NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

INTERDEPARTMENTAL POSTGRADUATE PROGRAM IN MICROELECTRONICS

MASTER THESIS

American Sign Language Recognition via Sensor glove data analysis with deep learning - An ARM Implementation

Theodoros K. Barmpakos

ATHENS

OCTOBER 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΣΤΗ ΜΙΚΡΟΗΛΕΚΤΡΟΝΙΚΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνώριση της Αμερικανικής Νοηματικής Γλώσσας μέσω ανάλυσης δεδομένων από γάντι αισθητήρων με δίκτυο βαθιάς μάθησης - Υλοποίηση σε επεξεργαστή ARM

Θεόδωρος Κ. Μπαρμπάκος

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2022

MASTER THESIS

American Sign Language Recognition via Sensor glove data analysis with deep learning - An ARM Implementation

Theodoros K. Barmpakos

A.M: MM289

SUPERVISOR: Elias S. Manolakos, Professor of National and Kapodistrian University of Athens

OCTOBER 2022

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αναγνώριση της Αμερικανικής Νοηματικής Γλώσσας μέσω ανάλυσης δεδομένων από γάντι αισθητήρων με δίκτυο βαθιάς μάθησης - Υλοποίηση σε επεξεργαστή ARM

Θεόδωρος Κ. Μπαρμπάκος

Α.Μ: ΜΜ289

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Ηλίας Σ. Μανωλάκος, Καθηγητής στο Εθνικό και Καποδιστριακό Πανεπηστήμιο Αθηνών

ΟΚΤΩΒΡΙΟΣ 2022

ABSTRACT

Deep Learning (DL), and especially Convolutional Neural Networks (CNNs), have been widely used to solve a large variety of problems in computer vision, including Sign Language Recognition (SLR). There have been many efforts towards designing systems using cameras that can translate signer gestures to text or even speech. However, these systems are very sensitive to factors such as light intensity, background color and motion occlusion etc.

In this thesis, we present the design of a simple end-to-end embedded system that translates continuous American Sign Language (ASL) into text based on inputs received from an instrumented low-cost sensor glove that we have created using flex sensors and an IMU device. To prove the concept, we first generated a limited dataset of 20 random ASL sentences using a 20-word vocabulary where we manually pre-label the time series data into 21 classes and simultaneously separate gesture and non-gesture (transition class) movement periods, by using an external button. Subsequently, a sliding window technique was used to extract overlapping labeled samples (time windows) for continuous SLR. After standardization, the data samples are fed to a simple 3-layer 1D CNN (conv1d - conv1d - fully connected) for classification.

Convolutional layers are useful for automated feature extraction and fully connected for classification. Our CNN achieves 93.40% accuracy on the test set (unseen data). For all practical purposes, the accuracy is actually 100% as vocabulary gestures are not confused for each other, and errors occur only in the transition from a gesture to a non-gesture transition movement window and vice versa. The CNN was trained and its hyperparameters tuned using the ATOM python-based framework. Its accuracy was compared and found to be slightly higher than that of other popular machine learning methods, such as the Random Forests, Support Vector Machines, and Extreme Gradient Boosted Trees (XGBoost).

Finally, we have developed an all-software implementation of the designed CNN (inference part) for the ARM Cortex A9 processor on the Zybo development board. Using the Xilinx SDK and Eigen library, we managed to design a real-time embedded system that can achieve an operating frequency much higher than the sampling frequency. Optimization, training, and testing of the CNN were performed on a PC using ATOM and the Keras library with a Tensorflow-GPU backend.

SUBJECT AREA: American Sign Language Recognition, Gesture Recognition

KEYWORDS: SLR, sensor glove, ARM, CNN, Machine Learning

ΠΕΡΙΛΗΨΗ

Η Βαθιά Μάθηση (DL), και ειδικότερα τα Νευρωνικά Δίκτυα Συνέλιξης (CNN), έχουν χρησιμοποιηθεί ευρέως για την επίλυση πλήθους προβλημάτων στη Μηχανική Όραση, περιλαμβανομένης και αυτού της Αναγνώρισης της Νοηματικής Γλώσσας (ΑΝΓ). Μέχρι σήμερα, έχουν γίνει πολλές προσπάθειες για τη σχεδίαση συστημάτων με χρήση κάμερας που μπορούν να μεταφράσουν τις χειρονομίες ενός ατόμου που μιλάει τη νοηματική γλώσσα σε κείμενο ή ακόμα και ομιλία. Ωστόσο, αυτά τα συστήματα είναι πολύ ευαίσθητα σε παράγοντες όπως η ένταση του φωτός, το χρώμα φόντου και η απόφραξη κίνησης κ.λπ.

Η παρούσα διπλωματική εργασία εστιάζει στην υλοποίηση ενός πλήρους συστήματος, το οποίο μεταφράζει σε συνεχή ροή λέξεις από την Αμερικανική Νοηματική Γλώσσα, σε κείμενο, με τη χρήση ενός γαντιού δεδομένων που κατασκευάστηκε με χαμηλό κόστος για τον ως άνω σκοπό και βασίζεται στη χρήση αισθητήρων κάμψης και μιας αδρανειακής μετρητικής συσκευής. Για την επίτευξη του στόχου, δημιουργήσαμε αρχικά ένα σύνολο δεδομένων από την καταγραφή χειρονομιών 20 τυχαίων παραγόμενων προτάσεων χρησιμοποιώντας ένα λεξιλόγιο 20 λέξεων. Κατά την καταγραφή και με τη χρήση ενός εξωτερικού κουμπιού αποδόθηκαν στα δεδομένα προ ετικέτες κατηγοριοποιώντας τα σε 21 κλάσεις και διαχωρίζοντας παράλληλα τις χρονικές περιόδους των χειρονομιών και μη χειρονομιών (κλάση μετάβασης). Στη συνέχεια, για την αντιμετώπιση της συνεχούς αναγνώρισης, εφαρμόζουμε τη μέθοδο ολισθαίνοντος παραθύρου και εξάγουμε τα αντίστοιχα αλληλεπικαλυπτόμενα δείγματα (χρονικά παράθυρα), τα οποία αφού κανονικοποιηθούν τροφοδοτούν ένα απλό Νευρωνικό Δίκτυο Συνέλιξης με τρία επίπεδα (conv1d - conv1d - fully connected).

Τα συνελικτικά επίπεδα συμβάλουν στην αυτόματη εξαγωγή ”χρήσιμων” χαρακτηριστικών ενώ το πλήρως συνδεδεμένο επίπεδο είναι υπεύθυνο για την κατηγοριοποίηση των δειγμάτων. Το προτεινόμενο νευρωνικό δίκτυο δοκιμάστηκε σε σύνολο δεδομένων που δεν είχε δεί ξανά, επιτυγχάνοντας ακρίβεια αναγνώρισης στις προκαθορισμένες χειρονομίες ίση με 93,40%. Στην πράξη, η ακρίβεια αναγνώρισης είναι 100%, καθώς δεν γίνονται λανθασμένες προβλέψεις μεταξύ χειρονομιών, αλλά μεταξύ μιας χειρονομίας και της μεταβατικής κλάσης τη στιγμή που το χρονικό παράθυρο εισέρχεται στα όρια της χειρονομίας ή εξέρχεται από αυτήν και για μόνο μερικά χρονικά βήματα. Η εκπαίδευση του νευρωνικού δικτύου και η ρύθμιση των υπερπαραμέτρων του, πραγματοποιήθηκε με τη χρήση του εργαλείου ATOM, που βασίζεται στη γλώσσα python. Επιπλέον, διεξήχθησαν δοκιμές και σε άλλα μοντέλα μηχανικής μάθησης όπως τα Random Forests, Support Vector Machines, και Extreme Gradient Boosted Trees (XGBoost), με αποτελέσματα που δείχνουν ότι το προτεινόμενο CNN πετυχαίνει ελαφρώς καλύτερο ποσοστό ακρίβειας αναγνώρισης.

Τέλος, αναπτύξαμε υλοποίηση του απλού Νευρωνικού Δικτύου Συνέλιξης τριών επιπέδων (για πρόβλεψη) στον επεξεργαστή ARM Cortex A9 που διαθέτει η πλακέτα ανάπτυξης Zybo. Χρησιμοποιώντας το περιβάλλον Xilinx SDK και την βιβλιοθήκη Eigen, καταφέραμε να σχεδιάσουμε ένα πραγματικού χρόνου ενσωματωμένο σύστημα, το οποίο λειτουργεί σε πολύ μεγαλύτερη συχνότητας από αυτή της δειγματοληψίας. Η εκπαίδευση και η δοκιμή του Νευρωνικού Δικτύου Συνέλιξης πραγματοποιήθηκε σε προσωπικό υπολογιστή χρησιμοποιώντας το εργαλείο ATOM και τη βιβλιοθήκη Keras βασισμένη στο Tensorflow-GPU.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αναγνώρηση Νοηματικής Γλώσσας, Αναγνώρηση Χειρονομίας

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ΑΝΓ, Γάντι Δεδομένων, ARM, CNN, Μηχανική Μάθηση

To my friends Tsipiras Dionysios and Kostantinos Kourkoulos.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor, Prof. Elias Manolakos for his trust in my abilities, his valuable guidance during the writing of this thesis and his patience in the revision process.

Next I would like to thank Prof. Antonis Paschalis and Dr. Alexandros Pino of the examination committee for their time spending on the review process and their helpful comments on the thesis.

Furthermore, I would like to thank Mr. Elias Kouskoumvekakis PhD student of Prof. Manolakos for his help and technical advice on embedded systems, as well as for teaching and training me on Xilinx FPGAs.

Last I would like to express my deepest gratitude to my family and friends for the support during this thesis and especially my cousin Periklis Barmpakos for his advice in Machine Learning.

CONTENTS

1 INTRODUCTION 23
1.1 Sign Language . . . 23
1.2 The ideal sign language recognition system - Specifications . . . 24
1.3 Thesis goals . . . 25
1.4 Thesis organization . . . 26

2 BACKGROUND AND RELATED WORK 27
2.1 Machine Learning . . . 27
2.2 Deep Learning . . . 27
2.3 Layers . . . 29
2.3.1 Convolution 1D Layer . . . 29
2.3.2 Fully Connected Layer . . . 30
2.4 Activation Functions . . . 31
2.5 Classification with Neural Networks . . . 32
2.5.1 One-hot encoded vector . . . 32
2.5.2 Multi-class and Multi-label Models . . . 33
2.6 Related Work . . . 34
2.7 Thesis contributions related to state of the art . . . 39

3 SYSTEM DESIGN - DATASET CREATION 41
3.1 The Sensor Glove . . . 41
3.2 Data Sampling . . . 42
3.3 Sliding Window . . . 44
3.4 Pre Processing . . . 45
3.5 Conclusions . . . 46

4 MODEL DEVELOPMENT 47
4.1 Baseline ML Methods . . . 47
4.2 Hyperparameter Tuning . . . 48
4.3 ML Results . . . 51
4.4 Using Convolution Neural Networks . . . 55
4.4.1 Model architecture . . . 55
4.4.2 CNN Hyperparameter tuning . . . 57
4.4.3 NN model on ATOM . . . 57

4.5 Results 58
4.6 Conclusions 59

5 AN ARM-BASED IMPLEMENTATION 61
5.1 Experimental System Overview 61
5.2 Software and Tools Used 63
5.3 Define the CNN using Eigen 64
5.4 Model transfer 68
5.5 Testset Loaded 69
5.6 Class prediction using Eigen 70
5.7 Conclusion 72

6 CONCLUSIONS AND FUTURE WORK 73

ABBREVIATIONS - ACRONYMS 75

APPENDICES 76

A Code for Baseline ML 77

B Code for Our Proposed Method 87

C ARM Main program 99

REFERENCES 111

LIST OF FIGURES

1 Gesture examples. (a) Twenty-four alphabet static gestures of the American sign language. (b) In the dynamic word 'Hello', hand posture is moving in the arrow direction. 23

2 A Feed-forward neural network structure. 28

3 A complete neural network structure with optimizer and loss function. 29

4 Applying 1D convolution to time-series data. The convolution kernel has four filters and size of three in this example. The output matrix dimensions are *timesteps - kernel size + 1* and *number of filters*. 30

5 Non-linear activation functions. (a) Relu. (b) Sigmoid 31

6 Example of Multi-class feed-forward Neural Network classification. 33

7 Example of Multi-label feed-forward Neural Network. 34

8 Our sensor glove with all its components packed away in a usb device. 41

9 (a) A thick laminate flex sensor. (b) An IC analog multiplexer device. 42

10 COM port display. Raw data samples flow over time. 43

11 The signal representation of the first sentence. Horizontal axis is time. Different colors distinguish the signal of each sensor. 44

12 Sliding window (green) process and overlapping class windows. The transition class is assigned to both #1 and #2 samples and 'd' class to sample #3. 45

13 ATOMClassifier log. 50

14 ATOMClassifier results. 51

15 Bootstrap method for the case n_bootstrap=3. 52

16 XGB confusion matrix 53

17 RF confusion matrix 53

18 LR confusion matrix 54

19 LSVM confusion matrix 54

20 The CNN model architecture - Bayesian optimization has been applied. Diagram exported from *Keras*. 55

21 Sequential model on keras. 56

22 Example of an 7-timestep feature extraction for 94 filters. 56

23 Hyper-parameter search space. 57
24 ATOMClassifier results on NN. 59
25 CNN confusion matrix. 59
26 System overview and bi-directional communication. The Zybo board holds the whole application while the input device with the PC work as a peripheral for input purposes. 61
27 The Zynq Processing System (adopted from [10]). 64
28 All necessary matrices we declare in order to run a feed forward pass. Hyper-parameters: x1=94 filters, y1=7 kernel_size, x2=86 filters, y2=5 kernel_size resulting from BO in section 4.5 according to our model architecture in section 4.4.1. Operator T gives the transpose of a matrix. 65
29 Keras model presented in chapter 4. Deleted layers are not used on inference. . 66

LIST OF TABLES

1 One-hot encoded representation for each case. If a dog exists in the image, then first coordinate of the vector has one, otherwise zero and so on. 32

2 The ASL vocabulary used for the project. 43

3 CNN Memory. The system performs on a Single-precision Floating-point format (FP32). 67

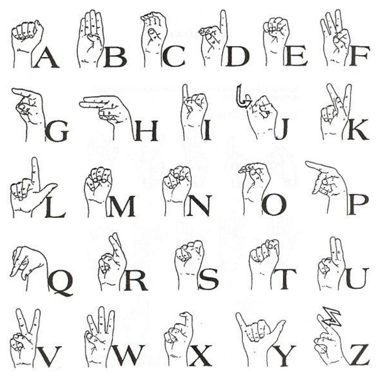
1. INTRODUCTION

1.1 Sign Language

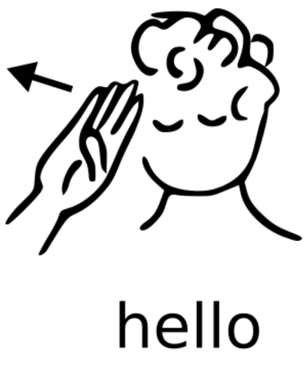
Sign Language (SL) is widely used for communication by deaf and mute people. It is not a simple hand-moving language but an entire body and face expression. SL is not standard in all communities globally, so there are more than two hundred different SLs [1] with their grammar and syntax. This thesis is concerned with the American Sign Language (ASL) because of its large number of speakers, which put it in the top three most widely spoken sign languages.

The essential part of ASL is the movement of the hands, which is called a *gesture*. Gestures can be static (a specific form of fingers and wrist) or dynamic [2]. Let us take a look at some static gestures in Figure 1a (except letters j and z), which present the American alphabet, and a dynamic one in Figure 1b, which is the word “Hello”. Thus, adding more and more gestures, static or dynamic, we construct the vocabulary of the ASL. There are more than 4,500 signs in the American vocabulary [3]. Applying some rules to it (e.g., grammar), the whole language is constructed. That is how signers can communicate with each other.

What about the communication between a signer and a speaker? As the former can not speak, the only way to communicate with each other is to use the same sign language. But, generally, speakers do not understand sign language and learning it is not an easy process. Thus, a communication gap between them is created. To minimize this gap and make communication possible is where gesture recognition plays a role.



(a)



(b)

Figure 1: Gesture examples. (a) Twenty-four alphabet static gestures of the American sign language. (b) In the dynamic word ’Hello’, hand posture is moving in the arrow direction.

Gesture recognition is a computing process that attempts to recognize, not only sign language but generally human gestures and interpret them using algorithms. It is an essential domain because it can facilitate communication and create an interface between humans and machines more naturally. For example, we can control devices such as drones or medical machines only by hand movements. Another important intent of gesture recognition focuses on virtual reality (VR) environments. By transferring human body movements from reality into virtual worlds, we can create sophisticated and interactive applications for training purposes or even for treatments of diseases such as Alzheimer's [4].

One powerful tool to solve this problem is Machine Learning (ML), which constantly provides solutions enhancing human capabilities in many real-world applications. Pattern recognition is one of the most distinct aspects in which we can draw insights through machine learning methods. Specifically, since our application concerns real-time response and user mobility, it is essential to investigate the potential of successful embedded machine learning solutions, which motivated the design of a portable embedded system for ASL.

1.2 The ideal sign language recognition system - Specifications

What makes a system for sign language recognition an ideal one? There are many requirements to meet to achieve perfection, but we will focus on the most important ones. Let us think of a signer standing in front of a speaker and trying to communicate. First of all, s/he needs a comfortable, portable, and easy-to-use device at a cost as low as possible to be affordable.

As speech is a continuous sequence of words, sign language is a continuous sequence of gestures. The signer must freely act while talking without having to worry about when a gesture starts or ends. The system has to recognize the boundaries of each gesture automatically, so we say that it works in continuous/online mode. In contrast to continuous mode, a version of isolated finite signals recognition exists but is far from ideal.

In addition, two significant specifications are vocabulary size and recognition accuracy. The ideal system must recognize the whole ASL dictionary without any wrong prediction. That means it has to classify more than 4,500 gestures and achieve 100% accuracy.

Finally, a real-time attribute is necessary to make communication as fast and interactive as it can be. In general, [5] [6] a real-time system responds within specified time constraints, which are often in the order of milliseconds, and sometimes microseconds. That means, the total processing time per sample including overhead (i.e., other steps required to complete a task such as read/write memory, etc.) should be less than the sampling period. Hence, this constrains the sampling rate of any signal processing system we may employ.

Sometimes, although the above conditions are satisfied (total processing time is less than sampling period), a throughput delay (latency) also exists e.g. latency in a pipeline system, adding some extra cost to the total processing time. In theory, even if this delay is large enough, the system is considered as a real-time one. But in practice, when talking about real-time bi-directional telecommunications, processing requires both real-time operation and a sufficient limit to that throughput delay. So systems with responses of less than 300 ms are met the above requirements and considered “acceptable” to avoid undesired “talk-over” in a conversation. To make it clear, we can think of a mobile phone conversation. If the voice from one speaker is delayed in reaching the other, then the latter starts talking and suddenly hear the delayed voice. This issue is repeated simultaneously in the other direction as well and makes the communication ineffective. Even though we face a one-way directional communication in our case/project (SLR), we can set the same response time (300 ms) as the upper bound of our system, hence it must translate each incoming sample/gesture to the corresponding lexicon word under 300 ms.

In summary, an efficient gesture recognition system should meet the following specifications:

- comfort, portability, and low cost
- continuous/online recognition
- large vocabulary
- high accuracy
- real-time response

1.3 Thesis goals

There have been many approaches since the 1990s that try to make communication between signers and speakers possible without a speaker needing to learn the corresponding language. Many of these use one or more cameras to aid on SLR. However, this is not easy to adopt as a portable solution because one or more cameras are needed to be set up in fixed places. This thesis aims to design and implement a portable end-to-end embedded system (software and hardware) that can translate American Sign Language into text using neural networks and a much simpler sensing system.

To manage the complexity of our project, we broke it down into smaller parts and set the following sub-goals that try to approximate the ideal system:

Goal 1: ***Construct a low-cost sensor glove.*** On a sensor-based SLR system, we need a device that collects data from hand movements, e.g. fingers’ bend, wrist direction, elbow position, etc. Commercial gloves are too expensive, so they are not suitable for everyone. Similar to other researches [7] [8] [9], flex sensors and Inertial Measurement Unit (IMU) devices are attached to ordinary gloves and make an affordable device with a cost of about 100€. At this point, we did not aim for a production-grade solution but a device we could use for experimental purposes.

Goal 2: ***Design an efficient neural network architecture model.*** The neural network is the main processing unit, and it is responsible for classifying incoming gestures.

Currently, the vocabulary size is small and is set to twenty gestures, static and dynamic. The network must be as simple as possible, suitable for an embedded solution, but achieve better accuracy than other ML methods.

Goal 3: ***Implement our SLR solution in an embedded system.*** The solution must fit into an embedded device and run in real-time in a continuous flow to make signers feel free while testing it. In addition, we selected to employ an FPGA since it was not clear at the beginning if we may need hardware acceleration capabilities for some parts of the design to reach real-time performance. The Zybo z7-10 development board by Xilinx [10], which includes a dual ARM processor and an FPGA fabric, was selected for this purpose.

Goal 4: ***One-way communication.*** As mentioned, communication between two people is a bi-directional process. This thesis focuses on one-way, where the signer talks and gestures get translated into text. So, a speaker interprets and understands the signer and not the other was around.

1.4 Thesis organization

The rest of the thesis is organized as follows:

- **Chapter 2:** We describe, in detail, the mathematical background of neural networks and especially of Convolutional Neural Networks (CNN), which is necessary for low-level implementation on embedded systems. We also review the state-of-the-art approaches for different SLR problems from 1991 to 2019 [11] and compare to our proposed method.
- **Chapter 3:** We present the design and implementation of a simple and inexpensive sensor glove, discuss our choices and finally capture the necessary dataset for training and testing various ML/DL models.
- **Chapter 4:** We apply different machine learning methods and present our continuous SLR approach based on CNNs. We build, optimize and test our model, and compare all methods.
- **Chapter 5:** We develop a bare-metal application running on the Zybo z7 development board, and especially its ARM processor. The FPGA part is not currently used, so we are presenting an all-software solution. Also, we describe how to load/transfer a trained model to our device, how much memory of DDR it needs, and how to use other peripheral parts (UART) of the Zybo board.
- **Chapter 6:** We summarize the main conclusions of this thesis and provide directions for future work and possible extensions.

2. BACKGROUND AND RELATED WORK

In this chapter we present a brief introduction to machine learning (ML) and neural networks (NN). We review how they work, what they consist of, and how they can solve a classification problem by applying multi-class and multi-label methods. Finally, we describe recent approaches for sign language recognition and position our work relatively to them.

2.1 Machine Learning

Machine Learning is a branch of Artificial Intelligence (AI) that gives systems the ability to learn from examples (observations) automatically, without being explicitly programmed. Instead of using handcraft rules to catch different statements (programming), it uses algorithms that build mathematical models based on collected data, called ”training data”. So after training, the system can make predictions and decisions without human intervention. These algorithms are divided into four broad categories [12]:

- ***Supervised Learning:*** Mainly consists of regression and classification models. A set of labelled examples (training set) is used for building the model. Then the model maps unseen data to known targets (e.g., optical character recognition, speech recognition, image classification, and language translation).
- ***Unsupervised Learning:*** Like supervised learning, it uses a set of examples for training a model, but *without* needing any label information. That leads to finding interesting and common representations on unseen data (e.g., clustering and dimensionality reduction).
- ***Self-Supervised Learning:*** It is supervised learning with self-generated labels used to train a model (e.g., auto-encoders [13] [14]).
- ***Reinforcement Learning:*** An agent receives information about its environment and learns to choose actions that will maximize some reward metric [15] (e.g., learn to play Atari games at maximum level).

This thesis focuses on supervised learning because we treat ASL recognition as a classification problem and use a labeled data set to train our ML models. More about the data set and the classes will be discussed in Chapter 3.

2.2 Deep Learning

Deep Learning (DL) is a member of a broader family of Machine Learning methods, including also *Probabilistic modeling*, *Kernel methods*, *Decision trees*, *Random Forest*, and *Gradient boosting machines* [12]. Deep Learning (DL) is implemented using *Neural Networks* (NN), and as a supervised ML method, it uses examples to build models. How does this work? Let us have a look at the structure of a neural network in Figure 2 below.

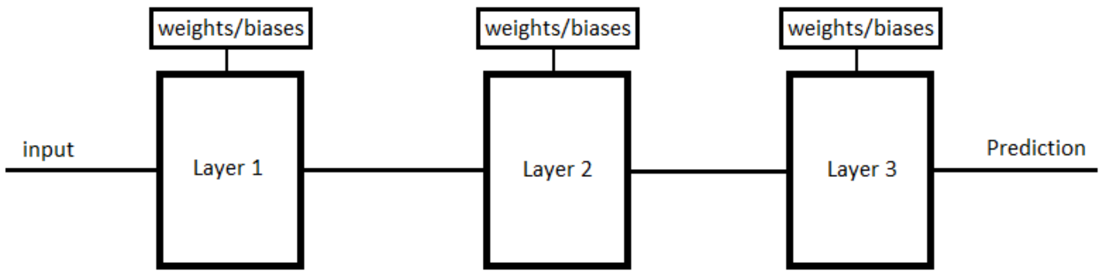


Figure 2: A Feed-forward neural network structure.

In general, a neural network can be thought as a sequence of *Layers* connected in series. The output of each layer is connected to the input of the next layer. The number of layers is called the *depth* of the network. The more the layers, the deeper the network. Each layer has a predefined number of parameters, *weights* and *biases*, forming the model's trainable parameters. Depending on its type (e.g., dense, convolution, max-pooling e.t.c.), a layer performs specific calculations as data flows from one layer to the next. Therefore, a neural network maps an input sample to a target, and so it is also called a *feed-forward neural network*. For example, consider a simple classification problem: recognizing handwritten digits (0 to 9). Also, assume that the network has already been trained, so weights have taken their expected/correct values. Then given a 28x28 pixel image with a handwritten digit as input of the network, it predicts what digit (class) is presented. The next question is how a neural network can be trained?

When initializing a network, weights and biases are being set randomly close to zero value. So prediction compared with the real target will be far away from what we expected. For this, first we need somehow to measure the size of this error and second to minimize it. The measurement is performed by some function which is called the *loss function*, and its output the *loss score*. Hence, the goal is to minimize the loss score. The *optimizer* is responsible for minimizing that loss score, making proper weight adjustments via the backpropagation algorithm. So, while training a model, new predictions are getting closer and closer to real targets at each step of the algorithm. Below we can see the corresponding diagram in Figure 3. As we mentioned before, we need a set of examples (training data). For our purpose (supervised learning), each data sample is labeled by its class.

In deep learning and generally in Machine Learning, models have two types of parameters [16]. The ones that are set before the training process (learning algorithm) is started, which are called *hyper-parameters*, and the others that are calculated automatically from the learning algorithm and constitute the trainable part of the model (weights and biases), which are simply called *parameters* of the model. For example, in the handwritten digit recognition problem mentioned earlier, both the input and output sizes of the network are static (input: 28x28 pixels, output: 10 classes) and belong to model's hyper-parameters. Hyper-parameters are also the number of the weights and biases of each layer but not their values. In general, hyper-parameters define the structure of a model and are shown for each layer in the next section.

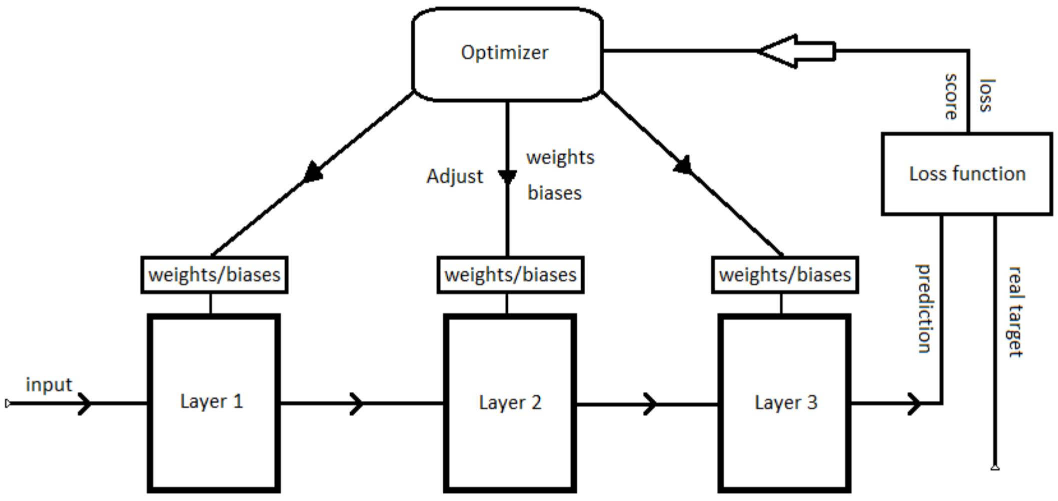


Figure 3: A complete neural network structure with optimizer and loss function.

2.3 Layers

In this part, we present the mathematical computations that take place behind each type of layer. That is necessary to design the architecture of our model on ARM later in Chapter 5. The two types of layers we will use are Convolution 1D and Fully Connected (dense) layer.

2.3.1 Convolution 1D Layer

Convolution layers apply a convolution operation to the input sample, passing the result to the next layer [17]. They can offer a fast alternative to other methods (e.g., RNNs) for *times-series* broadcasting because they can extract local 1D patches (subsequences) from sequences [12]. The two basic hyper-parameters of the conv1d filter are the *kernel size* or *kernel window* and the *number of filters*. Let us take a look at a simple version of our project. Raw data is collected from the sensor glove. Thus, a 2D matrix is created. One dimension is time-steps, and the other is the input features (the number of sensors - five flex sensors, a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer). As shown in Figure 4, the convolution kernel slides through the input frame one step at a time, convolve the corresponding matrices and store the output value. This procedure is repeated for each filter. It is important to note that the kernel's values are the trainable part of the layer. In our model implementation in section 4.4, we set the hyper-parameters of the first conv1d layer to be: *kernel size* = *7*, *number of filters* = *94* and of the second one: *kernel size* = *5*, *number of filters* = *86*. These values are selected from a specified search space through Bayesian Optimization method achieving best possible accuracy.

