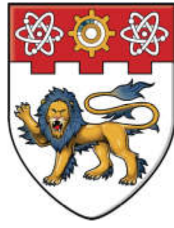

NANYANG
TECHNOLOGICAL
UNIVERSITY

Research Project as part of

Exchange Semester from ETH Zurich

EXPLORING GESTURE RECOGNITION USING DEEP LEARNING BASED ON GOOGLE SOLI

FELIX NICOLAS GRAULE
BSc in Electrical Engineering and Information Technology
N1604716B

Supervised by Prof. Dr. Junsong Yuan
SCHOOL OF ELECTRICAL AND ELECTRONICS ENGINEERING

MAY 2017

Abstract

The goal of this project was to gather first experiences in the field of artificial intelligence, gain an in-depth understanding of the deep learning gesture recognition pipeline using the Google Soli sensor proposed by S. Wang and J. Song (referring to [1]) and eventually improve its performance. The paper introduces a significant improvement over former pipelines while allowing a bigger set of gestures. The main use-case of Soli is acting as an input device for wearables, as it performs well as an human-computer interaction in scenarios with limited space and computing power.

To achieve this, I reproduced the findings from the mentioned paper to get familiar with the code and network structure. I first studied basic concepts in machine learning to understand the mathematical background of the method. Next, I learned how to evaluate the pre-trained network using Lua and Torch. This required fixing several bugs in the code published on GitHub. As all fixes were written back to the repository, the project contributed to further work on Soli gesture recognition pipelines. The resulting overall recognition accuracy was significantly lower than proposed by the paper. What caused this discrepancy should be investigated in a future project.

This report should further be used as an introductory overview about the paper and as instruction how to use the proposed network. Hence, it includes actual code and explanations on how exactly the paper could be reproduced. The project was conducted as part of an exchange semester in Nanyang Technological University (NTU).

Table of Contents

Abstract I
Table of Contents II

1 Introduction **1**
1.1 Motivation 1
1.2 Goal 1
1.3 Gesture Recognition in Wearables 2

2 Background and Theory **3**
2.1 Google Soli 3
2.1.1 Hardware 4
2.1.2 Signal Processing and Transformation 5
2.1.3 Original Gesture Recognition 7
2.1.4 Proposed Gesture Recognition 8
2.2 Neural Networks 10
2.2.1 CNN, RNN and LSTM 11
2.2.2 Relevant Layer Types 13
2.2.3 Relevant Software Frameworks 14

3 Reproducing the Paper **16**
3.1 Experimental Setup 16
3.2 Preprocessing 16
3.3 Layers of Existing Network 17
3.4 Using the Pre-trained Network 18
3.5 Fixed Bugs in the Code 19

4 Results **22**

4.1 Gesture Recognition Accuracy 22
4.2 Contribution to Paper 24

5 Conclusion **25**
5.1 Conclusion 25
5.2 Future Work 26
List of Figures 27

References **28**

Chapter 1

Introduction

1.1 Motivation

The key motivation for this project was my fascination for learning algorithms. Developing two image analysis algorithms using hard-coded decision rules during former projects showed me that true robustness can only be achieved through adapting threshold structures. I am amazed, how neural networks solve this issue by storing knowledge about a problem inside their weights and how well they perform as classifiers for highly non-linear problems.

Furthermore, I have a strong internal drive to do research and push the boundaries of technology. Working on this project gave me the opportunity to do so while allowing me to directly apply techniques studied in an introductory course on neural networks within Nanyang Technological University (NTU). This drastically improved my understanding of the course.

1.2 Goal

Through this project, I want to make first experiences with artificial intelligence and machine learning in general and deep neural networks in particular. I want to learn how to design, train and optimize networks while challenging myself by contributing to cutting-edge research having nearly no previous knowledge in the field.

Additionally, I aim to contribute to the underlaying paper (referring to [1]) and promote the developed pipeline. I want to help improving the performance of the proposed gesture recognition, thus further advancing the Soli sensor.

1.3 Gesture Recognition in Wearables

Wearables have gained increasing popularity, especially smartwatches and fitness equipment. These devices share a small display limiting interaction between user and device. While a wide range of approaches has been used to solve this (ranging from speech recognition to ultra-simplistic UI design), no approach achieved fully satisfying performance. Gesture recognition has the potential to take an important role in the final solution. Gestures do not rely on a display allowing them to be used in highly compact devices. They are also intuitive and can be designed to be more subtle than speech recognition.

Vision-based gesture recognition systems reach impressive accuracy in stationary situations, but often require calibration and direct line-of-sight. Methods based on electromagnetic field sensing provide poor spatial resolution, making accurate gesture recognition impossible. [1]

Recent millimeter-wave radar chips working in a high-frequency ultra-wide frequency band show potential for accurate short-range sensing while using little energy and space (see [2]). Combining high accuracy and not relying on direct line-of-sight, this new set of chips holds great potential.

Chapter 2

Background and Theory

2.1 Google Soli

The Advanced Technology And Project (ATAP) group within Google has recently revealed project Soli, which is a promising millimeter wave radar sensor that could be used in wearables (as explained in section 1.3). The sensor is described in great detail in a paper by the ATAP group at Google (please refer to [3]).

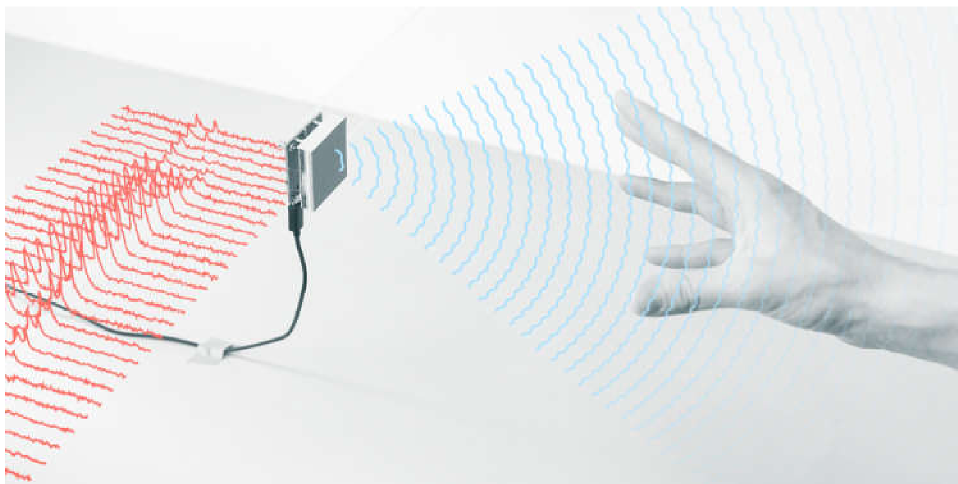

Figure 2.1: Overall principle of Soli with slow time signal (evolution of red lines), fast time (each of the red lines) signal and the hand scattering the radar waves

Figures in this chapter are taken from the papers [1] or [3] (if no other source is stated).

