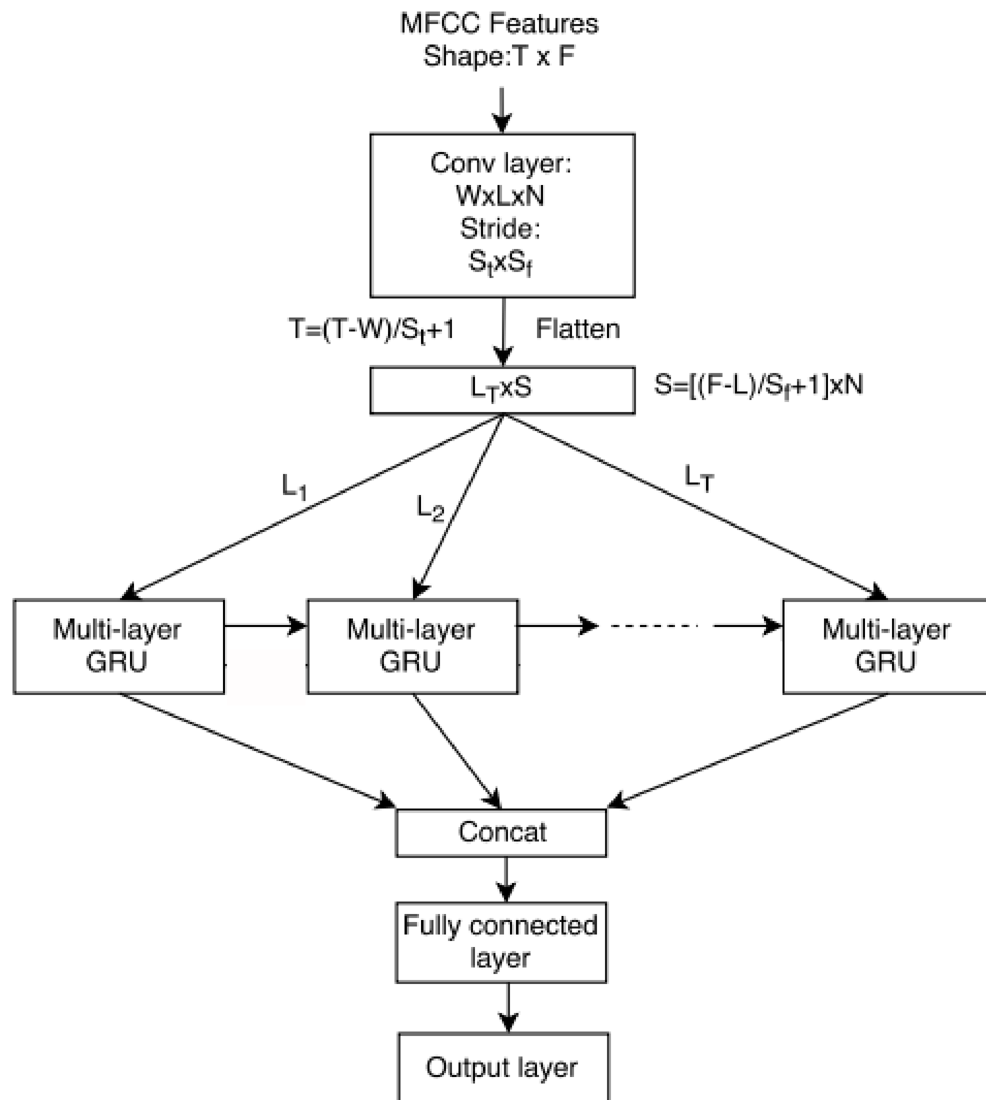
Figure 3.3. Basic architecture of an RNN Model.



3.5.2. CRNN A convolutional recurrent neural network is a hybrid of CNN and RNN, it exploits the local temporal/spatial correlation using convolution layers and global temporal dependencies in the speech features using recurrent layers. [3]

As shown in Figure 3.4 [3], the architecture of the CRNN model is composed of: a 2d convolutional layer, a reshape layer, multiple RNN layers (in this work 2) using a GRU cell, and finally 2 fully connected layers where the last one is the output layer.

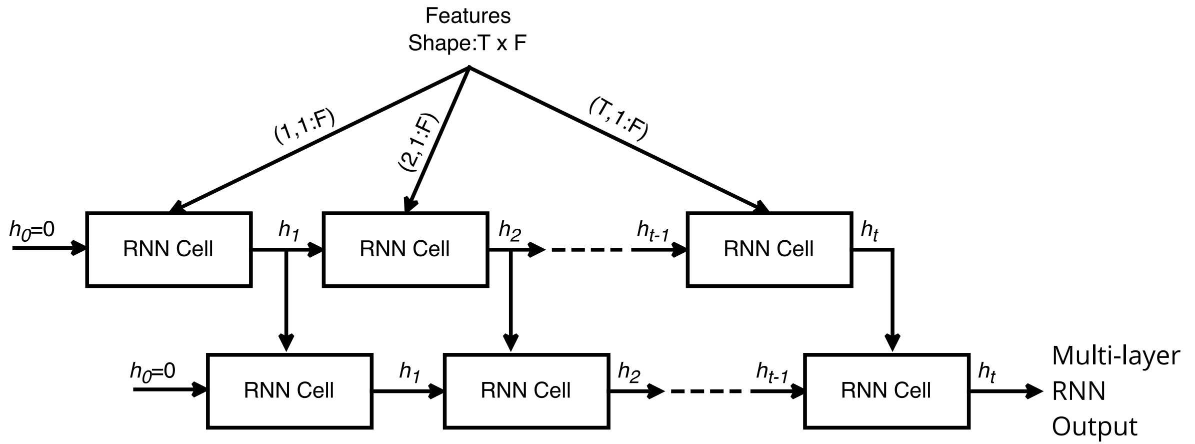
Figure 3.4. CRNN Model. Taken from



The functionality of a 2-layer RNN is shown in Figure 3.5 [3]. Generally, a single RNN layer takes an input of shape $T \times F$ and produces an output of shape *output_size* where *output_size* is the number of activations at the output of the RNN cell used by the layer. The RNN cell in the layer produces an output h_t for each time step in the input (every row), and the final output of the layer is generally h_T, the previous outputs h_1 to h_{T-1} are generally discarded, however, the RNN layer can be configured to output every h_t at every time-step, resulting in an output of shape $T\times$*output_size*, which is 2-dimensional and thus can be used as the input of another RNN layer, in TensorFlow this can be controlled with the parameter *return_sequences*, if *return_sequences = True*, the output of the RNN layer is $T\times$*output_size*, otherwise it is *output_size*. For the 2-layer GRU in the CRNN model, the first layer has *return_sequences = True* and the second layer has *return_sequences = False*.

The size of the model is controlled by the hyperparameters of the convolutional, recurrent, and fully connected layers. The convolutional layer has hyperparameters *N* (Number of filters/conv features), *W* (Height of the filters), *L* (Width of the filters), *St* (Stride in y direction), *Sf* (Stride in x direction). Both layers in the Multi-layer RNN have the same number of units *GRU_units* as hyperparameter (output size of the RNN cells used in the layers). The first fully connected layer has hyperparameter *FC_units* (output size of the layer). And the output size of the output layer (Fully connected layer) is the number of classes of the dataset.

Figure 3.5. 2-Layer RNN.



On Table 3.3, we see the accuracy comparison of different models found in [43], models from [3] have a suffix (L or S) according to the number of parameters of the model. The accuracy of the models is shown for Google's Speech Commands Dataset [16] version 2 of 12 classes (Except for FastGRNN which uses version 1).

Table 3.3. Comparison of different versions of the models found in the papers

Model	**Features**	**In (T,F)**[a]	**Accuracy [%]**		
			train	val	test
FastGRNN-LSQ	log-mel spectrogram	99, 32	-	-	93.18
[b]Basic LSTM-S	MFCCs	49, 10	98.2	91.5	92.0
[b]Basic LSTM-L	MFCCs	49, 10	99.1	93.0	93.4
GRU-S	MFCCs	25, 10	98.4	92.7	93.5
GRU-L	MFCCs	49, 10	99.2	93.9	93.7
CRNN-S	MFCCs	49, 10	98.4	93.6	94.1
CRNN-L	MFCCs	49, 10	99.1	94.4	95.0

[a]Input shape TxF (Timesteps x Features)

[b]In this work, we refer to the Basic LSTM model as just LSTM

4. LIBRARY DEVELOPMENT

Previous work related to the A-Connect library has implementations for the fully connected layer and the 2D convolutional layer.[7] The library also supports training with parameter quantization on the implemented layers. [10].

This work extends the scope of the library by implementing the A-Connect methodology to increase accuracy resilience to stochastic variability in different kinds of recurrent layers, namely FastGRNN, LSTM, and GRU, which are the recurrent layers used in the models we tested. This section describes each of the layers implemented with the A-Connect methodology.

4.1. GENERAL PROCEDURE

The implementation of A-Connect for each cell follows the same general procedure: First, we identify the parameters of the cell (weights, biases, etc.); then, during the forward propagation of a batch of inputs, we generate a batch of error matrices for each of the parameters of the cell (one error matrix per example in the batch, so we end up with *batch_size* error matrices per parameter); after that, we multiply each of the original parameters with the corresponding batch of error matrices resulting in a batch of noisy parameters; finally, we calculate the outputs of the cell by using the noisy parameters. By doing this, we effectively calculate each output in the batch using a different set of noisy parameters for each input.

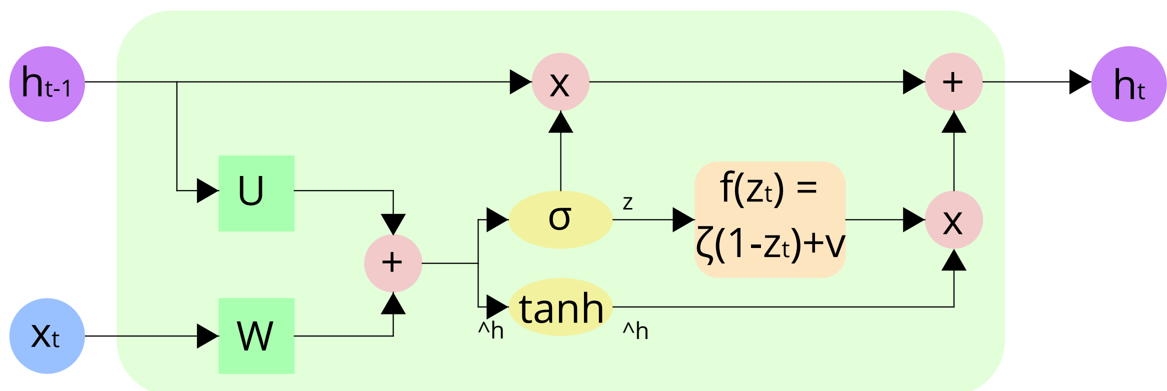
This procedure results in increased memory usage (as we now have *batch_size* different sets of parameters) and also in increased training time in many cases. We can reduce the memory usage and training time when using the A-Connect methodology by generating a lesser amount of error matrices using the parameter *pool*. This parameter sets the number of error matrices per parameter in a batch, so for every

batch of inputs we generate a batch of *pool* error matrices where *pool* <*batch_size*. This way, multiple outputs share the same set of noisy parameters. For example, if *batch_size*$= 100$ and *pool*$= 2$, this means we generate two sets of noisy parameters, and calculate outputs 1 to 50 with the first set of noisy parameters, and the outputs 51 to 100 using the other set of noisy parameters.