The convolution operation between two matrices, A and B with the same size MxN is given by the formula below:

$$conv(A, B) = \sum_{j=1}^{N} \sum_{i=1}^{M} A(i, j) * B(i, j) \tag{2.1}$$

where $(*)$ is element-wise matrix multiplication.

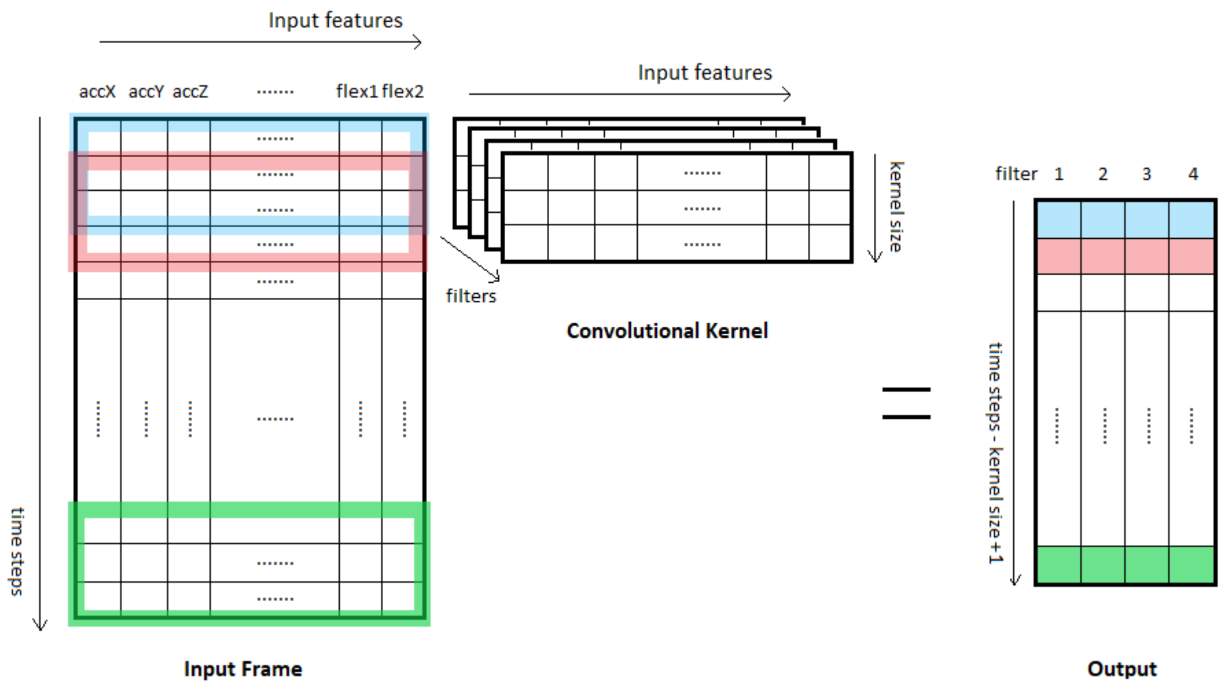


Figure 4: Applying 1D convolution to time-series data. The convolution kernel has four filters and size of three in this example. The output matrix dimensions are *timesteps - kernel size + 1* and *number of filters*.

2.3.2 Fully Connected Layer

The fully connected layer is the final layer of a model used to classify input data into the predefined classes. Unlike convolution layers that learn local patterns, fully connected layers recognize global patterns from input data. The corresponding operation is a linear transformation of its input *X* as formulated below [18]:

$$L = XW + B \tag{2.2}$$

where $X \in \mathbb{R}^{1\times l}$ is the flattened input (e.g. in handwritten digit example, a 28x28 image is reshaped into an 1x784 matrix, where 784 is the number of input neurons l), $W \in \mathbb{R}^{l\times classes}$ and $B \in \mathbb{R}^{1\times classes}$ are weights and biases, respectively (parameters of the layer that have to be trained). The input neurons l and $classes$ are layer's hyper-parameters.

2.4 Activation Functions

Activation functions are non-linear functions applied between layers to make a model more effective [18]. That is because the model's parameters are adjusted more independently (non-linear way) from layer to layer while training the network. Possible activation functions are *Rectified linear unit (Relu)*, *Leaky Relu*, *Sigmoid*, and *Softmax* [19]. *Relu* and *Softmax* are going to be used in this project. First, besides its simplicity, *Relu* is the most commonly used activation function defined as:

$$\rho(x) = max(x, 0) \tag{2.3}$$

Its graphical representation is shown in Figure 5a. *Relu* can be easily implemented either in hardware or in software. Second, *Softmax* is a convenient function for turning a finite set of numbers into a probability distribution. Given the set $X = \{x_1, x_2, ..., x_n\}, \quad x_i \in \mathbb{R}$, the probability distribution is the set $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$:

$$\sigma_i = \frac{e^{x_i}}{\sum\limits_{j=1}^{n} e^{x_j}} \tag{2.4}$$

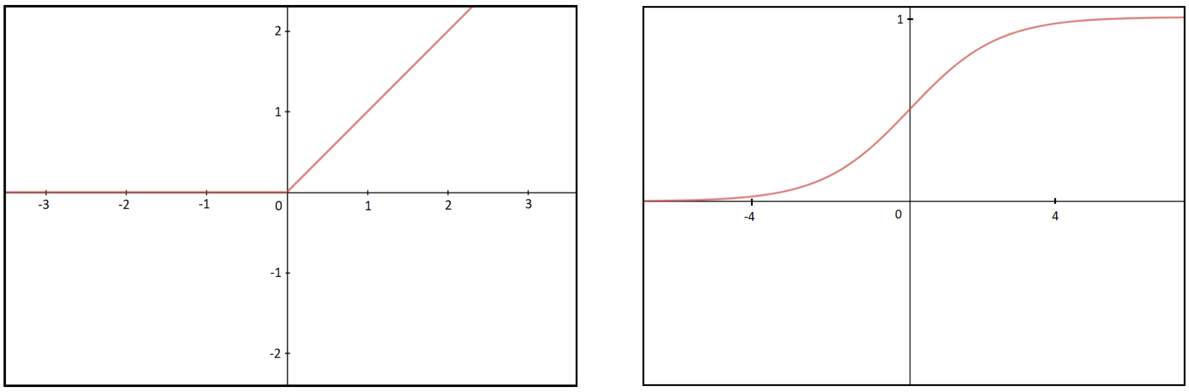


Figure 5: Non-linear activation functions. (a) Relu. (b) Sigmoid

In contrast with Relu, *Softmax* is applied on the final layer of the network to produce the probability of each class label. *Softmax* function contains massive exponential and division operations, making its implementation on hardware a complex one [20]. Both *Softmax* and *Relu* functions are implemented on the ARM processor in Chapter 5.

2.5 Classification with Neural Networks

Generally, there are two standard ways to address a classification problem; *multi-class* classification and *multi-label* classification.

In multi-label classification, a data sample may belong to multiple classes, whereas in multi-class, one data sample belongs exclusively to one class. In this section, according to [21], we give a general idea of how to implement a neural network for classification problems and apply it to build our model for SLR later on in Chapter 4.

2.5.1 One-hot encoded vector

Classification takes place by applying labels to data samples. These labels can be anything, like numbers, words, symbols, images, and others. For example, a label can be a bird, cat, dog, e.t.c. As Neural Networks perform arithmetic operations, it is necessary to transform labels into a suitable form. For our purpose, this form is the one-hot encoded vector and can be applied on both multi-class and multi-label classification.

Given a set of n classes $C = \{c_1, c_2, ...c_n\}$ we take its power set $D(C)$ (or 2^C) with 2^n elements and for an input sample X we define one-hot encoded representation as follows:

$$X \mapsto (v_1, v_2, ..., v_{2^n}), \text{ with} \tag{2.5}$$

$$v_i = \begin{cases} 1 & \text{, if } X \in c_i \\ 0 & \text{, otherwise} \end{cases}$$

Let us look at a simple image classification example with three classes: *dog*, *cat*, *bird*. Then one-hot encoded vector labels are:

Table 1: One-hot encoded representation for each case. If a dog exists in the image, then first coordinate of the vector has one, otherwise zero and so on.

No	image sample includes	label attached
0	null	(0, 0, 0)
1	dog	(1, 0, 0)
2	cat	(0, 1, 0)
3	bird	(0, 0, 1)
4	dog and cat	(1, 1, 0)
5	dog and bird	(1, 0, 1)
6	cat and bird	(0, 1, 1)
7	all	(1, 1, 1)

It is important to emphasize that, in a multi-class model, labels can have a unique coordinate with ’1’, and all the others are filled with zeros ’0’. That tells us that a data sample can belong to only one class. Instead, in multi-label, we can have more than one class being presented simultaneously.

2.5.2 Multi-class and Multi-label Models

The feed-forward Neural Network in Figure 6 has a Dense layer which handles the classification. This layer performs a multiplication between an input array and a matrix with weights. As shown in the Figure, image pixels are flattened into an input array, and scores are calculated as output results. Then, in the Multi-class case, the dense layer is followed by a Softmax activation function which transforms scores into probabilities that sum up to one. The position of largest probability then gives us a one-hot encoded vector with only one ’1’, which provides the predicted class label.

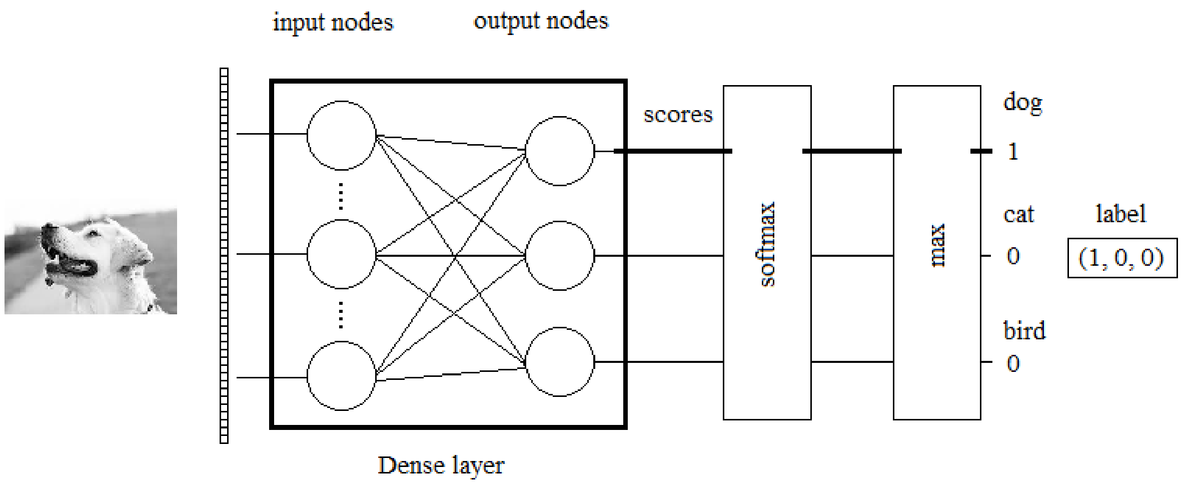


Figure 6: Example of Multi-class feed-forward Neural Network classification.

On the other hand, in the Multi-label case, the dense layer is followed by the Sigmoid activation function, which is applied separately on each score element and produces values between 0 to 1. If the value at a certain position is greater than 0.5, the one-hot encoded vector writes ’1’ at the same position, see Figure 7.

At this point, we can clearly understand that two parameters define the size of the dense layer, hence the size of our models: *input nodes* and *output nodes*. The number of output nodes equals the number of different classes of the specific classification problem, so we can not modify them.

What about the input nodes? They depend on the size of the input image. So large images imply more complex models, in terms of number of parameters (matrix of weights), and more time-consuming training and inference. A good solution is not to directly feed the network with the whole image but with useful features extracted from it.

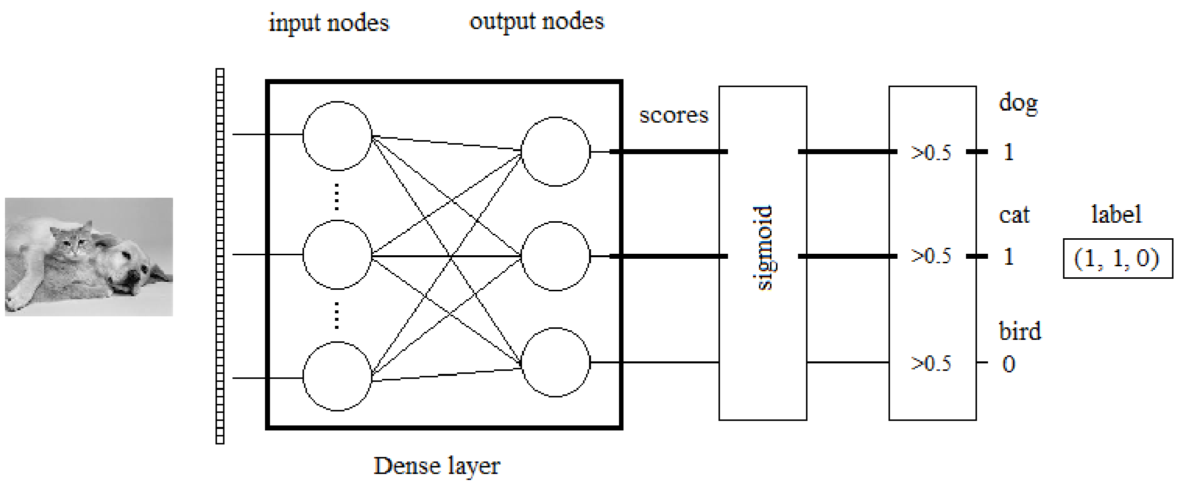


Figure 7: Example of Multi-label feed-forward Neural Network.

This can be done either by manual feature engineering or automatically using Convolutional layers. There are three widely used types of convolutional layers: conv1d, conv2d, conv3d which handle time series feature extraction, image feature extraction and video feature extraction, respectively. In this thesis, we chose CNNs using 1D Convolutional layers. So, by looking back at Figure 2, we can substitute Layer1 and Layer2 with two Convolutional layers to create our purposed CNN for SLR (see section 4.4.1). Using appropriate hyper-parameters (see section 2.3.1) to the convolutional layers, we can reduce the output of the second layer; hence, reduce the input nodes of the dense layer, which is the most complex layer in terms of memory (number of trainable parameters).

Once we design our Multi-class and Multi-label networks, it is time to train them. By following the structure in Figure 3, we need one final step. Define *loss function* and *optimizer* for each method. This thesis focus on inference and not on training. So, without any deeper explanation, we use the *RMSprop* optimizer for both networks and two variations of the cross-Entropy loss function, *Categorical Cross-Entropy*, and *Binary Cross-Entropy*, respectively [18].

As we mentioned before, in this thesis, we will not use NN models on images but on time-series data, and the above example is used to make it clearly understood how to face such a classification problem.

2.6 Related Work

Sign Language Recognition has been studied from early years, and different solutions, such as template matching, feature extraction, statistics, and learning algorithms, have been applied [23]. These approaches can be separated into three basic categories, depending on system design. First, is a *Vision-Based* approach in which systems contain at least one camera as the basic component and hence the recognition is using image processing methods. Second, we have a *Sensor-Based* approach. Time-series data is captured from wearable devices, such as sensor-gloves, tracking devices, and other sensors (e.g., IMUs, EMGs, flex sensors, etc). The last one is a *Hybrid* method, in which both approaches are combined.

To come up with a well-designed SLR system some of the challenges are common regardless of the chosen approach. We start from the size of the vocabulary; it can be some letters or numbers, a whole alphabet, a set of words, or even sentences. The larger the size of the vocabulary, the more (computationally) complex the solution. Another factor that increases the complexity is the number of hands used (one-handed recognition vs two-handed). In the case of two-handed, we need two wearable devices if we have, for example, a sensor-based system. This implies doubling the number of signal channels, so a more complex model is needed. The same is also true for a vision-based approach. If we think of a gesture in which one hand overlaps the other, then correct recognition requires a more powerful model.

Why do we need SLR solutions with low complexity? SLR is a real-time problem by its nature because communication is a direct interaction between people. Hence, a good system must respond fast, in an amount of time which is less than 300ms, as described in [24]. Finally, another big challenge is the acquisition mode of input data. It can be either isolated or continuous. In isolated mode, classification/recognition is made upon a gesture, one at a time. We feed the model with a gesture, and the model tells us what this gesture means. That can be done by using a button to capture the exact region of the gesture. On the other hand, in continuous mode, a frame of images or a series of data is captured and imported continuously into the model. So the model has not only to classify the frame, but it has to ensure that the input data corresponds to a valid gesture. Let us discuss below the different studies that have been reported in the literature for addressing SLR.

A. Wadhawan and P. Kumar [25] provide an overview of how researchers have approached the SLR problem until 2017. Most focus on video-based systems using a camera and classify static and one-handed gestures in isolated mode. The leading classification mechanism is Neural Networks (NNs) in vision-based systems followed by, Support Vector Machines (SVMs) and Hidden Markov Models (HMMs). Some vision-based system techniques are presented below:

Isolated

Q. Munib et al. [26] use a ANN with two hidden layers to classify 20 static gestures of the American sign language (14 letters, 3 numbers and 3 words). A dataset of 300 images (20 gestures x 15 times each gesture) pre-processed and a feature vector is constructed based on Hough transformation before feeding the model. 200 images (10 of each gesture) are used for training while the remaining 100 for testing. Their method achieves 90% recognition accuracy on unseen images.

W. Tangsuksant et al. [27] use an ANN with one hidden layer to classify the American alphabet (only static). They place two cameras to capture 6 colored markers on a glove from different angles and convert captured images into 3D object space coordinates using the DLT algorithm. 2,100 images are used for training while 480 (20 images per posture) for testing, achieving 95% accuracy.

M. M. Islam et al. [28] designed an android application to translate the alphabet and numbers of the American sign language by snapping images with the mobile camera. Then five features exported (fingertip finder, eccentricity, elongatedness, pixel segmentation and rotation) and feed an ANN with one hidden layer. The purposed model trained with a dataset of 1,850 samples of 37 signs (50 samples for each sign) and tested on 370 samples (5 signers x 2 times x 37 signs) captured in real time, achieving an accuracy of 94.32%.

M. Zamani and H. R. Kanan [29] present a method for recognizing American sign language alphabets and numbers (36 signs) based on saliency of images. After saliency detection, the output image processed by PCA and LDA methods to reduce its size and minimize/maximize an internal/external class distances respectively. Then the exported feature vectors feed an ANN with one hidden layer. The robustness of the model was examined through 4-Fold Cross Validation on a 2,520 sample dataset (70 times x 36 signs) where 1,890 samples was used for training and the remaining 630 for testing, achieving an average accuracy of 99.88%

Continuous

P. V. V. Kishore et al. [30], purposed a multi feature model for recognizing continuous gestures of Indian sign language with the classification stage be an ANN. A sentence of 58 words was captured by 10 subjects. Five of them was used for training and remaining for testing. Horn Schunck optical flow algorithm applied to extract tracking features (position vectors of hands) while Active Contour model extracts hand shapes features along with head portion. Combining the above features, they train and test the ANN achieving an accuracy around 90%.

D. Kelly et al. [31], presented a framework for continuous Irish sign language recognition. They use a camera to capture double handed dynamic signs as well as head movements (face position, width and eye position). Mean shift algorithm and haar cascade applied to extract features respectively. The classification stage is a Multi-channel HMM, which leads to the accuracy of 95.7%.

In [23], M.A. Ahmed et al. also present approaches from 2007 to 2017 and give us details about the hardware specifications they use (various handmade and commercial sensor gloves, microcontrollers etc.). Especially for sensor-based systems, K. Kudrinko et al. [11] presents approaches from 1991 until 2019. Some more recent are the followings:

Isolated

B.G. Lee and S.M. Lee [32] use a custom sensor glove with five (5) flex sensors, two (2) pressure sensors and a 9-Axis IMU to distinguish characters in the American Sign Language alphabet. Data from the sensor glove are captured via the Atmega328 microcontroller, which is the main computational core. Flex data are standardized while orientations (pitch, roll, yaw) are computed from IMU data. Hence a feature table is extracted, and the SVM classifier is run by the same microcontroller to predict the current sign. For training the classifier, a dataset was created by using twelve subjects with a total of near 6,500 samples (20 times x 28 signs x 12 subjects). Their method achieves 98,2% recognition accuracy on a 26-letter-and-two-sign vocabulary.

S. Jiang et al. [24] presents a wrist wearable device with four (4) sEMG sensors and one (1) IMU module, which is more comfortable than gloves, to recognize eight (8) air and four (4) surface gestures. Raw data captured from the device are divided into windows, of a predefined length, to extract features over a series of points and then predict the gesture. There are four features for sEMG sensors: mean absolute value, zero crossing, slope sign changes, waveform length, and two for IMU module: mean absolute value and waveform length. After feature extraction, an LDA (Linear Discriminant Analysis) classifier predicts the current sign every 100ms, implying a real-time system. Training of the classifier is done in two parts: *Build* the classifier and *Update* it. For the first, tree trials of data sets are captured using long-time training history data combined with short-time current training data to design a relatively robust classifier. The second one is applied for calibrating the classifier because sEMG signals may differ each time we put on and off the armband device. This method achieves 92.6% recognition accuracy on eight air (without touching a surface) gestures and 88.8% on four surface (touching a ground surface) gestures with two distinct force levels.

S. Yin et al. [33] uses a custom sensor glove with five flex sensors (one for each finger) controlled by an STM32 microcontroller and focus on the recognition of static gestures, setting the intuitive distinction between the digit gestures 1-6 and alphabet gestures A, T, W. Their approach is a combination of template matching method followed by a NN, to achieve a better recognition accuracy 99.8% than achieved individually (96.7% and 98.4%, respectively). For implementing this, a data set of 9,000 samples (5 subjects x 200 times x 9 gestures) is captured, and a template base (the average of values for each gesture and each finger sensor) is created. Then a template matching algorithm checks for the similarity between current sensor data and a template base using Euclidean distance. These outputs are normalized first and feed a NN afterwards, which improves the recognition rate and accuracy.

S. P. Y. Jane and S. Sasidhar [34] use a Myo armband to recognize 48 words from Signing Exact English (SEE-II) lexicon. An accuracy of 97.12% was achieved. Their method applies a wavelet de-noising filter to the incoming data, and data segmented using Teager-Kaiser energy operator (TKEO) thresholds. Then a 12 element feature vector is extracted like *Maximum Amplitude*, *Mean Absolute Value*, *Zero Crossing*, *Modified Mean Frequency*, *Maximum Energy Frequency*, *Wavelet Energy* etc. These features feed a NN classifier with three hidden layers. A data set of 4,927 samples is captured and split into three parts, 60%, 20%, and 20%, which are used for training, evaluation, and testing.

J. Gałka et al. in [35] presents a wearable device that consists of seven 3-axis accelerometers (one for each finger, one for the wrist, and the last for upper arm) to recognize 40 gestures describing days of the week, months, basic numerals, and names of medical specialties. He uses a PaHMM (one HMM for each channel/sensor) with the combination of a joint HMM model added as a new parallel channel to extract about 40 features. Then the classification is done by a token passing algorithm which finally achieves a 97.75% recognition accuracy (same as the single joint HMM) but with less error (about 60%). A data set of 2,000 (5 signers x 10 times x 40 gestures) recordings is used to validate the solution.

All previous studies focus on isolated gesture recognition using sensor data. However, to implement a sign recognition system that can correspond to real-world conditions, this system must be designed for continuous SLR, which is a more challenging project. Only about 20% percentage of the reported studies works on this problem.

Continuous

Kehuang Li, Z. Zhou, and C. Lee in [36] use two custom sensor gloves with gyroscopes and accelerometers set in the middle of every bone of both hands. Their method is based on the ASR (Automatic Speech Recognition) framework using HMMs, connecting two models, *transition* and *static*. These models were trained in the same way as ASR phoneme modeling using a data set of 9,216 (6 signers x 3 times x 512 words) samples with Chinese signs and 2,580 (6 signers x 2 times x 215 sentences) samples with sentences. The method achieves 87.4% recognition accuracy on 1,024 test sentences.

N. Tubaiz, T. Shanableh, and K. Assaleh in [37] present a continuous Arabic Sign Language recognition of 40 sentences consisting of an 80-word lexicon using two DG5-VHand data gloves. These forty sentences were recorded ten times by one subject. Acquisition frequency is set to 30 readings per second, and a sliding window is used. This sliding window with size w aims to extract local features such as standard deviation and means for each sensor. Then these features are appended to the original sensor readings to reserve long-term trends. This method works as an lowpass filter and it results in a smoothed version of the original signal by containing information about past and future sensor readings. Pre-processed features are stored and then labeled manually from a video captured simultaneously with data. For classification, a modified K-Nearest Neighbors classifier was proposed. The method achieved 98.9% recognition accuracy on testing data (30% of the original data set).

In [38], Yan Li et al. implemented an automatic continuous Chinese Sign Language (CSL) recognition system using a 3-axis accelerometer and four EMG sensors in each hand. Raw data from sensors are collected with a sampling rate of 1kHz. Then segmentation is performed using the amplitude of EMG sensors which is a good reference for the automatic detection of subword segments within continuous streams signal. These boundaries are then applied to accelerometers signals too. Three parallel classifiers (for handshape, orientation, and movement) are evaluated individually and then integrated for subword-level classification. The first two extract features (means absolute value, 4-order auto-regressive coefficients from 3-axis accelerometers, and onset/offset orientation features from SMG sensors, respectively) and then LDC (Linear Discriminant Classifier) is used for the training. The movement classifier is a multi-stream HMM classifier that uses extracted features from data of all sensors simultaneously. Finally, a two-stage integration is performed, which combines these three classifiers into steps to produce the current prediction. For training and testing the whole model, a data set of 40 sentences consisting of 116 signs is captured by two subjects, two times and one time, respectively. This method has achieved 97.6% recognition accuracy.

2.7 Thesis contributions related to state of the art

In contrast with vision-based approaches in which CNNs take the lead, in sensor-based approaches CNNs are not widely employed. Even though 1D CNNs are used on extracting patterns from time-series data, the only report we found that applies them is presented by Wang F. et al. in [39].

In [39] the authors build a Recognition-Verification mechanism to address the Chinese SLR problem in continuous mode. Two models are used in parallel to classify 86 sign language categories, including an empty class—classification model based on VGG (Visual Geometry Group) [40] and verification on the Siamese network [41]. The whole process is based on a sliding window and each time window moves a step forward, VGG performs classification while Siamese judges the correctness of recognition of the first.

In this thesis, we focus on creating a sensor-based system using a hand-made sensor glove. This glove consists of five bend sensors and a 9-axis IMU device to recognize 20 one-handed, static and dynamic, gestures plus an extra one (transition class) from American Sign Language vocabulary. Two approaches, one for isolated recognition and the other for continuous, are implemented during this writing. However, we will present only the continuous/online one as being the more challenging case. That is because the adequate segmentation information of real-time input data is not apparent, so we do not know the boundaries of each gesture while moving the sliding window.

In contrast with the recognition-verification mechanism, we use a single CNN network to address this difficulty by checking if the sliding window covers more percentage of a gesture and a little of the transition period or the opposite while on training time. As a result, some wrong predictions are displayed as the sliding window enters or gets out of a gesture, but this does not affect gesture recognition. More on this technique will be discussed in Chapter 3. Our approach is based exclusively on 1D CNNs and all necessary features are extracted automatically by them, without requiring to find useful representations from the raw data following state of the art methods.

3. SYSTEM DESIGN - DATASET CREATION

In this chapter, we present our system for ASL recognition, which uses signals collected from a wearable device (sensor glove). Compared to vision-based approaches, this method is computationally less expensive and remains unaffected by light intensity, background color and motion occlusion factors. Also, there is no need to set up cameras or other detecting devices, but all we need is one portable device.

3.1 The Sensor Glove

The sensor glove we used is a handmade device that has five flex sensors (Sparkfun 4.7 inches) for measuring fingers' bend and a 9-axis Inertial Measurement Unit (IMU) (Adafruit LSM9DS1 with 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer) for measuring wrest angle and rotation. All sensors are connected to the Arduino Uno R3 board, which is responsible for reading their values with a sampling frequency of about 30/100 Hz (continuous/isolated).

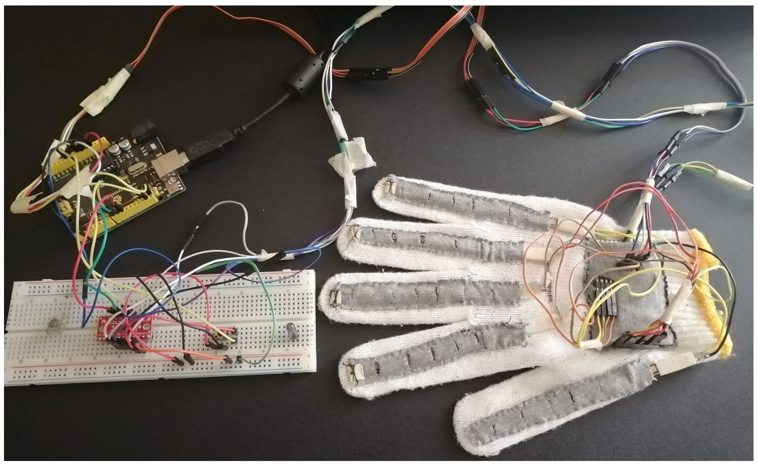


Figure 8: Our sensor glove with all its components packed away in a usb device.