2.1.1 Hardware

The sensor is implemented on a single, small-size chip (see figure 2.2), including antennas and all single processing hardware on-chip. This was achieved through a drastic shrinking process over 18 months. The sensor takes advantage of recent improvements in silicon fabrication processes that allow integrated RF-chip design over 30 GHz and direct integration of antennas onto a chip. The miniaturization also drastically decreases cost and power consumption of the sensor. [3]

Figure 2.2: Radar chip (left) embedded on a development board

The ATAP group developed two distinct chips that implement Soli's function. Through the *Hardware Abstraction Layer* (HAL) depicted in figure 2.3, all higher-level software can be used for both sensors interchangeably. The sensors feature these important functions:

- Operating frequency of 60 GHz which allows small antennas
- Broad beam (150-degree) for full illumination of hand even at close distance
- Low noise and power consumption (300mW with duty cycling)
- Ability to detect sub-millimeter movement

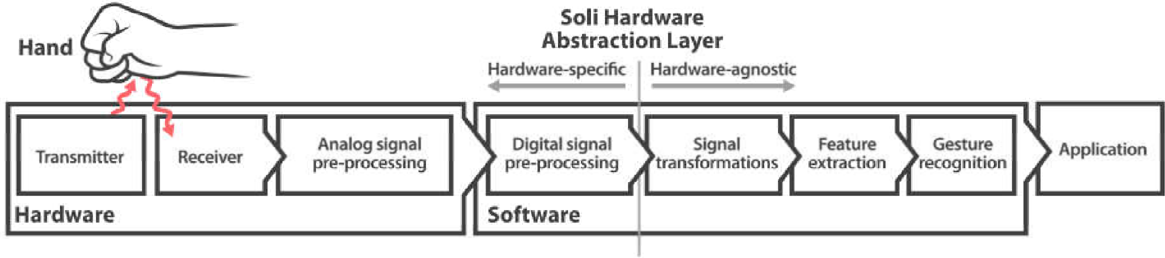

Figure 2.3: Overview of hard- and software components of Soli

2.1.2 Signal Processing and Transformation

The overall idea of Soli is that the chip emits radar waves modulated with a high frequency signal, which are partially reflected back to the sensor by the hand and then measured by the receiver antennas.

Figure 2.3 gives an overview about the different steps from the raw signal up to the actual recognition of a gesture. In my project, I mainly focused on the last two software steps: feature extraction and gesture recognition. Nevertheless, the lower parts of the pipeline are introduced in the following section to provide the reader with a better understanding of what is fed into the neural network.

The Soli pipeline models the fingers of the user's hand as five distinct scattering centers (as seen in figure 2.4) to reconstruct more information from the signal. Each of the fingers is assigned a radial distance $r_i(T)$ and a complex reflectivity $\rho_i(T)$. The overall signal is then interpreted as a superposition of all the reflected signals as described in equation 2.1.

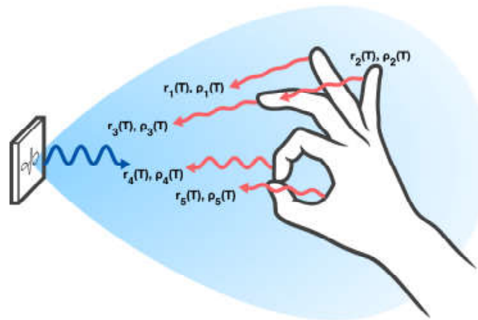

Figure 2.4: Overall principle: Radar signal illuminates hand, scattered waves are measured

$$y(r,T) = \sum_{i=1}^{N_{sc}} \rho_i(T)\delta(r - r_i(T)) \tag{2.1}$$

Unlike most classical radar systems, Soli does not rely on high spatial, but high temporal resolution. This is achieved by dividing the signal into two time scales: slow and fast time (the signal is chopped apart in constant intervals). We refer to the signal in between two cuts as the *fast time* signal. It carries information about the scattering center range and reflectivity at one point in time (even though it not exactly a point, but rather a short interval in time). On the other hand, the evolution of the fast time signal over time is called *slow time* signal. It carries information about the dynamics of the scattering centers, like their velocity and change in geometry. The different signals are depicted in figure 2.1, where the red lines represent the fast time signal. Multiple fast time signal are plotted along the slow signal time axis, which is why there is multiple red lines.

Through preprocessing and a range of transformations we end up with a set of signal representation shown in figure 2.5. The Range-Doppler image (RDI) representation (on the left in the lower right rectangle) is of special interest to us.