4.2. FastGRNN

Fast, Accurate, Stable, and Tiny Gated Recurrent Neural Network (FastGRNN) was developed to address the RNN limitations of inaccurate training and inefficient prediction [4]. FastGRNN uses shared matrices W, U to compute both the hidden state (h_t) as well as the gate (z_t), resulting in less number of parameters compared to the GRU and LSTM [4].

Figure 4.1. FastGRNN Cell



$N_{m,n}(1, \sigma^2_{\%})$ is a matrix of shape $m \times n$ with random values from a distribution of mean 1 and standard deviation $\sigma^2_{\%}$, The value $\sigma^2_{\%}$ depends on the hyperparameters *Wstd* and *Bstd* in the TensorFlow implementation. *tanh* denotes the hyperbolic tangent function, and σ denotes the sigmoid function. (This also applies for Algorithms 2 and 3)

Algoritmo 1: FastGRNN cell with AConnect

Input: Input Data (x), Previous hidden State (h_{t-1})
Output: Cell output (h), Hidden state (h)
function FASTGRNN_CELL(x, h_{t-1})

$W_{err} \leftarrow N_{m_w,n}(1, \sigma^2_{\%})$

$U_{err} \leftarrow N_{m_u,n}(1, \sigma^2_{\%})$

$[b_{zerr}, b_{herr}] \leftarrow N_{2n,1}(1, \sigma^2_{\%})$

$[\zeta_{err}, \nu_{err}] \leftarrow N_{1,2}(1, \sigma^2_{\%})$

$W \leftarrow W \cdot W_{err}$

$U \leftarrow U \cdot U_{err}$

$b_z \leftarrow b_z \cdot b_{zerr}$

$b_h \leftarrow b_h \cdot b_{herr}$

$\zeta \leftarrow \zeta \cdot \zeta_{err}$

$\nu \leftarrow \nu \cdot \nu_{err}$

$z \leftarrow \sigma(Wx + Uh_{t-1} + b_z)$

$\hat{h} \leftarrow tanh(Wx + Uh_{t-1} + b_h)$

$h \leftarrow (\zeta(1 - z) + \nu)\hat{h} + z \odot h_{t-1}$

return h, h

end function

The first step to start the implementation of the A-Connect methodology was to replicate the base implementation of the FastGRNN cell found in the GitHub repository [22] in TensorFlow v2.x (In the repository the implementation is done in TensorFlow v1.x).

For the A-Connect version, we determine the parameters contained in the layer, as shown in Figure 4.1. We have the W and U weight matrices, b_z and b_h bias vectors,

[22] Dennis, Don Kurian and Gaurkar, Yash and Gopinath, Sridhar and Goyal, Sachin and Gupta, Chirag and Jain, Moksh and Jaiswal, Shikhar and Kumar, Ashish and Kusupati, Aditya and Lovett, Chris and Patil, Shishir G and Saha, Oindrila and Simhadri, Harsha Vardhan. *EdgeML: Machine Learning for resource-constrained edge devices*. Ver. 0.4.

and the ζ and ν scalar parameters, and then continue with the general procedure 4.1.

4.3. LSTM

Long Short-Term Memory (LSTM) is a special type of RNN, capable of learning long-term dependencies in the input data [23]. The LSTM cell uses the computation of the simple RNN ($h_{rnn} = tanh(Wx_t + Uh_{t-1} + b)$) as an intermediate candidate for the internal memory cell state ($\hat{c}$), and adds it in a (element-wise) weighted-sum to the previous value of the internal memory state (c_{t-1}), to produce the current value of the memory cell state (c), as shown in Algorithm 2 [24].
The weighted sum is implemented in the equation $c = c_{t-1} \odot f + i \odot \hat{c}$ via element-wise (Hadamard) multiplication (denoted by $\odot$) with gating signals. The gating (control) signals i, f and o denote, respectively, the input, forget, and output gating signals. These control gating signals are a replica of the simple RNN equation h_{rnn}, each of them with its own parameters, the σ limits the gating signals to within 0 and 1.[24].
For the A-Connect implementation of the LSTM cell, we used the base implementation of the LSTM cell found in TensorFlow [25] and modified it to apply A-Connect. For that, we first determine the trainable parameters contained in the layer, as shown in Figure 4.2 and Algorithm 2 we have the W_x (W_{xi}, W_{xf}, W_{xo}, W_{xc}) and W_h (W_{hi}, W_{hf}, W_{ho}, W_{hc}) weight matrices, and bias vectors (b_i, b_f, b_o, b_c). We then continue with the general procedure 4.1.

---

[23] *Understanding LSTM Networks*.

[24] Rahul Dey y Fathi M. Salem. “Gate-variants of Gated Recurrent Unit (GRU) neural networks”. En: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, págs. 1597-1600. DOI: 10.1109/MWSCAS.2017.8053243.

[25] TensorFlow. *tf.keras.layers.LSTMCell*.

Algoritmo 2: LSTM cell with AConnect

Input: Input data (x), Previous cell state (c_{t-1}), Previous output (h_{t-1})
Output: Cell output (h), Cell state (c)
function LSTM_CELL(x, c_{t-1}, h_{t-1})

$[W_{xierr}, W_{xferr}, W_{xoerr}, W_{xcerr}] \leftarrow N_{m_x,4n}(1, \sigma^2_{\%})$

$[W_{hierr}, W_{hferr}, W_{hoerr}, W_{hcerr}] \leftarrow N_{m_h,4n}(1, \sigma^2_{\%})$

$[b_{ierr}, b_{ferr}, b_{oerr}, b_{cerr}] \leftarrow N_{4n,1}(1, \sigma^2_{\%})$

$W_{xi} \leftarrow W_{xi} \cdot W_{xierr}$

$W_{hi} \leftarrow W_{hi} \cdot W_{hierr}$

$W_{xf} \leftarrow W_{xf} \cdot W_{xferr}$

$W_{hf} \leftarrow W_{hf} \cdot W_{hferr}$

$W_{xo} \leftarrow W_{xo} \cdot W_{xoerr}$

$W_{ho} \leftarrow W_{ho} \cdot W_{hoerr}$

$W_{xc} \leftarrow W_{xc} \cdot W_{xcerr}$

$W_{hc} \leftarrow W_{hc} \cdot W_{hcerr}$

$b_i \leftarrow b_i \cdot b_{ierr}$

$b_f \leftarrow b_f \cdot b_{ferr}$

$b_o \leftarrow b_o \cdot b_{oerr}$

$b_c \leftarrow b_c \cdot b_{cerr}$

$i \leftarrow \sigma(W_{xi}x + W_{hi}h_{t-1} + b_i)$

$f \leftarrow \sigma(W_{xf}x + W_{hf}h_{t-1} + b_f)$

$o \leftarrow \sigma(W_{xo}x + W_{ho}h_{t-1} + b_o)$

$\hat{c} \leftarrow tanh(W_{xc}x + W_{hc}h_{t-1} + b_c)$

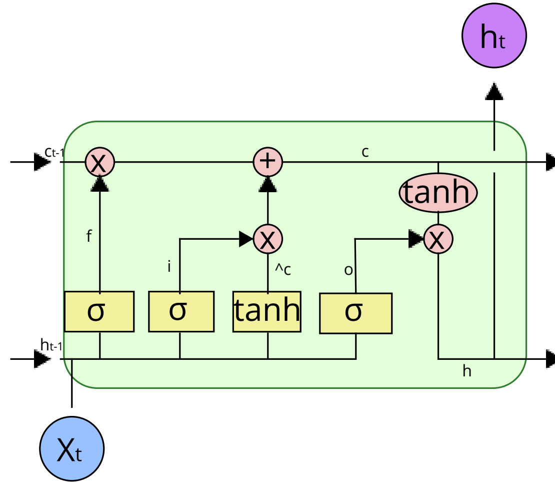
$c \leftarrow c_{t-1} \odot f + i \odot \hat{c}$

$h \leftarrow tanh(c) \odot o$

return h, c

end function

Figure 4.2. LSTM Cell



4.4. GRU

GRU reduces the gating signals to two compared to the LSTM. The two gates are called the update gate z and the reset gate r. The output h is calculated using the update gate z and a candidate $\hat{h}$, where $\hat{h}$ is calculated using the simple RNN equation but gating h_{t-1} using the reset gate r via element-wise multiplication. [24]

For the A-Connect implementation of the GRU cell, we started with the base implementation of the GRU cell found in TensorFlow [26] and modified it to apply A-Connect. For that, we first determine the trainable parameters contained in the layer, as shown in Figure 4.3 and Algorithm 3 we have the W (W_z, W_r, W_h) and U (U_z, U_r, U_h) weight matrices, and bias vectors (b_z, b_r, b_h, $b_r z$, $b_r r$, $b_r h$). We then continue with the general procedure 4.1. Note that the GRU cell implementation has two variants controlled by the hyperparameter *reset_after*, if *reset_after = True* we have 6 bias vectors (b_z, b_r, b_h ,b_{rz}, b_{rr}, b_{rh}), if not, we only have 3 bias vectors (b_z, b_r, b_h).

---

[26] TensorFlow. *tf.keras.layers.GRUCell*.