In more detail, a flex sensor is a thick laminate where its resistance changes on different bend angles. Values scale between 24 kOhm (no bending) and 48 kOhm (full bending). We use a voltage divider with a constant resistor of 48 KOhm in series and an input voltage of 5 Volts to capture its behavior. The output voltage is formulated as

$$V_{out} = \frac{RR_1}{R + R_1} V_{in} \tag{3.1}$$

where R is the sensor's variable resistance, R_1 is the constant resistor, and V_{in} is the input voltage. To read the value of V_{out} we use Arduino's input analog ports. Due to the limitation of these ports (only four supported when I2C communication is used simultaneously), we add an analog 5-to-1 multiplexer, and values of each sensor are read one after the other in circular mode. Multiplexer's voltage and other specifications are suitable with Arduino board.

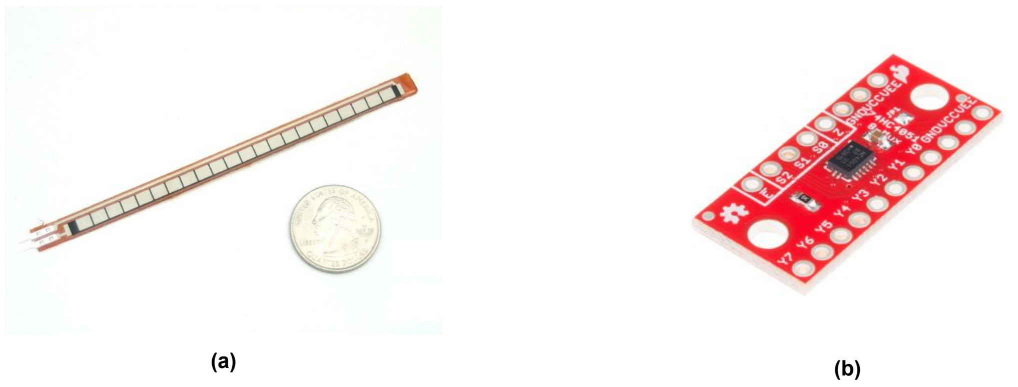

(a) (b)

Figure 9: (a) A thick laminate flex sensor. (b) An IC analog multiplexer device.

The LSM9DS1 device includes both an I2C serial bus interface and an SPI serial standard interface. In this project we connect the device to the arduino through I2C interface and we use an already created by the manufacturer C library to read its values. So, by using high-level functions, we can easily read the appropriate values, gyroscope, accelerometer, and magnetometer, without any further knowledge.

Before combining all these into an Arduino IDE project and uploading it to the Uno board, we need to face another challenge. We have to attach correct "pre"-labels (will discuss this later on in section 3.2) to the recorded time-series data, to separate gesture periods from transition ones. To achieve this, in [35] and [37] a camera was used and a manual method applied frame by frame. Similarly, in our project, we add an external button to the Arduino board, and our device's hardware is ready to use. As we mentioned before, we developed two different source codes (firmware) for our two modes: *isolated* and *continuous*. However, we present here only the continuous SLR case, which is more challenging. In contrast to the isolation case, we do not have to press any button (on inference) to assign a gesture's borders, so we can freely move our hands more naturally. That allow us to extend our project using two-handed words in future work.

3.2 Data Sampling

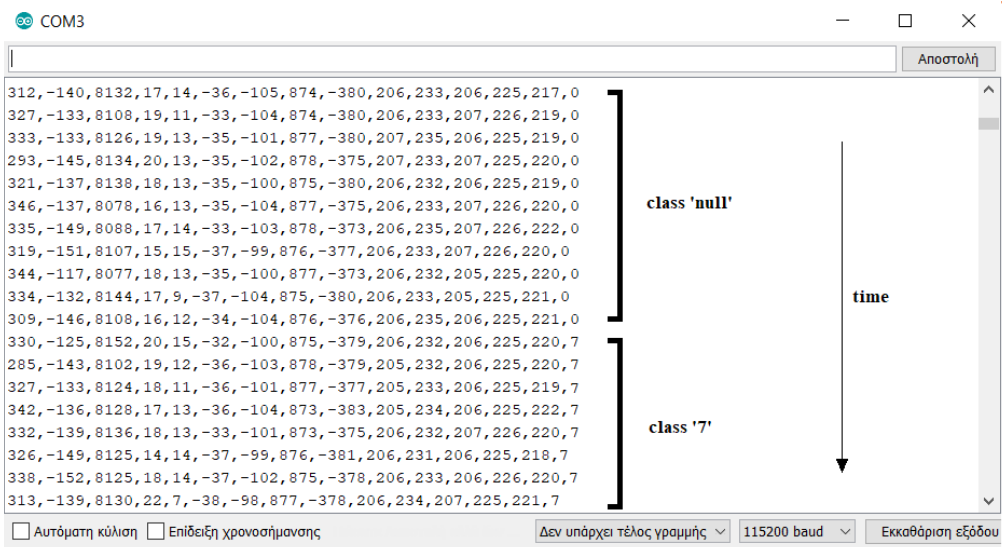
After the sensor glove was made, we created a dataset for training, validating, and testing our network. For simplicity and due to time limitations, all recorded gestures were captured by one signer (the author of this document) and cover a vocabulary of twenty gestures. It includes the first ten letters of the American alphabet and the ten most common ASL words plus one extra class: the null/transition class, as shown in Table 2 with the corresponding class label attached.

Table 2: The ASL vocabulary used for the project.

alphabet : class		words : class	
a : 1	f : 6	home : 11	hat : 16
b : 2	g : 7	thank : 12	eat : 17
c : 3	h : 8	hello : 13	happy : 18
d : 4	i : 9	drink : 14	sorry : 19
e : 5	j : 10	apple : 15	go : 20
null/transition : 0			

In more detail, 20 sentences of 5 words each, were generated randomly and captured 5 to 7 times one by one separately to make the process more general. We use five words in a sentence to avoid making mistakes while recording and re-capturing it.

Let us take the first sentence ***”g d hat f apple”*** and see how it works. We plug the sensor glove via Arduino into the PC and open the serial COM port. Then, Arduino starts reading the sensor data one by one (3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer, five flex sensors) and printing them to the serial port. As shown in Figure 10 below, before moving on to the next timestep, we append a ”pre-label” and then go to the next line.



```
312,-140,8132,17,14,-36,-105,874,-380,206,233,206,225,217,0
327,-133,8108,19,11,-33,-104,874,-380,206,233,207,226,219,0
333,-133,8126,19,13,-35,-101,877,-380,207,235,206,225,219,0
293,-145,8134,20,13,-35,-102,878,-375,207,233,207,225,220,0
321,-137,8138,18,13,-35,-100,875,-380,206,232,206,225,219,0
346,-137,8078,16,13,-35,-104,877,-375,206,233,207,226,220,0
335,-149,8088,17,14,-33,-103,878,-373,206,235,207,226,222,0
319,-151,8107,15,15,-37,-99,876,-377,206,233,207,226,220,0
344,-117,8077,18,13,-35,-100,877,-373,206,232,205,225,220,0
334,-132,8144,17,9,-37,-104,875,-380,206,233,205,225,221,0
309,-146,8108,16,12,-34,-104,876,-376,206,235,206,225,221,0
330,-125,8152,20,15,-32,-100,875,-379,206,232,206,225,220,7
285,-143,8102,19,12,-36,-103,878,-379,205,232,206,225,220,7
327,-133,8124,18,11,-36,-101,877,-377,205,233,206,225,219,7
342,-136,8128,17,13,-36,-104,873,-383,205,234,206,225,222,7
332,-139,8136,18,13,-33,-101,873,-375,206,232,207,226,220,7
326,-149,8125,14,14,-37,-99,876,-381,206,231,206,225,218,7
338,-152,8125,18,14,-37,-102,875,-378,206,233,206,226,220,7
313,-139,8130,22,7,-38,-98,877,-378,206,234,207,225,221,7
```

Figure 10: COM port display. Raw data samples flow over time.

Since the sampling frequency is 30Hz, a new line is printed in 1/30 sec, and so is our timestep.

It is clear that our signal has two regions: *active* and *transition*. Active regions are the periods of the signal where a gesture evolves and transition zones where the movement of the hand has no useful meaning, in general. At the same time, it goes from one gesture to another, or relaxing. To separate these two regions, we use a left-hand button when recording instead of manually separating video as described in [37].

While we are in the active region, we press the button, and the pre-label is set to the response class. Otherwise, the button is not pressed, and the pre-label is set to ”0”. In the example sentence, we start with a transition region, so the first eleven timesteps are marked as ”0”, and then the gesture ”g” is performed with the class ’7’ (button is pressed). In the following Figure 11, we can see the graphical signal representation of the whole sentence, where we can distinguish the active from the transition regions. The vertical red lines mark the boundaries of the gestures and show the moments when we press and release the button at recording time.

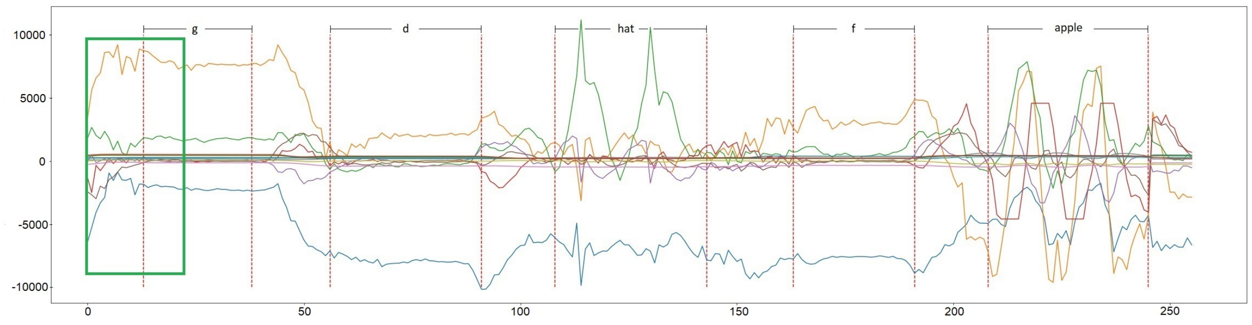


Figure 11: The signal representation of the first sentence. Horizontal axis is time. Different colors distinguish the signal of each sensor.

3.3 Sliding Window

We now have a long sequence of 23,862 timesteps that involves a full recording of 13.25 minutes (23,862/30 steps per second/60 sec per min) and includes all the sentences with all the repetitions in a text file. Next, we manually divide it into two parts: *train* and *test* part. The test part consists of the last recording of each sentence and the train one of the remaining recordings. It is more practical and safe to handle two datasets instead of one, to avoid mixing them during model training.

The final dataset construction and specification is not yet finished. We also need an additional procedure for extracting samples in a suitable format which is based on a **sliding window** method, see Figure 12. An arbitrarily sized window slides along the signal on the time axis one step (point) at a time, and a sample is exported (exported samples are overlapped). A unique class label should characterize each sample, but it becomes evident that there may be an overlap between two or more classes during sliding. In this case, the assigned label to the exported sample is the one that covers the majority of the time frame, thus occupying the largest percentage in the specific sliding window. The whole process was implemented using Python and applied to both training and testing datasets individually.

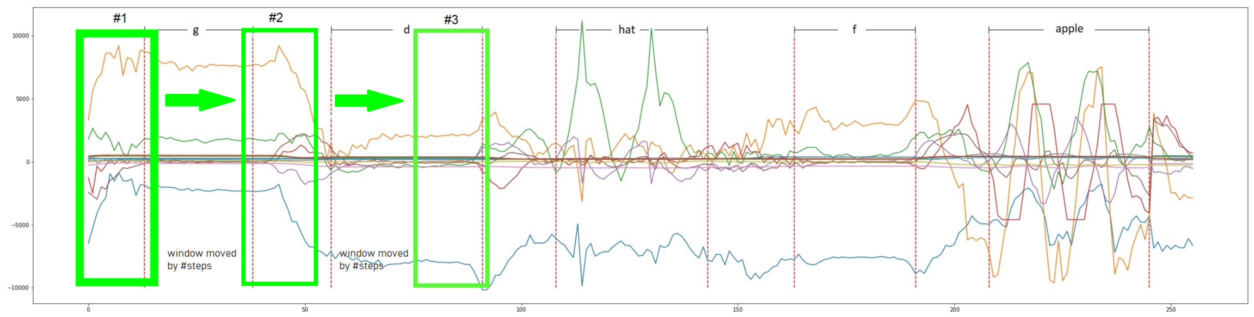


Figure 12: Sliding window (green) process and overlapping class windows. The transition class is assigned to both #1 and #2 samples and 'd' class to sample #3.

A window size of 20 was chosen for our experiments, but we can easily modify it for further tests. A brief experiment (not presented here) with other sizes, e.g. 15, 25, 30 yielded similar results. The reason we chose values near 30 is to be close to the sampling frequency. Another reason is that we do not want the window to include many gestures together, if we consider that the majority of the gestures takes less than a second to be completed. Applying a larger window might be a good choice if we intend to implement a multi-label method, as described in section 2.5.1. So, by using a window size of 20, the final dataset consist of 29,362 overlapped samples (20x14 matrices) where 23,841 (81%) are used for training and the remaining 5,521 (19%) for testing.

3.4 Pre Processing

Most of the time, data have a different scale for each feature. There are four types of data in our case, one for each type of sensor (accelerometer, gyroscope, magnetometer, and flex sensor). As a result, the trained ML model may give more attention to some features with bigger values than others with smaller ones. To address this issue, there are two widely used methods, *Normalization* and *Standardization*. The second method is used in our implementation via the *scikit-learn* library [42].

- ***Normalization:*** An estimator scales and translates all data values in the range of 0 to 1, individually for each feature i, by using the formula below:

$$y(x) = \frac{x - x_{min}}{x_{max} - x_{min}}, \tag{3.2}$$

- ***Standardization:*** Similarly, this estimator scales and centers values, individually for each feature, so data follows normal distribution (mean:0 , variance: 1), by using the formula:

$$y(x) = \frac{x - \mu}{\sigma} \tag{3.3}$$

For the training set, consisting of 23,841 overlapping sliding windows with shape = (23841, 20, 14), we first apply the NumPy *reshape* method to convert the shape to (23841*20,

14) and then use a *scaler.fit_transform* method from scikit-learn standard scaler library. Reshaping is necessary as the standard scaler can handle a 2-dimension input and also we want to standardize values of each feature in overall and not per sample; hence μ and σ have the same 14 length size. The test set follows the same process but instead of the *scaler.fit_transform* method, we apply *scaler.transform* one, in which transformation uses the previously calculated vectors μ and σ.

3.5 Conclusions

In this chapter, we first presented a low-cost sensor glove device using five flex sensors and an IMU device. We used it to capture 20 different gestures into 20 randomly generated sentences of 5 words each and create a dataset of 29,362 samples (2D arrays size of 20x14), where 23,841 (81%) is used for training and 5,521 (19%) for testing our model for continuous SLR. In the next chapter we will discuss how different Machine Learning models, including our proposed one, behave on our dataset.

4. MODEL DEVELOPMENT

In this Chapter, we apply different Machine Learning methods for SLR using our previous captured and pre-processed dataset. In addition, we design a simple three-layer CNN model. Finally, we compare all models to find a winner, i.e. the one with the best accuracy on the test dataset.

4.1 Baseline ML Methods

As we want to address the SLR problem in a data-driven way and not by using handcraft rules, Machine Learning is the right choice. Among different classical ML methods for classification, e.g., Support Vector Machine (SVM) [43] [44] [45], Random Forests (RF) [46] [47], Linear Discriminant Analysis (LDA) [24] [44], Neural Networks (NN) [48], XGBoost [48], etc. we need to investigate which one is the most appropriate for our problem. Thus, we have to measure the effectiveness of these models for our dataset. We use accuracy as a metric for this, so the model with the highest correct predictions on unseen data (test dataset) is declared the winner. A little attention is needed on interpreting erroneous predictions. An error between two non-transition classes is vital because gestures should not be confused for each other. However, an error between a transition and an active class is acceptable when it appears at the beginning or at the end of a gesture. We will explain this point in more detail later on in this chapter.

Before applying different ML methods to our dataset and choosing the best one, we must first tune their hyperparameters. Model tuning is necessary for almost every ML problem and helps us find the best model corresponding to a given dataset. As described in [49], tools such as *ray tune*, *hyperopt*, *tensorboard*, *scikit-learn* libraries can be used to achieve this goal. In this thesis, we will use *ATOM* (Automated Tool for Optimized Modelling) [50], a high-level package based on *scikit-learn* [51], which gives us the ability to run experiments quickly and efficiently without requiring any profound knowledge.

ATOM has a large set of predefined ML models that we can try and an API to incorporate other more complicated models compatible with *scikit-learn*. For our purpose, we have chosen the following five well-known models:

- Linear Regression (LR) [52]
- Random Forests (RF) [53]
- Linear Discriminant Analysis (LDA) [54]
- Linear Support Vector Machine (LSVM) [55]
- XGBoost (XGB) [56]

We tried to include classical ML models and more recent one, such as XGBoost; a powerful ML algorithm often declared as a winner in Kaggle competitions.

From the above methods only RF, LDA, SVM have been applied before on sensor-based SLR, based on [11] (Table I). For instance, in [43], R. Fatmi et al. used SVMs to recognise 13 signs of the American SL, using two Myo Armband devices achieving 85,5% test accuracy. In [24], as described in Chapter 2, an LDA method is applied interpreting 8 gestures, using a 4-channel sEMG and IMU device with an 92,6% accuracy. In [44], both LDA and SVM methods are used to recognise 10 ASL letters, using a custom device consisting of 5 flex sensors, and achieving 97,81% and 97,87% accuracy respectively. In [46] and [47], the authors use RF to recognise 22 French SL letters and 26 ASL + 1 sign with a 92,95% and 79.35% test accuracy respectively. Finally, another research [45], with a larger vocabulary of 80 commonly used ASL signs, achieved 96,16% test accuracy using SVMs. It is important to emphasize that all previous attempts are focused on isolated SLR.

On the other hand only few researches have been taken place to address continues SLR while the majority applies HHMs as a baseline model: Simple HMM [36] [57] [58] [59], Multi-Stream HMM [38] [60] [61] [62], HMM combined with RNN (Recurrent Neural Network) [63] [64] and HMM combined with NN [65]. In addition, two other works found using other models. In [37] the authors apply a K-Nearest Neighbour model and in [39], apply a parallel combination of a CNN with a Siamese network, as described in sections 2.6 and 2.7 respectively.

4.2 Hyperparameter Tuning

Now it is time to set up our baseline models. There are three basic ways [66] to do this: *Grid Search*, *Random Search* and *Bayesian Optimization*. In all methods, the corresponding algorithm tries different hyperparameter values from a given space. It trains all different sub-models on a training dataset, evaluates them on an evaluation dataset, and returns hyperparameters with the higher (or lower) metric score, which is the accuracy score in our case. Finally, we train the best sub-model on the whole (train + evaluation) dataset and test it on unseen data (test dataset). Accuracy shows how good or bad the model is. This process is repeated for each LR, RF, LDA, LSVM, and XGB sub-models, and the accuracy scores are compared. In the Grid-Search method, the hyperparameter space consists of discrete values, and the search for the best sub-model is done considering all combinations. This makes it a slow method. On the other hand, Random search selects random values for each hyperparameter in the search space. So, if we are lucky enough, we can achieve good results, but if search space is large, it is more difficult to obtain good results.

In this thesis, we will apply [67] *Bayesian Optimization*. In contrast to Grid Search and Random Search, BO exploits previous tries and finds parameter regions with promising results [68], so we keep exploring these regions rather than other areas. In complex hyperparameter spaces, Bayesian Optimization is the best method for fine-tuning a model.

Let us start the python code using ATOM (see Appendix A for full code). Implementation is simple and includes three steps:

1. Loading standardized train and test datasets as numpy arrays with corresponding labels. As ATOM can handle directly a dataset up to two dimensions (samples, features) we flatten/reshape a 20x14 sample into a 280 element array.

   Step 1

   ```
   dataSet      = np.load(”train_dataset_20.npy”)
   labels       = np.load(”train_labels_20.npy”)
   test_dataSet = np.load(”test_dataset_20.npy”)
   test_labels  = np.load(”test_labels_20.npy”)
   dataSet      = dataSet.reshape(23841, 280)
   test_dataSet = test_dataSet.reshape(5521, 280)
   ```

2. Import ATOMClassifier class and create an atom classifier object called ”atom”. Dataset attached in $(X\_train, y\_train), (X\_test, y\_test)$ format and $n\_jobs = -1$ allow us to use all available CPU cores at optimization time.

   Step 2

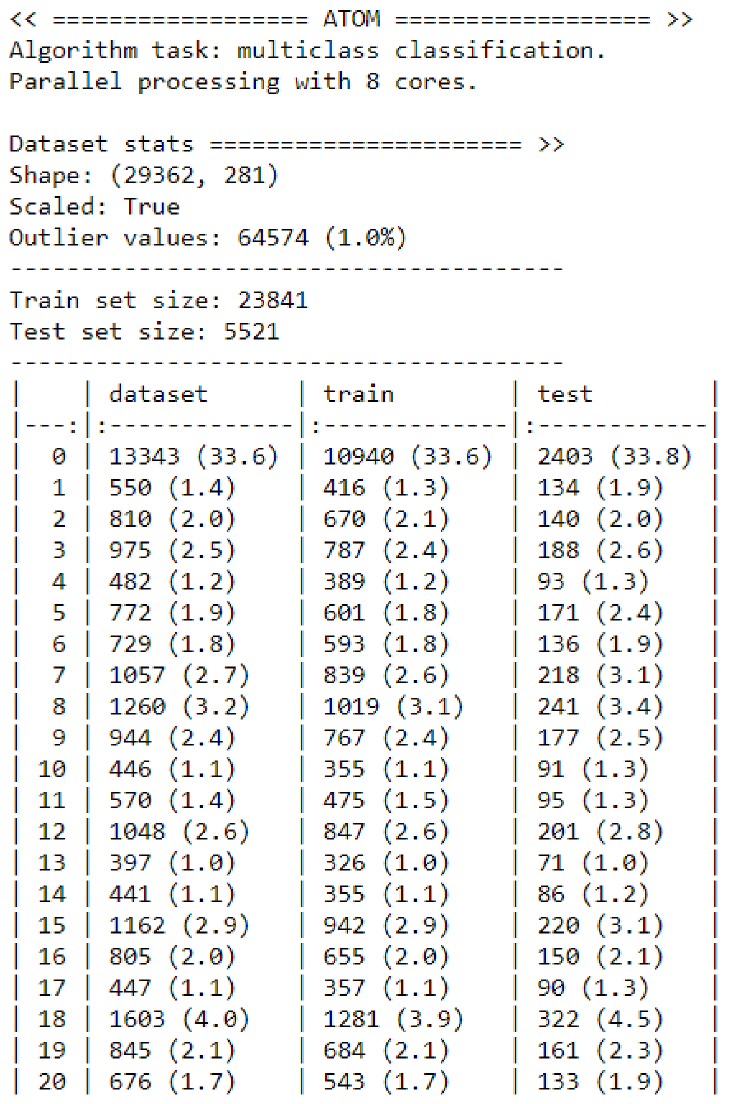
   ```
   from atom import ATOMClassifier
   atom = ATOMClassifier((dataSet, labels),
                         (test_dataSet, test_labels),
                         warnings='ignore', logger=”auto”,
                         n_jobs=-1, verbose=2)
   ```

   The output of this code prints us information about the atom object, see Figure 13. As we can see in the figure below, our task is a multi-class classification. The number of classes is 21 and is labeled from 0 to 20. The dataset consists of 29,362 samples and has 280 features plus one, which is the target label. Also, ATOM recognizes that data are scaled and shows us a table on how samples are distributed in each class.

3. Running Bayesian Optimization on LDA, LR, RF, LSVM, and XGB models. As mentioned at the beginning of this section, accuracy is used as a metric for tuning the model's hyperparameters, so the *run* method tries to maximize this metric, for each individual model.

Step 3

```
atom.run(
        models=[”LDA”, ”LR”, ”RF”, ”lSVM”, ”XGB”],
        metric=”accuracy”,
        n_calls=25,
        n_initial_points=10,
        bo_params={”early_stopping”: 0.1, ”cv”:5},
        n_bootstrap=5,
)
```



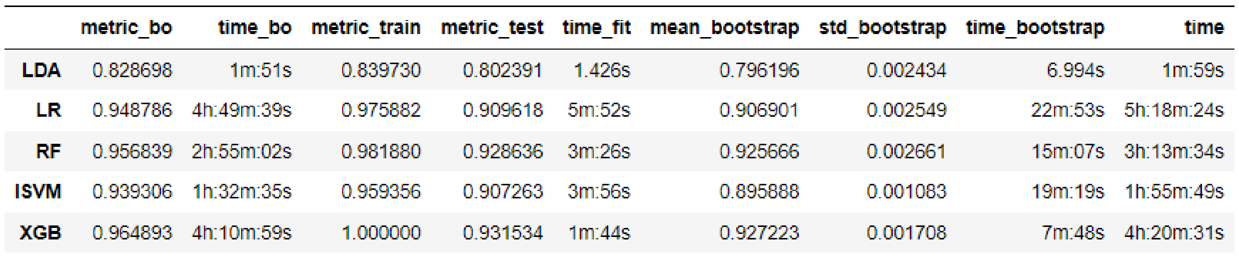
```
<< ================== ATOM ================== >>
Algorithm task: multiclass classification.
Parallel processing with 8 cores.

Dataset stats ====================== >>
Shape: (29362, 281)
Scaled: True
Outlier values: 64574 (1.0%)
---------------------------------------
Train set size: 23841
Test set size: 5521
---------------------------------------
|    | dataset      | train        | test        |
|---:|:-------------|:-------------|:------------|
|  0 | 13343 (33.6) | 10940 (33.6) | 2403 (33.8) |
|  1 | 550 (1.4)    | 416 (1.3)    | 134 (1.9)   |
|  2 | 810 (2.0)    | 670 (2.1)    | 140 (2.0)   |
|  3 | 975 (2.5)    | 787 (2.4)    | 188 (2.6)   |
|  4 | 482 (1.2)    | 389 (1.2)    | 93 (1.3)    |
|  5 | 772 (1.9)    | 601 (1.8)    | 171 (2.4)   |
|  6 | 729 (1.8)    | 593 (1.8)    | 136 (1.9)   |
|  7 | 1057 (2.7)   | 839 (2.6)    | 218 (3.1)   |
|  8 | 1260 (3.2)   | 1019 (3.1)   | 241 (3.4)   |
|  9 | 944 (2.4)    | 767 (2.4)    | 177 (2.5)   |
| 10 | 446 (1.1)    | 355 (1.1)    | 91 (1.3)    |
| 11 | 570 (1.4)    | 475 (1.5)    | 95 (1.3)    |
| 12 | 1048 (2.6)   | 847 (2.6)    | 201 (2.8)   |
| 13 | 397 (1.0)    | 326 (1.0)    | 71 (1.0)    |
| 14 | 441 (1.1)    | 355 (1.1)    | 86 (1.2)    |
| 15 | 1162 (2.9)   | 942 (2.9)    | 220 (3.1)   |
| 16 | 805 (2.0)    | 655 (2.0)    | 150 (2.1)   |
| 17 | 447 (1.1)    | 357 (1.1)    | 90 (1.3)    |
| 18 | 1603 (4.0)   | 1281 (3.9)   | 322 (4.5)   |
| 19 | 845 (2.1)    | 684 (2.1)    | 161 (2.3)   |
| 20 | 676 (1.7)    | 543 (1.7)    | 133 (1.9)   |
```

Figure 13: ATOMClassifier log.

We choose a small number of bo_iterations, $n\_calls = 25$ for each individual model, to keep running time low and a random starting state $n\_initial\_points = 10$ to initialize, through 10 random tests, the hyperparameters before fitting the surrogate function. After the hyperparameters' initialization, BO runs for additional $25 - 10 = 15$ iterations. Also, *bo_parameters* keep default/auto values like $k = 5$ for 5-fold cross-validation, but early_stopping is set at 0.1, which let us stop training if the model did not improve in the last 10% steps (is available only for models that allow in-training evaluation. XGB is not compatible). Finally, we apply the bootstrap algorithm by setting the optional *n_bootstrap* parameter to 5, and we are ready to run the code. The whole process takes about 15 hours to complete on a laptop with an Intel Core i5-8250U CPU and 12GB of RAM.

4.3 ML Results

Once the whole process is completed, we can easily compare models. ATOM's *result* method gives us a table with every model's performance in the Figure below.



	metric_bo	time_bo	metric_train	metric_test	time_fit	mean_bootstrap	std_bootstrap	time_bootstrap	time
LDA	0.828698	1m:51s	0.839730	0.802391	1.426s	0.796196	0.002434	6.994s	1m:59s
LR	0.948786	4h:49m:39s	0.975882	0.909618	5m:52s	0.906901	0.002549	22m:53s	5h:18m:24s
RF	0.956839	2h:55m:02s	0.981880	0.928636	3m:26s	0.925666	0.002661	15m:07s	3h:13m:34s
lSVM	0.939306	1h:32m:35s	0.959356	0.907263	3m:56s	0.895888	0.001083	19m:19s	1h:55m:49s
XGB	0.964893	4h:10m:59s	1.000000	0.931534	1m:44s	0.927223	0.001708	7m:48s	4h:20m:31s

Figure 14: ATOMClassifier results.