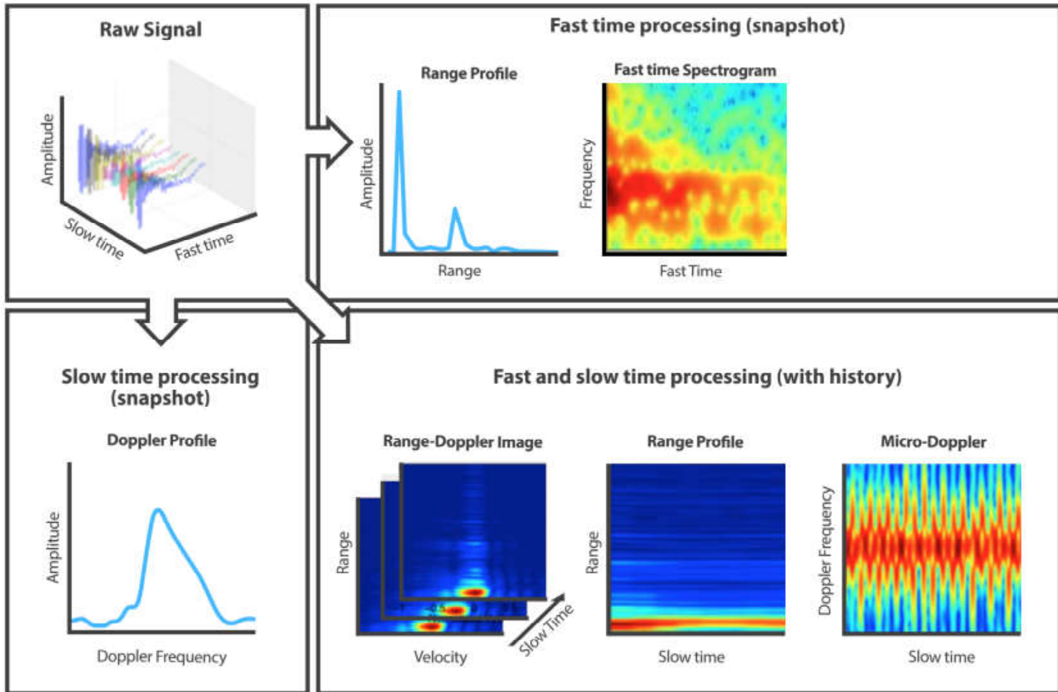

Figure 2.5: Different visual representations of the signal

The proposed recognition pipeline (referring to [1]) focuses entirely on the Range-Doppler images. In a RDI, we plot the intensity of the reflected light of a scattering center in terms of range from the sensor and velocity of the scattering center as shown in figure 2.6. The figure shows the reflections of two walking persons, where the left person walks with negative velocity (relative to the sensor's coordinates) and is closer to the sensor (smaller radial value). The right person walks with positive velocity, but it further away from the sensor.

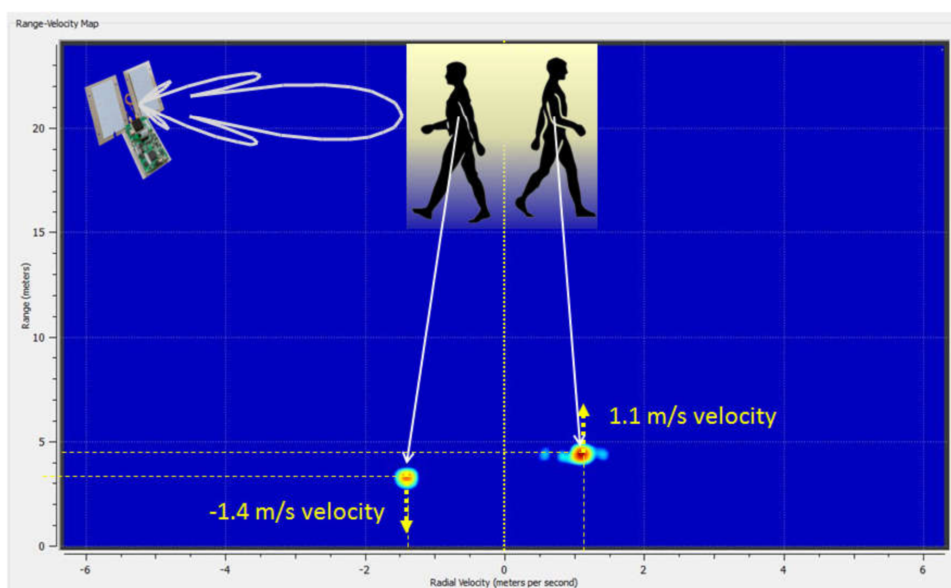

Figure 2.6: Explanation of Range-Doppler image
Source: http://ancortek.com/wp-content/uploads/2015/03/TwoPersonWalking.jpg

2.1.3 Original Gesture Recognition

The original approach to recognize gestures was based on manually selected features within the different signal representations shown in figure 2.5. The extracted features are low-dimensional and can be grouped into:

- Scattering center trackers
- Descriptors of physical RF measurement
- Data-centric machine learning

A detailed explanation on this does not fit the frame of this project, so please refer [3] for further information on this. The essence is that the original gesture recognition mechanism relied on a fairly high number of manually selected features based on different representations.

This method allows very accurate gesture recognition (> 90%) on a small set of highly distinctive gestures (four gestures, see figure 2.7).

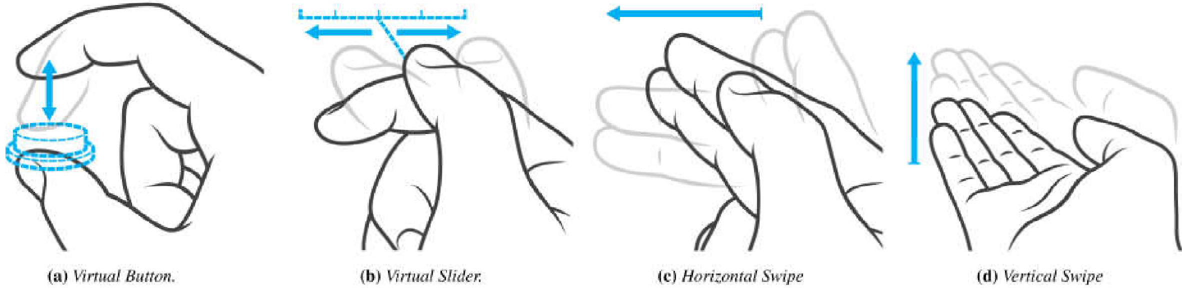

Figure 2.7: Four different gestures used in original version of Soli

2.1.4 Proposed Gesture Recognition

Contrary to the original approach, the gesture recognition introduced by S. Wang and J. Song is based on the Range-Doppler images only and uses automatically extracted features.

To achieve this, a *Convolutional Neural Network* (CNN) and a *Recurrent Neural Network* (RNN) are combined into one end-to-end model. The overall structure of the final network is shown in figure 2.8. The network structure itself is discussed in full detail in section 3.3.

The key idea is to use the CNN for feature extraction at each point in slow time and then use the RNN to model the dynamics of the scattering centers. The information learned (an intermediate representation of the Range-Doppler images) in every step is propagated along the time axis through *Long Short-Term Memory* (LSTM). Futher, in every time step a *Fully Connected* (FC) layer and a *Softmax* layer is used to give per-frame gesture predictions. During training, the final gesture recognition is used directly as optimization goal. Hence, no attempt to actually reconstruct the spatial gesture shape is made.