Algoritmo 3: GRU cell with AConnect

Input: Input data (x), Previous hidden state (h_{t-1})
Output: Cell output (h), Hidden state (h)
function GRU_CELL(x, h_{t-1})

$[W_{zerr}, W_{rerr}, W_{herr}] \leftarrow N_{m_w,3n}(1, \sigma^2_{\%})$

$[U_{zerr}, U_{rerr}, U_{herr}] \leftarrow N_{m_u,3n}(1, \sigma^2_{\%})$

$[b_{zerr}, b_{rerr}, b_{herr}] \leftarrow N_{3n,1}(1, \sigma^2_{\%})$

$W_z \leftarrow W_z \cdot W_{zerr}$

$W_r \leftarrow W_r \cdot W_{rerr}$

$W_h \leftarrow W_h \cdot W_{herr}$

$U_z \leftarrow U_z \cdot U_{zerr}$

$U_r \leftarrow U_r \cdot U_{rerr}$

$U_h \leftarrow U_h \cdot U_{herr}$

$b_z \leftarrow b_z \cdot b_{zerr}$

$b_r \leftarrow b_r \cdot b_{rerr}$

$b_h \leftarrow b_h \cdot b_{herr}$

if *reset_after* **then**

$[b_{rzerr}, b_{rrerr}, b_{rherr}] \leftarrow N_{3n,1}(1, \sigma^2_{\%})$

$b_{rz} \leftarrow b_{rz} \cdot b_{rzerr}$

$b_{rr} \leftarrow b_{rr} \cdot b_{rrerr}$

$b_{rh} \leftarrow b_c \cdot b_{rherr}$

$z \leftarrow \sigma(W_z x + b_z + U_z h_{t-1} + b_{rz})$

$r \leftarrow \sigma(W_r x + b_r + U_r h_{t-1} + b_{rr})$

$\hat{h} \leftarrow (U_h h_{t-1} + b_{rh}) \odot r$

else

$z \leftarrow \sigma(W_z x + b_z + U_z h_{t-1})$

$r \leftarrow \sigma(W_r x + b_r + U_r h_{t-1})$

$\hat{h} \leftarrow U_h(h_{t-1} \odot r)$

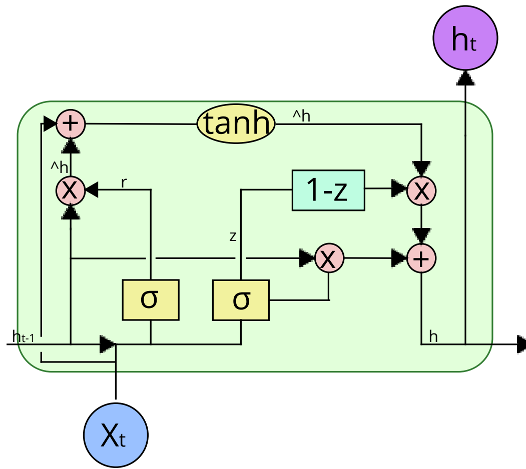
$\hat{h} \leftarrow tanh(\hat{h} + W_h x + b_h)$

$h \leftarrow z \odot h_{t-1} + (1 - z) \odot \hat{h}$

return h, h

end function

Figure 4.3. GRU Cell



4.5. HYPERPARAMETERS

Each of the cells has a set of hyperparameters shown in Table 4.1, which are adjusted based on the implemented model. Some of these hyperparameters are of each cell itself, and some are parameters of the A-Connect methodology.

Table 4.1. Cell hyperparameters

Hyperparameter	**Description**
units	The number of neurons or output activations
activation	Activation function to use
recurrent_activation [a]	Activation function to use for the recurrent step
update_activation [b]	Activation function to use.
Wstd	Float between 0 and 1 for the weights standard deviation for training.
Bstd	Float between 0 and 1 for the bias standard deviation for training.
isBin	String yes or no for weights binarization.
pool	Integer for the pool size of the weights and bias error matrices.
errDistr	Error distribution
d_type	Type of the trainable parameters and error matrices.
reset_after [c]	GRU convention (whether to apply reset gate after or before matrix multiplication). False = "before", True = .[a]fter"

[a]Only for LSTM and GRU cells

[b]Only for FastGRNN cell

[c]Only for GRU cell

5. RESULTS

In this section, we present the results obtained using the implemented layers [Section 4] with the FastGRNN, LSTM, GRU, and CRNN models [Section 3.5] using the Spoken Digits 3.3.4 and Google's Speech Commands datasets 3.3.3.

5.1. SPOKEN DIGITS DATASET

LSTM, GRU, and CRNN models are trained on the Spoken Digits dataset. Then, a 1000-sample Monte Carlo simulation is performed with a simulation error of 30 % 50 % and 70 %, obtaining the results of Table 5.1. We only use the small (S) variants of the models.

5.1.1. Hyperparameters The dataset is split into train, validation, and test with size 80 %/10 %/10 %, resulting in a training split with 2000 examples, and validation and test split with 250 examples each. Note that the dataset is very small.

Each of the models was trained with the following common training parameters: a pool of 2, a batch size of 100, using the Adam optimizer, for a duration of 90 epochs, with a learning rate (lr) $lr = 5e-4$ for the first 29 epochs and $lr = 1e-4$ for the remaining epochs.

The following are the model-specific parameters.

- LSTM-S with units = 118.
- GRU-S with units = 154.
- CRNN-S with GRU_units = 60, N = 48, W = 10, L = 4, St = 2, Sf = 1, and FC_units = 84

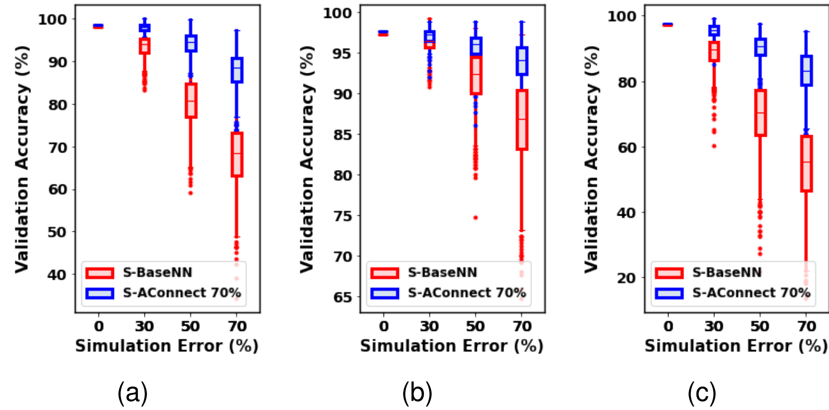
The feature extraction parameters are the same for all the models: we used 10 MFCCs, performed no data augmentation nor feature standardization, used a stride of 20 ms and a window size of 40 ms.

5.1.2. Monte Carlo experiments In the results, we observe that the models trained with A-Connect, and in particular those trained with a higher variability perform better than the other models when increasing the simulation error. For example, for the LSTM model tested with 70 % simulation error, the Base model gives an accuracy of 68.4 %. On the other hand for the model trained using A-Connect with 70 % variability, the accuracy is 88.4 %, which is 20 % higher than that of the base model. We see similar results for the CRNN model with 70 % simulation error, where the accuracy of the model trained using A-Connect with 70 % variability is 28 % higher than that of the base model.

Table 5.1. Spoken Digits Results

Model	**Sim. Error**	**Base-NN**	**A-Connect**		
			30 %	**50 %**	**70 %**
LSTM	**0 %**	*98.00 %/0.00 %*	*98.79 %/0.00 %*	*96.79 %/0.00 %*	*98.40 %/0.00 %*
	30 %	*93.99 %/3.20 %*	*95.59 %/1.99 %*	*96.39 %/1.2 %*	*98.00 %/1.2 %*
	50 %	*80.80 %/7.99 %*	*86.00 %/6.40 %*	*91.60 %/4.39 %*	*94.4 %/3.59 %*
	70 %	*68.4 %/9.99 %*	*75.59 %/8.79 %*	*83.20 %/7.99 %*	*88.40 %/5.59 %*
GRU	**0 %**	*97.2 %/0.00 %*	*96.79 %/0.00 %*	*96.79 %/0.00 %*	*97.6 %/0.00 %*
	30 %	*96.39 %/1.60 %*	*95.99 %/1.19 %*	*96.79 %/1.20 %*	*97.2 %/1.20 %*
	50 %	*92.40 %/4.40 %*	*93.99 %/2.40 %*	*95.2 %/2.39 %*	*95.99 %/1.99 %*
	70 %	*86.79 %/7.19 %*	*91.20 %/4.39 %*	*93.59 %/3.20 %*	*93.99 %/3.19 %*
CRNN	**0 %**	*97.20 %/0.00 %*	*98.40 %/0.00 %*	*98.00 %/0.00 %*	*97.60 %/0.00 %*
	30 %	*89.60 %/5.59 %*	*94.80 %/3.20 %*	*95.99 %/2.39 %*	*95.59 %/2.39 %*
	50 %	*70.39 %/13.60 %*	*83.20 %/10.39 %*	*88.60 %/7.20 %*	*90.79 %/4.79 %*
	70 %	*55.19 %/16.8 %*	*69.99 %/13.19 %*	*78.79 %/11.19 %*	*83.20 %/8.79 %*

Figure 5.1. Results of Spoken Digits Dataset for: a. LSTM-S BaseNN and A-Connect 70 % b. GRU-S BaseNN and A-Connect 70 % c. CRNN-S BaseNN and A-Connect 70 %



5.2. GOOGLE'S SPEECH COMMANDS DATASET

FastGRNN, LSTM, GRU, and CRNN models are trained on Google's Speech Commands Dataset (a large dataset commonly used for KWS applications). Then, a 1000-sample Monte Carlo simulation is performed with a simulation error of 30 %, 50 %, and 70 %, obtaining the results in Tables 5.3, 5.4, 5.5 and 5.6.

5.2.1. Hyperparameters Most hyperparameters are taken from [4] and [3]. Each of the LSTM, GRU, and CRNN models were trained with the following common training parameters: a pool of 2, a batch size of 100, using the Adam optimizer, for a duration of 30 epochs with a learning rate (lr) $lr = 5e-4$ for the first 9 epochs, $lr = 1e-4$ between epochs 10 and 19 and $lr = 2e-5$ for the remaining epochs.
The following are the LSTM, GRU, and CRNN model-specific parameters. The description of each parameter is found in Section 3.5.

- LSTM size S with units = 118, size M with units = 214, size L with units = 344
- GRU size S with units = 154, size M with units = 250, size L with units = 400

- CRNN with L = 4, St = 2, Sf = 1, W = 10 for all sizes
  - S with GRU_units = 60, N = 48, FC_units = 84
  - M with GRU_units = 76, N = 128, FC_units = 164
  - L with GRU_units = 136, N = 100, FC_units = 188

The feature extraction parameters are the same for LSTM, GRU (M, L), and CRNN models, we used 10 MFCCs, performed data augmentation applying random time shift of up to 100 ms and background noise to around 80 % of the samples with a volume of up to 10 %, and performed no feature standardization, used a stride of 20 ms and a window size of 40 ms. The GRU of size S uses a stride of 40 ms and all of the other feature extraction parameters are the same as for the other models.
The FastGRNN model was trained with the following training parameters: a pool of 2, a batch size of 100, using the SGD optimizer with Nesterov momentum of 0.1, for a duration of 300 epochs, with a learning rate (lr) $lr = 1e-2$ for the first 200 epochs, and $lr = 1e-3$ for the remaining epochs. There are two versions of the FastGRNN model, one with units = 100 and a bigger version with units = 400.
The feature extraction parameters for the FastGRNN model are: log-Mel spectrogram of 32 features, performed no data augmentation nor feature standardization, we used a stride of 10 ms, and a window size of 25 ms.