XGB model took the lead with 93.15% test accuracy, followed by Random Forests with 92.86%. Linear Regression and Linear Support Vector Machine take third and fourth place, with 90.96% and 90.72% accuracy, respectively. Finally, Linear Discriminant Analysis achieves only 80.23% accuracy, which is low compared with the other methods; thus, it is out of competition. At this point, we will refer to two other statistical values, *mean_bootstrap* and *std_bootstrap*, which are used at step 3 ($n\_bootstrap = 5$). Once the best sub-model is exported from Bayesian optimization ATOM applies bootstrap technique [69] to assess the robustness of the model. This technique creates several new training datasets (5 in our case) selecting random samples from the original training set (with replacement) and evaluates them on the same test set by calculating corresponding accuracy values. This way we get a distribution of the performance of the model. Hence, we can estimate the skill of the current machine learning model when making predictions on data not included in the training data by computing the mean and standard deviation of this distribution (*mean_bootstrap* and *std_bootstrap*). As we can see, *mean_bootstrap* is close to *metric_test* for every model with a small *std_bootstrap* value. That means that models are acting as well as possible on unseen data.

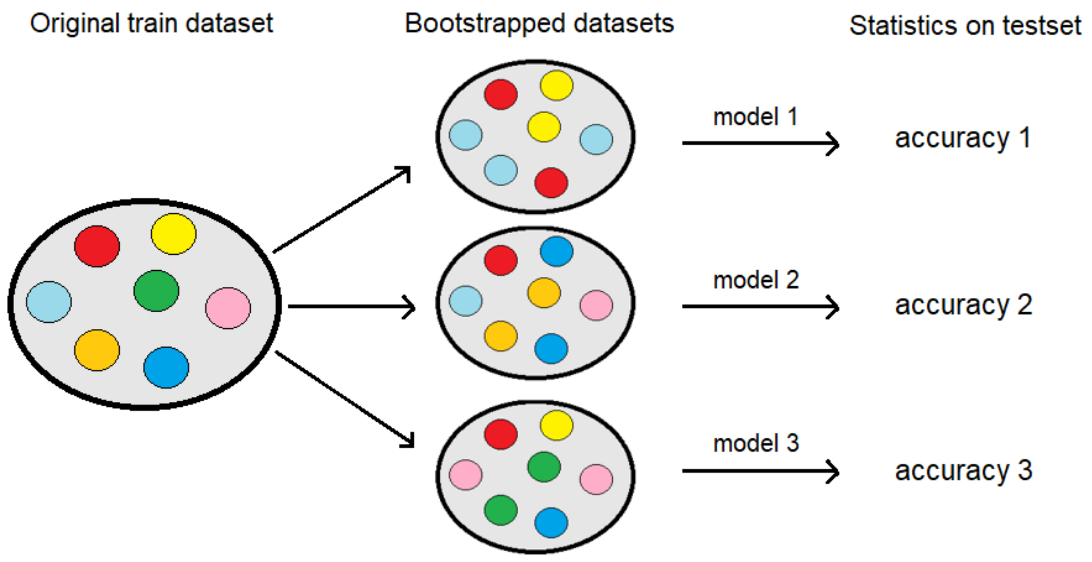


Figure 15: Bootstrap method for the case n_bootstrap=3.

Another critical point we must take into consideration is interpreting wrong predictions. Which classes are been confused? We can check results by plotting corresponding confusion matrices (figures 16 - 19), with an ATOM built-in function: *plot_confusion_matrix*. Ignoring transition class (class 0), we see that only the XGB model makes all predictions correct (every values are zero except on the diagonal). Thus XGB does not confuse gestures like the other models. Now, what is happening with the transition class? If we do not shuffle the test set and let having samples in a natural time series way, we will see that the wrong predictions appear at the beginning or/and at the end of a gesture. It means that the model recognizes a gesture a little bit later/earlier than it truly starts/ends. However, in reality, there is not an exact starting and ending point for a gesture. It depends on how we captured it and when we pressed the button during recording time. Since there is no wrong or missing gesture during continuous recognition, we can accept that the real accuracy of the XGB model is 100% for all practice cases.

On the other hand LR, LSVM and RF (Figures 17-19) made 16, 13 and 7 wrong predictions between remarkable classes, respectively. In fact, this should not be a significant issue, because a wrong prediction means that the model did not act correctly for an $1/30$ percentage of a second (sampling frequency is 30Hz). A real problem would be if more wrong predictions are happening on continuous samples in timeseries data. For example, in Figure 19, LSVM confused class number 5 with class number 17 in total of 5 times in continuous samples while in Figure 18, LR confused class number 4 with class number 12 in total of 5 times. As these errors are in consecutive windows (we printed predictions in an original timeseries form via python), their importance depends on the overall length of the current gesture. In some case we may prevent such errors by using a sorts of additional code. To sum up, without any further analysis, although the above models achieved lower accuracy they perform quite well on our dataset, too.

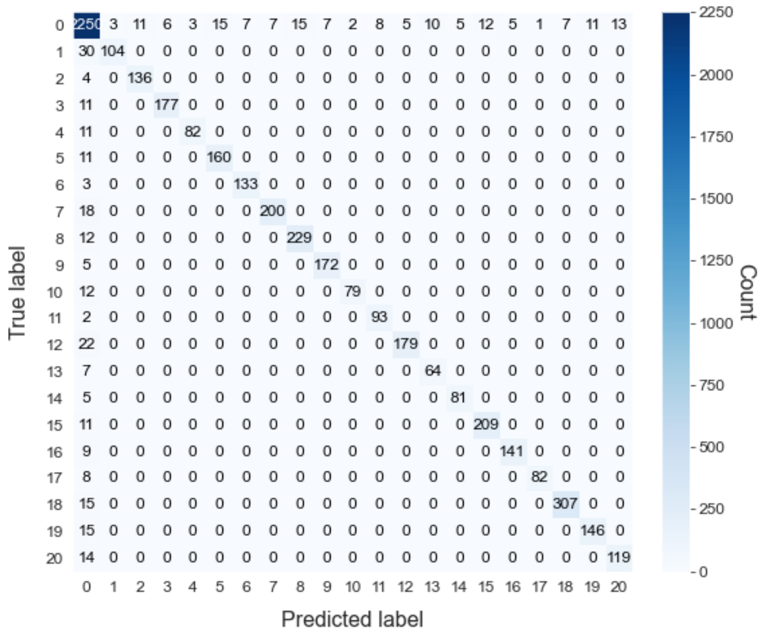


Figure 16: XGB confusion matrix

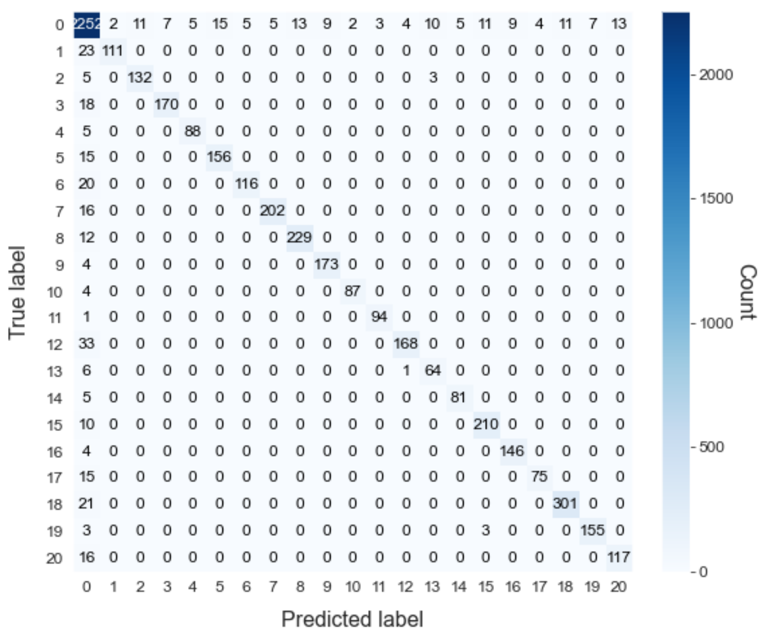
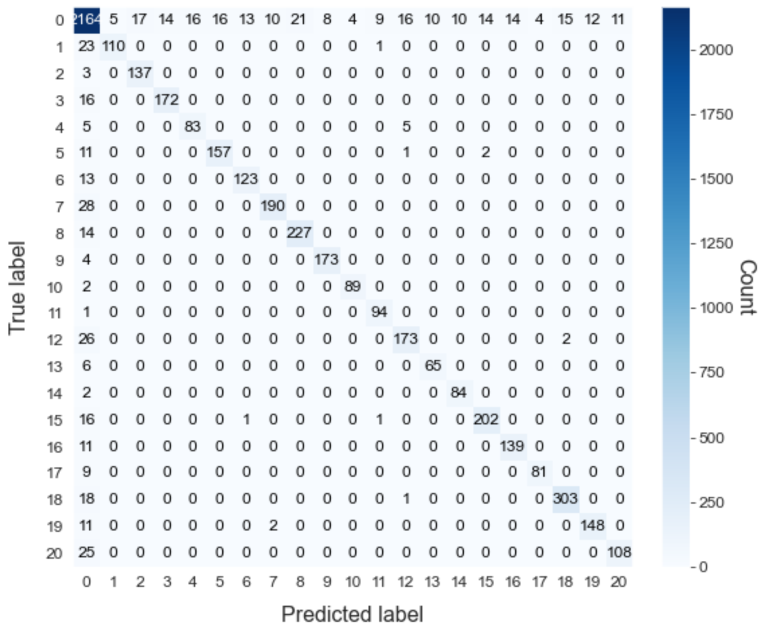
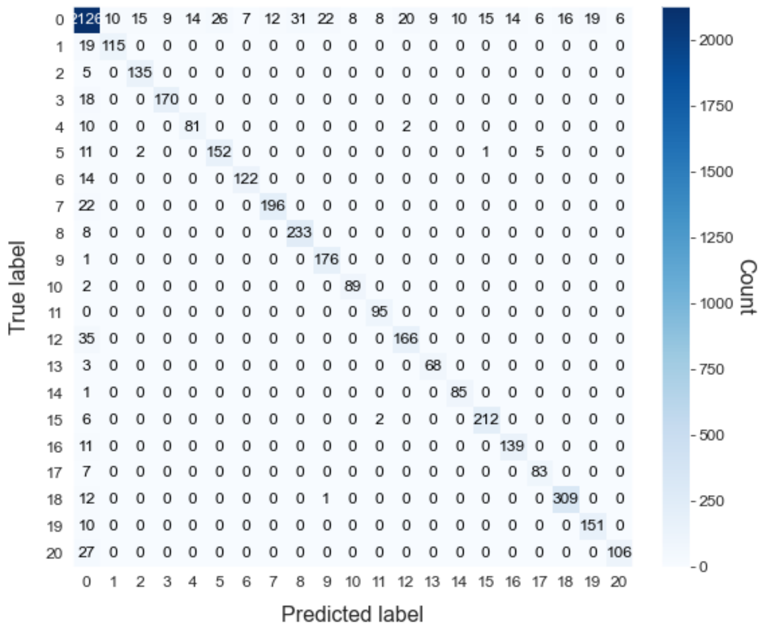


Figure 17: RF confusion matrix



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	2164	5	17	14	16	16	13	10	21	8	4	9	16	10	10	14	14	4	15	12	11
1	23	110	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
2	3	0	137	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	16	0	0	172	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	5	0	0	0	83	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0
5	11	0	0	0	0	157	0	0	0	0	0	0	1	0	0	2	0	0	0	0	0
6	13	0	0	0	0	0	123	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	28	0	0	0	0	0	0	190	0	0	0	0	0	0	0	0	0	0	0	0	0
8	14	0	0	0	0	0	0	0	227	0	0	0	0	0	0	0	0	0	0	0	0
9	4	0	0	0	0	0	0	0	0	173	0	0	0	0	0	0	0	0	0	0	0
10	2	0	0	0	0	0	0	0	0	0	89	0	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	94	0	0	0	0	0	0	0	0	0
12	26	0	0	0	0	0	0	0	0	0	0	0	173	0	0	0	0	0	2	0	0
13	6	0	0	0	0	0	0	0	0	0	0	0	0	65	0	0	0	0	0	0	0
14	2	0	0	0	0	0	0	0	0	0	0	0	0	0	84	0	0	0	0	0	0
15	16	0	0	0	0	0	1	0	0	0	0	1	0	0	0	202	0	0	0	0	0
16	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	139	0	0	0	0
17	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	81	0	0	0
18	18	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	303	0	0
19	11	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	148	0
20	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	108

Figure 18: LR confusion matrix



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	2126	10	15	9	14	26	7	12	31	22	8	8	20	9	10	15	14	6	16	19	6
1	19	115	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	5	0	135	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	18	0	0	170	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	10	0	0	0	81	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0
5	11	0	2	0	0	152	0	0	0	0	0	0	0	0	0	1	0	5	0	0	0
6	14	0	0	0	0	0	122	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	22	0	0	0	0	0	0	196	0	0	0	0	0	0	0	0	0	0	0	0	0
8	8	0	0	0	0	0	0	0	233	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	176	0	0	0	0	0	0	0	0	0	0	0
10	2	0	0	0	0	0	0	0	0	0	89	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	95	0	0	0	0	0	0	0	0	0
12	35	0	0	0	0	0	0	0	0	0	0	0	166	0	0	0	0	0	0	0	0
13	3	0	0	0	0	0	0	0	0	0	0	0	0	68	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0	0	0	0	0	0	0	85	0	0	0	0	0	0
15	6	0	0	0	0	0	0	0	0	0	0	2	0	0	0	212	0	0	0	0	0
16	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	139	0	0	0	0
17	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	83	0	0	0
18	12	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	309	0	0
19	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	151	0
20	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	106

Figure 19: LSVM confusion matrix

4.4 Using Convolution Neural Networks

Convolution Neural Networks (CNNs) are used to solve a wide range of problems in computer vision, such as image classification, segmentation, etc. According to [12], they can handle time-series data too. Can we build a CNN model, especially a simple one, that achieves good results on the SLR problem? This section presents a simple three-layer CNN and evaluates how it performs on our dataset compared to baseline ML models. We target a simple deep learning CNN model to keep complexity low so that it can run in real-time on an embedded system (see chapter 5).

4.4.1 Model architecture

To build the model, we will use of the *Keras* framework [70] [71] and follow the multi-class method described on section 2.5.2. So, we define a sequential model as shown in the next diagram (Figure 20) and its equivalent code (Figure 21):

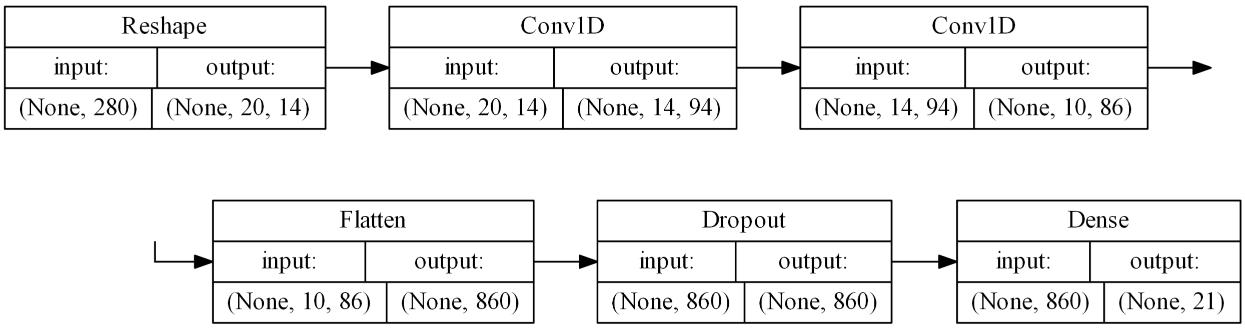
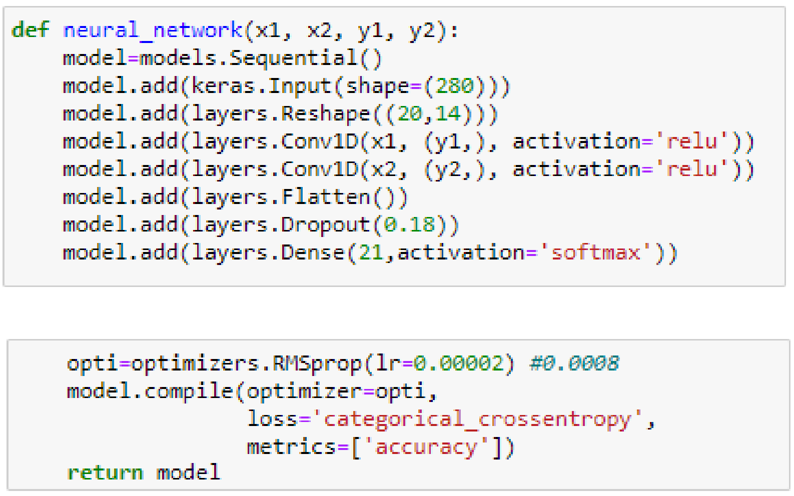


Figure 20: The CNN model architecture - Bayesian optimization has been applied. Diagram exported from *Keras*.

At first sight, it does not seem to be a three-layer model as we mentioned before. However, suppose we ignore the dropout layer, which is used to prevent overfitting only at training time, and the flatten one, which is used to adjust dimensions and make the connection between 2nd conv1d and dense layer compatible without any extra cost, we actually have a conv1d-conv1d-dense model (lines 5, 6 and 9 of the code in Figure 21). The model's code consists of two parts: *feed-forward model* and *optimizer-loss function*.

The whole model is created as a python function which is necessary to convert it into an ATOM model later on in the next section. Arguments x1, y1, x2, y2 are its hyper-parameters need to be tuned. As in previous ML methods, we are going to use the ATOM tool to achieve highest accuracy. In line 2, we define our model as a sequential one and in line 3 we set its input to be a sample of 280 elements/features (see Figure 13). In line 4, we reshape the sample into its original dimensions 20x14 (see step 1 in section 4.2).

The first crucial layer, in line 5, is a convolutional layer with hyper parameters: *filters = x1*, *kernel size = y1*, followed by a non linear activation "relu". This means that the current layer



```
def neural_network(x1, x2, y1, y2):
    model=models.Sequential()
    model.add(keras.Input(shape=(280)))
    model.add(layers.Reshape((20,14)))
    model.add(layers.Conv1D(x1, (y1,), activation='relu'))
    model.add(layers.Conv1D(x2, (y2,), activation='relu'))
    model.add(layers.Flatten())
    model.add(layers.Dropout(0.18))
    model.add(layers.Dense(21,activation='softmax'))
```

```
    opti=optimizers.RMSprop(lr=0.00002) #0.0008
    model.compile(optimizer=opti,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Figure 21: Sequential model on keras.

learns patterns of y1 timesteps from incoming signal in x1 different ways. For example, if *filter* = *94* and *kernel size* = *7* then it will extract $20 - 7 + 1 = 14$ patches of 8 timesteps for each filter as show at Figure 22 below:

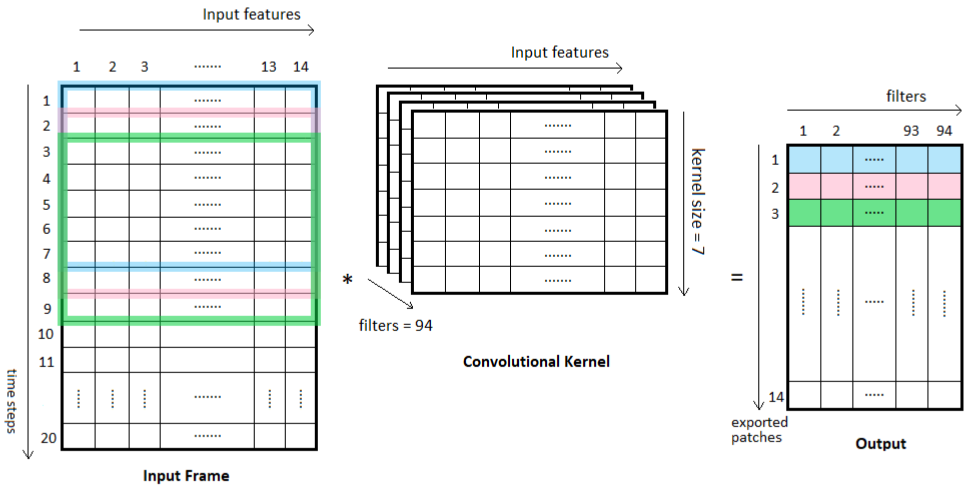


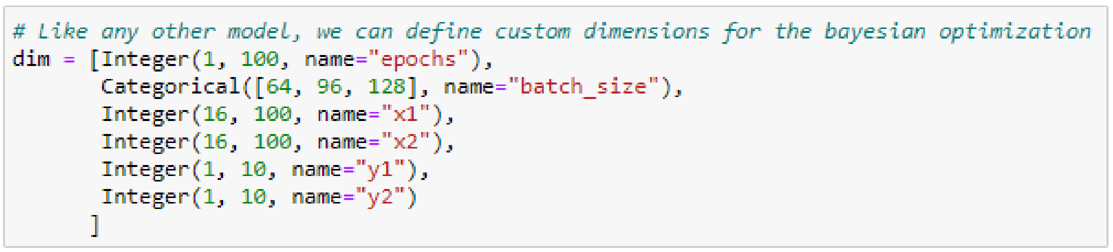
Figure 22: Example of an 7-timestep feature extraction for 94 filters.

In line 6, the second critical layer is also 1d convolutional layer, with *filters* = *x2*, *kernel size* = *y2*, which works the same way (extracting patches) and finally, through flatten and drop out layers, we end up with a dense layer with softmax activation, which takes care of the classification by matching input samples onto corresponding one-hot encoded labels. Values for dropout layer (:= 0.18) and optimizer's learning rate (:= 0.00002) are selected through some manual tests. The type of optimizer ("RMSprop") and loss function ("categorical_crossentropy") are defined in section 2.5.2.

Even though our model responded with high accuracy on the dataset, for different values of its hyper-parameters by setting them up manually, nevertheless we applied Bayesian optimization for more reliability. For this, we define the search space of these hyper-parameters and let ATOM choose which ones are the best.

4.4.2 CNN Hyperparameter tuning

As any other neural network tuning tool, ATOM can handle different formats for variables. In our case we use two of them: *Integer* and *Categorical*. The integer format is followed by a range where a variable can take (integer) values, and the categorical one can take any discrete value from a predefined set. The Figure below presents corresponding lines of code. Except from variables x1, y1, x2, y2 there are also two extra variables, *epochs* and *batch_size* which are hyper-parameters of the training algorithm and not model parameters (see section 2.2). To reach an efficient training of the model, we have to tune them too.



```
# Like any other model, we can define custom dimensions for the bayesian optimization
dim = [Integer(1, 100, name="epochs"),
       Categorical([64, 96, 128], name="batch_size"),
       Integer(16, 100, name="x1"),
       Integer(16, 100, name="x2"),
       Integer(1, 10, name="y1"),
       Integer(1, 10, name="y2")
      ]
```

Figure 23: Hyper-parameter search space.

Here we have to pay attention on kernel size search spaces (variables y1 and y2) which are dependent to each other. For input sample 20x14, y1 determines the first dimension of the output of the first convolution layer to be 20 - y1 + 1. Subsequently, y2 determines the first dimension of the output of the second convolution layer to be (20 - y1 + 1) - y2 + 1 = 22 - (y1 + y2). To ensure that the final dimension exists, y1 + y2 must be lower than 22. Hence, we set the same range for each one to be equal to (1, 10). For the other four hyper-parameters, range values are chosen through some tests.

4.4.3 NN model on ATOM

Running a neural network on ATOM is similar to other ML methods, with one exception. Neural networks are not included in standard ATOM's model library, and since ATOM uses the sci-kit learn API, we can use Keras' wrapper to run them.

Let us start python code using ATOM. Implementation includes one extra step than in the previous section:

1. Apply steps 1 and 2 (see 4.2) to reshape the dataset and create an ATOM classifier object. Reshaping dataset from three to two dimensions is not necessary when

using NNs. The reason we do this is for being similar to the previous section. To invert it, there is an extra reshape layer on our sequential model.

2. Create an ATOM neural network model. At line 1, Keras' sequential model (Figure 20), from the defined "neural_network" function, is converted into sklearn's model, and at line 2, the latter is converted into Atom's one.

Step 3

```
   # Since ATOM uses sklearn's API, use Keras' wrapper
1  model = KerasClassifier(neural_network, verbose=2)

   # Convert the model to an ATOM model
2  model = ATOMModel(model,
                     acronym="NN",
                     fullname="Neural network")
```

3. Running Bayesian Optimization on our NN model and trying to maximize accuracy, which is the selected metric, under the defined searching space.

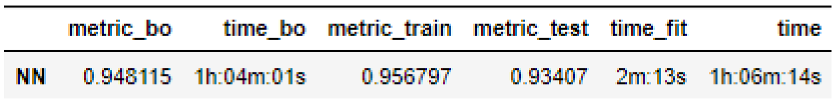
Step 4

```
atom.run(
        models=model,
        metric="accuracy",
        n_calls=25,
        n_initial_points=10,
        bo_params="dimensions": dim, "early_stopping": 0.1, "cv":5,
)
```

Parameter *n_bootstrap* which was used before, is not compatible with Keras' NN, so it is not included. A sorts of manual bootstrapping, which is not presented in this thesis, was applied to the exported best model giving a stable accuracy on five new random datasets created from the original sampling (with replacement, see Figure 15). Search space is passing into bo_params through the "dimensions" element.

4.5 Results

Model tuning took about an hour to complete and best model exported with training parameters: 'epochs': 100, 'batch_size': 128 and model's hyper-parameters: 'x1': 94, 'x2': 86, 'y1': 7, 'y2': 5 (see Figure 21). By calling ATOM's *result* method as before, we obtain the above results in Figure 24.

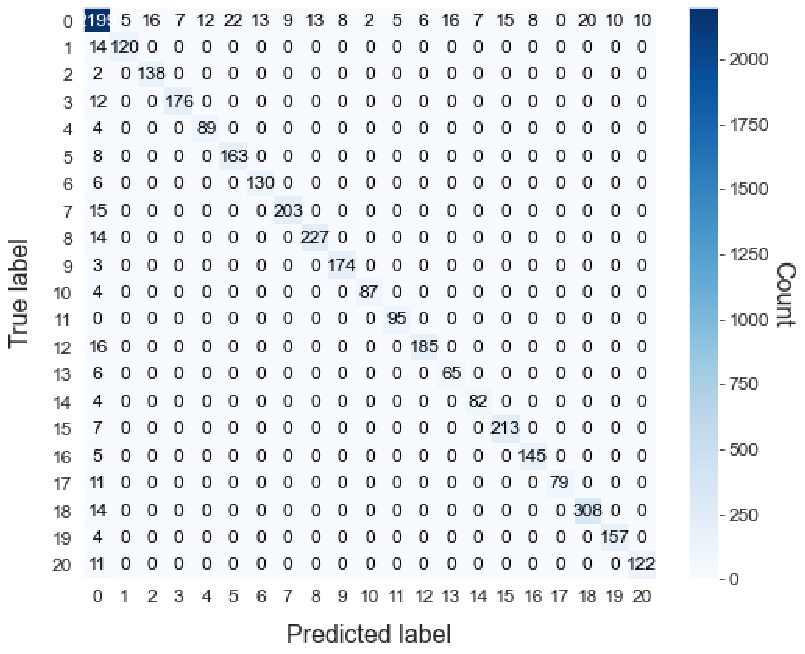


	metric_bo	time_bo	metric_train	metric_test	time_fit	time
NN	0.948115	1h:04m:01s	0.956797	0.93407	2m:13s	1h:06m:14s

Figure 24: ATOMClassifier results on NN.

As we can see, our neural network achieves 93.40% accuracy on the test set, which is slightly higher than the XGBoost model's accuracy (93.15%). At first sight, our simple CNN is a close winner on the Sign Language Recognition problem. What about the remaining 6.60% of wrong predictions? As we mentioned before, XGB did not confuse classes 1 to 21 together. Wrong predictions appear only between transition class and one other while entering or/and coming out a gesture. For being more accurate, we have to check if this is true for the CNN model.

Figure 24 shows the corresponding confusion matrix, in which every element is equal to zero except elements on first row, first column and main diagonal. As described at section 4.3 for XGB model, our continuous SLR method using a simple neural network is not mixing up gestures with each other and so we can accept that real_accuracy is 100%.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	219	5	16	7	12	22	13	9	13	8	2	5	6	16	7	15	8	0	20	10	10
1	14	120	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	0	138	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	12	0	0	176	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	4	0	0	0	89	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	8	0	0	0	0	163	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	6	0	0	0	0	0	130	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	15	0	0	0	0	0	0	203	0	0	0	0	0	0	0	0	0	0	0	0	0
8	14	0	0	0	0	0	0	0	227	0	0	0	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	174	0	0	0	0	0	0	0	0	0	0	0
10	4	0	0	0	0	0	0	0	0	0	87	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	95	0	0	0	0	0	0	0	0	0
12	16	0	0	0	0	0	0	0	0	0	0	0	185	0	0	0	0	0	0	0	0
13	6	0	0	0	0	0	0	0	0	0	0	0	0	65	0	0	0	0	0	0	0
14	4	0	0	0	0	0	0	0	0	0	0	0	0	0	82	0	0	0	0	0	0
15	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	213	0	0	0	0	0
16	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	145	0	0	0	0
17	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	79	0	0	0
18	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	308	0	0
19	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	157	0
20	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	122

Figure 25: CNN confusion matrix.

4.6 Conclusions

ATOM is a very useful tool to run and automate fine-tuning ML models, including deep learning models, easy and fast, without having to worry about the large set of their hyper-parameters. Applying Bayesian optimization to determine these hyper-parameters, we conclude that the proposed CNN achieves 93.40% accuracy on the test set (real accuracy 100% on gesture recognition), which is near but a bit higher than all the other tested baseline ML models. In contrast to accuracy, our model has a training time of about 1 minute higher than the powerful XGBoost. The sample rate of data acquisition is set at 30Hz to keep the model's input at a small size. In the next chapter we will program an ARM processor to run a copy of the current CNN architecture and predict gestures in real.

5. AN ARM-BASED IMPLEMENTATION

In this chapter, we present an all-software real-time application for continuous SLR on the ARM Cortex A9 processor via the Zybo z7-10 development board. We also present a simple approach to handle linear algebra operations on ARM, which are necessary for implementing neural networks.

5.1 Experimental System Overview

Even though an SLR system has to be portable for comfortable use, here we use a PC to transfer data from the input device to the Zybo board [10]. That makes communication between the Arduino and the ARM processor easier (for testing purposes), without additional hardware, due to the different logic ”HIGH” voltage they use. Arduino’s I/O pins rely on a 5V logic level, but Zybo’s rely on either 1.8V, 2.5V, or 3.3V, so we do not want to connect ports directly together to avoid any damage. In the following figure, we can see a diagram of our experimental system.

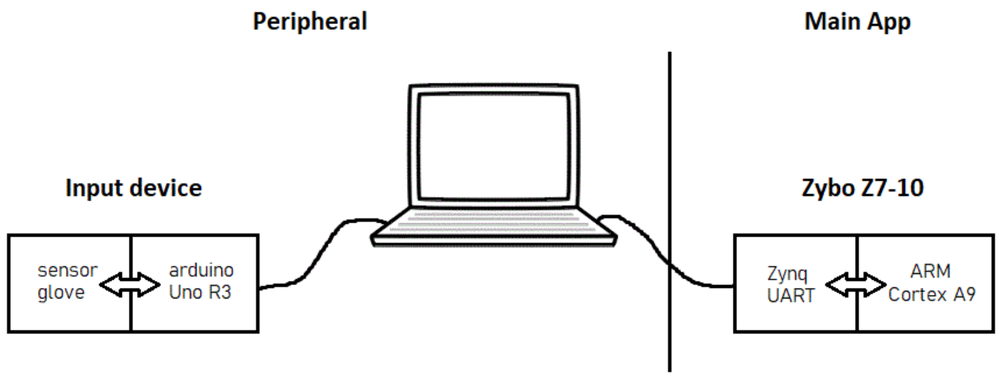


Figure 26: System overview and bi-directional communication. The Zybo board holds the whole application while the input device with the PC work as a peripheral for input purposes.

The ARM processor runs the main application. It is a wide loop that waits for a character from the keyboard to perform a predefined task. This makes the application more dynamic and easily configurable. Below we can see a description of our code and related tasks. Results and messages are printed on the display.

Task 1: Assign zeros to the model’s weights/biases, to scaler’s and input sample’s values(it is used only for testing/debugging purposes).

Task 2: Transfer the model’s weights/biases and scaler’s values from binary file to Zybo’s DDR (first we train our model in a PC using Keras with TensorFlow-GPU backend).

Application Program Structure