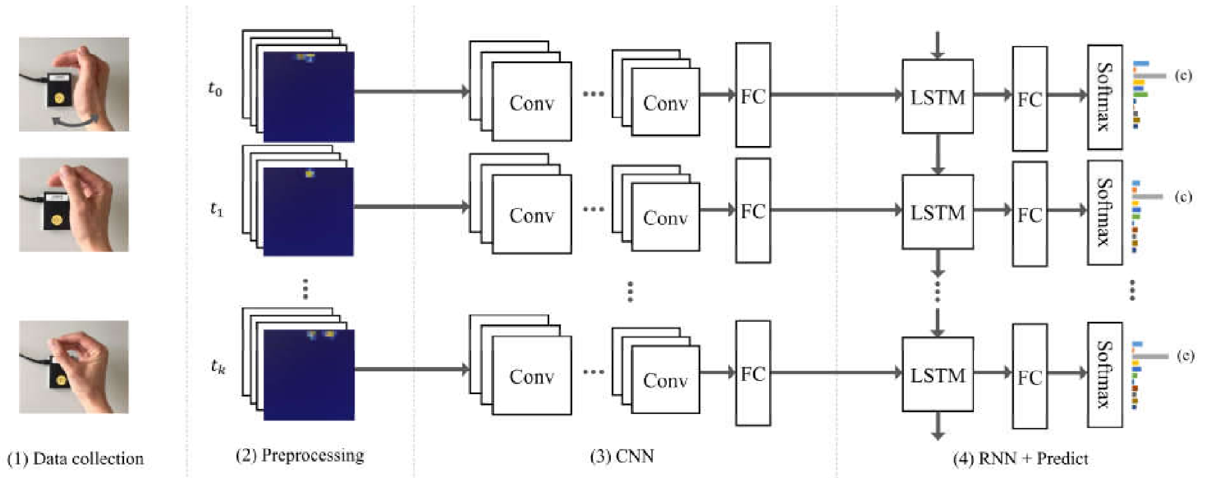

Figure 2.8: Overall structure of end-to-end model

As explained in section 2.1.2, the raw signal is divided into slow and fast time. To take in account the temporal information of the signal, we create one RDI for every time-step in fast time and then stack multiple RDIs along the slow time axis. Therefore, the network input consists of a stack of RDIs for every time-step in slow time (see (2) in figure 2.8). Each RDI has a resolution of only 32x32 pixels. The images are normalized before fed into the network, so every pixel corresponds to a value between 0 and 1. As we group four consecutive RDIs on one stack, every stack is a 3-dimensional tensor of size 4x32x32. We further group 16 of these tensors into one batch to train and evaluate the network in batch-mode. This is done frequently to exploit the parallel computing power of the GPU. So finally, the network takes a 4-dimensional tensor of size 16x4x32x32 as an input, which makes 65'536 elements per 16 time-steps in slow time. However, the items within one batch are used sequentially, so the highest network layer deals with a 3-dimensional tensor. The computational power required for recognizing a gesture hence depends on the gesture duration, the temporal resolution and eventual down-sampling.

The proposed pipeline achieves highly accurate recognition ($\approx$ 87%) over a large set of gestures (eleven gestures, see figure 2.9).

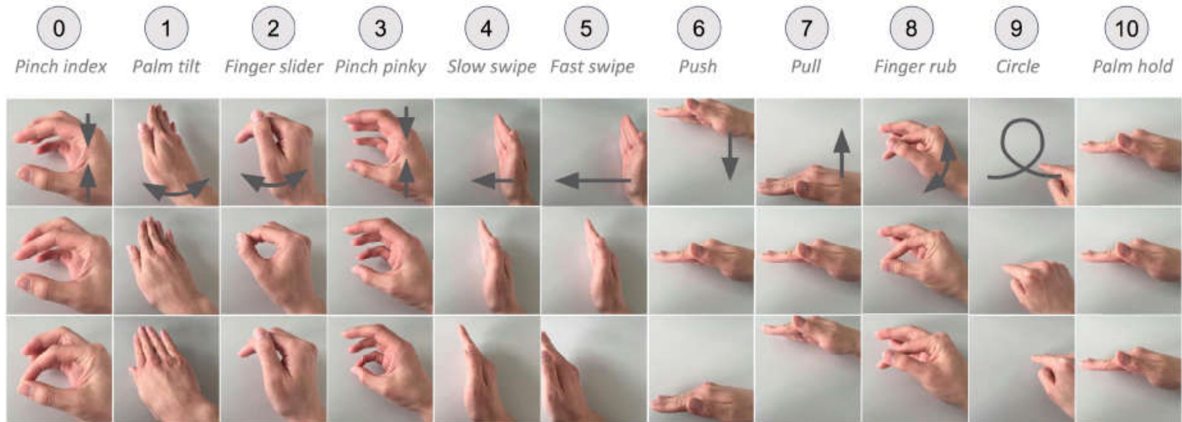

Figure 2.9: Eleven different gestures used in the newly proposed pipeline

2.2 Neural Networks

The working principle of neural networks is inspired by the human brain, where a large number of highly interconnected nodes work together. In neural networks, we refer to the nodes as *neurons*. They act as threshold units that fire when their input is high enough. Connections between nodes are weighted, meaning that different inputs to one neuron have varying contribution the total input signal. The intelligence of the network lies in the weights between the nodes. Hence, the network does not posses any intelligence or knowledge directly after initialization, it needs to learn first. This is achieved through training, during which we alter the weights in the network to get optimal results.

As neural networks are often used to classify inputs to a certain group (such as hand movement to a gesture in our case), super-vised learning is a widely used training method. During this procedure, we present the network with a manually classified input, look at its output (prediction) and adjust the weights according to the classification error.

The neural network seen in figure 2.10 only contains one hidden layer. Recently, networks with a larger number of hidden layers (called *deep neural networks*) gained popularity as they can solve highly non-linear classification problems. Especially the use of specialized multi-layer *perceptrons* with specific preprocessing attached lead to extraordinary results. An example for this are convolutional neural networks (see section 2.2.1).