5.2.2. Comparison of the base model and base implemented version We perform an accuracy comparison between the results obtained in the papers [43] and the results we obtained (Shown in Table 5.2). Note that the difference in accuracy between the published results and the version we implemented might be due to a number of factors, one of them being the distribution of the examples in each split, given that the results on the test split are low in comparison even though we achieved better results in the validation split. Also, note that in the paper [4], the FastGRNN model

was trained with the 0.0.1 version of Google's Speech Commands dataset.

Table 5.2. Accuracy Comparison

NN Model	Base Model (paper) [%]			Base Model (Ours) [%]		
	train	val	test	train	val	test
FastGRNN	-	-	93.18	97.31	94.63	90.06
LSTM-L	99.1	93.0	93.4	98.14	95.42	90.12
GRU-L	99.2	93.9	93.7	98.43	95.88	91.61
CRNN-L	99.1	94.4	95.0	97.86	95.51	9139

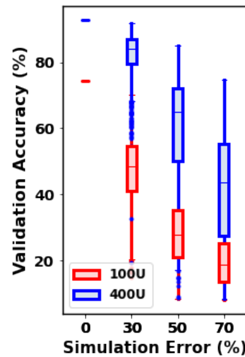
5.2.3. Monte Carlo experiments

FastGRNN For the FastGRNN model, two versions of the model are trained, the first one using 100 output activations in the recurrent layer (which is the number of output activations used in [4]) and the second one using 400 output activations. As shown in Table 5.3 and Figure 5.2, an improvement in the accuracy resilience is observed when increasing the number of units, this is expected, given that increasing the output size increases the number of parameters of the model, and neural networks with higher amount of parameters are less vulnerable to parameter stochastic variability.

Table 5.3. FastGRNN Results

Model Size # units.	Sim. Error	Base-NN	A-Connect		
			30 %	50 %	70 %
100	**0 %**	*90.06 %/0.00 %*	*89.63 %/0.00 %*	*76.69 %/0.00 %*	*74.19 %/0.00 %*
	30 %	*34.51 %/18.35 %*	*53.47 %/18.75 %*	*43.47 %/15.64 %*	*48.42 %/13.72 %*
	50 %	*16.28 %/9.46 %*	*24.31 %/16.16 %*	*23.87 %/13.36 %*	*27.74 %/14.57 %*
	70 %	*12.01 %/5.51 %*	*15.69 %/9.78 %*	*16.82 %/10.25 %*	*18.67 %/11.70 %*
400	**0 %**	*92.17 %/0.00 %*	*92.78 %/0.00 %*	*91.94 %/0.00 %*	*92.84 %/0.00 %*
	30 %	*70.96 %/24.80 %*	*78.64 %/13.63 %*	*81.27 %/7.84 %*	*83.91 %/7.52 %*
	50 %	*39.08 %/29.14 %*	*48.36 %/26.56 %*	*58.89 %/28.88 %*	*64.99 %/22.10 %*
	70 %	*22.33 %/18.35 %*	*31.51 %/21.85 %*	*36.73 %/29.33 %*	*43.40 %/27.85 %*

Figure 5.2. Results for: FastGRNN 100 units A-Connect 70 % and FastGRNN 400 units A-Connect 70 %



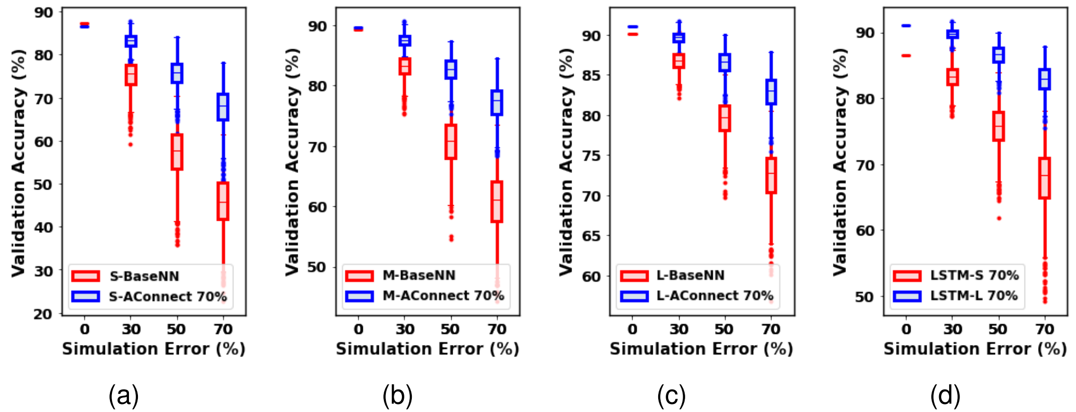
LSTM Table 5.4 shows the results obtained with the Monte Carlo simulation of the three sizes for different noise levels and A-Connect. We can see that in the version of size S an accuracy of 45.78 % is obtained with a simulation error of 70 %, and when applying A-Connect at 70 %, the accuracy is higher, reaching a value of 68.25 %.
In the box diagrams of Figures 5.4(a), 5.4(b), and 5.4(c) we can observe what's previously mentioned when comparing the accuracy between the base-NN and A-Connect at 70 % for the three sizes. On the other hand, the box diagram (d) shows the comparison between the versions of sizes S and L with A-Connect at 70 %, going from 68.25 % accuracy to 83.01 %, because the number of parameters is higher.

GRU Table 5.5 shows the results of the Monte Carlo simulation for different simulation errors and A-Connect levels. In Figures 5.5(a), 5.5(b), and 5.5(c), we can see the comparison between the Base-NN and GRU with A-Connect 70 % for each of the sizes, where better accuracy is obtained by using A-Connect. Also, for each of the GRU model sizes, it is evident that it performs better than the LSTM model, for example, achieving about 5 % more in the L-size model with 70 % simulation error and A-Connect 70 %.

Table 5.4. LSTM Results

Model Size # Params.	Sim. Error	Base-NN	A-Connect 30 %	A-Connect 50 %	A-Connect 70 %
S 62,316	0 %	*87.30 %/0.00 %*	*88.20 %/0.00 %*	*87.60 %/0.00 %*	*86.52 %/0.00 %*
	30 %	*75.65 %/4.41 %*	*81.21 %/3.13 %*	*83.25 %/2.26 %*	*83.33 %/2.22 %*
	50 %	*57.74 %/4.41 %*	*68.75 %/6.21 %*	*74.02 %/4.50 %*	*75.82 %/4.27 %*
	70 %	*45.78 %/8.34 %*	*57.88 %/7.49 %*	*65.24 %/6.05 %*	*68.25 %/6.03 %*
M 195,180	0 %	*89.34 %/0.00 %*	*89.85 %/0.00 %*	*90.14 %/0.00 %*	*89.57 %/0.00 %*
	30 %	*83.33 %/2.43 %*	*85.68 %/1.89 %*	*87.17 %/1.59 %*	*87.57 %/1.41 %*
	50 %	*70.85 %/5.46 %*	*77.46 %/3.95 %*	*81.18 %/3.27 %*	*82.68 %/2.78 %*
	70 %	*61.13 %/6.53 %*	*69.01 %/5.32 %*	*74.82 %/4.17 %*	*77.50 %/3.82 %*
L 492,620	0 %	*90.12 %/0.00 %*	*90.28 %/0.00 %*	*91.28 %/0.00 %*	*91.08 %/0.00 %*
	30 %	*86.74 %/1.51 %*	*88.11 %/1.34 %*	*89.69 %/0.96 %*	*89.75 %/0.98 %*
	50 %	*79.68 %/3.16 %*	*83.14 %/2.60 %*	*86.02 %/2.04 %*	*86.68 %/2.02 %*
	70 %	*72.76 %/4.27 %*	*77.68 %/3.62 %*	*81.64 %/3.05 %*	*83.01 %/2.84 %*

Figure 5.3. Results for Google Speech Commands Dataset: a. LSTM-S BaseNN and A-Connect 70 % b. LSTM-M BaseNN and A-Connect 70 % c. LSTM-L BaseNN and A-Connect 70 % d. LSTM-S A-Connect 70 % and LSTM-L A-Connect 70 %

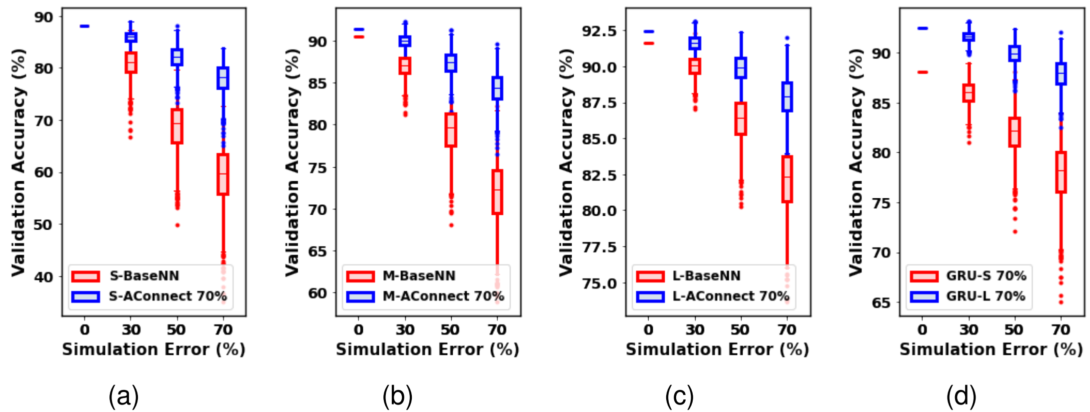


CRNN Table 5.6 and Figure 5.5 show the results obtained after the Monte Carlo simulation. We can see a big difference between the Base and A-Connect 70 % with a simulation error of 70 %.