```
1:  #define myNetwork
2:  procedure main ()
3:      while (True)
4:              Print_Message()
5:              ch ← keyboard.read()
6:          if (ch==’1’) then
7:              Zero_Initialize_CNNweights()                    Task 1
8:          else if (ch==’2’) then
9:              Load_CNNweights_from_Binary_File()              Task 2
10:         else if (ch==’3’) then
11:             Load_Testset_Samples_From_Binary_File()         Task 3
12:         else if (ch==’4’) then
13:             Enable_Aqcuisition_Mode()                       Task 4
14:         else if (ch==’5’) then
15:             Run_Inference_on_Testset_or_AcquisitionData()   Task 5
16:         else if (ch==’6’) then
17:             exit()
18:         else
19:             print(”Invalid character”)
20:     loop

21:     return
```

Task 3: After our model is loaded (Task 2), then we transfer sample(s) from the test set that we want to classify (inference). Values are being standardized during this task.

Task 4: Enable acquisition mode for real time inference.

Task 5: Start inference on loaded samples (Task 3) and output relative predictions. If Task 4 occurs, then an acquisition mode is running in which data comes directly from the sensor glove. This task holds the main processing core of the CNN.

In addition to the above Task 5 when running in acquisition mode, it is necessary to describe the whole process in more detail. Let us take a look at Figure 10, which shows a data sample to be processed. The Input device sends these values to the PC, one by one, in row-major order. At the same time, the PC is running a python script that reads incoming values via a serial communication protocol (UART) and organizes them in an array with 14 elements (as many as the number of sensors). Each time the array is filled up, the PC sends it to Zybo for further processing (calculations of our model), and the process is repeated again. It is necessary to tell that both Uart communications, the first between the input device and the PC and the second between the PC and the Zybo, are synchronized to ensure a proper data flow through UART FIFOs and avoid losing or stacking any data. How do we implement the above program, and what tools we used?

5.2 Software and Tools Used

When buying a new Zybo z7-10 development board [10], it comes with the Vivado Suite and Xilinx SDK, so these are the two main software tools we use. Another significant dilemma we come across during our research is what kind of application to create. A Linux application or a bare-metal one? Because of our interest in the architecture of Zybo's hardware (FPGA and ARM), a good choice was a bare-metal app. So C++ was used as the programming language.

As we saw in section 2.3, a feed-forward CNN is a series of multiplications and additions/subtractions, either matrix multiplications or matrix element-wise multiplications. To perform these matrix operations, we can use ”pure” C++ code such as the ”for...loop” control structure. Is that fast enough for ARM to complete a feed-forward pass of our model under 300ms, which is the upper limit of a real-time app, or is there a better solution? After some investigation, we found Eigen [72], a C++ library for implementing mathematical operations, like NumPy in python, which is faster and easier to use than pure C++ code.

What makes Eigen so fast is that it takes advantage of ARM's architecture, specifically of a block called Neon Engine [73]. ARM Neon technology is an advanced Single Instruction Multiple Data (SIMD) architecture, which performs multiple-element operations in parallel. According to its official website [73], Neon can accelerate signal processing algorithms to speed up applications such as audio and video processing, voice and facial recognition, computer vision, and deep learning (which is our goal). How to use Neon technology and achieve parallel processing? There are several ways, namely: Neon intrinsics which are function calls that the compiler replaces with an appropriate Neon instruction or sequence of Neon instructions to directly ”talk” to the Neon Engine, Neon-enabled libraries e.g. Ne10, Libyuv, Skia and ARM Compute Library, Auto-vectorization by the compiler e.g. GCC, where it can automatically analyze our code and identify opportunities to optimize performance with Neon, and finally Hand-coded Neon assembler, a low-level programming method with very high performance. As Eigen is a free open source software which makes use of NEON and supports NEON instructions (it has its own vectorization system which is enhanced by the compiler [72]), we chose the first method that does not require any further low-level knowledge about ARM's SIMD arhitecture and let the compiler do its job.

Another thing to discuss is the communication between the ARM processor and the serial port driver chip on the Zybo development board. The Xilinx SDK takes care of this by offering a built-in library with the necessary serial port functions that allows writing our bare metal application. In Figure 27 we can see the Xilinx chip with Neon (SIMD) Engine and UART I/O driver. The Processing System (PS), also contains some extra drivers/MIO interfaces, like SPI, I2C and GPIO which can be used for direct connection between the sensor glove and Zybo without using Arduino and the PC to read and send data. This was discovered during elaboration of this thesis and is highlighted for future work.

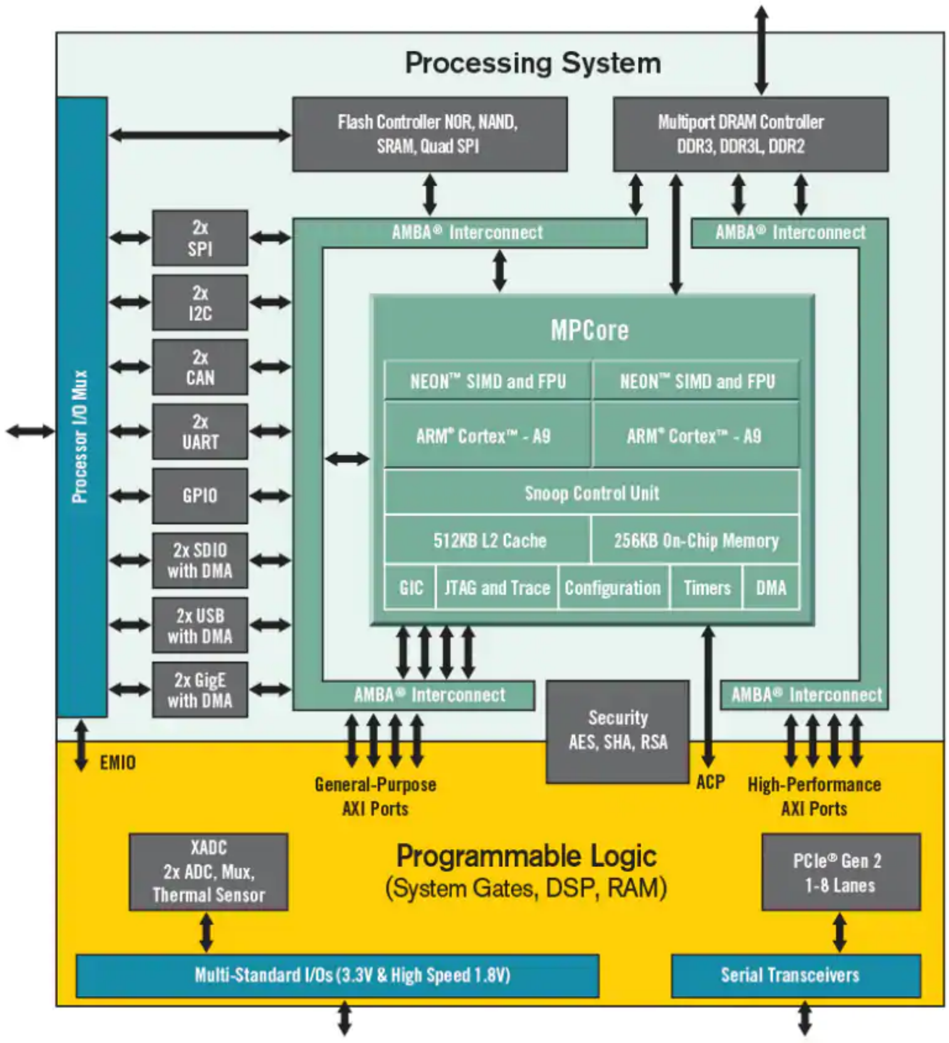


Figure 27: The Zynq Processing System (adopted from [10]).

5.3 Define the CNN using Eigen

Once we have selected all the tools we will use, its time to implement our proposed CNN (see section 4.4) on the ARM processor. Firstly, we have to allocate all necessary space, in Zynq's DDR memory, to hold parameters and other useful values for computing a feed-forward pass. This space, as shown in Figure 28, consist of the following arrays/matrices:

- Convolutional kernels (weights and biases) for each conv1d layer
- Dense layer's weights and biases
- Input sample matrix
- Input/Output of each layer.

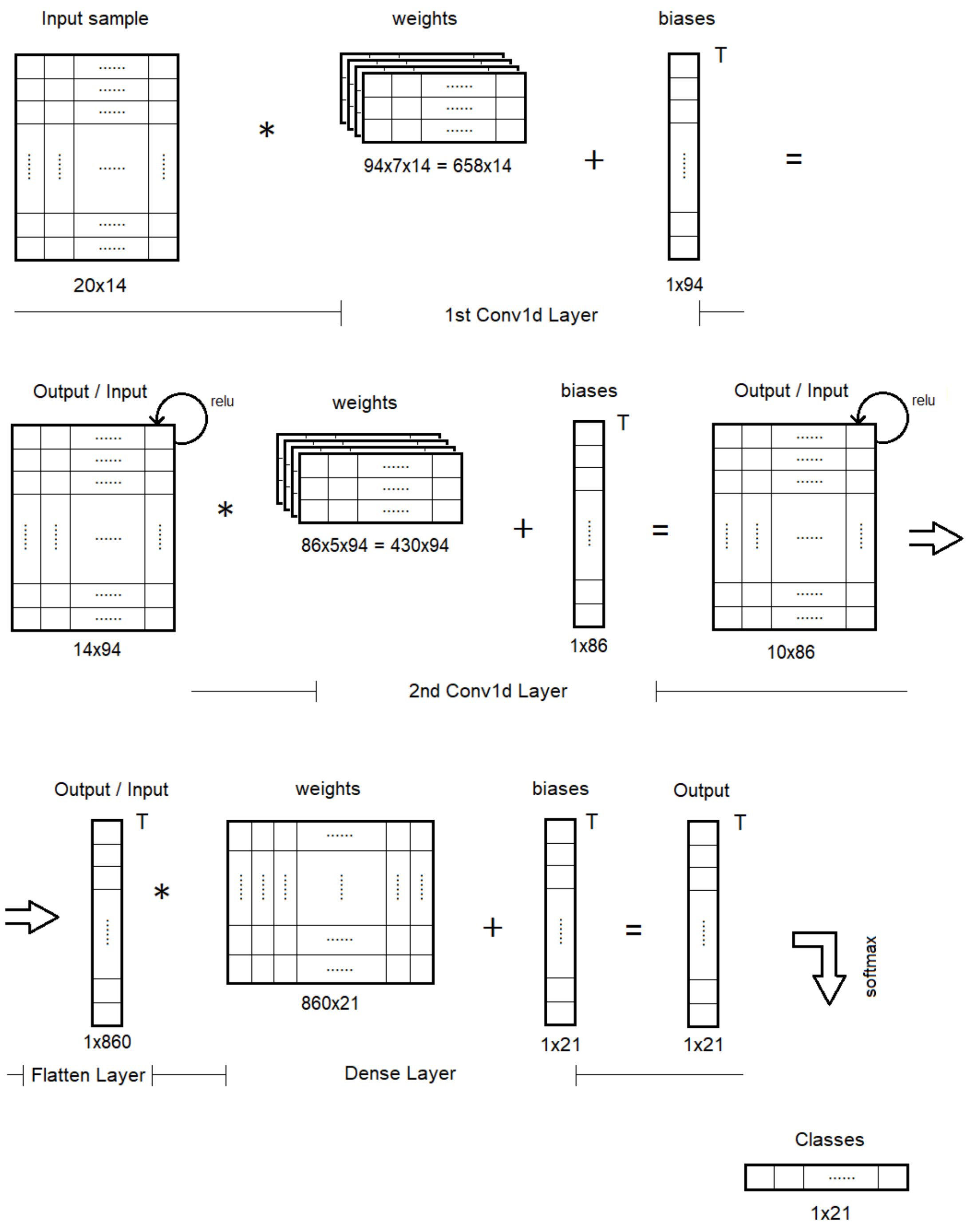


Figure 28: All necessary matrices we declare in order to run a feed forward pass. Hyper-parameters: x1=94 filters, y1=7 kernel_size, x2=86 filters, y2=5 kernel_size resulting from BO in section 4.5 according to our model architecture in section 4.4.1. Operator T gives the transpose of a matrix.

Lets describe in more detail, what these matrices represent and how they are linked with each layer of our purposed CNN:

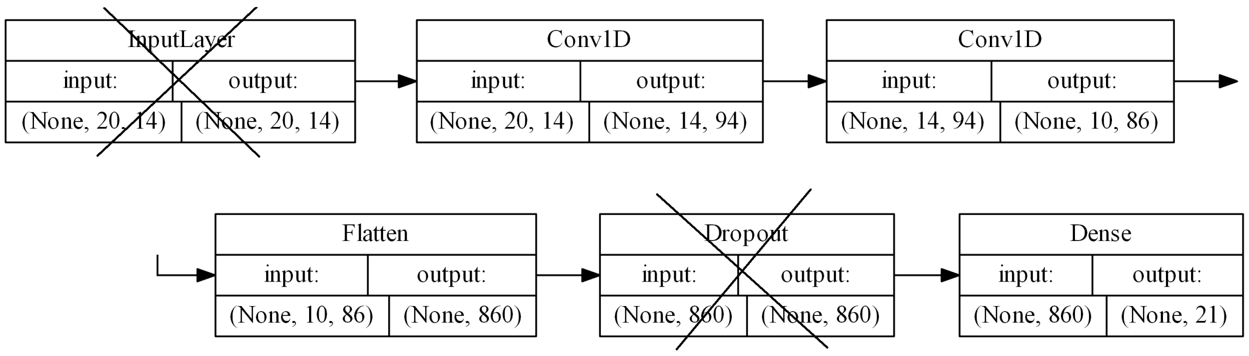


Figure 29: Keras model presented in chapter 4. Deleted layers are not used on inference.

Firstly, the input of our network has **20x14** neurons, so we need a same size matrix, **Input sample**, to store the current sample. As shown in Figure 4, rows represent the timesteps and columns hold sensors' values for each timestep.

Secondly, in Keras to hold its parameters, an 1D convolutional layer has a *kernel matrix* and a *bias vector*. As shown in Figure 4, kernel is a 3-dimensions matrix and its size depends on the layer's hyper-parameters: filters, kernel_size and also the size of the second dimension (number of columns) of layer's input. In our case, the first conv1d layer needs a kernel matrix of 94x7x14 to store its weights. Some attention is needed at this point. As Eigen can handle matrix operations up to 2-dimensions, we have to reshape kernel matrix by reducing one dimension, as shown in Figure 29. Thus, the kernel/**weight** matrix for the 1st conv1d layer is transformed into **658x14** (94*7x14).

To calculate the size of the bias vector we need to apply first the result of the first multiplication's operator which outputs the convolution product between input sample and kernel matrix. The output, as described in section 2.3.1, is a 2-dimensional matrix and its size depends on the size of the first dimension (number of rows) of layer's input, and on layer's hyper-parameters: filters, kernel_size. In our case, the first conv1d layer needs an **output** matrix of **14x94** (20-7+1x94).

Finally, the **bias** is a 94 element vector (**1x94**) which is added to each row of the previous output matrix. Relu activation function needs no extra space because it is a wise element function that stores results into the original matrix.

To calculate corresponding matrices for the second conv1d layer we follow the same method. Now the input matrix is identified with the previous output one of size 14x94. Subsequently, the kernel/weight matrix has 3-dimensions of size 86x5x94 (filter x kernel_size x 94) which is transformed into 430x94 (86*5x94). The convolution output between input and kernel matrices is a 10x86 (14-5+1x86) matrix and the bias an 86 element vector (1x86). Like before Relu activation function does not need extra space.

The next layer of our sequential model is a **flatten** one that reshapes the previous output matrix (10x86) to vector with 860 elements (**1x860**). These 860 elements are the input neurons of the dense layer. Flattening does not need extra space (and time), because Eigen creates a map to the memory that can read its data in a specific order. Similar to 1D convolutional layer, dense layer's parameters are stored in *kernel matrix* and a *bias vector*. In our case, the kernel/**weight** matrix has 2-dimensions of size **860x21** (input_neurons x classes) and the **bias** a 21 element vector (**1x21**). Dense layer implements a matrix multiplication (see section 2.3.2), hence the **output** is an **1x21** vector and then is added with bias. Softmax activation function needs no extra space; it uses the last output 21-element vector to store predictions.

The total memory of the purposed CNN model is shown in the next Table 3.

Table 3: CNN Memory. The system performs on a Single-precision Floating-point format (FP32).

Required Memory			
matrix	weights (bytes)	bias (bytes)	Overall (bytes)
Input Sample	1,120	-	1,120
Conv1	36,848	376	37,224
Out1	5,264	-	5,264
Conv2	161,680	344	162,024
Out2	3,440	-	3,440
Dense4	82,560	84	82,644
Output	84	-	84
	Total		291,800

Finally, in this method, we define the size of each matrix in the preamble of our code (see Appendix C), so it is a static method and takes place on compiling. By changing hyper-parameters manually, we can implement a set of different CNN models while keeping the same architecture (conv1d/relu - conv1d/relu - flatten - dense/softmax). Future interesting work is to create a CNN generator for the Eigen library, to be able to define every possible network.

To sum up we present the relative part of the code with all declared matrices using Eigen's class Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>:

```
//====================Allocate memory for CNN matrices==================//
Matrix<float, In_shape_i, In_shape_j> Input;                           //
                                                                       //
Matrix<float, conv1_shape_i, conv1_shape_j> conv1;                     //
Matrix<float, 1, bias1_len> bias1;                                     //
Matrix<float, Out1_shape_i, Out1_shape_j> Out1;                        //
                                                                       //
Matrix<float, conv2_shape_i, conv2_shape_j> conv2;                     //
Matrix<float, 1, bias2_len> bias2;                                     //
Matrix<float, Out2_shape_i, Out2_shape_j, RowMajor> Out2;              //
```

```
                                                                             //
  //Create a map to the memory that can read Out2 matrix as flatten          //
  Map<Matrix<float, 1, dense4_shape_i>> Out3(Out2.data(), Out2.size());      //
                                                                             //
  Matrix<float, dense4_shape_i, dense4_shape_j> dense4;                      //
  Matrix<float, 1, bias4_len> bias4;                                         //
                                                                             //
  Matrix<float, 1, classes> Output;                                          //
                                                                             //
  Matrix<float, 1, scaler_mean_len> Scaler_mean;                             //
  Matrix<float, 1, scaler_std_len> Scaler_std;                               //
                                                                             //
  //This matrix stores testset's samples for inference                        //
  Matrix<float, samples_num*conv1_shape_i, conv1_shape_j> DataSetMatrix;     //
  //=========================================================================//
```

Listing 5.1: Application Program - Task 2

In the above code there are two more matrices 1x14 to store scaler's values (mean and std), and another one to store a fraction of the test set (samples_num*20x14). These matrices do not belong to the CNN but are part of the whole program. By defining samples_num equal to 50 an extra memory space is 56,112 bytes.

5.4 Model transfer

After training the model on the PC using Keras, we export the trained parameters in binary format. Once the model is already defined on ARM in the previous section, it is time to fill the necessary matrices with trained values (weights and biases). Task 2 is responsible for this action. When bytes are available in ARM's UART RX FIFO, the program reads and stores them one by one in a byte array *buffer*. Then we cast the buffer to a 32-bit float variable **tmp** and store it in the matrix. Both systems, Keras and Eigen work with 32-bit float numbers.

```
else if(c=='2'){ //Task 2 - Initialize CNN matrices from file - Transfer CNN
  //==========================START COEFF FROM UART======================//
  // This part transfers the trained CNN from PC to ARM byte by byte from
  // file.bin. Values for each matrix are 32-bit float numbers.
  print("\nSend a .bin file\n\r");
  InitMatrixFromFile(conv1);          //Read 36,848 bytes for conv1 weights.
  InitMatrixFromFile(bias1);          //Read 376 bytes for conv1 biases.
  print("Conv1 weights and biases loaded...ok\n\r");
  InitMatrixFromFile(conv2);          //Read 161,680 bytes for conv2 weights.
  InitMatrixFromFile(bias2);          //Read 344 bytes for conv2 biases.
  print("Conv2 weights and biases loaded...ok\n\r");
  InitMatrixFromFile(dense4);         //Read 72,240 bytes for dense weights.
  InitMatrixFromFile(bias4);          //Read 84 bytes for dense biases.
  print("Dense weights and biases loaded...ok\n\r");
  InitMatrixFromFile(Scaler_mean);    //Read 56 bytes for scaler mean.
  InitMatrixFromFile(Scaler_std);     //Read 56 bytes for scaler std.
  print("Scaler mean and std values loaded...ok\n\r");
  //========================END COEFF FROM UART=========================//
}
```

Listing 5.2: Application Program - Task 2

The domain function in Task 2 is the custom *InitMatrixFromFile()* template function as presented below:

```
/*  InitMatrixFromFile()
 *  This function reads bytes from UART and cast them as a 32-bit float
 *  in order to fill an Eigen Matrix.
 *  Argument: An Eigen Matrix.
*/
template <typename M>
void InitMatrixFromFile(DenseBase<M>& A){
  u8 buffer[4];
  float tmp;

  //loop through each matrix's element
  for (int i = 0; i < A.rows(); i++) {
    for (int j = 0; j < A.cols(); j++) {
      //ARM reads 4 bytes from UART's FIFO
      buffer[0]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      buffer[1]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      buffer[2]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      buffer[3]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);

      //cast buffer[4] to a 32-bit float number
      memcpy(&tmp, &buffer, sizeof(float));

      A(i,j)=tmp; //copy tmp to matrix
    }
  }
}
```

Listing 5.3: InitMatrixFromFile() function

5.5 Testset Loaded

As previously, Task 3 applies the *InitMatrixFromFile()* function to transfer a predefined number of samples into ARM's DDR memory. This number is defined in the preamble of the program. As samples are loaded successfully they need to be standardized before feeding the model. *Substraction* and *cwiseQuotient* are element-wise operations between Eigen arrays.

```
else if(c=='3'){//Task 3 - Load testset from file ==========================//
                                                                            //
  isRealTimeEnabled = false;          //realtime is disabled by default------//
  print("\nSend a .bin file\n");                                             //
                                                                            //
  InitMatrixFromFile(DataSetMatrix); //load dataset from uart               //
                                                                            //
  //----------Standardize values row by row using mean and std--------------//
  for(int i=0; i<samples_num*In_shape_i; i++){                               //
    DataSetMatrix.row(i) = (DataSetMatrix.row(i)-Scaler_mean).cwiseQuotient(
   Scaler_std);
  }                                                                         //
}                                                                           //
else if(c=='4'){//Task 4 - Enables realtime acquisition                     //
  isRealTimeEnabled = true;                                                 //
}                                                                           //
//==========================================================================//
```

Listing 5.4: Application Program - Task 3 and Task 4

Task 4 is only used to enable real-time acquisition instead of using test set in Task 5.

5.6 Class prediction using Eigen

In this section, we describe step by step the procedure of a feed-forward pass and present our C++ code. Task 5 is responsible for this action and is based on Eigen block operations and element-wise operations, which are so simple to work with. The whole task is separated in two (almost same) parts; one for disabled real time acquisition (see Appendix C) and the other for enabled.

At first, the input matrix (20x14) with row index count from 0 to 19 is empty. In each loop, a block starting from 1 to 19 is copied to equal size block 0 to 18, and new 14 sensor values fill the last row of the input matrix one by one. That creates a continuous data flow which constitutes our sliding window. Before heading to the second step, we have to standardize incoming raw data, so we subtract *mean* from row 19 and then divide by *standard deviation*. Both of them are element-wise operations.