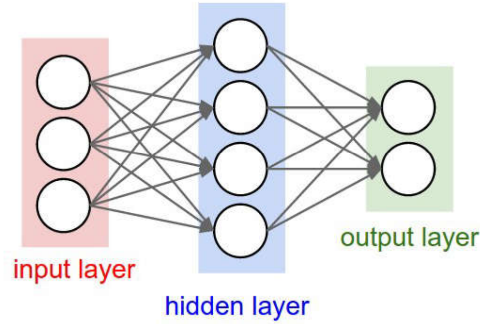

Figure 2.10: Simple example of a neural network [13]

Each neuron has an activation function that describes its input/output relation. A widely used activation function is the sigmoid function shown in equation 2.2, which normalizes the sum of all inputs to a value between 0 and 1. This guarantees that the weights inside the network cannot diverge during training.

$$\phi(v_i) = \frac{1}{1 + e^{-v_i}} \tag{2.2}$$

2.2.1 CNN, RNN and LSTM

As mentioned earlier, the proposed pipeline uses two types of networks (see figure 2.8).

The first type the input gets fed through is a *Convolutional Neural Network* (CNN). These networks are inspired by the visual cortex in animals and humans and widely used in image recognition and natural language processing. They are feed-forward networks, so data can only propagate from input to output. As the name suggests, the most important layers are convolutions. In a convolutional layer, each neuron uses an entire region of the input data as input instead of using only one pixel or the whole image. Through this, the spatial relations between neighboring pixels can be exploited. Further layers are used to lower the dimensionality of the data. An example for a CNN is shown in figure 2.11.

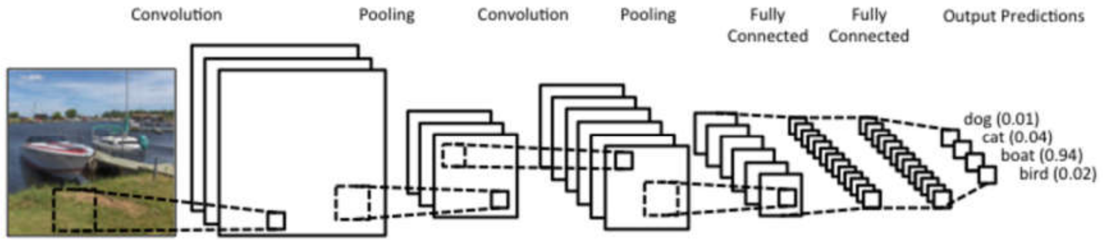

Figure 2.11: Example of a CNN [14]

The second type of network is a *Recurrent Neural Network* (RNN). Contrary to CNNs, these networks are not feed-forward, as the data travels in directed circles inside the network. Depending on what data is currently cycling through them, the network is in a different state. Like this, the network can capture temporal information through its dynamic behavior. An example for such a RNN is depicted in figure 2.12.

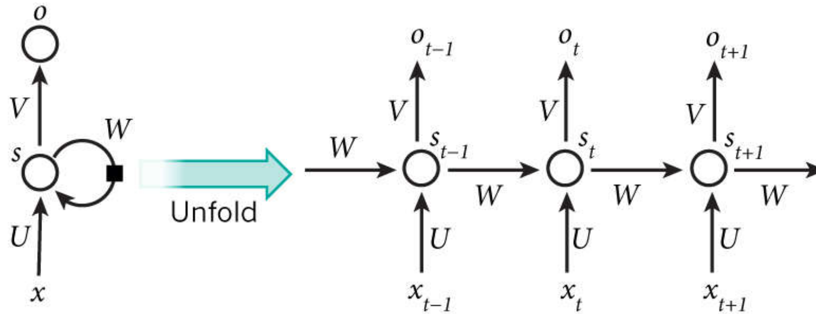

Figure 2.12: Example of a RNN (left) and how it can be unfolded in time (right) [15]

As classical RNNs also suffer from the vanishing gradient problem, they are often extended with *Long Short-Term Memory* (LSTM). They key idea behind LSTM units is that they store certain information over time without degrading its value by applying an activation function during each time-step. LSTM units often have between three and four gates to control the information flow through them. Adjusting these parameters is complex and requires knowledge of the problem that is to be solved. An example for a LSTM unit is displayed in figure 2.13.

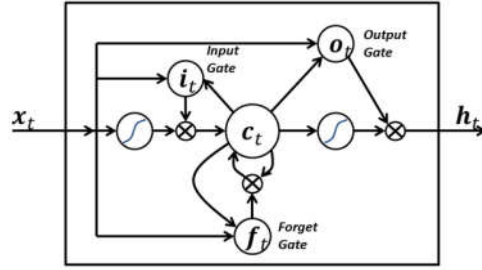

Figure 2.13: Example of a LSTM unit [16]

2.2.2 Relevant Layer Types

The following layers are important in the proposed pipeline and therefore briefly described. All these layers are part of the CNN and will be included into a *Sequential* container, so they are fully connected. The layer names correspond to the Torch framework, layers with equal functionality might have different names in other frameworks like TensorFlow or Theano.

Linear Represents a normal fully connected (FC) layer of neurons. The number of inputs, hidden units and outputs can be adjusted manually.

SpatialConvolution Takes a 3D tensor (consisting of multiple planes of 2D images) and applies a 2D convolution to every plane. The width and height of the convolution kernel can be adjusted manually, same for the step size in both dimensions.

ReLU Applies a rectifier linear unit (ReLU) element-wise to the input tensor. The activation function is defined as $f(x) = \max(0, x)$. The output is a tensor of same dimensionality.

Dropout Mask used during training with binary samples from a Bernoulli distribution with parameter p (default value is 0.5). Hence, every element in a tensor might be dropped with probability p. The remaining elements are scaled by $1/(1 - p)$. This layer helps to prevent co-adaption of neurons (see [4]). During evaluation, this layer has no effect at all.

SpatialDropout Same as Dropout, but assumes that two right-most dimensions of the input are spatial (3D).