Table 5.5. GRU Results

Model Size # Params.	Sim. Error	Base-NN	A-Connect 30 %	A-Connect 50 %	A-Connect 70 %
S 78,090	**0 %**	*88.22 %/0.00 %*	*88.61 %/0.00 %*	*88.24 %/0.00 %*	*88.08 %/0.00 %*
	30 %	*81.21 %/3.53 %*	*85.21 %/1.88 %*	*85.90 %/1.78 %*	*86.05 %/1.57 %*
	50 %	*69.45 %/6.31 %*	*78.45 %/4.04 %*	*81.06 %/3.23 %*	*82.13 %/2.87 %*
	70 %	*59.79 %/7.52 %*	*71.39 %/5.54 %*	*76.48 %/4.37 %*	*78.18 %/3.90 %*
M 198,762	**0 %**	*90.59 %/0.00 %*	*91.47 %/0.00 %*	*91.77 %/0.00 %*	*91.39 %/0.00 %*
	30 %	*87.09 %/1.77 %*	*89.01 %/1.34 %*	*90.28 %/1.10 %*	*90.06 %/1.00 %*
	50 %	*79.63 %/3.83 %*	*84.06 %/2.82 %*	*87.10 %/2.24 %*	*87.44 %/1.92 %*
	70 %	*72.20 %/5.07 %*	*78.71 %/3.762 %*	*83.36 %/2.90 %*	*84.39 %/2.59 %*
L 498,012	**0 %**	*91.61 %/0.00 %*	*91.86 %/0.00 %*	*92.43 %/0.00 %*	*92.45 %/0.00 %*
	30 %	*90.04 %/0.98 %*	*90.83 %/0.88 %*	*91.42 %/0.80 %*	*91.62 %/0.72 %*
	50 %	*86.44 %/2.13 %*	*88.24 %/1.65 %*	*89.40 %/1.53 %*	*89.93 %/1.37 %*
	70 %	*82.35 %/3.12 %*	*85.27 %/2.51 %*	*87.05 %/2.08 %*	*87.93 %/1.98 %*

Figure 5.4. Results for Google Speech Commands Dataset: a. GRU-S BaseNN and A-Connect 70 % b. GRU-M BaseNN and A-Connect 70 % c. GRU-L BaseNN and A-Connect 70 % d. GRU-S A-Connect 70 % and GRU-L A-Connect 70 %

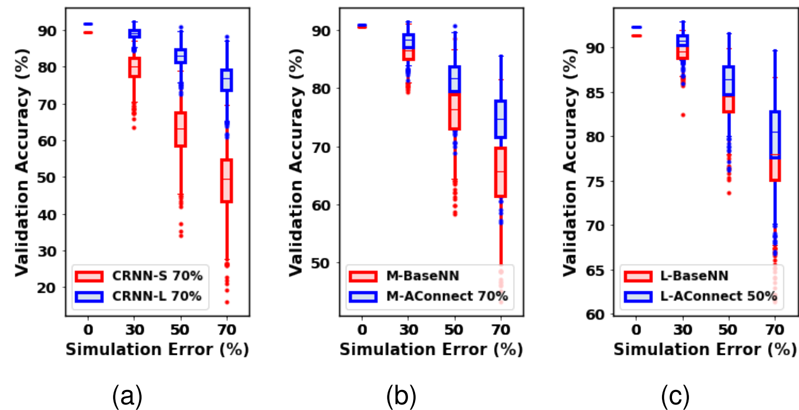


CRNN is a more robust model, but when looking at the results in Table 5.6, the accuracy obtained is lower compared to the LSTM and GRU models since it contains the convolutional layer and the fully connected layer that were not trained with the A-Connect implementation.

Table 5.6. CRNN Results

Model Size # Params.	Sim. Error	Base-NN	A-Connect 30 %	A-Connect 50 %	A-Connect 70 %
S 75,432	**0 %**	*89.47 %/0.00 %*	*90.43 %/0.00 %*	*90.12 %/0.00 %*	*90.37 %/0.00 %*
	30 %	*80.02 %/4.73 %*	*83.92 %/3.59 %*	*83.48 %/3.57 %*	*84.93 %/3.41 %*
	50 %	*63.30 %/9.04 %*	*71.71 %/7.87 %*	*72.12 %/7.24 %*	*74.37 %/6.92 %*
	70 %	*49.63 %/11.16 %*	*59.86 %/10.06 %*	*61.27 %/10.04 %*	*63.92 %/10.07 %*
M 189,032	**0 %**	*90.61 %/0.00 %*	*90.59 %/0.00 %*	*91.80 %/0.00 %*	*91.00 %/0.00 %*
	30 %	*86.50 %/2.72 %*	*86.81 %/2.52 %*	*88.34 %/2.23 %*	*88.28 %/2.15 %*
	50 %	*76.39 %/5.92 %*	*78.64 %/4.77 %*	*81.33 %/4.50 %*	*81.76 %/4.10 %*
	70 %	*65.61 %/8.37 %*	*70.04 %/7.54 %*	*73.94 %/6.20 %*	*74.78 %/6.14 %*
L 485,004	**0 %**	*91.39 %/0.00 %*	*91.10 %/0.00 %*	*92.37 %/0.00 %*	*91.84 %/0.00 %*
	30 %	*89.55 %/1.39 %*	*88.75 %/1.68 %*	*90.82 %/1.10 %*	*89.23 %/1.87 %*
	50 %	*84.56 %/3.30 %*	*82.55 %/3.50 %*	*86.45 %/3.13 %*	*83.10 %/3.58 %*
	70 %	*78.00 %/5.10 %*	*75.92 %/5.61 %*	*80.50 %/5.16 %*	*76.85 %/5.67 %*

Figure 5.5. Results for Google Speech Commands Dataset: a. CRNN-S BaseNN and A-Connect 70 % b. CRNN-M BaseNN and A-Connect 70 % c. CRNN-L BaseNN and A-Connect 50 %



6. FPGA IMPLEMENTATION

In this section, we describe the implementation of a model trained on the google speech commands dataset [16] to run inference on an FPGA. It was written in Verilog using the Xilinx Vivado IDE for the Nexys4 DDR FPGA board. Documentation and source files can be found in the GitHub repository [27].

We first choose a model to implement, based mainly on the accuracy achieved by the model on Google's Speech Commands dataset, but also on the simplicity of the model architecture to make the implementation easier. The FastGRNN model [4] is the simplest model we tested, however, the GRU model [3] is also fairly simple and achieves a better accuracy, so we chose the GRU model. We make some modifications to the base model to achieve a fast implementation that is also power efficient and doesn't use more resources than the ones available on the FPGA we are using.

We perform quantization after training on the parameters and activations of the model (which are represented originally in single precision (32-bit) floating point) to reduce memory footprint and memory bandwidth utilization, we also replace some of the functions used in the model to allow us to perform all operations using only fixed-point integer arithmetic.

6.1. FIXED-POINT INTEGER ARITHMETIC

Most operations performed by the model (such as vector-matrix multiplication, element-wise vector multiplication/addition, etc.) are just a collection of basic operations like multiplication, addition, and subtraction, all of which can be implemented using fixed-point integer arithmetic, however, the model also uses the non-linear sigmoid (sigm)

---

[27] Jose Amaya, Karen Leon. *FPGA-KWS GitHub repository*.

and hyperbolic tangent (tanh) functions which cannot be implemented using fixed-point integer arithmetic, so we train the model with piecewise-linear approximations of these functions (qsigm and qtanh) (taken from [22]), described in Equations (6) and (7), allowing us to do all of the computations with fixed-point integer arithmetic.

$$qsigm(x) = \begin{cases} 0 & (x+1)/2 \leqslant 0 \\ 1 & (x+1)/2 \geqslant 1 \\ (x+1)/2 & 0 < (x+1)/2 < 1 \end{cases} \tag{6}$$

$$qtanh(x) = \begin{cases} -1 & x \leqslant -1 \\ 1 & x \geqslant 1 \\ x & -1 < x < 1 \end{cases} \tag{7}$$

6.2. QUANTIZATION APPROACH

We perform quantization after training using a simpler version of the approach described in [28]. The parameters and activations are represented as 8-bit signed binary fixed-point values, this results in $1/4$ of the memory usage of the original model.
We specify the parameters of the number format using the Q notation. A number in Qm.n format has 1+m+n bits, where *m* is the number of bits used for the integer part of the value (the bits before the binary point) and *n* is the number of fraction bits (bits after the binary point), it can represent values from -2^m to $2^m - 2^{-n}$ (2^{-n} is the resolution) [29]

---

[28] ARM Software. *Keyword spotting for Microcontrollers - Quantization guide*.

[29] Larry D. Pyeatt. "Chapter 8 - Non-Integral Mathematics". En: *Modern Assembly Language Programming with the ARM Processor*. Ed. por Larry D. Pyeatt. Newnes, 2016, págs. 219-264. DOI: `https://doi.org/10.1016/B978-0-12-803698-3.00008-5`.

To quantize the model parameters and activations we use the TensorFlow function *tf.quantization.fake_quant_with_min_max_args* [30], and to find the values *m* and *n* such that after we quantize the model and evaluate it against the test set we get the highest accuracy we do the following steps.

1. Do steps 2-4 for values of *m* from 0 to 7 (*n* = $7 - m$)
2. Replace each layer of the model with a quantized version, which has *fake_quant* applied after every operation (*fake_quant* arguments are min = -2^m, max = $2^m - 2^{-n}$)
3. Apply *fake_quant* to the parameters (weights and biases) of every layer in the model
4. Evaluate the accuracy of the quantized model on the test set
5. Select the values *m* and *n* which result in the highest accuracy

For the implemented model, we achieved the best accuracy using the Q3.4 format. We used a Python script to quantize the model parameters and save them to a file to use with Verilog.

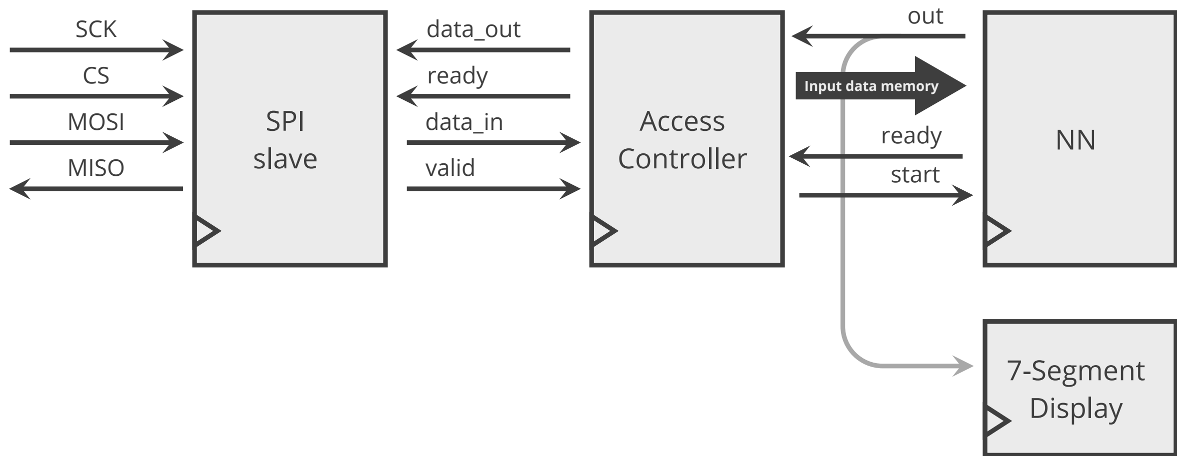
6.3. MODULES

The top module of the implementation consists of the four modules shown in Figure 6.1. In short, the *SPI slave* allows PC-FPGA communication, the *Access Controller* controls the top module based on the data received from the PC, the *NN* module performs all of the processing, and the *7-Segment Display* module displays the predicted word using the 7-Segment display on the FPGA board.