```
 else if(c=='5'){//Task 5 - Inference either on real time samples or test
set
   if (isRealTimeEnabled == false){
                 ......                     //See Appendix C
   }
   else{//Real-time acqusision enabled
     print("Open UART\n\r");
     //---Clear ARM's UART RX FIFO by reading available bytes---//
     XUartPs_RecvByte(UART_MEM_BASE_ADDR);                      //
     sleep(3);                                                  //
     while(XUartPs_IsReceiveData(UART_MEM_BASE_ADDR)){          //
```

```
      XUartPs_RecvByte(UART_MEM_BASE_ADDR);                        //
    }                                                              //
    print("waiting bytes\n");                                      //
    //-------------------------------------------------------------//
    while (1){//Sample processing and predict loop---------------------//
                                                                       //
      //Start counter to compute running time                          //
      XTime_GetTime((XTime *) &tStart);                                //
                                                                       //
      //Sliding window moving - Drop first row                         //
      Input.block<In_shape_i-1, In_shape_j>(0, 0)=Input.block<In_shape_i
-1, In_shape_j>(1, 0);
                                                                       //
      //read 14(sensor values)*4-bytes from FIFO                       //
      for(int i=0; i<buffer_size; i++){                                //
        *(buffer+i)=XUartPs_RecvByte(UART_MEM_BASE_ADDR);              //
      }                                                                //
                                                                       //
      //cast bytes to 14 32-bit float numbers                          //
      memcpy(&income_line, buffer, In_shape_j*sizeof(float));          //
                                                                       //
      //Fill last row with new 14 sensor values                        //
      Input.row(In_shape_i-1) = Input_c;                               //
                                                                       //
      //Standardize last row of the matrix                             //
      Input.row(In_shape_i-1) = (Input.row(In_shape_i-1)-Scaler_mean).
cwiseQuotient(Scaler_std);
    //-----------------------------------------------------------------//
      //=======================Start Inference======================//
      //-------------First convolutional layer-----------//
      for (int j = 0; j < conv1_filters; j++)
        for (int i = 0; i < Out1_shape_i; i++)
          Out1(i,j) = Input.block<conv1_kernel, In_shape_j>(i, 0).
cwiseProduct(conv1.block<conv1_kernel, conv1_shape_j>(j * conv1_kernel,0))
.sum() + bias1(j);

      //relu activation function
      Out1 = Out1.cwiseMax(0);

      //-------------Second convolutional layer----------//
      for (int j = 0; j < conv2_filters; j++)
        for (int i = 0; i < Out2_shape_i; i++)
          Out2(i, j) = Out1.block<conv2_kernel, Out1_shape_j>(i, 0).
cwiseProduct(conv2.block<conv2_kernel, conv2_shape_j>(j * conv2_kernel, 0)
).sum() + bias2(j);

      //relu activation function
      Out2 = Out2.cwiseMax(0);

      //-------------Flatten layer-----------------------//
      //Map has already done when allocate memory. Outputs Out3 matrix
      //-------------Fully connected layer---------------//
      Output = Out3 * dense4 + bias4;
```

```
            //softmax activation function
            Output = Output.array().exp();
            sum = Output.sum();
            Output=Output/sum;

            max = Output.maxCoeff(&maxCol);

            //Print current prediction class
            cout << "class: " <<  maxCol << "\tpossibility: " << max << endl;

            //End counter, compute usec and print time to terminal
            XTime_GetTime((XTime *) &tEnd);
            f=1.0*(tEnd - tStart) / (COUNTS_PER_SECOND/1000000);
            cout << "Completed\n\rIt took: "<< f << "us"<< endl;
            //===============================================================//
        }
    }
}
```

Listing 5.5: Application Program - Task 5

Second, we apply the conv1d layer as described in Figure 2, add biases, and store results in the corresponding pre-defined Output matrix. Then relu activation function, figure 5a, applied to itself and the process repeated for the 2nd convd1d layer and its activation function. Also, to prepare matrix dimensions for the next dense layer, a flatten one is necessary. Finally, a simple matrix multiplication takes place (biases added) and softmax (see equation 2.4) computes the output probabilities. The maximum one is the predicted class.

ARM's results, output probabilities, and classes for every sample of the test set are the same as those when running the model on PC with python. So we know that everything works perfectly. The sample processing time is in the order of a few milliseconds ($\sim$ 3.56 msec), a value which is near nine times smaller than the sampling period of the system (33 msec). That leads us to a real-time system.

5.7 Conclusion

In this Chapter, we singled out tools and libraries that can easily and effectively handle matrix operations on ARM. We analyse the memory we allocate to store our purposed CNN ($\sim$ 300 Kbytes), so embedded device that do not meet this requirement e.g. arduino UNO, MEGA, even DUE, cannot run it. Also we exploit ARM's NEON Engine using a NEON enabled library, Eigen and achieve to speed up inference to 3.56 msec per sample. Consequently, micro-controllers running at frequencies under 66Mhz even if they can handle parallel processing, like NEON Engine, would not achieve real-time inference. Finally, we give a simple idea on how to define and run Neural Networks on a bare-metal application using Xillinx SDK.

6. CONCLUSIONS AND FUTURE WORK

In this thesis, we presented a sensor-based approach, via flex sensors and an IMU device, for continuous American SLR. We validated it for a 20-word vocabulary of the most common words and few letters of the American alphabet. To collect data we constructed a custom sensor glove instead of using a commercial one and achieved to create a stable acquisition device simply by placing sensors on a common rubber glove and reading values using an Arduino UNO board. Raw data were captured with a sample frequency about 30Hz, pre-labeled in a continual way, and pre-processed in terms of standardizing sensors readings, individually for each sensor data type. A sliding window method was adopted to export samples, time series frames of 20 data points length, from the overall signal with a window step of 1 and assign labels by calculating the maximum percentage of pre-labels in the current frame.

The classification was performed by applying LDA, LR, RF, LSVM and XGB Machine Learning models using ATOM tool. We explained how to create an ATOM classifier and run Bayesian Optimization to tune the hyper-parameters of these models in order to maximize the accuracy metric. The procedure was simple enough because ATOM can handle the whole process in an automated way. The results indicate that all models perform very well on the dataset with XGB being the leader reaching an accuracy of 93,15%, followed by the RF model with accuracy 92,86%. Analysing the errors we can say that the real accuracy is actually 100% because the samples that were not predicted correctly do not affect the way we address the SLR problem.

In addition to the above ML methods, we have purposed a simple three-layer CNN deep learning model to address the continuous SLR problem. Two 1D convolutional layers are used to extract features from preprocessed data while a fully connected layer at the end takes care of the classification. The given results, using the ATOM tool, are similar to the XGB and RF baseline models with an accuracy reaching 93,40% and real accuracy of 100%.

Another challenge we faced on in this thesis was to design an experimental end-to-end system and test in real-time. An ARM-based solution was developed using the Zybo z7-10 development board in which we have managed to develop a bare-metal application that can efficiently run our purposed CNN model. Our software was designed to be generic, in order to run multiple instances of the same three-layer architecture and also to take advantage of the ARM's NEON Engine for faster calculations. The processing time of our purposed model is $\sim$3.56 msec according to the sampling period which is about 33 msec, so our system can be considered as a real-time one.

Since the time of this thesis was limited, we quote some short extensions that can improve our work. First, for better results we need to calibrate the sensor glove and especially the IMU device. Calibration can prevent errors that arise from deterioration of the micro mechanical part of the IMU that are caused during its lifetime. Another point is to reset IMU values before every use. Due to the fact that all gestures are captured while looking in one direction, if we change it while on testing, results would be a mess. Resetting the

IMU means that we have to add an offset to every value of the IMU to keep the signal unaffected by our current position and direction. Furthermore, although we wanted our system to be portable, we connected both sensor glove and Zybo board to the PC to simplify things. A good improvement will be to bypass the Arduino board and PC, and directly attach the sensors to the Zybo board, so that the ARM can read raw data on its own.

Although we have managed to implement this project without deviating from our original goals, many things need further research and study. In current work we have used a limited vocabulary that does not measure up to the original American Sign Language lexicon. So as future work, we have to increase the size of this vocabulary. This comes at the expense of the overall complexity of the project. From gesture recording, that must be done by real signers in that case to more computational complex model (our purposed 1D CNN architecture) needed for inference. How far can we go and how model's complexity changes according to the size of the vocabulary keeping accuracy high enough? Another work is to create a generator for implementing different CNN models on ARM using the Eigen library in a complete, user-friendly application to initialize and run various models. Finally, we can create a hardware accelerator that can run on small resources FPGAs, such as Zybo, in combination with the ARM processor to take full advantage of an embedded system and run more complex CNN models in a more extensive vocabulary.

Although this thesis presents the design and implementation of an end-to-end system to address gesture recognition, it more generally demonstrates a feasible method to handling time series based classification by exploiting the power of 1D CNNs. In SLR, we add a new sensor-based method, using 1D CNNs relative to the state of the art Recognition-Verification mechanism in [39], that promises an efficient way to classify gestures from a large vocabulary and can run in small resources embedded systems.

ABBREVIATIONS - ACRONYMS

SL	Sign Language
ASL	American Sign Language
SLR	Sign Language Recongition
VR	Virtual Reality
ML	Machine Learning
DL	Deep Learning
IMU	Inertial Measurement Unit
FPGA	Field Programmable Gate Array
ARM	Advanced RISC Machines
CNN	Convolutional Neural Network
UART	Universal Asynchronous Receiver/Transmitter
AI	Artificial Intelligence
RNN	Recurrent Neural Network
EMG	Electromyography
sEMG	Surface Electromyography
LDA	Linear Discriminant Analysis
SVM	Support Vector Machine
HMM	Hidden Markov Model
PaHMM	Parallel Hiden Markov Model
ASR	Automatic Speech Recognition
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
IDE	Integrated Development Environment
COM	Communication
SPI	Serial Peripheral Interface
GPIO	General Purpose Input/Output

APPENDIX A. CODE FOR BASELINE ML

Python code for fine-tuning baseline methods: *AtomML.ipynb*

[1]:
```
import numpy as np
from sklearn.preprocessing import StandardScaler
```

[2]:
```
dataSet=np.load("train_dataset_20.npy")
labels=np.load("train_labels_20.npy")
print(np.shape(dataSet))
```

```
(23841, 20, 14)
```

[3]:
```
dataSet=dataSet.reshape(23841,280)
```

[4]:
```
test_dataSet=np.load("test_dataset_20.npy")
test_labels=np.load("test_labels_20.npy")
print(np.shape(test_dataSet))
```

```
(5521, 20, 14)
```

[5]:
```
test_dataSet=test_dataSet.reshape(5521,280)
```

[6]:
```
scaler1 = StandardScaler()
dataSet=scaler1.fit_transform(dataSet)
test_dataSet=scaler1.transform(test_dataSet)
```

[7]:
```
#Import ATOM library and create an ATOMClassifier
from atom import ATOMClassifier
atom = ATOMClassifier((dataSet, labels),           #train set, targets
                      (test_dataSet, test_labels), #test set, targets
                      warnings='ignore', logger="auto",
                      n_jobs=-1, verbose=2)
```

```
<< ================== ATOM ================== >>
Algorithm task: multiclass classification.
Parallel processing with 4 cores.

Dataset stats ====================== >>
Shape: (29362, 281)
Scaled: True
Outlier values: 64574 (1.0%)
---------------------------------------
Train set size: 23841
Test set size: 5521
---------------------------------------
|    | dataset       | train         | test         |
|---:|:--------------|:--------------|:-------------|
|  0 | 13343 (33.6)  | 10940 (33.6)  | 2403 (33.8)  |
|  1 | 550 (1.4)     | 416 (1.3)     | 134 (1.9)    |
|  2 | 810 (2.0)    | 670 (2.1)    | 140 (2.0)   |
|  3 | 975 (2.5)    | 787 (2.4)    | 188 (2.6)   |
|  4 | 482 (1.2)    | 389 (1.2)    | 93 (1.3)    |
|  5 | 772 (1.9)    | 601 (1.8)    | 171 (2.4)   |
|  6 | 729 (1.8)    | 593 (1.8)    | 136 (1.9)   |
|  7 | 1057 (2.7)   | 839 (2.6)    | 218 (3.1)   |
|  8 | 1260 (3.2)   | 1019 (3.1)   | 241 (3.4)   |
|  9 | 944 (2.4)    | 767 (2.4)    | 177 (2.5)   |
| 10 | 446 (1.1)    | 355 (1.1)    | 91 (1.3)    |
| 11 | 570 (1.4)    | 475 (1.5)    | 95 (1.3)    |
| 12 | 1048 (2.6)   | 847 (2.6)    | 201 (2.8)   |
| 13 | 397 (1.0)    | 326 (1.0)    | 71 (1.0)    |
| 14 | 441 (1.1)    | 355 (1.1)    | 86 (1.2)    |
| 15 | 1162 (2.9)   | 942 (2.9)    | 220 (3.1)   |
| 16 | 805 (2.0)    | 655 (2.0)    | 150 (2.1)   |
| 17 | 447 (1.1)    | 357 (1.1)    | 90 (1.3)    |
| 18 | 1603 (4.0)   | 1281 (3.9)   | 322 (4.5)   |
| 19 | 845 (2.1)    | 684 (2.1)    | 161 (2.3)   |
| 20 | 676 (1.7)    | 543 (1.7)    | 133 (1.9)   |
```

[8]:
```
#To run BO for XGB - comment all other models and "early stopping".
#To run BO for remaining models - Comment "XGB" and uncomment everything else.
atom.run(
    models=["LDA",
            "LR",
            "RF",
            "lSVM"
            #"XGB"
    ],
    metric="accuracy",
    n_calls=25,
    n_initial_points=10,
    bo_params={"early_stopping": 0.1,
               "cv":5},
    n_bootstrap=5,
)
```

[9]:
```
#Save Atom class to file
atom.save("final_atom",save_data=True)
```

```
ATOMClassifier saved successfully!
```

[10]:
```
from atom import ATOMLoader
```

[11]:
```
atom_2=ATOMLoader("final_atom", verbose=0)
```

```
ATOMClassifier loaded successfully!
```

[12]:
```
atom_2.results
```

[12]:
```
      metric_bo     time_bo  metric_train  metric_test time_fit  \
LDA    0.828698       1m:51s      0.839730     0.802391   1.426s
LR     0.948786  4h:49m:39s      0.975882     0.909618   5m:52s
RF     0.956839  2h:55m:02s      0.981880     0.928636   3m:26s
lSVM   0.939306  1h:32m:35s      0.959356     0.907263   3m:56s

      mean_bootstrap  std_bootstrap time_bootstrap        time
LDA         0.796196       0.002434         6.994s      1m:59s
LR          0.906901       0.002549        22m:53s  5h:18m:24s
RF          0.925666       0.002661        15m:07s  3h:13m:34s
lSVM        0.895888       0.001083        19m:19s  1h:55m:49s
```

[13]:
```
#Atom runs BO for XGB model - "early stopping isn't compatible with XGB".
#To run BO for remaining models - Comment "XGB" and uncomment everything else.
atom_2.run(
    models=[#"LDA",
            #"LR",
            #"RF",
            #"lSVM"
        "XGB"
    ],
    metric="accuracy",
    n_calls=25,
    n_initial_points=10,
    bo_params={#"early_stopping": 0.1,
               "cv":5},
    n_bootstrap=5,
)
```

```
Training ===================================== >>
Models: XGB
Metric: accuracy


Running BO for XGBoost...
Initial point 1 ---------------------------------
Parameters --> {'n_estimators': 62, 'learning_rate': 0.04, 'max_depth': 1,
'gamma': 0.59, 'min_child_weight': 1, 'subsample': 0.8, 'colsample_bytree': 0.8,
'reg_alpha': 10.0, 'reg_lambda': 0.1}

2021/07/15 15:52:53 WARNING mlflow.tracking.context.git_context: Failed to
import Git (the Git executable is probably not on your PATH), so Git SHA is not
available. Error: Failed to initialize: Bad git executable.
The git executable must be specified in one of the following ways:
    - be included in your $PATH
    - be set via $GIT_PYTHON_GIT_EXECUTABLE
    - explicitly set via git.refresh()

All git commands will error until this is rectified.

This initial warning can be silenced or aggravated in the future by setting the
$GIT_PYTHON_REFRESH environment variable. Use one of the following values:
    - quiet|q|silence|s|none|n|0: for no warning or exception
    - warn|w|warning|1: for a printed warning
    - error|e|raise|r|2: for a raised exception

Example:
    export GIT_PYTHON_REFRESH=quiet


Evaluation --> accuracy: 0.6980  Best accuracy: 0.6980
Time iteration: 1m:26s   Total time: 1m:26s
Initial point 2 --------------------------------
Parameters --> {'n_estimators': 362, 'learning_rate': 0.01, 'max_depth': 3,
'gamma': 0.43, 'min_child_weight': 8, 'subsample': 0.6, 'colsample_bytree': 0.9,
'reg_alpha': 1.0, 'reg_lambda': 100.0}
Evaluation --> accuracy: 0.9171  Best accuracy: 0.9171
Time iteration: 21m:31s   Total time: 22m:57s
Initial point 3 --------------------------------
Parameters --> {'n_estimators': 204, 'learning_rate': 0.27, 'max_depth': 5,
'gamma': 0.87, 'min_child_weight': 17, 'subsample': 0.7, 'colsample_bytree':
0.7, 'reg_alpha': 0.1, 'reg_lambda': 0.1}
Evaluation --> accuracy: 0.9615  Best accuracy: 0.9615
Time iteration: 5m:03s   Total time: 28m:00s
Initial point 4 --------------------------------
Parameters --> {'n_estimators': 398, 'learning_rate': 0.63, 'max_depth': 4,
'gamma': 0.91, 'min_child_weight': 9, 'subsample': 0.8, 'colsample_bytree': 0.6,
'reg_alpha': 1.0, 'reg_lambda': 1.0}
Evaluation --> accuracy: 0.9546  Best accuracy: 0.9615
Time iteration: 8m:41s   Total time: 36m:42s
Initial point 5 --------------------------------
Parameters --> {'n_estimators': 484, 'learning_rate': 0.11, 'max_depth': 1,
'gamma': 0.29, 'min_child_weight': 14, 'subsample': 0.9, 'colsample_bytree':
0.6, 'reg_alpha': 0.1, 'reg_lambda': 0.1}
Evaluation --> accuracy: 0.9567  Best accuracy: 0.9615
Time iteration: 5m:58s   Total time: 42m:39s
Initial point 6 --------------------------------
Parameters --> {'n_estimators': 76, 'learning_rate': 0.94, 'max_depth': 1,
'gamma': 0.25, 'min_child_weight': 6, 'subsample': 0.9, 'colsample_bytree': 0.7,
'reg_alpha': 100.0, 'reg_lambda': 100.0}
Evaluation --> accuracy: 0.8435  Best accuracy: 0.9615
Time iteration: 1m:19s   Total time: 43m:58s
Initial point 7 --------------------------------
```

```
Parameters --> {'n_estimators': 68, 'learning_rate': 0.26, 'max_depth': 7,
'gamma': 0.86, 'min_child_weight': 14, 'subsample': 0.7, 'colsample_bytree':
0.8, 'reg_alpha': 0.01, 'reg_lambda': 1.0}
Evaluation --> accuracy: 0.9594  Best accuracy: 0.9615
Time iteration: 3m:07s   Total time: 47m:06s
Initial point 8 --------------------------------
Parameters --> {'n_estimators': 102, 'learning_rate': 0.05, 'max_depth': 4,
'gamma': 0.66, 'min_child_weight': 14, 'subsample': 0.8, 'colsample_bytree':
0.3, 'reg_alpha': 0.0, 'reg_lambda': 0.01}
Evaluation --> accuracy: 0.9523  Best accuracy: 0.9615
Time iteration: 2m:37s   Total time: 49m:42s
Initial point 9 --------------------------------
Parameters --> {'n_estimators': 402, 'learning_rate': 0.02, 'max_depth': 4,
'gamma': 0.64, 'min_child_weight': 11, 'subsample': 0.8, 'colsample_bytree':
0.4, 'reg_alpha': 10.0, 'reg_lambda': 0.01}
Evaluation --> accuracy: 0.9511  Best accuracy: 0.9615
Time iteration: 11m:57s   Total time: 1h:01m:39s
Initial point 10 --------------------------------
Parameters --> {'n_estimators': 97, 'learning_rate': 0.14, 'max_depth': 6,
'gamma': 0.07, 'min_child_weight': 5, 'subsample': 1.0, 'colsample_bytree': 0.9,
'reg_alpha': 0.01, 'reg_lambda': 0.1}
Evaluation --> accuracy: 0.9649  Best accuracy: 0.9649
Time iteration: 6m:34s   Total time: 1h:08m:13s
Iteration 11 ------------------------------------
Parameters --> {'n_estimators': 500, 'learning_rate': 1.0, 'max_depth': 10,
'gamma': 1.0, 'min_child_weight': 4, 'subsample': 0.8, 'colsample_bytree': 0.3,
'reg_alpha': 100.0, 'reg_lambda': 100.0}
Evaluation --> accuracy: 0.9057  Best accuracy: 0.9649
Time iteration: 4m:51s   Total time: 1h:13m:04s
Iteration 12 ------------------------------------
Parameters --> {'n_estimators': 20, 'learning_rate': 1.0, 'max_depth': 1,
'gamma': 1.0, 'min_child_weight': 20, 'subsample': 1.0, 'colsample_bytree': 1.0,
'reg_alpha': 0.0, 'reg_lambda': 100.0}
Evaluation --> accuracy: 0.9212  Best accuracy: 0.9649
Time iteration: 26.632s   Total time: 1h:13m:31s
Iteration 13 ------------------------------------
Parameters --> {'n_estimators': 227, 'learning_rate': 1.0, 'max_depth': 7,
'gamma': 0.48, 'min_child_weight': 6, 'subsample': 1.0, 'colsample_bytree': 0.8,
'reg_alpha': 0.01, 'reg_lambda': 0.1}
Evaluation --> accuracy: 0.9486  Best accuracy: 0.9649
Time iteration: 5m:14s   Total time: 1h:18m:46s
Iteration 14 ------------------------------------
Parameters --> {'n_estimators': 20, 'learning_rate': 1.0, 'max_depth': 10,
'gamma': 0.0, 'min_child_weight': 20, 'subsample': 1.0, 'colsample_bytree': 1.0,
'reg_alpha': 0.0, 'reg_lambda': 0.0}
Evaluation --> accuracy: 0.9373  Best accuracy: 0.9649
Time iteration: 58.296s   Total time: 1h:19m:45s
Iteration 15 ------------------------------------
```

```
Parameters --> {'n_estimators': 20, 'learning_rate': 1.0, 'max_depth': 10,
'gamma': 0.0, 'min_child_weight': 8, 'subsample': 1.0, 'colsample_bytree': 0.8,
'reg_alpha': 0.0, 'reg_lambda': 0.0}
Evaluation --> accuracy: 0.9443  Best accuracy: 0.9649
Time iteration: 1m:01s   Total time: 1h:20m:46s
Iteration 16 ------------------------------------
Parameters --> {'n_estimators': 180, 'learning_rate': 1.0, 'max_depth': 7,
'gamma': 0.58, 'min_child_weight': 20, 'subsample': 0.6, 'colsample_bytree':
0.4, 'reg_alpha': 0.0, 'reg_lambda': 0.01}
Evaluation --> accuracy: 0.9431  Best accuracy: 0.9649
Time iteration: 2m:01s   Total time: 1h:22m:47s
Iteration 17 ------------------------------------
Parameters --> {'n_estimators': 76, 'learning_rate': 0.04, 'max_depth': 7,
'gamma': 0.0, 'min_child_weight': 6, 'subsample': 0.9, 'colsample_bytree': 0.7,
'reg_alpha': 0.0, 'reg_lambda': 0.1}
Evaluation --> accuracy: 0.9570  Best accuracy: 0.9649
Time iteration: 6m:12s   Total time: 1h:28m:60s
Iteration 18 ------------------------------------
Parameters --> {'n_estimators': 148, 'learning_rate': 0.05, 'max_depth': 2,
'gamma': 0.0, 'min_child_weight': 3, 'subsample': 1.0, 'colsample_bytree': 0.4,
'reg_alpha': 0.1, 'reg_lambda': 0.01}
Evaluation --> accuracy: 0.9436  Best accuracy: 0.9649
Time iteration: 3m:04s   Total time: 1h:32m:04s
Iteration 19 ------------------------------------
Parameters --> {'n_estimators': 500, 'learning_rate': 1.0, 'max_depth': 7,
'gamma': 0.73, 'min_child_weight': 16, 'subsample': 1.0, 'colsample_bytree':
0.5, 'reg_alpha': 1.0, 'reg_lambda': 0.0}
Evaluation --> accuracy: 0.9444  Best accuracy: 0.9649
Time iteration: 6m:11s   Total time: 1h:38m:16s
Iteration 20 ------------------------------------
Parameters --> {'n_estimators': 311, 'learning_rate': 1.0, 'max_depth': 9,
'gamma': 0.59, 'min_child_weight': 14, 'subsample': 0.6, 'colsample_bytree':
0.4, 'reg_alpha': 0.1, 'reg_lambda': 0.0}
Evaluation --> accuracy: 0.9462  Best accuracy: 0.9649
Time iteration: 3m:22s   Total time: 1h:41m:38s
Iteration 21 ------------------------------------
Parameters --> {'n_estimators': 110, 'learning_rate': 0.01, 'max_depth': 10,
'gamma': 0.19, 'min_child_weight': 1, 'subsample': 0.7, 'colsample_bytree': 0.3,
'reg_alpha': 0.01, 'reg_lambda': 0.0}
Evaluation --> accuracy: 0.9607  Best accuracy: 0.9649
Time iteration: 5m:39s   Total time: 1h:47m:18s
Iteration 22 ------------------------------------
Parameters --> {'n_estimators': 500, 'learning_rate': 0.01, 'max_depth': 9,
'gamma': 0.0, 'min_child_weight': 1, 'subsample': 0.9, 'colsample_bytree': 1.0,
'reg_alpha': 0.0, 'reg_lambda': 0.0}
Evaluation --> accuracy: 0.9632  Best accuracy: 0.9649
Time iteration: 1h:34m:47s   Total time: 3h:22m:05s
Iteration 23 ------------------------------------
```

```
Parameters --> {'n_estimators': 500, 'learning_rate': 1.0, 'max_depth': 9,
'gamma': 1.0, 'min_child_weight': 20, 'subsample': 0.9, 'colsample_bytree': 0.7,
'reg_alpha': 0.01, 'reg_lambda': 100.0}
Evaluation --> accuracy: 0.9533  Best accuracy: 0.9649
Time iteration: 15m:06s   Total time: 3h:37m:12s
Iteration 24 -----------------------------------
Parameters --> {'n_estimators': 371, 'learning_rate': 0.01, 'max_depth': 1,
'gamma': 0.92, 'min_child_weight': 1, 'subsample': 1.0, 'colsample_bytree': 0.6,
'reg_alpha': 0.0, 'reg_lambda': 0.01}
Evaluation --> accuracy: 0.7658  Best accuracy: 0.9649
Time iteration: 5m:25s   Total time: 3h:42m:37s
Iteration 25 -----------------------------------
Parameters --> {'n_estimators': 500, 'learning_rate': 0.01, 'max_depth': 3,
'gamma': 0.0, 'min_child_weight': 20, 'subsample': 0.5, 'colsample_bytree': 1.0,
'reg_alpha': 100.0, 'reg_lambda': 0.1}
Evaluation --> accuracy: 0.8497  Best accuracy: 0.9649
Time iteration: 28m:21s   Total time: 4h:10m:59s

Results for XGBoost:
Bayesian Optimization ---------------------------
Best parameters --> {'n_estimators': 97, 'learning_rate': 0.14, 'max_depth': 6,
'gamma': 0.07, 'min_child_weight': 5, 'subsample': 1.0, 'colsample_bytree': 0.9,
'reg_alpha': 0.01, 'reg_lambda': 0.1}
Best evaluation --> accuracy: 0.9649
Time elapsed: 4h:10m:59s
Fit ---------------------------------------------
Train evaluation --> accuracy: 1.0
Test evaluation --> accuracy: 0.9315
Time elapsed: 1m:44s
Bootstrap ---------------------------------------
Evaluation --> accuracy: 0.9272 ± 0.0017
Time elapsed: 7m:48s
-------------------------------------------------
Total time: 4h:20m:31s


Final results ========================= >>
Duration: 4h:20m:31s
------------------------------------------
XGBoost --> accuracy: 0.9272 ± 0.0017
```

[14]:
```
# "XGB" model has been added to pipeline.
atom_2.results
```

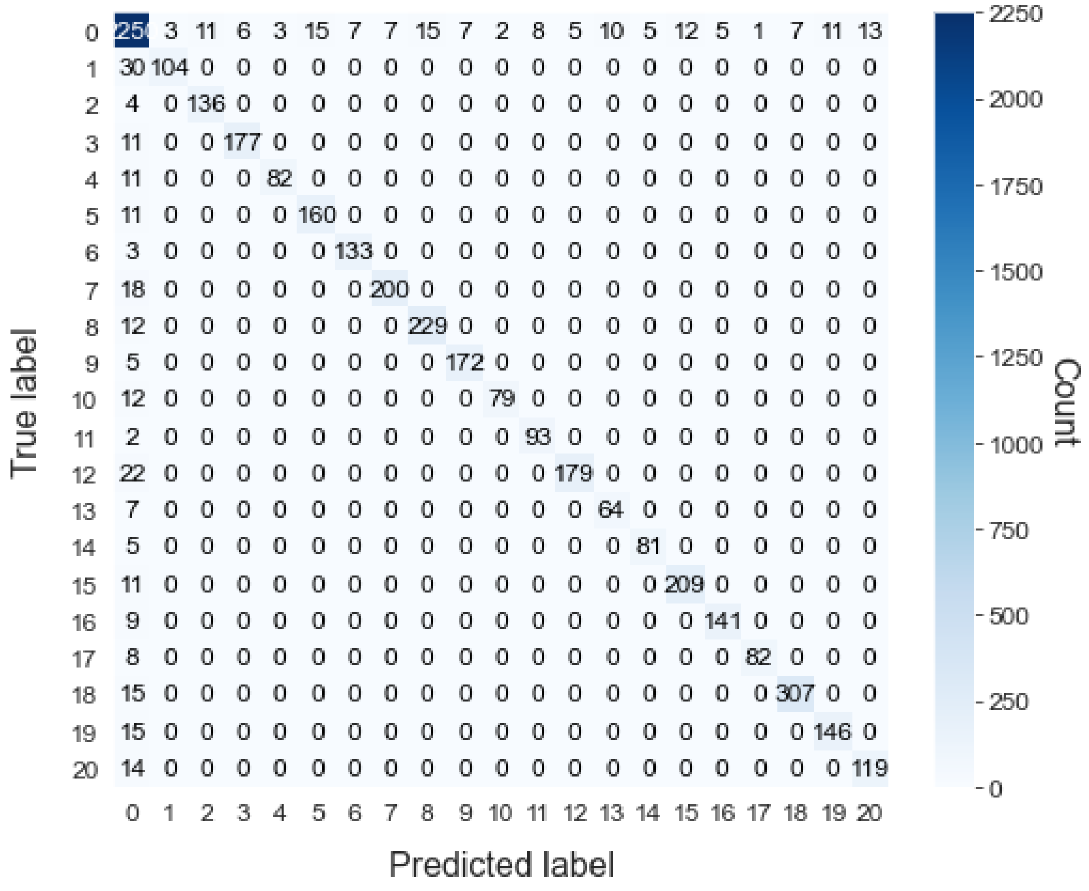
[14]:
```
       metric_bo     time_bo  metric_train  metric_test time_fit  \
LDA     0.828698      1m:51s      0.839730     0.802391   1.426s
LR      0.948786  4h:49m:39s      0.975882     0.909618   5m:52s
```

```
RF      0.956839  2h:55m:02s      0.981880     0.928636    3m:26s
lSVM    0.939306  1h:32m:35s      0.959356     0.907263    3m:56s
XGB     0.964893  4h:10m:59s      1.000000     0.931534    1m:44s

      mean_bootstrap  std_bootstrap  time_bootstrap        time
LDA         0.796196       0.002434          6.994s      1m:59s
LR          0.906901       0.002549         22m:53s  5h:18m:24s
RF          0.925666       0.002661         15m:07s  3h:13m:34s
lSVM        0.895888       0.001083         19m:19s  1h:55m:49s
XGB         0.927223       0.001708          7m:48s  4h:20m:31s
```

[15]:
```
#Save Atom class to file
atom_2.save("final_atom",save_data=True)
```