LogSoftMax Applies the logsoftmax-function to an n-dimensional input tensor. Often used to get final output value of network. The logsoftmax-function is defined as

$$f_i(x) = \log\left(\frac{1}{ae^{x_i}}\right) \tag{2.3}$$

2.2.3 Relevant Software Frameworks

For the project, I used the following programming languages and software frameworks:

Lua A light-weight scripting language with support for object-oriented programming. Ideal for configuration, scripting and rapid-prototyping. This is exactly what we use it for, as we build the network and training environment using Lua. [5]

CUDA An application programming interface (API) provided by Nvidia to deploy heavily parallel tasks to the GPU. It can be used with C++ and gives the programmer direct access to the GPU's virtual instruction set and parallel computational elements. [6]

Torch A scientific computing framework for machine learning with emphasis on GPU support and high performance (based on Lua and C/CUDA). The most recent version of Troch is torch7. As many other frameworks, Torch consists of several packages [7]. The most important ones are listed here:

> **Neural Network package - nn** Contains the `Module` class from which all other modules inherit. Further contains the `Containers`, so classes like `Sequential` which is used as a box for fully connected, feed-forward network structures. [8]
>
> **Recurrent Neural Network package - rnn** Implements the `Recursor` module which is used as a container for recurrent networks, also including LSTM units. [9]
>
> **Cutorch** Back-end that brings CUDA support to Torch by substituting the normal `torch.FloatTensor` with the GPU-optimized `torch.CudaTensor` (available for all types besides float). [10]

Nvidia Cuda Deep Neural Network library- cuDNN Provides GPU-optimized versions of standard operations used in deep neural networks such as forward and backward convolution, pooling, normalization, and activation layers. Drastically decreases the overall training time of neural networks. [11]

GitHub A web-based version control repository widely used for distributed software development, especially open-source software. Allows developers to track bugs, come up with suggestions (through pull requests) and raise issues and questions on the code. [12]

Chapter 3

Reproducing the Paper

3.1 Experimental Setup

All experiments were performed on a shared, high-performance server in the Institute for Media Innovation (IMI) within Nanyang Technological University. The server was running Ubuntu 14.04 LTS with all the required dependencies (such as Torch, CUDA, MATLAB, and others). The hardware specification of the server goes as follows:

- CPU: Intel Core i7-5930K (Hexa-Core @ 3.50 GHz)
- Installed RAM: 64 GB (4 elements @ 2133 MHz)
- GPU: Nvidia Quadro K5200 (8GB GDDR5, 2304 CUDA cores @ 666 MHz)

This powerful setup allowed fast evaluation of the network ($<$ 5 min for evaluating over whole dataset). To allow simultaneous usage of the server, it was accessed through SSH tunneling using MobaXterm Personal Edition.

3.2 Preprocessing

The preprocessing code is written in Python and provided in the GitHub repository of the proposed paper ([1]). Hence, we simply need to run it from the terminal using bash.

Unpacking Data As all the RDIs were published in a HDF5 archive, they first need to be unpacked for further processing. This is done using the following code:

Listing 3.1: HDF5 to image

```
python pre/main.py --op image --file [dataset folder] --target
[target image folder] --channel 4 --originsize 32 --outsize 32
```

All the image data is saved to the directory specified in `[target image folder]`.

Creating Meanfile To arrive at a normalized version of the unpacked data, we need to find the mean for every image and store it into the `meanfile`. This is achieved using the following command:

Listing 3.2: Creating the meanfile

```
python pre/main.py --op mean --file [image folder] --target
[mean file name] --channel 4 --outsize 32
```

3.3 Layers of Existing Network

CNN The CNN is made up by the following layers:

(1): cudnn.SpatialConvolution(4 − > 32, 3x3, 2,2)
(2): nn.SpatialBatchNormalization (4D) (32)
(3): cudnn.ReLU
(4): cudnn.SpatialConvolution(32 − > 64, 3x3, 2,2)
(5): nn.SpatialBatchNormalization (4D) (64)
(6): cudnn.ReLU
(7): nn.SpatialDropout(0.400000)
(8): cudnn.SpatialConvolution(64 − > 128, 3x3, 2,2)
(9): nn.SpatialBatchNormalization (4D) (128)
(10): cudnn.ReLU
(11): nn.SpatialDropout(0.400000)
(12): nn.Reshape(1152)
(13): nn.Linear(1152 − > 512)

(14): nn.BatchNormalization (2D) (512)
(15): cudnn.ReLU
(16): nn.Dropout(0.5, busy)
(17): nn.Linear(512 − > 512)

The optimal configuration of a neural network is often found by trial-and-error, as the layer structure and parameters are subject to optimization. Nevertheless, the overall structure of the network is typical for image recognition networks: multiple convolutional layers are used to turn high surface/low depth data into low surface/high depth data. In layer (1) we go from depth 4 to 32, in layer (4) from 32 to 64, and in layer (8) from 64 to 128. The deeper the data representation becomes, the higher the correlation within it - we reach intermediate, more meaningful image representations. Finally, we can use the two fully connected linear layers (13) and (17) to extract 512 features from these representations that are then fed into the RNN.

RNN The RNN is much shallower and has the following structure:

(18): nn.LSTM(512 − > 512)
(19): nn.Dropout(0.5, busy)
(20): nn.Linear(512 − > 13)
(21): cudnn.LogSoftMax

The information gathered over time is stored in the LSTM units in layer (18) and propagates along the slow-time axis. At every point in time, we use the fully connected linear layer (20) to extract the gesture probabilities for the 11 gestures from the 512 features extracted by the CNN. However, we notice that the output layer actually consists of 13 nodes. One of these extra classes is used as a reference signal, e.g. when no hand is present. The other one is required due to internal processing of the `MaskZero` module.

3.4 Using the Pre-trained Network

The final step of the experiment is to start evaluating the pre-trained network. This can be done with the code shown in listing 3.3. As we now actually deal with the network itself, we are executing Torch code instead of Python code.

Listing 3.3: Evaluating the network

```
th net/main.lua --file [image folder] --list [train/test sequence
split file] --load [model file] --inputsize 32 --inputch 4 --label 13
--datasize 32 --datach 4 --batch 16 --maxseq 40 --cuda --cudnn
```

This will invoke a series of Lua scripts that load the model, feed in the testing data and finally output the gesture recognition accuracy. To understand how this works in detail, please refer to figure 3.1.