---

[30] TensorFlow. *tf.quantization.fake_quant_with_min_max_args*.

Figure 6.1. NN controller module (Top module)



The *SPI Slave* module enables the communication between the PC and the FPGA. We send the input data from the PC to the FPGA and send the result of the inference (predicted class) from the FPGA to the PC. The PC is the SPI master meaning it always starts the communication, and to receive data from the FPGA it has to send data first.

The *Access Controller* module controls the *NN* module based on the data it receives from the *SPI Slave* module. The process of performing inference for an input data is illustrated in Figure 6.2 and is the following: the PC sends the characters 'STW' through the SPI interface followed by the input data; the *Access Controller* after detecting the sequence 'STW' starts writing the data it receives from the SPI interface to the *input data* memory inside the *NN* module; after all the input data is written, the access controller instructs the *NN* module to start the inference (*NN start = 1*); when it finishes, the result of the prediction can be read through the SPI interface. While the *NN* module is processing the data (*NN ready = 0*), the *Access Controller* instructs the *SPI Slave* to send the byte value 0xFF to the PC, and when the *NN* module is idle (*NN ready = 1*), it sends the last result of the prediction, that way on the PC we can know that the inference has finished if the value read is different from 0xFF. From

the PC, we send 0x00 to the FPGA when we want to read from it.

Figure 6.2. Example of signals for a single inference

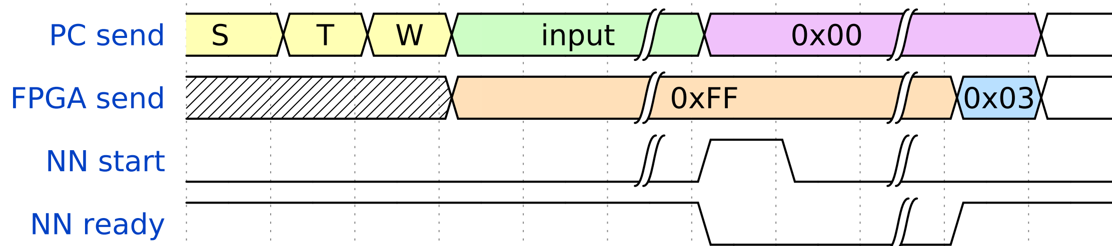


Figure 6.3. NN module

memories
params
input data
temp1
temp2
mem temp1
memories selector
(mux/demux)
mem outputs
mem IN1/IN2 outputs
mem inputs
mem IN1/IN2/OUT inputs
mem IN1
mem IN2
mem OUT
config
memory select IN1/IN2/OUT
vecMatMult
Wx
vecVecFunc
x1+x2
x1⊙x2
vecFunc
σ(x)
tanh(x)
1-x
argMax
out
+1
next Instruction
program counter
program memory

The *NN* module shown in Figure 6.3 is what performs all the computations of the model. It consists of the following blocks:

- A set of memories, where *params* is a ROM used to store the parameters of the model, *input data* is a RAM that contains the input data of the model (the PC writes the data to this memory before starting the inference), *temp1* and *temp2* are both RAMs which are used to store the activations of the model. All of these memories have the same width which by default is 64 bits (meaning that, for example, every time we read from the *params* memory, we are reading a set of 8 values of 8-bits each).

- A set of processing blocks. *vecMatMult* which performs vector-matrix multiplication. *vecVecFunc* which performs either element-wise vector addition or multiplication. *vecFunc* which applies a function to each element of a vector, either qsigm(x), qtanh(x), or f(x) = 1-x. And *argMax* which finds the index of the maximum value in a set of values saved in memory, this is used to get the predicted class.

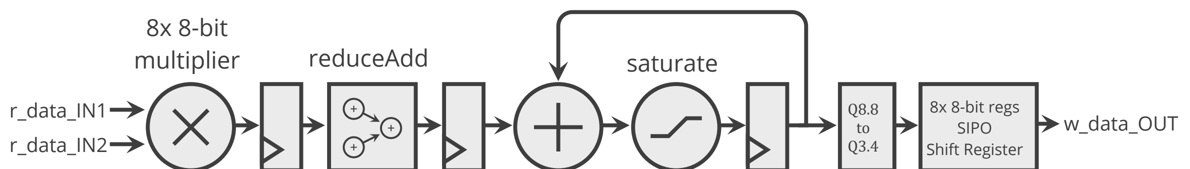
- A set of muxes/demuxes which allow us to choose which memories to read from to get the input data for an operation, and in which memory to write the result of the operation. For example, to perform a matrix multiplication between part of the input data, and a set of parameters while saving the output in *temp1*, we would configure *mem IN1* to be connected to the *params* memory, *mem IN2* to be connected to the *input data* memory and *mem OUT* to be connected to the *temp1* memory.

The processing blocks have a set of wires to configure which memory addresses contain the input data and the addresses where the results of the operation are to be stored. These wires are represented by the *config* input.

The NN is a simple processor, and the blocks that coordinate which processing steps to perform to calculate the output are *program counter* and *program memory*.

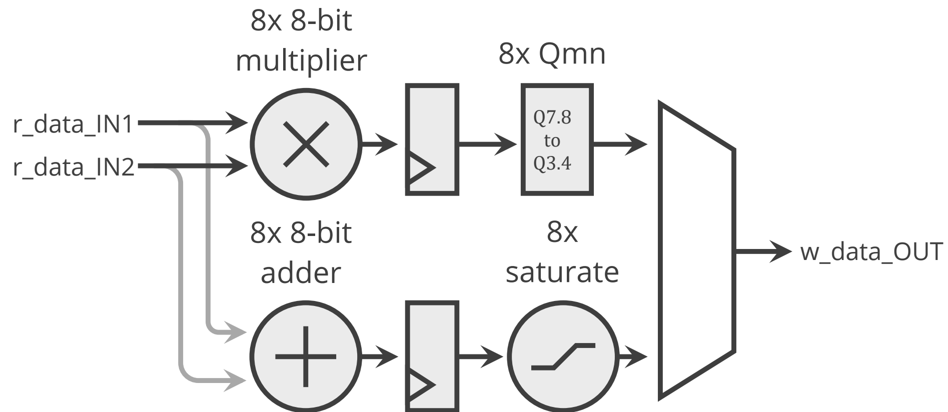
The *program memory* simply outputs a set of configurations given the value of the *program counter*. These configurations are a set of wires called *config* and they control multiple things: The *memories selector* to select the input memories and output memory; the *program counter* to perform a jump to a different instruction; and which processing block is going to run at a certain step.

Figure 6.4. DataPath of vecMatMult module



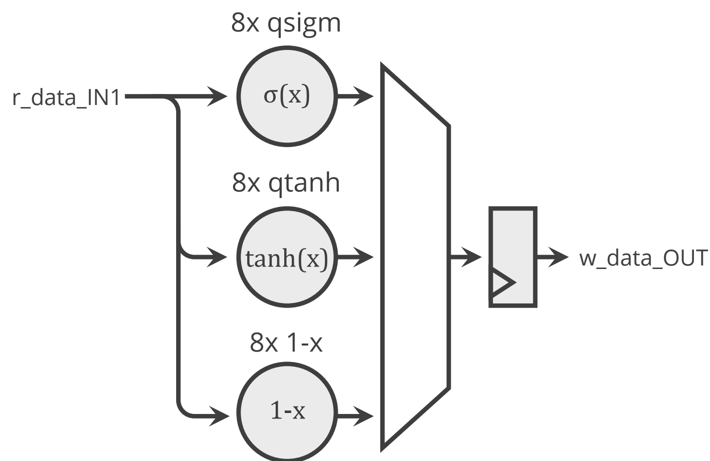
To perform vector-matrix multiplication, the block (Figure 6.4) first takes the data from both input memories (8x 8-bit values each) and multiplies them element-wise, resulting in 8x 16-bit values, these values are added using an adder tree, resulting in one 17-bit value (this value is the dot product of the input data), this value is then fed to an accumulator, which saturates the value after each addition to keep the result of 17-bits and prevents a wraparound when the result of the addition is higher than what can be represented by 17-bits, the output of the accumulator then goes through a block that performs a conversion of the Qm.n format (To keep the format of the result in Q3.4 which is Q7.8 after the multiplication, and Q8.8 after the addition), the result goes to a SIPO shift register of 8x 8-bit values which holds the data to write to the output memory. A state machine controls which addresses to read based on *config*, when to reset the accumulator, when to shift the data in the SIPO SR, and when to write the output data.

Figure 6.5. DataPath of vecVecFunc module



To perform element-wise vector multiplication/addition, the block (Figure 6.5) first takes the data from both input memories (8x 8-bit values each) and sends it through 2 paths. On one path we multiply the data element-wise resulting in 8x 16-bit values in format Q7.8, then we convert each value back to Q3.4 and this result goes to a mux. On the other path, we add the data element-wise resulting in 8x 9-bit values, each of these values is converted to 8-bit by using a saturate block and this result goes to a mux. The mux is controlled with a wire in *config* which defines the function we need to perform.

Figure 6.6. DataPath of vecFunc module



To apply qsigm(x), qtanh(x), or f(x) = 1-x to a vector, the block (Figure 6.6) first takes the data from both input memories (8x 8-bit values each) and sends it through 3 paths, each path applies either qsigm, qtanh or 1-x element-wise and the result goes to a mux. The mux is controlled with a wire in *config* which defines the function we need to perform.

6.4. DEMONSTRATION

A demo of the FPGA implementation is created using the gradio library [31], which provides a web interface that allows the user to record a clip of audio and perform inference on the FPGA.