```
ATOMClassifier saved successfully!
```

[16]:
```
#Plot the confusion matrix for the "XGB"
atom_2.xgb.plot_confusion_matrix()
```



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	25(	3	11	6	3	15	7	7	15	7	2	8	5	10	5	12	5	1	7	11	13
1	30	104	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	4	0	136	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	11	0	0	177	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	11	0	0	0	82	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	11	0	0	0	0	160	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	3	0	0	0	0	0	133	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	18	0	0	0	0	0	0	200	0	0	0	0	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	229	0	0	0	0	0	0	0	0	0	0	0	0
9	5	0	0	0	0	0	0	0	0	172	0	0	0	0	0	0	0	0	0	0	0
10	12	0	0	0	0	0	0	0	0	0	79	0	0	0	0	0	0	0	0	0	0
11	2	0	0	0	0	0	0	0	0	0	0	93	0	0	0	0	0	0	0	0	0
12	22	0	0	0	0	0	0	0	0	0	0	0	179	0	0	0	0	0	0	0	0
13	7	0	0	0	0	0	0	0	0	0	0	0	0	64	0	0	0	0	0	0	0
14	5	0	0	0	0	0	0	0	0	0	0	0	0	0	81	0	0	0	0	0	0
15	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	209	0	0	0	0	0
16	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	141	0	0	0	0
17	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	82	0	0	0
18	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	307	0	0
19	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	146	0
20	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	119

In line 13, ATOM's *run* method, runs BO for the XGB model. The output log, prints the steps of the procedure. Ten random hyper-parameter tests are done first (Initial point 1 - 10). Then the best one fits the surrogate function (BO algorithm). BO runs for 15 times to reach $n\_call = 25$ and best hyper-parameters are exported with an evaluation accuracy 93,15%. Finally, a bootstrap algorithm take place. The same procedure is repeated for the remaining models and results are printed by *result* function in line 14.

Best Hyper-parameters are presented below:

1. **LDA:** Best parameters –> {'solver': 'svd'}
2. **LR:** Best parameters –> {'penalty': 'l2', 'C': 2.033, 'solver': 'newton-cg', 'max_iter': 108}
3. **RF:** Best parameters –> {'criterion': 'entropy', 'max_depth': 9, 'max_features': 0.6, 'max_samples': 0.7, 'min_samples_split': 12, 'n_estimators': 376}
4. **LSVM:** Best parameters –> {'C': 0.984, 'dual': False, 'penalty': 'l1'}

APPENDIX B. CODE FOR OUR PROPOSED METHOD

Python code for fine-tuning our proposed CNN method: *AtomCNN.ipynb*

```
[1]: # Disable annoying tf warnings
     import logging
     import tensorflow as tf
     tf.get_logger().setLevel(logging.ERROR)
     import os
     os.environ["GIT_PYTHON_REFRESH"] = "quiet"
```

```
[2]: from __future__ import absolute_import, division, print_function,␣
      ↪unicode_literals
     import tensorflow as tf
     from tensorflow import keras
     from keras import layers
     from keras import models
     import numpy as np
     from sklearn.preprocessing import StandardScaler
     from sklearn.metrics import confusion_matrix
     from skopt.space.space import Integer, Categorical, Real
     from keras import optimizers
     from keras.utils.vis_utils import plot_model
```

```
[3]: dataSet=np.load("train_dataset_20.npy")
     labels=np.load("train_labels_20.npy")
     print(np.shape(dataSet))
```

```
(23841, 20, 14)
```

```
[4]: dataSet=dataSet.reshape(23841*20,14)
```

```
[5]: test_dataSet=np.load("test_dataset_20.npy")
     test_labels=np.load("test_labels_20.npy")
     print(np.shape(test_dataSet))
```

```
(5521, 20, 14)
```

```
[6]: test_dataSet=test_dataSet.reshape(5521*20,14)
```

```
[7]: scaler1 = StandardScaler()
     dataSet=scaler1.fit_transform(dataSet)
     test_dataSet=scaler1.transform(test_dataSet)
```

```
[8]: dataSet=dataSet.reshape(23841, 280)
     test_dataSet=test_dataSet.reshape(5521, 280)
```

```
[9]: #define our CNN model
     def neural_network(x1, x2, y1, y2):
         model=models.Sequential()
```

```
    model.add(keras.Input(shape=(280)))
    model.add(layers.Reshape((20,14)))
    model.add(layers.Conv1D(x1, (y1,), activation='relu'))
    model.add(layers.Conv1D(x2, (y2,), activation='relu'))
    model.add(layers.Flatten())
    model.add(layers.Dropout(0.18))
    model.add(layers.Dense(21,activation='softmax'))


    opti=optimizers.RMSprop(lr=0.00002
    model.compile(optimizer=opti,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model
```

[10]:
```
# Import standard packages
from atom import ATOMClassifier, ATOMModel
```

[11]:
```
# Like any other model, we can define custom dimensions for the bayesian␣
 ↪optimization
dim = [Integer(1, 100, name="epochs"),
       Categorical([64, 96, 128], name="batch_size"),
       Integer(16, 100, name="x1"),
       Integer(16, 100, name="x2"),
       Integer(1, 10, name="y1"),
       Integer(1, 10, name="y2")
      ]
```

[12]:
```
# Since ATOM uses sklearn's API, use Keras' wrapper
model = KerasClassifier(neural_network, verbose=2)
```

[13]:
```
# Convert the model to an ATOM model
model = ATOMModel(model, acronym="NN", fullname="Neural network")
```

[14]:
```
atom = ATOMClassifier(dataSet, test_dataSet, labels, test_labels, n_rows=1,␣
 ↪n_jobs=2, verbose=2)
#atom.add(StandardScaler()) -- We do not use the following functions
#atom.impute()
#atom.encode()
#atom.feature_selection()
```

```
<< ================== ATOM ================== >>
Algorithm task: multiclass classification.
Parallel processing with 2 cores.

Dataset stats ====================== >>
Shape: (29362, 281)
```

```
Scaled: True
Outlier values: 64574 (1.0%)
---------------------------------------
Train set size: 23841
Test set size: 5521
---------------------------------------
|    | dataset       | train         | test         |
|---:|:--------------|:--------------|:-------------|
|  0 | 13343 (33.6)  | 10940 (33.6)  | 2403 (33.8)  |
|  1 | 550 (1.4)     | 416 (1.3)     | 134 (1.9)    |
|  2 | 810 (2.0)     | 670 (2.1)     | 140 (2.0)    |
|  3 | 975 (2.5)     | 787 (2.4)     | 188 (2.6)    |
|  4 | 482 (1.2)     | 389 (1.2)     | 93 (1.3)     |
|  5 | 772 (1.9)     | 601 (1.8)     | 171 (2.4)    |
|  6 | 729 (1.8)     | 593 (1.8)     | 136 (1.9)    |
|  7 | 1057 (2.7)    | 839 (2.6)     | 218 (3.1)    |
|  8 | 1260 (3.2)    | 1019 (3.1)    | 241 (3.4)    |
|  9 | 944 (2.4)     | 767 (2.4)     | 177 (2.5)    |
| 10 | 446 (1.1)     | 355 (1.1)     | 91 (1.3)     |
| 11 | 570 (1.4)     | 475 (1.5)     | 95 (1.3)     |
| 12 | 1048 (2.6)    | 847 (2.6)     | 201 (2.8)    |
| 13 | 397 (1.0)     | 326 (1.0)     | 71 (1.0)     |
| 14 | 441 (1.1)     | 355 (1.1)     | 86 (1.2)     |
| 15 | 1162 (2.9)    | 942 (2.9)     | 220 (3.1)    |
| 16 | 805 (2.0)     | 655 (2.0)     | 150 (2.1)    |
| 17 | 447 (1.1)     | 357 (1.1)     | 90 (1.3)     |
| 18 | 1603 (4.0)    | 1281 (3.9)    | 322 (4.5)    |
| 19 | 845 (2.1)     | 684 (2.1)     | 161 (2.3)    |
| 20 | 676 (1.7)     | 543 (1.7)     | 133 (1.9)    |
```

[15]:
```
# Train the models using early stopping. An early stopping value of 0.1 means
# that the model will stop if it didn't improve in the last 10% of it's␣
 ↪iterations

atom.run(
    model,
    metric="accuracy",
    n_calls=25,
    n_initial_points=10,
    bo_params={"dimensions": dim, "early_stopping": 0.1, "cv":5},
)
```

```
Training ===================================== >>
Models: NN
Metric: accuracy
```

```
Running BO for Neural network...
Initial point 1 --------------------------------
Parameters --> {'epochs': 23, 'batch_size': 96, 'x1': 38, 'x2': 55, 'y1': 6,
'y2': 2}
Evaluation --> accuracy: 0.9072  Best accuracy: 0.9072
Time iteration: 1m:36s   Total time: 1m:36s
Initial point 2 --------------------------------
Parameters --> {'epochs': 69, 'batch_size': 96, 'x1': 26, 'x2': 57, 'y1': 9,
'y2': 3}
Evaluation --> accuracy: 0.9339  Best accuracy: 0.9339
Time iteration: 2m:24s   Total time: 4m:01s
Initial point 3 --------------------------------
Parameters --> {'epochs': 14, 'batch_size': 96, 'x1': 76, 'x2': 35, 'y1': 4,
'y2': 4}
Evaluation --> accuracy: 0.8736  Best accuracy: 0.9339
Time iteration: 52.157s   Total time: 4m:53s
Initial point 4 --------------------------------
Parameters --> {'epochs': 18, 'batch_size': 96, 'x1': 36, 'x2': 42, 'y1': 2,
'y2': 2}
Evaluation --> accuracy: 0.8632  Best accuracy: 0.9339
Time iteration: 42.769s   Total time: 5m:36s
Initial point 5 --------------------------------
Parameters --> {'epochs': 38, 'batch_size': 96, 'x1': 82, 'x2': 91, 'y1': 3,
'y2': 2}
Evaluation --> accuracy: 0.9363  Best accuracy: 0.9363
Time iteration: 2m:48s   Total time: 8m:24s
Initial point 6 --------------------------------
Parameters --> {'epochs': 2, 'batch_size': 96, 'x1': 56, 'x2': 51, 'y1': 3,
'y2': 4}
Evaluation --> accuracy: 0.4611  Best accuracy: 0.9363
Time iteration: 10.380s   Total time: 8m:34s
Initial point 7 --------------------------------
Parameters --> {'epochs': 5, 'batch_size': 64, 'x1': 49, 'x2': 88, 'y1': 3,
'y2': 6}
Evaluation --> accuracy: 0.7933  Best accuracy: 0.9363
Time iteration: 26.301s   Total time: 9m:01s
Initial point 8 --------------------------------
Parameters --> {'epochs': 36, 'batch_size': 128, 'x1': 94, 'x2': 96, 'y1': 7,
'y2': 9}
Evaluation --> accuracy: 0.9384  Best accuracy: 0.9384
Time iteration: 3m:05s   Total time: 12m:06s
Initial point 9 --------------------------------
Parameters --> {'epochs': 91, 'batch_size': 64, 'x1': 25, 'x2': 45, 'y1': 4,
'y2': 9}
Evaluation --> accuracy: 0.9419  Best accuracy: 0.9419
Time iteration: 3m:37s   Total time: 15m:43s
Initial point 10 -------------------------------
```

```
Parameters --> {'epochs': 100, 'batch_size': 128, 'x1': 94, 'x2': 86, 'y1': 7,
'y2': 5}
Evaluation --> accuracy: 0.9481  Best accuracy: 0.9481
Time iteration: 8m:16s   Total time: 23m:59s
Iteration 11 ------------------------------------
Parameters --> {'epochs': 42, 'batch_size': 96, 'x1': 97, 'x2': 92, 'y1': 2,
'y2': 7}
Evaluation --> accuracy: 0.9405  Best accuracy: 0.9481
Time iteration: 5m:26s   Total time: 29m:26s
Iteration 12 ------------------------------------
Parameters --> {'epochs': 94, 'batch_size': 96, 'x1': 57, 'x2': 84, 'y1': 3,
'y2': 3}
Evaluation --> accuracy: 0.9450  Best accuracy: 0.9481
Time iteration: 6m:10s   Total time: 35m:36s
Iteration 13 ------------------------------------
Parameters --> {'epochs': 66, 'batch_size': 96, 'x1': 24, 'x2': 18, 'y1': 2,
'y2': 3}
Evaluation --> accuracy: 0.9133  Best accuracy: 0.9481
Time iteration: 1m:38s   Total time: 37m:14s
Iteration 14 ------------------------------------
Parameters --> {'epochs': 8, 'batch_size': 64, 'x1': 33, 'x2': 90, 'y1': 1,
'y2': 7}
Evaluation --> accuracy: 0.8636  Best accuracy: 0.9481
Time iteration: 34.821s   Total time: 37m:49s
Iteration 15 ------------------------------------
Parameters --> {'epochs': 4, 'batch_size': 64, 'x1': 25, 'x2': 93, 'y1': 6,
'y2': 5}
Evaluation --> accuracy: 0.6725  Best accuracy: 0.9481
Time iteration: 15.375s   Total time: 38m:04s
Iteration 16 ------------------------------------
Parameters --> {'epochs': 6, 'batch_size': 64, 'x1': 59, 'x2': 55, 'y1': 5,
'y2': 9}
Evaluation --> accuracy: 0.8065  Best accuracy: 0.9481
Time iteration: 29.101s   Total time: 38m:34s
Iteration 17 ------------------------------------
Parameters --> {'epochs': 87, 'batch_size': 128, 'x1': 20, 'x2': 96, 'y1': 3,
'y2': 5}
Evaluation --> accuracy: 0.9375  Best accuracy: 0.9481
Time iteration: 4m:04s   Total time: 42m:38s
Iteration 18 ------------------------------------
Parameters --> {'epochs': 43, 'batch_size': 96, 'x1': 61, 'x2': 30, 'y1': 5,
'y2': 4}
Evaluation --> accuracy: 0.9263  Best accuracy: 0.9481
Time iteration: 2m:08s   Total time: 44m:46s
Iteration 19 ------------------------------------
Parameters --> {'epochs': 13, 'batch_size': 96, 'x1': 70, 'x2': 26, 'y1': 3,
'y2': 3}
Evaluation --> accuracy: 0.8042  Best accuracy: 0.9481
```

```
Time iteration: 41.791s   Total time: 45m:28s
Iteration 20 ------------------------------------
Parameters --> {'epochs': 25, 'batch_size': 96, 'x1': 42, 'x2': 35, 'y1': 3,
'y2': 7}
Evaluation --> accuracy: 0.9071  Best accuracy: 0.9481
Time iteration: 1m:16s   Total time: 46m:43s
Iteration 21 ------------------------------------
Parameters --> {'epochs': 8, 'batch_size': 96, 'x1': 53, 'x2': 36, 'y1': 3,
'y2': 5}
Evaluation --> accuracy: 0.7139  Best accuracy: 0.9481
Time iteration: 28.558s   Total time: 47m:12s
Iteration 22 ------------------------------------
Parameters --> {'epochs': 59, 'batch_size': 96, 'x1': 87, 'x2': 99, 'y1': 5,
'y2': 10}
Evaluation --> accuracy: 0.9457  Best accuracy: 0.9481
Time iteration: 6m:04s   Total time: 53m:16s
Iteration 23 ------------------------------------
Parameters --> {'epochs': 32, 'batch_size': 96, 'x1': 78, 'x2': 61, 'y1': 7,
'y2': 3}
Evaluation --> accuracy: 0.9310  Best accuracy: 0.9481
Time iteration: 1m:53s   Total time: 55m:09s
Iteration 24 ------------------------------------
Parameters --> {'epochs': 23, 'batch_size': 128, 'x1': 93, 'x2': 97, 'y1': 8,
'y2': 9}
Evaluation --> accuracy: 0.9290  Best accuracy: 0.9481
Time iteration: 1m:49s   Total time: 56m:58s
Iteration 25 ------------------------------------
Parameters --> {'epochs': 90, 'batch_size': 96, 'x1': 84, 'x2': 54, 'y1': 5,
'y2': 5}
Evaluation --> accuracy: 0.9455  Best accuracy: 0.9481
Time iteration: 7m:02s   Total time: 1h:04m:00s

Results for Neural network:
Bayesian Optimization ---------------------------
Best parameters --> {'epochs': 100, 'batch_size': 128, 'x1': 94, 'x2': 86, 'y1':
7, 'y2': 5}
Best evaluation --> accuracy: 0.9481
Time elapsed: 1h:04m:01s
Epoch 1/100
187/187 - 1s - loss: 2.4798 - accuracy: 0.3717
Epoch 2/100
187/187 - 1s - loss: 1.7621 - accuracy: 0.4805
Epoch 3/100
187/187 - 1s - loss: 1.3206 - accuracy: 0.5968
Epoch 4/100
187/187 - 1s - loss: 1.0111 - accuracy: 0.6816
Epoch 5/100
187/187 - 1s - loss: 0.7973 - accuracy: 0.7323
```

```
Epoch 6/100
187/187 - 1s - loss: 0.6434 - accuracy: 0.7857
Epoch 7/100
187/187 - 1s - loss: 0.5372 - accuracy: 0.8244
Epoch 8/100
187/187 - 1s - loss: 0.4609 - accuracy: 0.8498
Epoch 9/100
187/187 - 1s - loss: 0.4066 - accuracy: 0.8683
Epoch 10/100
187/187 - 1s - loss: 0.3703 - accuracy: 0.8784
Epoch 11/100
187/187 - 1s - loss: 0.3383 - accuracy: 0.8881
Epoch 12/100
187/187 - 1s - loss: 0.3170 - accuracy: 0.8929
Epoch 13/100
187/187 - 1s - loss: 0.2977 - accuracy: 0.8988
Epoch 14/100
187/187 - 1s - loss: 0.2841 - accuracy: 0.9055
Epoch 15/100
187/187 - 1s - loss: 0.2695 - accuracy: 0.9077
Epoch 16/100
187/187 - 1s - loss: 0.2633 - accuracy: 0.9107
Epoch 17/100
187/187 - 1s - loss: 0.2495 - accuracy: 0.9144
Epoch 18/100
187/187 - 2s - loss: 0.2426 - accuracy: 0.9162
Epoch 19/100
187/187 - 1s - loss: 0.2329 - accuracy: 0.9192
Epoch 20/100
187/187 - 1s - loss: 0.2279 - accuracy: 0.9199
Epoch 21/100
187/187 - 1s - loss: 0.2230 - accuracy: 0.9217
Epoch 22/100
187/187 - 1s - loss: 0.2181 - accuracy: 0.9216
Epoch 23/100
187/187 - 1s - loss: 0.2103 - accuracy: 0.9260
Epoch 24/100
187/187 - 1s - loss: 0.2048 - accuracy: 0.9266
Epoch 25/100
187/187 - 1s - loss: 0.2015 - accuracy: 0.9264
Epoch 26/100
187/187 - 1s - loss: 0.1981 - accuracy: 0.9279
Epoch 27/100
187/187 - 1s - loss: 0.1953 - accuracy: 0.9292
Epoch 28/100
187/187 - 1s - loss: 0.1948 - accuracy: 0.9291
Epoch 29/100
187/187 - 1s - loss: 0.1881 - accuracy: 0.9321
```

```
Epoch 30/100
187/187 - 1s - loss: 0.1871 - accuracy: 0.9308
Epoch 31/100
187/187 - 1s - loss: 0.1861 - accuracy: 0.9311
Epoch 32/100
187/187 - 1s - loss: 0.1803 - accuracy: 0.9338
Epoch 33/100
187/187 - 1s - loss: 0.1773 - accuracy: 0.9350
Epoch 34/100
187/187 - 1s - loss: 0.1743 - accuracy: 0.9348
Epoch 35/100
187/187 - 1s - loss: 0.1728 - accuracy: 0.9365
Epoch 36/100
187/187 - 1s - loss: 0.1700 - accuracy: 0.9360
Epoch 37/100
187/187 - 1s - loss: 0.1696 - accuracy: 0.9362
Epoch 38/100
187/187 - 1s - loss: 0.1682 - accuracy: 0.9374
Epoch 39/100
187/187 - 1s - loss: 0.1648 - accuracy: 0.9389
Epoch 40/100
187/187 - 2s - loss: 0.1661 - accuracy: 0.9388
Epoch 41/100
187/187 - 1s - loss: 0.1624 - accuracy: 0.9391
Epoch 42/100
187/187 - 2s - loss: 0.1594 - accuracy: 0.9396
Epoch 43/100
187/187 - 1s - loss: 0.1598 - accuracy: 0.9388
Epoch 44/100
187/187 - 1s - loss: 0.1591 - accuracy: 0.9396
Epoch 45/100
187/187 - 1s - loss: 0.1549 - accuracy: 0.9419
Epoch 46/100
187/187 - 2s - loss: 0.1554 - accuracy: 0.9412
Epoch 47/100
187/187 - 2s - loss: 0.1540 - accuracy: 0.9407
Epoch 48/100
187/187 - 1s - loss: 0.1529 - accuracy: 0.9412
Epoch 49/100
187/187 - 1s - loss: 0.1518 - accuracy: 0.9416
Epoch 50/100
187/187 - 1s - loss: 0.1513 - accuracy: 0.9421
Epoch 51/100
187/187 - 1s - loss: 0.1496 - accuracy: 0.9423
Epoch 52/100
187/187 - 1s - loss: 0.1483 - accuracy: 0.9429
Epoch 53/100
187/187 - 1s - loss: 0.1471 - accuracy: 0.9436
```

```
Epoch 54/100
187/187 - 1s - loss: 0.1461 - accuracy: 0.9437
Epoch 55/100
187/187 - 1s - loss: 0.1459 - accuracy: 0.9443
Epoch 56/100
187/187 - 1s - loss: 0.1447 - accuracy: 0.9435
Epoch 57/100
187/187 - 2s - loss: 0.1420 - accuracy: 0.9443
Epoch 58/100
187/187 - 1s - loss: 0.1384 - accuracy: 0.9459
Epoch 59/100
187/187 - 1s - loss: 0.1407 - accuracy: 0.9464
Epoch 60/100
187/187 - 1s - loss: 0.1398 - accuracy: 0.9464
Epoch 61/100
187/187 - 1s - loss: 0.1379 - accuracy: 0.9458
Epoch 62/100
187/187 - 1s - loss: 0.1382 - accuracy: 0.9459
Epoch 63/100
187/187 - 1s - loss: 0.1371 - accuracy: 0.9465
Epoch 64/100
187/187 - 1s - loss: 0.1368 - accuracy: 0.9471
Epoch 65/100
187/187 - 1s - loss: 0.1354 - accuracy: 0.9465
Epoch 66/100
187/187 - 1s - loss: 0.1363 - accuracy: 0.9460
Epoch 67/100
187/187 - 1s - loss: 0.1351 - accuracy: 0.9472
Epoch 68/100
187/187 - 1s - loss: 0.1347 - accuracy: 0.9472
Epoch 69/100
187/187 - 1s - loss: 0.1329 - accuracy: 0.9490
Epoch 70/100
187/187 - 1s - loss: 0.1328 - accuracy: 0.9471
Epoch 71/100
187/187 - 1s - loss: 0.1315 - accuracy: 0.9481
Epoch 72/100
187/187 - 1s - loss: 0.1304 - accuracy: 0.9479
Epoch 73/100
187/187 - 1s - loss: 0.1282 - accuracy: 0.9503
Epoch 74/100
187/187 - 1s - loss: 0.1304 - accuracy: 0.9478
Epoch 75/100
187/187 - 1s - loss: 0.1295 - accuracy: 0.9495
Epoch 76/100
187/187 - 1s - loss: 0.1281 - accuracy: 0.9492
Epoch 77/100
187/187 - 1s - loss: 0.1278 - accuracy: 0.9496
```

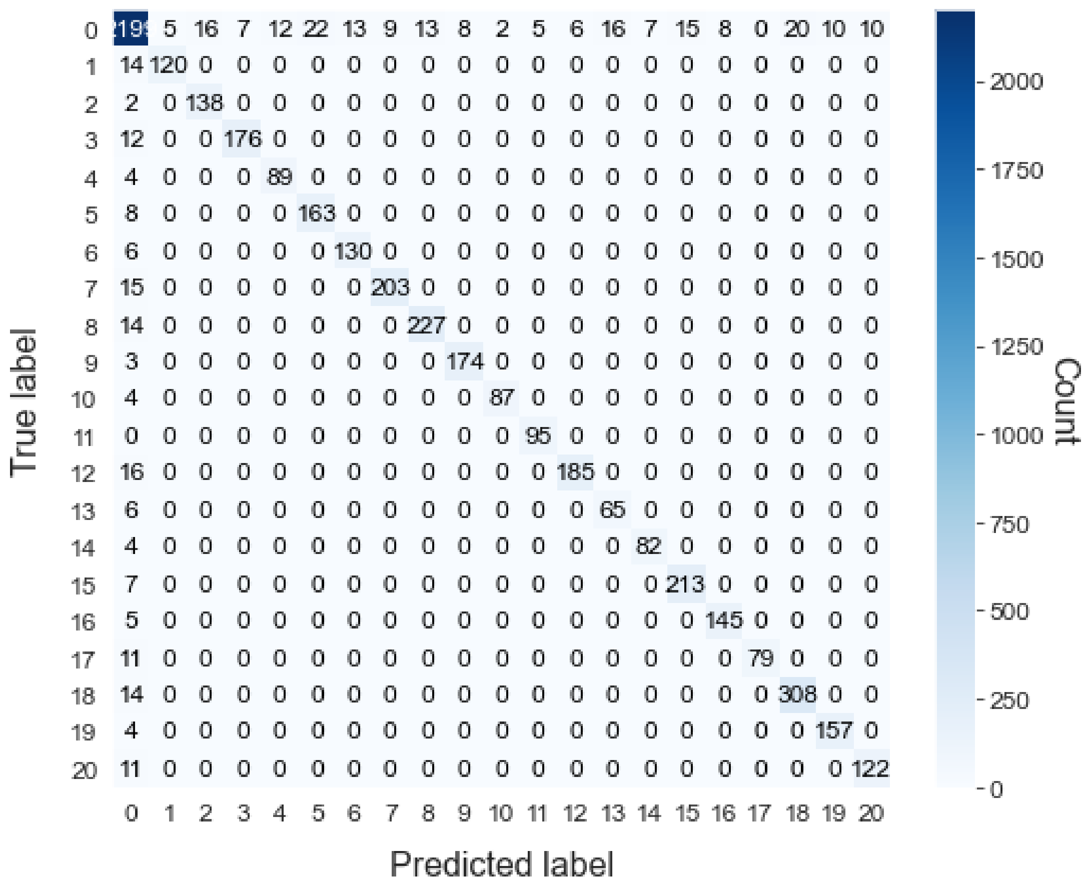
```
Epoch 78/100
187/187 - 1s - loss: 0.1253 - accuracy: 0.9500
Epoch 79/100
187/187 - 1s - loss: 0.1281 - accuracy: 0.9482
Epoch 80/100
187/187 - 1s - loss: 0.1261 - accuracy: 0.9506
Epoch 81/100
187/187 - 1s - loss: 0.1245 - accuracy: 0.9502
Epoch 82/100
187/187 - 1s - loss: 0.1239 - accuracy: 0.9501
Epoch 83/100
187/187 - 1s - loss: 0.1264 - accuracy: 0.9503
Epoch 84/100
187/187 - 1s - loss: 0.1235 - accuracy: 0.9508
Epoch 85/100
187/187 - 1s - loss: 0.1235 - accuracy: 0.9497
Epoch 86/100
187/187 - 1s - loss: 0.1219 - accuracy: 0.9513
Epoch 87/100
187/187 - 1s - loss: 0.1226 - accuracy: 0.9515
Epoch 88/100
187/187 - 1s - loss: 0.1224 - accuracy: 0.9502
Epoch 89/100
187/187 - 1s - loss: 0.1195 - accuracy: 0.9522
Epoch 90/100
187/187 - 1s - loss: 0.1196 - accuracy: 0.9527
Epoch 91/100
187/187 - 1s - loss: 0.1198 - accuracy: 0.9513
Epoch 92/100
187/187 - 1s - loss: 0.1203 - accuracy: 0.9520
Epoch 93/100
187/187 - 1s - loss: 0.1183 - accuracy: 0.9513
Epoch 94/100
187/187 - 1s - loss: 0.1176 - accuracy: 0.9527
Epoch 95/100
187/187 - 1s - loss: 0.1180 - accuracy: 0.9528
Epoch 96/100
187/187 - 1s - loss: 0.1170 - accuracy: 0.9523
Epoch 97/100
187/187 - 1s - loss: 0.1180 - accuracy: 0.9514
Epoch 98/100
187/187 - 1s - loss: 0.1168 - accuracy: 0.9536
Epoch 99/100
187/187 - 1s - loss: 0.1167 - accuracy: 0.9519
Epoch 100/100
187/187 - 1s - loss: 0.1156 - accuracy: 0.9522
187/187 - 0s
44/44 - 0s
```

```
Fit ---------------------------------------------------
Train evaluation --> accuracy: 0.9568
Test evaluation --> accuracy: 0.9341
Time elapsed: 2m:13s
-------------------------------------------------
Total time: 1h:06m:14s

Final results ========================= >>
Duration: 1h:06m:14s
------------------------------------------
Neural network --> accuracy: 0.9341
```

[16]:
```
atom.results
```

[16]:
```
     metric_bo     time_bo  metric_train  metric_test time_fit        time
NN    0.948115  1h:04m:01s      0.956797      0.93407   2m:13s  1h:06m:14s
```

[17]:
```
atom.nn.plot_confusion_matrix(dataset="test", normalize=False, title=None,␣
    ↪figsize=None, filename="NN_confusion2", display=True)
```



APPENDIX C. ARM MAIN PROGRAM

C++ code for the bare-metal application: *app.cpp*

```
//set eigen to operate on free size matricied
#define EIGEN_STACK_ALLOCATION_LIMIT 0

#include <iostream>
#include <Eigen/Dense>
#include "xparameters.h"
#include "xil_io.h"
#include "xuartps.h"
#include "xtime_l.h"
#include "sleep.h"

using namespace std;
using namespace Eigen;

#define classes 21

#define In_shape_i 20
#define In_shape_j 14

#define conv1_kernel 7
#define conv1_filters 94
#define conv1_shape_i conv1_kernel*conv1_filters
#define conv1_shape_j In_shape_j
#define bias1_len conv1_filters
#define Out1_shape_i In_shape_i - conv1_kernel + 1
#define Out1_shape_j conv1_filters