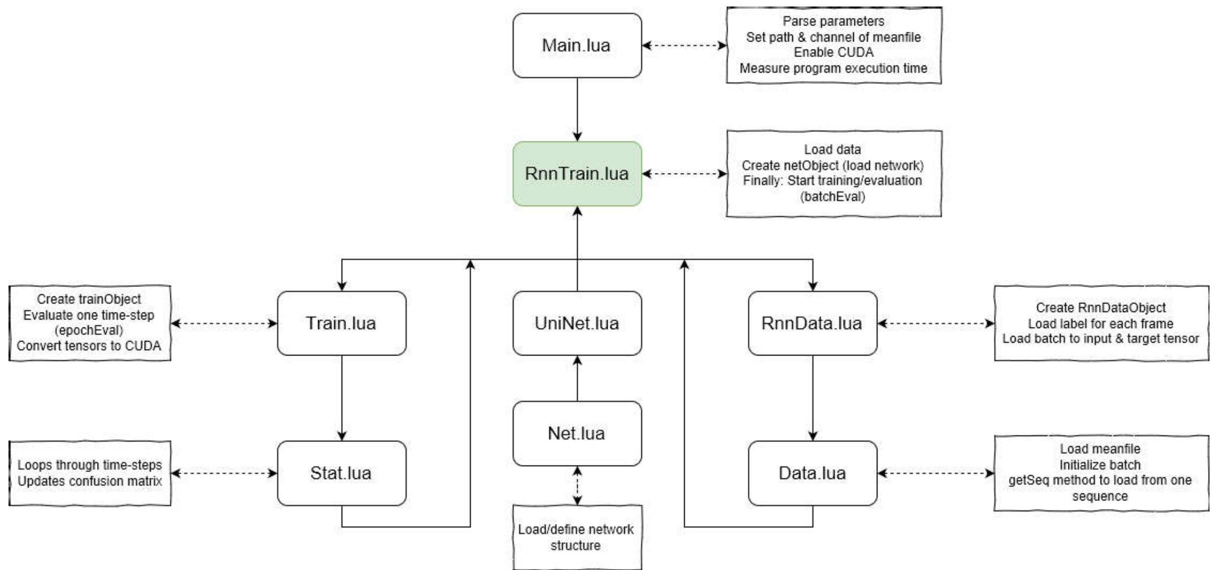

Figure 3.1: Overview over program flow during evaluation

3.5 Fixed Bugs in the Code

Before the code from GitHub could actually be used, a number of bugs had to be fixed. As all my fixes were later written back to the repository, the code should now run smoothly without further adjustments. The major bugs and their fixes are listed below. Even if most of the fixes might seem trivial, finding the cause for a specific error often took multiple hours, since the error messages were not very informative and the code structure can be quite confusing.

Finding `meanfile` The code for creating the meanfile (see listing 3.2) suggests that later during evaluation, the meanfile should be located in its original path. However, a different

path was hard-coded into the `Main.lua` script. By changing this path accordingly (should be `'../meanfile/mean_32.h5'`), this issue can be resolved.

Syntax Error: Additional `end` In the script `RnnData.lua`, an additional `end` command in line 41 caused a syntax error hindering the rest of the code from executing. Deleting this line of code obviously solved the issue.

Parsing of cuDNN-Flag Due to a missing variable assignment, the boolean `useCudnn` remained `false` even if the cuDNN-flag was set. This could be fixed by assigning the boolean in line 43 of `Main.lua` accordingly.

Wrong number of Data Channels Originally, the number of data channels was set to be three, causing a tensor dimension mismatch. This issue could be solved by altering the prompt shown in 3.3 from `--datach = 3` to `--datach = 4`. We need four channels, since for each time-step we use four RDIs stacked on top of each other. This stack is treated like a multi-channel image of depth 4. The bug itself arises as the data has four channels, but the input and output tensors are initiated according to the provided number of channels. We therefore end up with dimensionally unmatched data.

Wrong number of Labels As explained in section 3.3 in paragraph *RNN*, we deal with 13 different classes. Hence, the length of our label vector should be 13 too. However, in the original code the `label`-flag was set to be 11. Changing this to `--label = 13` solved the issue.

Error after Batch Evaluation This bug only arises whenever an entire batch has been evaluated. In that case, we enter another if-clause in the `nn.Sequencer` script and encounter the array `self.tableoutput` being of value `nil` (not allocated). By manually initializing the array for this if-clause as well (line 73 in `Sequencer.lua`), this issue can be avoided.

Version Compatibility Different versions of the `nn.MaskZero` script handle the tensors optimized for CUDA differently. This can lead to incompatibility and a fatal error in line 49 of `MaskZero.lua`. By wrapping around the mask as seen in listing 3.4, we can ensure version compatibility.

Listing 3.4: Wrapping around the mask

```
if(zeroMask:type() == 'torch.CudaTensor') then
        zeroMask = zeroMask:cudaByte()
end
output:maskedFill(zeroMask, 0)
if(zeroMask:type() == 'torch.CudaByteTensor') then
        zeroMask = zeroMask:cuda()
end
```

Chapter 4

Results

4.1 Gesture Recognition Accuracy

The evaluation lead to an overall gesture recognition accuracy of 0.711933, so roughly 71%. The per-gesture accuracy can be seen in table 4.1.

Gesture	Accuracy
01	58
02	96
03	62
04	73
05	80
06	43
07	98
08	91
09	93
10	64
11	25

Table 4.1: Recognition accuracy for each gesture

We notice that the achieved accuracy is significantly lower than the accuracy mentioned in the paper which was around 87% (see [1]). This discrepancy should be investigated further,

as it is definitely not a randomly appearing fluctuation. Further, it can be seen that gestures (02), (07), (08) and (09) achieve very high recognition accuracy (> 90%). Looking at figure 2.9 (gesture (01) corresponds to gesture (0) in the figure), we see that these corresponds to *Palm tilt*, *Push*, *Pull* and *Finger rub*. Most of these gestures include large scale movement into a distinct direction, which apparently is well-recognized by this pipeline. The evaluation takes around 5 minutes and the recognition accuracy remained unchanged over several iterations. This makes sense, as neither the test dataset nor the weights in the pre-trained network change.