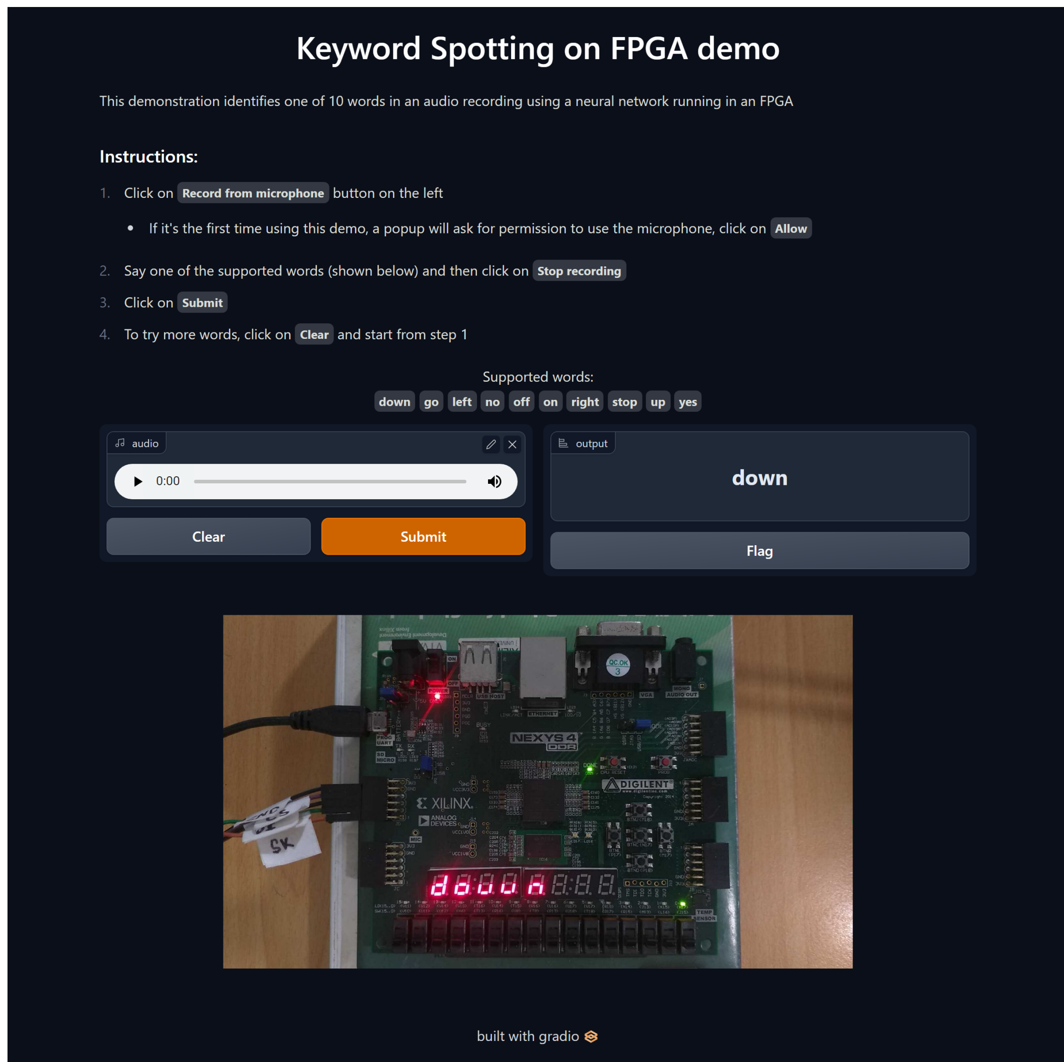
On the demo, each audio received is first resampled to 16kHz and cut to 16000 samples to match the format of the audio in the dataset, then, we perform feature extraction on the audio to get the input features of the model and finally quantize to Q3.4 format. This data is then sent to the FPGA through the SPI interface using the pyftdi library [32], and the result is read through the SPI interface to be displayed on the web interface.

---

[31] Abubakar Abid et al. "Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild". En: *arXiv preprint arXiv:1906.02569* (2019).

[32] Emmanuel Blot,Emmanuel Bouaziz. *PyFtdi*.

Figure 6.7. FPGA demo web interface



6.5. RESULTS

The implementation results are shown for different versions of the GRU model, either by changing hyperparameters of the model, training parameters, or implementation parameters. We run inference on the examples in the test set of Google's Speech Commands Dataset (which contains 4890 examples). We calculate the accuracy and record the time taken to perform inference on all of the examples, from that we

estimate the inference latency using (8) and the throughput using (9).

$$\text{Avg. latency} = \frac{\text{Test time}}{\text{\# of examples}} \tag{8}$$

$$\text{Throughput} = \frac{1}{\text{Avg. latency}} \tag{9}$$

We trained two versions of the GRU model, one large version (L) (with 535212 parameters) and one small version (SC) (with 90732 parameters). The small model was also trained with A-Connect at 30 %, 50 %, and 70 % variability to compare accuracy resilience using Monte Carlo experiments. We also performed post-quantization training on both models. hyperparameters are shown in Table 6.1.
The comparison of the accuracy between the floating point models and the implemented quantized models (in Q3.4 format) with and without post-quantization training (PQT) is shown in Table 6.2. We see that models trained with A-Connect suffer less accuracy degradation when implemented, as the quantization can be considered a type of variability. We also see that post-quantization training can help us achieve the accuracy of the original model (or even higher) and doesn't require training for extended periods (9 epochs for the results obtained).

Table 6.1. Implemented models hyperparameters

Hyperparameter	**Value**
Model	
GRU layer output size	L: 400
	SC: 160
activation	qtanh
recurrent activation	qsigm
Dataset and feature extraction	

Table 6.1. Implemented models hyperparameters

Hyperparameter	**Value**
dataset	google speech commands
feature type	log-mel spectrogram
window size	40 ms
stride	L: 20 ms
	SC: 40 ms
num. features	L: 40
	SC: 24
data augmentation	yes
feature standardization	yes
Training	
optimizer	Adam
batch size	100
epochs	30
learning rate	epochs 1 to 9: 5e-4
	epochs 10 to 19: 1e-4
	epochs 20 to 30: 2e-5
Post-Quantization training	
epochs	9
learning rate	epochs 1 to 2: 5e-4
	epochs 3 to 5: 1e-4
	epochs 6 to 9: 2e-5
A-Connect	
pool	2
errDistr	"normal"

Table 6.2. Accuracy comparison

Model	Accuracy [%]		
	Tensorflow (FP32)	FPGA (Q3.4)	FPGA (Q3.4-PQT)
L	94.21	90.10	94.78
SC	91.33	79.83	92.35
SC-AC30 %	90.63	86.46	-
SC-AC50 %	89.18	86.97	-
SC-AC70 %	89.30	87.11	-

Table 6.3. FPGA implementation results

Model	Accuracy [%]	Test time [s]	Average latency [ms]	Estimated throughput [infer/s]
L	90.10	180.9	36.99	27.031
L-PQT	94.78	180.9	36.99	27.031
L-PQT-128b	94.78	105	21.47	46.571
SC	79.83	24.4	4.989	200.40
SC-AC30 %	86.46	24.4	4.989	200.40
SC-AC50 %	86.97	24.4	4.989	200.40
SC-AC70 %	87.11	24.4	4.989	200.40
SC-PQT	92.35	24.4	4.989	200.40
SC-PQT-128b	92.35	19.9	4.069	245.72
SC-PQT-256b	92.35	17.3	3.537	282.65

PQT: With post-quantization training (No PQT by default)
128b/256b: Describes the width for the memories (64b default)
AC30/50/70 %: Variability applied during training (0 % default)

The SC model has higher throughput because it has fewer parameters, and a smaller input shape. Note that increasing the width of the memories means there are more operations done per second, and that's why the throughput increases.

6.5.1. Power consumption We measured the current draw of the FGPA on an idle state and while performing inference with a Fluke 117 multimeter using the scheme shown in Figure 6.8, connecting the multimeter in series between the 5 volts line of the USB ports on the PC and the FPGA board. Photos of example measurements are shown in Figure 6.9 for the idle state, and 6.10 while performing inference.
We measured an average current consumption of 116 [mA] on the idle state, and a peak current consumption of around 172 [mA] while performing inference. For 5 [V] voltage, the average power draw is 0.58 [W] on the idle state, and the peak power draw is 0.86 [W].

Figure 6.8. FPGA current measurement scheme

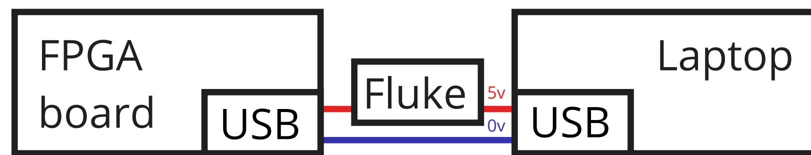


Figure 6.9. Current draw on idle state

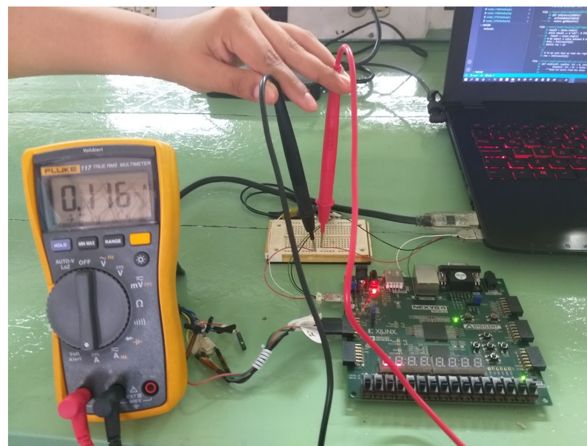
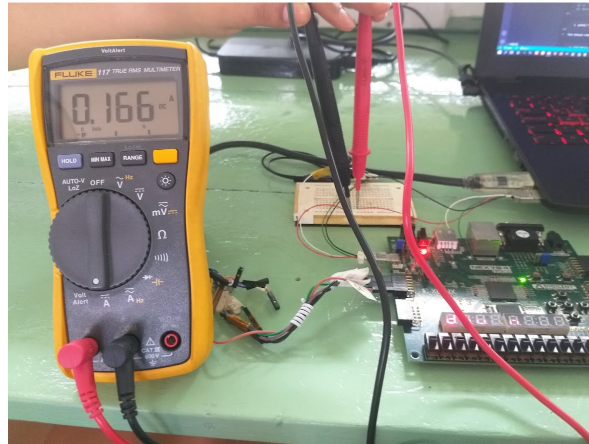


Figure 6.10. Current draw while performing inference



6.5.2. FPGA Monte Carlo experiments We perform Monte Carlo experiments of 100 samples for 30 %, 50 %, and 70 % variability on the SC, SC-AC30 %, SC-AC50 %, and SC-AC70 % models using the FPGA, for each of the samples we do the following steps:

1. For each of the quantized parameters in the model, apply the variability (multiplying the parameters with an error matrix), then quantize the noisy parameters and save them to a file.