#define conv2_kernel 5
#define conv2_filters 86
#define conv2_shape_i conv2_kernel*conv2_filters
#define conv2_shape_j Out1_shape_j
#define bias2_len conv2_filters
#define Out2_shape_i Out1_shape_i - conv2_kernel + 1
#define Out2_shape_j conv2_filters

#define dense4_shape_i (Out2_shape_i)*Out2_shape_j
#define dense4_shape_j classes
#define bias4_len classes

#define scaler_mean_len In_shape_j
#define scaler_std_len In_shape_j

#define buffer_size In_shape_j*4
#define samples_num 50

#define UART_MEM_BASE_ADDR  XPAR_XUARTPS_0_BASEADDR

char Uart_Menu(void);
```

```
template <typename M>
void InitMatrixFromFile(DenseBase<M>& A);

template <typename M>
void InitMatrixZeros(DenseBase<M>& A);

int main()
{
  char c;
  bool isRealTimeEnabled = true;
  float sum, max, f;

  u8* buffer = new u8[buffer_size];
  float income_line[In_shape_j];

  XTime tStart, tEnd;

  //Create a map to the memory where c++ array exists
  Map<Array<float, 1, In_shape_j>> Input_c(income_line);

  Matrix<float, In_shape_i, In_shape_j> Input;

  Matrix<float, conv1_shape_i, conv1_shape_j> conv1;
  Matrix<float, 1, bias1_len> bias1;
  Matrix<float, Out1_shape_i, Out1_shape_j> Out1;

  Matrix<float, conv2_shape_i, conv2_shape_j> conv2;
  Matrix<float, 1, bias2_len> bias2;
  Matrix<float, Out2_shape_i, Out2_shape_j, RowMajor> Out2;

  //Create a map to the memory that can read Out2 matrix as flatten
  Map<Matrix<float, 1, dense4_shape_i>> Out3(Out2.data(), Out2.size());

  Matrix<float, dense4_shape_i, dense4_shape_j> dense4;
  Matrix<float, 1, bias4_len> bias4;

  Matrix<float, 1, classes> Output;
  MatrixXf::Index maxCol;

  Matrix<float, 1, scaler_mean_len> Scaler_mean;
  Matrix<float, 1, scaler_std_len> Scaler_std;

  //This matrix stores testset's samples for inference
  Matrix<float, samples_num*conv1_shape_i, conv1_shape_j> DataSetMatrix;

  while(1){
    //display message and return character//
    c = Uart_Menu();
    if (c == '1'){
      //============================================================//
      //===============START Input sample generation================//
      // This part fills with zeros each matrix of the CNN including the
     Scaler.
    // Values for each matrix are 32-bit float numbers.
    print("\nStarting ...\n\r");
    InitMatrixZeros(Input);            //
    print("Conv1 weights and biases loaded with zeros ... ok\n\r");
    InitMatrixZeros(conv1);
    InitMatrixZeros(bias1);
    print("Conv1 weights and biases loaded with zeros ... ok\n\r");
    InitMatrixZeros(conv2);
    InitMatrixZeros(bias2);
    print("Conv2 weights and biases loaded with zeros ... ok\n\r");
    InitMatrixZeros(dense4);
    InitMatrixZeros(bias4);
    print("Dense weights and biases loaded with zeros ... ok\n\r");
    InitMatrixZeros(Scaler_mean);
    InitMatrixZeros(Scaler_std);
    print("Scaler mean and std values loaded with zeros ... ok\n\r");
    print("Finished ... ok!\n\r");
    //================================================================//
    //====================END Input sample generation=================//
  }
  else if(c == '2'){ //Initialize CNN from file - Transfer CNN
    //================================================================//
    //======================START COEFF FROM UART=====================//
    // This part transfers the trained CNN from PC to ARM byte by byte from
file.bin.
    // Values for each matrix are 32-bit float numbers.
    print("\nSend a .bin file\n\r");
    InitMatrixFromFile(conv1);        //Read 36,848 bytes for conv1 weights.
    InitMatrixFromFile(bias1);        //Read 376 bytes for conv1 biases.
    print("Conv1 weights and biases loaded...ok\n\r");
    InitMatrixFromFile(conv2);        //Read 161,680 bytes for conv2 weights
.
    InitMatrixFromFile(bias2);        //Read 344 bytes for conv2 biases.
    print("Conv2 weights and biases loaded...ok\n\r");
    InitMatrixFromFile(dense4);       //Read 72,240 bytes for dense weights.
    InitMatrixFromFile(bias4);        //Read 84 bytes for dense biases.
    print("Dense weights and biases loaded...ok\n\r");
    InitMatrixFromFile(Scaler_mean);      //Read 56 bytes for scaler mean.
    InitMatrixFromFile(Scaler_std);       //Read 56 bytes for scaler std.
    print("Scaler mean and std values loaded...ok\n\r");
    //================================================================//
    //=======================END COEFF FROM UART======================//
  }
  else if(c == '3'){//-------Choose to load samples from file
    isRealTimeEnabled = false;      //---realtime disabled
    print("\nSend a .bin file\n");

    InitMatrixFromFile(DataSetMatrix); //load dataset from uart binary file

    //--------Standardize values row by row using mean and std--------//
    for(int i=0; i<samples_num*In_shape_i; i++){
      DataSetMatrix.row(i) = (DataSetMatrix.row(i)-Scaler_mean).
cwiseQuotient(Scaler_std);
```

```
      }
    }
    else if(c=='4'){
      isRealTimeEnabled = true;
    }
    else if(c=='5'){
      if (isRealTimeEnabled == false){
        for (int i=0; i<samples_num; i++){
          //XTime_GetTime((XTime *) &tStart); //Uncomment to start timer
          Input = DataSetMatrix.block<In_shape_i, In_shape_j>(i*In_shape_i, 0)
;

          //----------------------Start Inference------------------------//
          //-------------First convolutional layer-----------//
          for (int j = 0; j < conv1_filters; j++)
            for (int i = 0; i < Out1_shape_i; i++)
              Out1(i,j) = Input.block<conv1_kernel, In_shape_j>(i, 0).
cwiseProduct(conv1.block<conv1_kernel, conv1_shape_j>(j * conv1_kernel,0))
.sum() + bias1(j);
          //relu activation function
          Out1 = Out1.cwiseMax(0);

          //-------------Second convolutional layer----------//
          for (int j = 0; j < conv2_filters; j++)
            for (int i = 0; i < Out2_shape_i; i++)
              Out2(i, j) = Out1.block<conv2_kernel, Out1_shape_j>(i, 0).
cwiseProduct(conv2.block<conv2_kernel, conv2_shape_j>(j * conv2_kernel, 0)
).sum() + bias2(j);

          //relu activation function
          Out2 = Out2.cwiseMax(0);

          //-------------Flatten layer-----------------------//
          Map<Matrix<float, 1, dense4_shape_i>> Out3(Out2.data(), Out2.size())
;
          //-------------Fully connected layer---------------//
          Output = Out3 * dense4 + bias4;

          //softmax activation function
          Output = Output.array().exp();
          sum = Output.sum();
          Output=Output/sum;

          max = Output.maxCoeff(&maxCol);

          cout << "sample" << i << "\tclass: " <<  maxCol << "\tpossibility: "
 << max << endl;
          XTime_GetTime((XTime *) &tEnd);
          f=1.0*(tEnd - tStart) / (COUNTS_PER_SECOND/1000000);
          cout << "Completed\n\rIt took: "<< f << "us"<< endl;

          //=====================================================//
          //break;
      }
    }
    else{
      print("Open UART\n\r");
      XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      sleep(3);
      while(XUartPs_IsReceiveData(UART_MEM_BASE_ADDR)){

        XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      }
      print("waiting bytes\n");

      while (1){

        //XTime_GetTime((XTime *) &tStart);

        Input.block<In_shape_i-1, In_shape_j>(0, 0)=Input.block<In_shape_i
-1, In_shape_j>(1, 0);

        for(int i=0; i<buffer_size; i++){
          *(buffer+i)=XUartPs_RecvByte(UART_MEM_BASE_ADDR);
        }
        memcpy(&income_line, buffer, In_shape_j*sizeof(float));
        Input.row(In_shape_i-1) = Input_c;
        Input.row(In_shape_i-1) = (Input.row(In_shape_i-1)-Scaler_mean).
cwiseQuotient(Scaler_std);

        //---------------------Start Inference--------------------------//
        //-------------First convolutional layer-----------//
        for (int j = 0; j < conv1_filters; j++)
          for (int i = 0; i < Out1_shape_i; i++)
            Out1(i,j) = Input.block<conv1_kernel, In_shape_j>(i, 0).
cwiseProduct(conv1.block<conv1_kernel, conv1_shape_j>(j * conv1_kernel,0))
.sum() + bias1(j);

        //relu activation function
        Out1 = Out1.cwiseMax(0);

        //-------------Second convolutional layer----------//
        for (int j = 0; j < conv2_filters; j++)
          for (int i = 0; i < Out2_shape_i; i++)
            Out2(i, j) = Out1.block<conv2_kernel, Out1_shape_j>(i, 0).
cwiseProduct(conv2.block<conv2_kernel, conv2_shape_j>(j * conv2_kernel, 0)
).sum() + bias2(j);

        //relu activation function
        Out2 = Out2.cwiseMax(0);

        //-------------Flatten layer-----------------------//
        Map<Matrix<float, 1, dense4_shape_i>> Out3(Out2.data(), Out2.size())
;
        //-------------Fully connected layer---------------//
        Output = Out3 * dense4 + bias4;
```

```
          //softmax activation function
          Output = Output.array().exp();
          sum = Output.sum();
          Output=Output/sum;

          max = Output.maxCoeff(&maxCol);

          cout << "class: " <<  maxCol << "\tpossibility: " << max << endl;
          //XTime_GetTime((XTime *) &tEnd);
          //f=1.0*(tEnd - tStart) / (COUNTS_PER_SECOND/1000000);
          //cout << "Completed\n\rIt took: "<< f << "us"<< endl;
          //=========================================================//
          //break;
        }
      }
    }
    else if(c=='6'){
      print("Okay, exiting...\n\r");
      break;
    }
  }
}

/* Uart_Menu
 * Prints a menu to terminal for the user and reads a byte from UART.
 * Return: a char value e.g. the key from keyboard tha was sended from
    terminal.
 */
char Uart_Menu(void){
  //Print Message
  print("\n\n\r=================Inference===================\n\r");
  print("What would you like to do?\n\r");
  print("1. Generate CNN weights randomly--for test only\n\r");
  print("2. Load CNN weights from .bin file\n\r");
  print("3. Load sample(s) from .bin file\n\r");
  print("4. Enable realtime acqusition\n\r");
  print("5. Run Inference\n\r");
  print("6. Exit\n\r");

  return XUartPs_RecvByte(UART_MEM_BASE_ADDR);
}

/*  InitMatrixFromFile
 *  This function reads bytes from UART and cast them as a 32-bit float
 *  in order to fill an Eigen Matrix.
 *  Argument: An Eigen Matrix.
*/
template <typename M>
void InitMatrixFromFile(DenseBase<M>& A){
  u8 buffer[4];
  float tmp;
  //loop through each matrix's element
  for (int i = 0; i < A.rows(); i++) {
    for (int j = 0; j < A.cols(); j++) {
      //ARM reads 4 bytes from UART's FIFO
      buffer[0]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      buffer[1]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      buffer[2]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);
      buffer[3]=XUartPs_RecvByte(UART_MEM_BASE_ADDR);

      //cast buffer[4] to a 32-bit float number
      memcpy(&tmp, &buffer, sizeof(float));

      A(i,j)=tmp; //copy tmp to matrix
    }
  }
}

/*  InitMatrixZeros
 *  This function fills an Eigen Matrix with zeros.
 *  Argument: An Eigen Matrix.
*/
template <typename M>
void InitMatrixZeros(DenseBase<M>& A){
  for (int i = 0; i < A.rows(); i++) {
    for (int j = 0; j < A.cols(); j++) {
      A(i,j)=0;
    }
  }
}
```

REFERENCES

[1] Wikipedia Contributors, “List of sign languages,” *Wikipedia*, Apr. 19, 2019. https://en.wikipedia.org/wiki/List_of_sign_languages (accessed Oct. 11, 2021).

[2] S. Mitra and T. Acharya, “Gesture Recognition: A Survey,” *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 3, pp. 311–324, May 2007, doi: 10.1109/tsmcc.2007.893280.

[3] E. Costello, *Random House Webster’s concise American Sign Language dictionary*. New York: Bantam, 2002.

[4] M. S. Kibbanahalli Shivalingappa, H. Ben Abdessalem, and C. Frasson, “Real-Time Gesture Recognition Using Deep Learning Towards Alzheimer’s Disease Applications,” *Brain Function Assessment in Learning*, pp. 75–86, 2020, doi: 10.1007/978-3-030-60735-7_8.

[5] B. Hudgins, P. Parker, and R. N. Scott, “A new strategy for multifunction myoelectric control,” *IEEE Transactions on Biomedical Engineering*, vol. 40, no. 1, pp. 82–94, 1993, doi: 10.1109/10.204774.

[6] Wikipedia Contributors, “Real-time computing,” *Wikipedia*, Apr. 29, 2019. https://en.wikipedia.org/wiki/Real-time_computing (accessed Oct. 11, 2021).

[7] M. I. Sadek, M. N. Mikhael, and H. A. Mansour, “A new approach for designing a smart glove for Arabic Sign Language Recognition system based on the statistical analysis of the Sign Language,” *2017 34th National Radio Science Conference (NRSC)*, Mar. 2017, doi: 10.1109/nrsc.2017.7893499.

[8] R. M. McGuire, J. Hernandez-Rebollar, T. Starner, V. Henderson, H. Brashear, and D. S. Ross, “Towards a one-way American Sign Language translator,” *Sixth IEEE International Conference on Automatic Face and Gesture Recognition, 2004. Proceedings.*, doi: 10.1109/afgr.2004.1301602.

[9] P. Vijayalakshmi and M. Aarthi, “Sign language to speech conversion,” *2016 International Conference on Recent Trends in Information Technology (ICRTIT)*, Apr. 2016, doi: 10.1109/icrtit.2016.7569545.

[10] “Zybo Z7 Reference Manual - Digilent Reference,” *digilent.com*. https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual (accessed Apr. 30, 2022).

[11] K. Kudrinko, E. Flavin, X. Zhu, and Q. Li, “Wearable Sensor-Based Sign Language Recognition: A Comprehensive Review,” *IEEE Reviews in Biomedical Engineering*, vol. 14, pp. 82–97, 2021, doi: 10.1109/rbme.2020.3019769.

[12] F. Chollet, *Deep Learning with Python.* Shelter Island (New York, Estados Unidos): Manning, Cop, 2018.

[13] A. Dertat, “Applied Deep Learning - Part 3: Autoencoders,” *Medium*, Oct. 03, 2017. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[14] J. Zhai, S. Zhang, J. Chen and Q. He, ”Autoencoder and Its Various Variants,” *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2018, pp. 415-419, doi: 10.1109/SMC.2018.00080.

[15] F. Moreno-Vera, ”Performing Deep Recurrent Double Q-Learning for Atari Games,” *2019 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, 2019, pp. 1-4, doi: 10.1109/LA-CCI47412.2019.9036763.

[16] K. Nyuytiymbiy, “Parameters and Hyperparameters in Machine Learning and Deep Learning,” *Medium*, Apr. 05, 2021. https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac.

[17] Pitsis, George. (2018). *Design and Implementation of an FPGA-Based Convolutional Neural Network Accelerator.*

[18] E. Charniak, *Introduction to deep learning.* Cambridge, Ma: Mit Press, 2018.

[19] B. Ding, H. Qian and J. Zhou, ”Activation functions and their characteristics in deep neural networks,” *2018 Chinese Control And Decision Conference (CCDC)*, 2018, pp. 1836-1841, doi: 10.1109/CCDC.2018.8407425.

[20] Z. Li, H. Li, X. Jiang, B. Chen, Y. Zhang and G. Du, ”Efficient FPGA Implementation of Softmax Function for DNN Applications,” *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018, pp. 212-216, doi: 10.1109/ICASID.2018.8693206.

[21] S. Verma, “Multi-Label Image Classification with Neural Network | Keras,” *Medium*, Oct. 05, 2021. https://towardsdatascience.com/multi-label-image-classification-with-neural-network-keras-ddc1ab1afede (accessed Oct. 24, 2021).

[22] K. O. Jimoh, A. O. Ajayi, and I. K. Ogundoyin, “Template Matching Based Sign Language Recognition System for Android Devices,” *FUOYE Journal of Engineering and Technology*, vol. 5, no. 1, Mar. 2020, doi: 10.46792/fuoyejet.v5i1.465.

[23] M. A. Ahmed, B. B. Zaidan, A. A. Zaidan, M. M. Salih, and M. M. bin Lakulu, “A Review on Systems-Based Sensory Gloves for Sign Language Recognition State of the Art between 2007 and 2017,” *Sensors*, vol. 18, no. 7, p. 2208, Jul. 2018, doi: 10.3390/s18072208.

[24] S. Jiang et al., “Feasibility of Wrist-Worn, Real-Time Hand, and Surface Gesture Recognition via sEMG and IMU Sensing,” *IEEE Transactions on Industrial Informatics*, vol. 14, no. 8, pp. 3376–3385, Aug. 2018, doi: 10.1109/TII.2017.2779814.

[25] A. Wadhawan and P. Kumar, “Sign Language Recognition Systems: A Decade Systematic Literature Review,” *Archives of Computational Methods in Engineering*, Dec. 2019, doi: 10.1007/s11831-019-09384-2.

[26] [63]Q. Munib, M. Habeeb, B. Takruri, and H. A. Al-Malik, “American sign language (ASL) recognition based on Hough transform and neural networks,” *Expert Systems with Applications*, vol. 32, no. 1, pp. 24–37, Jan. 2007, doi: 10.1016/j.eswa.2005.11.018.

[27] W. Tangsuksant, S. Adhan and C. Pintavirooj, ”American Sign Language recognition by using 3D geometric invariant feature and ANN classification,” *The 7th 2014 Biomedical Engineering International Conference*, 2014, pp. 1-5, doi: 10.1109/BMEiCON.2014.7017372.

[28] M. M. Islam, S. Siddiqua and J. Afnan, ”Real time Hand Gesture Recognition using different algorithms based on American Sign Language,” *2017 IEEE International Conference on Imaging*, Vision & Pattern Recognition (icIVPR), 2017, pp. 1-6, doi: 10.1109/ICIVPR.2017.7890854.

[29] M. Zamani and H. R. Kanan, ”Saliency based alphabet and numbers of American sign language recognition using linear feature extraction,” 2014 4th International Conference on Computer and Knowledge Engineering (ICCKE), 2014, pp. 398-403, doi: 10.1109/ICCKE.2014.6993442.

[30] P. V. V. Kishore, M. V. D. Prasad, D. A. Kumar and A. S. C. S. Sastry, ”Optical Flow Hand Tracking and Active Contour Hand Shape Features for Continuous Sign Language Recognition with Artificial Neural Networks,” *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, 2016, pp. 346-351, doi: 10.1109/IACC.2016.71.

[31] D. Kelly, J. Reilly Delannoy, J. Mc Donald, and C. Markham, “A framework for continuous multimodal sign language recognition,” Proceedings of the 2009 international conference on Multimodal interfaces - ICMI-MLMI ’09, 2009, doi: 10.1145/1647314.1647387.

[32] B. G. Lee and S. M. Lee, “Smart Wearable Hand Device for Sign Language Interpretation System With Sensors Fusion,” *IEEE Sensors Journal*, vol. 18, no. 3, pp. 1224–1232, Feb. 2018, doi: 10.1109/jsen.2017.2779466.

[33] S. Yin et al., “Research on Gesture Recognition Technology of Data Glove Based on Joint Algorithm,” *Proceedings of the 2018 International Conference on Mechanical, Electronic, Control and Automation Engineering (MECAE 2018)*, 2018, doi: 10.2991/mecae-18.2018.8.

[34] S. P. Y. Jane and S. Sasidhar, “Sign Language Interpreter: Classification of Forearm EMG and IMU Signals for Signing Exact English *,” *2018 IEEE 14th International Conference on Control and Automation (ICCA)*, Jun. 2018, doi: 10.1109/icca.2018.8444266.

[35] J. Galka, M. Masior, M. Zaborski, and K. Barczewska, “Inertial Motion Sensing Glove for Sign Language Gesture Acquisition and Recognition,” *IEEE Sensors Journal*, vol. 16, no. 16, pp. 6310–6316, Aug. 2016, doi: 10.1109/jsen.2016.2583542.

[36] K. Li, Z. Zhou, and C.-H. Lee, “Sign Transition Modeling and a Scalable Solution to Continuous Sign Language Recognition for Real-World Applications,” *ACM Transactions on Accessible Computing*, vol. 8, no. 2, pp. 1–23, Jan. 2016, doi: 10.1145/2850421.

[37] N. Tubaiz, T. Shanableh, and K. Assaleh, “Glove-Based Continuous Arabic Sign Language Recognition in User-Dependent Mode,” *IEEE Transactions on Human-Machine Systems*, vol. 45, no. 4, pp. 526–533, Aug. 2015, doi: 10.1109/thms.2015.2406692.

[38] Y. Li, X. Chen, X. Zhang, K. Wang, and J. Yang, “Interpreting sign components from accelerometer and sEMG data for automatic sign language recognition,” *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, Aug. 2011, doi: 10.1109/iembs.2011.6090910.

[39] F. Wang, S. Zhao, X. Zhou, C. Li, M. Li, and Z. Zeng, “An Recognition–Verification Mechanism for Real-Time Chinese Sign Language Recognition Based on Multi-Information Fusion,” *Sensors*, vol. 19, no. 11, p. 2495, May 2019, doi: 10.3390/s19112495.

[40] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *arXiv.org*, 2014. https://arxiv.org/abs/1409.1556.

[41] S. B. J, “A friendly Introduction to Siamese Networks,” *Medium*, Jan. 29, 2021. https://towardsdatascience.com/a-friendly-introduction-to-siamese-networks-85ab17522942.

[42] P. Deekshith chary, Dr.R.P.Singh ”Review on Advanced Machine Learning Model: Scikit-Learn” *International Journal of Scientific Research and Engineering Development (IJSRED)*, vol. 3, no. 4 pp. 526-529.

[43] R. Fatmi, S. Rashad, and R. Integlia, “Comparing ANN, SVM, and HMM based Machine Learning Methods for American Sign Language Recognition using Wearable Motion Sensors,” *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2019, doi: 10.1109/ccwc.2019.8666491.

[44] L. Li, S. Jiang, P. B. Shull, and G. Gu, “SkinGest: artificial skin for gesture recognition via filmy stretchable strain sensors,” *Advanced Robotics*, vol. 32, no. 21, pp. 1112–1121, Jul. 2018, doi: 10.1080/01691864.2018.1490666.

[45] J. Wu, L. Sun, and R. Jafari, “A Wearable System for Recognizing American Sign Language in Real-Time Using IMU and Surface EMG Sensors,” *IEEE Journal of Biomedical and Health Informatics*, vol. 20, no. 5, pp. 1281–1290, Sep. 2016, doi: 10.1109/jbhi.2016.2598302.

[46] C. Mummadi et al., “Real-Time and Embedded Detection of Hand Gestures with an IMU-Based Glove,” *Informatics*, vol. 5, no. 2, p. 28, Jun. 2018, doi: 10.3390/informatics5020028.

[47] C. Savur and F. Sahin, “American Sign Language Recognition system by using surface EMG signal,” *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct. 2016, doi: 10.1109/smc.2016.7844675.

[48] J. Wu, Y. Li, and Y. Ma, “Comparison of XGBoost and the Neural Network model on the class-balanced datasets,” IEEE Xplore, Nov. 01, 2021. https://ieeexplore.ieee.org/document/9647373 (accessed Sep. 02, 2022).

[49] S. Y. Mudugandla, “10 Hyperparameter optimization frameworks.,” *Medium*, Feb. 15, 2021. https://towardsdatascience.com/10-hyperparameter-optimization-frameworks-8bc87bc8b7e3 (accessed Apr. 09, 2022).

[50] M. vd Boom, “ATOM: A Python package for fast exploration of machine learning pipelines,” *Medium*, Sep. 18, 2021. https://towardsdatascience.com/atom-a-python-package-for-fast-exploration-of-machine-learning-pipelines-653956a16e7b (accessed Apr. 09, 2022).

[51] scikit-learn, “scikit-learn: machine learning in Python — scikit-learn 0.20.3 documentation,” *Scikit-learn.org*, 2019. https://scikit-learn.org/stable/

[52] S. Rong and Z. Bao-wen, “The research of regression model in machine learning field,” *MATEC Web of Conferences*, vol. 176, p. 01033, 2018, doi: 10.1051/matecconf/201817601033.

[53] M. Schonlau and R. Y. Zou, “The random forest algorithm for statistical learning,” *The Stata Journal: Promoting communications on statistics and Stata*, vol. 20, no. 1, pp. 3–29, Mar. 2020, doi: 10.1177/1536867x20909688.

[54] A. Tharwat, T. Gaber, A. Ibrahim, and A. E. Hassanien, “Linear discriminant analysis: A detailed tutorial,” *AI Communications*, vol. 30, no. 2, pp. 169–190, May 2017, doi: 10.3233/aic-170729.

[55] S. Suthaharan, “Support Vector Machine,” *Machine Learning Models and Algorithms for Big Data Classification*, vol. 36, pp. 207–235, 2016, doi: 10.1007/978-1-4899-7641-3_9.

[56] C. Wade, *Hands-on gradient boosting with XGBoost and scikit-learn : perform accessible machine learning and extreme gradient boosting with python.* Birmingham: Packt Publishing, 2020.

[57] W. Gao et al., “HandTalker: A Multimodal Dialog System Using Sign Language and 3-D Virtual Human,” *Advances in Multimodal Interfaces — ICMI 2000*, pp. 564–571, 2000, doi: 10.1007/3-540-40063-x_74.

[58] R. M. McGuire, J. Hernandez-Rebollar, T. Starner, V. Henderson, H. Brashear and D. S. Ross, ”Towards a one-way American sign language translator,” *Sixth IEEE International Conference on Automatic Face and Gesture Recognition*, 2004. Proceedings., 2004, pp. 620-625, doi: 10.1109/AFGR.2004.1301602.

[59] W. Gao, Gaolin Fang, Debin Zhao and Yiqiang Chen, ”Transition movement models for large vocabulary continuous sign language recognition,” *Sixth IEEE International Conference on Automatic Face and Gesture Recognition*, 2004. Proceedings., 2004, pp. 553-558, doi: 10.1109/AFGR.2004.1301591.

[60] Y. Li, X. Chen, X. Zhang, K. Wang and Z. J. Wang, ”A Sign-Component-Based Framework for Chinese Sign Language Recognition Using Accelerometer and sEMG Data,” in *IEEE Transactions on Biomedical Engineering*, vol. 59, no. 10, pp. 2695-2704, Oct. 2012, doi: 10.1109/TBME.2012.2190734.

[61] M. Maebatake, I. Suzuki, M. Nishida, Y. Horiuchi and S. Kuroiwa, ”Sign Language Recognition Based on Position and Movement Using Multi-Stream HMM,” *2008 Second International Symposium on Universal Communication*, 2008, pp. 478-481, doi: 10.1109/ISUC.2008.56.

[62] Rung-Huei Liang and Ming Ouhyoung, ”A real-time continuous gesture recognition system for sign language,” *Proceedings Third IEEE International Conference on Automatic Face and Gesture Recognition*, 1998, pp. 558-567, doi: 10.1109/AFGR.1998.671007.

[63] Gaolin Fang and W. Gao, ”A SRN/HMM system for signer-independent continuous sign language recognition,” *Proceedings of Fifth IEEE International Conference on Automatic Face Gesture Recognition*, 2002, pp. 312-317, doi: 10.1109/AFGR.2002.1004172.

[64] W. Gao, G. Fang, D. Zhao, and Y. Chen, “A Chinese sign language recognition system based on SOFM/SRN/HMM,” *Pattern Recognition*, vol. 37, no. 12, pp. 2389–2402, Dec. 2004, doi: 10.1016/j.patcog.2004.04.008.

[65] W. GAO, J. MA, J. WU, and C. WANG, “SIGN LANGUAGE RECOGNITION BASED ON HMM/ANN/DP,” International Journal of Pattern Recognition and Artificial Intelligence, vol. 14, no. 05, pp. 587–602, Aug. 2000, doi: 10.1142/s0218001400000386.

[66] L. Yang and A. Shami, “On hyperparameter optimization of machine learning algorithms: Theory and practice,” *Neurocomputing*, vol. 415, pp. 295–316, Nov. 2020, doi: 10.1016/j.neucom.2020.07.061.

[67] M. vd Boom, “Multiclass classification - ATOM,” *tvdboom.github.io*. https://tvdboom.github.io/ATOM/v4.13/examples/multiclass_classification/ (accessed Apr. 09, 2022).

[68] W. Koehrsen, “A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning,” *Medium*, Jun. 24, 2018. https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f.

[69] L. Yen, “An Introduction to the Bootstrap Method,” *Medium*, Jan. 26, 2019. https://towardsdatascience.com/an-introduction-to-the-bootstrap-method-58bcb51b4d60

[70] R. Wood, “Deep Learning Primer with Keras,” *Medium*, Feb. 03, 2021. https://towardsdatascience.com/deep-learning-primer-with-keras-3958705882c5 (accessed Apr. 10, 2022).

[71] Keras, “Home - Keras Documentation,” Keras.io, 2019. https://keras.io/

[72] “Eigen,” *eigen.tuxfamily.org.* https://eigen.tuxfamily.org/index.php?title=Main_Page (accessed Apr. 30, 2022).

[73] A. Ltd, “SIMD ISAs | Neon,” *Arm Developer*. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon (accessed Jan. 30, 2022).