2. Update the *params* memory of the original model bitstream using Vivado and the *updatemem* command with the noisy parameters.

3. Upload the bitstream containing the noisy parameters to the FPGA using Vivado and perform inference on the test set, then calculate the accuracy.

The steps are all done automatically in python, and each Monte Carlo experiment takes around 40 minutes for 100 samples. The results of the experiments are shown in Table 6.4. We see that generally, the accuracy resilience increases when training with a higher variability and the model trained with 70 % variability leads to the best accuracy for all simulation errors.

Figure 6.11. Results for SC - BaseNN and A-Connect 70 %

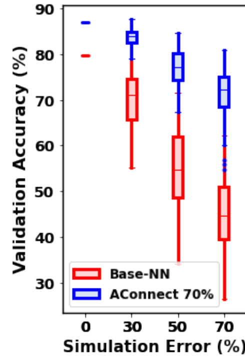


Table 6.4. FPGA Monte Carlo experiments results

Model	Sim. Error	Base-NN	A-Connect 30 %	A-Connect 50 %	A-Connect 70 %
SC	**0 %**	*79.83 %/0.00 %*	*86.46 %/0.00 %*	*86.97 %/0.00 %*	*87.11 %/0.00 %*
	30 %	*71.10 %/8.79 %*	*80.73 %/4.63 %*	*83.49 %/2.48 %*	*83.98 %/2.35 %*
	50 %	*54.66 %/13.40 %*	*67.20 %/7.06 %*	*75.42 %/5.40 %*	*77.28 %/5.75 %*
	70 %	*44.65 %/11.31 %*	*57.99 %/8.91 %*	*68.25 %/7.57 %*	*72.19 %/6.46 %*

6.5.3. Implementation details We used the Nexys4 DDR FPGA board [33] which features the Xilinx XC7A100T FPGA, 128 MiB of DDR memory, 100 MHz clock, 4 Pmod ports, and a range of several I/O devices. The Xilinx XC7A100T features 15850 Logic Slices, 4860 Kbits of Block RAM, 240 DSP Slices, and supports an internal clock of more than 450MHz.

The design is developed in the Xilinx Vivado IDE using the Verilog hardware description language, the memories were implemented with Xilinx Parametrized Macros (XPMs) although they could be implemented using Verilog modules to make the design platform independent. An exception is made for the L-128b model where the *params* memory was implemented using the Block Memory Generator LogiCORE™

[33] Digilent. *Nexys 4 DDR*.

IP, as using XPMs resulted in over-utilization of the block RAM resource.

The implemented design runs at 100 MHz, uses Pmod port JD to expose the SPI peripheral connectivity (SCK in JD1, MOSI in JD2, MISO in JD3, and CS in JD4), uses Led 0 to display if the FPGA is ready to start inference, uses the 7-segment display to show the predicted word, and uses the center button (BTNC) to reset the accelerator.

We used the C232HM cable [34] to provide an SPI interface for the computer to communicate with the FPGA, we run the SPI clock at 6 Mhz to ensure data transmission is reliable.

Resource utilization for implemented models (L and SC) reported by Vivado is shown in Tables 6.5 and 6.6 for memories of 64-bit width.

Resource	**Utilization**	**Available**	**Utilization %**
LUT	4273	63400	6.74
LUTRAM	1227	19000	6.46
FF	2883	126800	2.27
BRAM	96	135	71.11
IO	22	210	10.48

Table 6.5. Resource utilization for L model

[34] Future Technology Devices International Ltd. *C232HM USB 2.0 HI-SPEED TO MPSSE CABLE Datasheet*.

Resource	**Utilization**	**Available**	**Utilization %**
LUT	3298	63400	5.20
LUTRAM	587	19000	3.09
FF	2282	126800	1.80
BRAM	18.50	135	13.70
IO	22	210	10.48

Table 6.6. Resource utilization for SC model

7. FURTHER CONTRIBUTIONS

A-Connect is still in development since the base work (Ph.D. Thesis) is not finished yet. A-Connect needs more work to test the methodology with more complex models and other types of neural networks. Also, A-Connect needs more work on memory usage optimization.

- Include parameter quantization on RNNs during training using the A-Connect methodology.
- To extend A-Connect to other types of neural network architectures used in differents applications such as Natural Language Processing.

Finally, the works mentioned above will be included in a new version of the library.

8. CONCLUSION

This paper presents the extension of A-Connect to 3 types of recurrent neural networks used in speech recognition applications (FastGRNN, LSTM, and GRU). Favorable results are obtained after implementation, showing that models trained with the A-Connect methodology perform better in terms of accuracy than models trained without the methodology when the parameters are subject to noise. In particular, for the GRU model of size L subject to 70 % variability, we achieve 87.93 % accuracy on the speech commands dataset using the A-Connect version trained with 70 % variability, which is 5.58 % higher than the base version. Similarly, for the LSTM model of size S, the A-Connect version achieves 68.25 % accuracy, which is 22.78 % higher than the base version. The source files of the TensorFlow implementation can be found in the GitHub repository [6].

Finally, the methodology could be validated in an FPGA implementation by performing Monte Carlo experiments and obtaining, for a variability of 70 %, an accuracy of 72.19 % in the speech commands dataset using A-Connect trained with 70 % variability, which is 27.54 % higher than the base model. Documentation and source files can be found in the GitHub repository [27].

BIBLIOGRAPHY

François Chollet. *Deep learning API Keras*. [Online] Available: https://keras.io/. 2015 (vid. pág. 15).

Abid, Abubakar et al. “Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild”. En: *arXiv preprint arXiv:1906.02569* (2019) (vid. pág. 59).

Amaya, J. et al. *Library for A-Connect*. 2022 (vid. págs. 15, 70).

Amodei, Dario et al. *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*. 2015. DOI: 10.48550/ARXIV.1512.02595 (vid. pág. 14).

ARM Software. *Keyword spotting for Microcontrollers - Quantization guide* (vid. pág. 52).

Dennis, Don Kurian and Gaurkar, Yash and Gopinath, Sridhar and Goyal, Sachin and Gupta, Chirag and Jain, Moksh and Jaiswal, Shikhar and Kumar, Ashish and Kusupati, Aditya and Lovett, Chris and Patil, Shishir G and Saha, Oindrila and Simhadri, Harsha Vardhan. *EdgeML: Machine Learning for resource-constrained edge devices*. Ver. 0.4 (vid. págs. 35, 52).

Dey, Rahul y Fathi M. Salem. “Gate-variants of Gated Recurrent Unit (GRU) neural networks”. En: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, págs. 1597-1600. DOI: 10.1109/MWSCAS.2017.8053243 (vid. págs. 36, 38).

Digilent. *Nexys 4 DDR* (vid. pág. 66).

Emmanuel Blot,Emmanuel Bouaziz. *PyFtdi* (vid. pág. 59).

Future Technology Devices International Ltd. *C232HM USB 2.0 HI-SPEED TO MPSSE CABLE Datasheet* (vid. pág. 67).

Google. *Machine Learning Glossary* (vid. pág. 25).

Google brain team. *Machine learning platform TensorFlow*. 2015 (vid. pág. 15).

He, Yanzhang et al. *Streaming End-to-end Speech Recognition For Mobile Devices*. 2018. DOI: 10.48550/ARXIV.1811.06621 (vid. pág. 14).

Huang, Xuedong, Alex Acero, Hsiao-Wuen Hon y Raj Reddy. “Spoken Language Processing: A Guide to Theory, Algorithm and System Development”. En: 2001 (vid. pág. 27).

IBM Cloud Education. “Speech Recognition”. En: (2020) (vid. pág. 21).

Jackson, Zohar. *Spoken Digit*. 2016 (vid. pág. 24).

Jiménez, Edward Albán Silva, Ricardo Matheo Vergel Sanabria, Luis Rueda y Elkim Roa. “Library Development For A-Connect”. previous undergraduate project. 2021 (vid. págs. 15, 18, 19, 33).

Jose Amaya, Karen Leon. *FPGA-KWS GitHub repository* (vid. págs. 51, 70).

Kusupati, Aditya et al. *FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network*. 2019. DOI: 10.48550/ARXIV.1901.02358 (vid. págs. 14, 32, 34, 44-46, 51).

Lyons, James. *Mel Frequency Cepstral Coefficient (MFCC) tutorial* (vid. pág. 26).

Ochoa, Andrés Felipe Centeno, Luis Alejandro Hernández, Luis Rueda y Elkim Roa. “Parameters quantization with A-Connect”. previous undergraduate project. 2022 (vid. págs. 19, 33).

Panayotov, Vassil, Guoguo Chen, Daniel Povey y Sanjeev Khudanpur. “Librispeech: an ASR corpus based on public domain audio books”. En: *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE. 2015, págs. 5206-5210 (vid. pág. 23).

Pyeatt, Larry D. “Chapter 8 - Non-Integral Mathematics”. En: *Modern Assembly Language Programming with the ARM Processor*. Ed. por Larry D. Pyeatt. Newnes, 2016, págs. 219-264. DOI: https://doi.org/10.1016/B978-0-12-803698-3.00008-5 (vid. pág. 52).

Rousseau, Anthony, Paul Deléglise y Yannick Estève. “TED-LIUM: an Automatic Speech Recognition dedicated corpus”. En: *Conference on Language Resources and Evaluation (LREC)*. 2012, págs. 125-129 (vid. pág. 23).

Rueda, Luis y Elkim Roa. “A-Connect: Enabling Imprecise Analog Computation”. (unpublished work) (vid. pág. 15).

speech recognition. DOI: 10.1093/oi/authority.20110803100522540 (vid. pág. 20).

TensorFlow. *tf.keras.layers.GRUCell* (vid. pág. 38).

— *tf.keras.layers.LSTMCell* (vid. pág. 36).

— *tf.quantization.fake_quant_with_min_max_args* (vid. pág. 53).

Understanding LSTM Networks (vid. pág. 36).

Warden, P. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. En: *ArXiv e-prints* (abr. de 2018). arXiv: 1804.03209 [cs.CL] (vid. págs. 24, 26, 32, 51).

Warden, Pete. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018. DOI: 10.48550/ARXIV.1804.03209 (vid. págs. 20, 23).

Young, Steve et al. “The HTK book”. En: *Cambridge university engineering department* 3.175 (2002), pág. 12 (vid. págs. 26-28).

Zhang, Yundong, Naveen Suda, Liangzhen Lai y Vikas Chandra. *Hello Edge: Keyword Spotting on Microcontrollers*. 2017. DOI: 10.48550/ARXIV.1711.07128 (vid. págs. 14, 21, 25, 29-32, 44, 45, 51).