Looking at the resulting confusion matrix in figure 4.1, we notice that while classifying between eleven gestures, we get 13 rows in the resulting confusion matrix. As explained in section 3.3 under paragraph *RNN*, two additional classes are used. Class (12) is carrying a background noise signal, which corresponds to no hand being above the sensor. This class does not hold any predictions. Class (13) is used for internal processing in the `MaskZero.lua` script. It is interesting how this internal data got misclassified as gesture (01). According to S. Wang, this does not affect the other prediction probabilities and can be ignored.

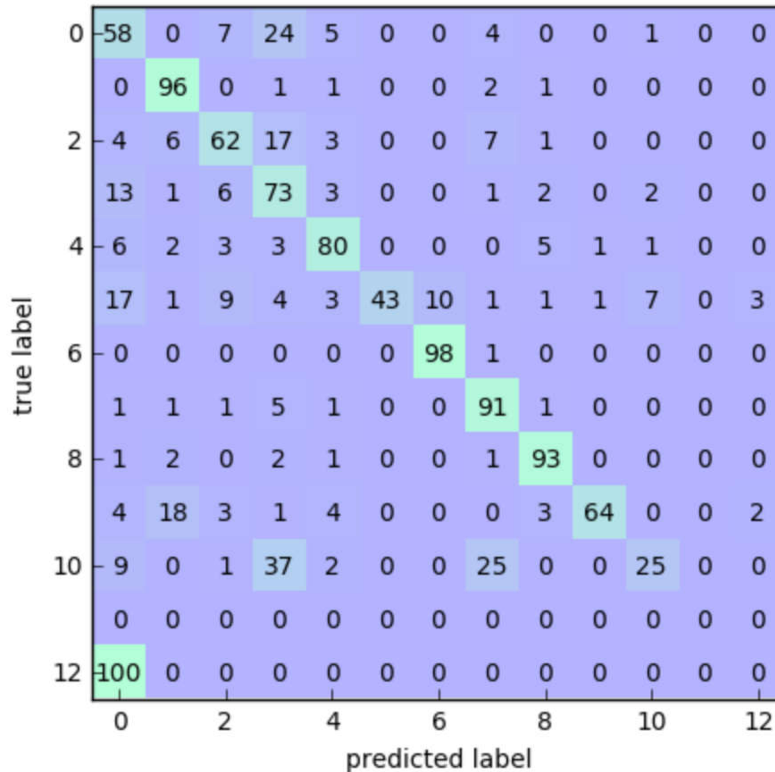

Figure 4.1: Resulting confusion matrix over all 13 classes

4.2 Contribution to Paper

While reproducing the paper, I came across various bugs in the code and unclear formulations in the documentation. I believe that solving the bugs within the code provided will contribute to the paper. As I consequently shared the fixes through GitHub, it is now a lot easier to experiment with the pipeline and eventually optimize it for even better performance.

Chapter 5

Conclusion

5.1 Conclusion

The main goal of the project was to gather first experiences in the field of artificial intelligence and getting familiar with neural networks. I achieved this through intensively working with the code provided. Especially, figuring out how to fix some of the bugs required me to get a deep understanding of the code and methods behind it. Through my *Computational Intelligence* class within NTU and several web classes about machine learning, I further received a basic understanding of the mathematical background. This will help me in my future classes in artificial intelligence, a field I definitely want to continue my education in.

However, reproducing the paper itself required a lot more work than I originally expected. Setting up all the dependencies correctly, understanding the code and fixing bugs delayed my project significantly. Hence, I could not design my own network, train it myself and then optimize it towards a higher recognition accuracy. The project also showed me the importance of community support for software, as I often struggled to find fixes online due to low popularity of certain packages.

The second major goal was contributing to the proposed paper and improve its performance. Unfortunately, I did not get to the point where I could start to optimize the network and increase its recognition accuracy. Nevertheless, I believe I partly achieved this goal by fixing several bugs and uploading these fixes to the GitHub repository. It will now be easier to experiment with the pipeline which could lead to further improvement of the performance through another project.

5.2 Future Work

Further projects related to the proposed paper could focus on the following aspects:

Explaining Recognition Accuracy Drop The discrepancy between the achieved accuracy while reproducing the paper and the original accuracy is high. One could further investigate the cause for this which will lead to deeper understanding of the pipeline and could even reveal a method to improve the overall performance of the network.

Rebuilding the Network A key step towards increasing the pipeline's performance is to rebuild the network and train it using the data provided. As soon as this is achieved, the network structure and parameters can be altered to optimize the network further.

Test the Pipeline in Real-time It would interesting to test the network in a more realistic scenario, meaning that an actual Soli sensor and commodity hardware for signal processing are used. According to the ATAP group, the second version of the Soli sensor should be announced soon and will be available to a broader spectrum of developers than the Alpha version.

List of Figures

2.1 Overall principle of Soli 3
2.2 Radar chip (left) embedded on a development board 4
2.3 Overview of hard- and software components of Soli 5
2.4 Overall principle: Radar signal illuminates hand, scattered waves are measured . 5
2.5 Different visual representations of the signal 6
2.6 Explanation of Range-Doppler image 7
2.7 Four different gestures used in original version of Soli 8
2.8 Overall structure of end-to-end model 9
2.9 Eleven different gestures used in the newly proposed pipeline 10
2.10 Simple example of a neural network 11
2.11 Example of a CNN 12
2.12 Example of a RNN 12
2.13 Example of a LSTM unit 13

3.1 Overview over program flow during evaluation 19

4.1 Resulting confusion matrix over all 13 classes 23

References

[1] S. Wang, J. Song, J. Lien, I. Poupyrev, and O. Hilliges. *Interacting with Soli: Exploring Fine-Grained Dynamic Gesture Recognition in the Radio-Frequency Spectrum*, Proceedings of the 29th Annual Symposium on User Interface Software and Technology, Tokyo, 2016

[2] C.J. Hansen. *Wigig: Multi-gigabit wireless communications in the 60 ghz band*, Wireless Communications, IEEE, 2011

[3] J. Lien, N. Gillian, P. Amihood, and I. Poupyrev. *Soli: Ubiquitous Gesture Sensing with Millimeter Wave Radar*, ACM Transactions on Graphics, 2016

[4] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. *Improving neural networks by preventing co-adaptation of feature detectors*, Cornell University, 2012

[5] Official Lua webpage, *https://www.lua.org/about.html*, May 2017

[6] Wikipedia article on CUDA, *https://en.wikipedia.org/wiki/CUDA*, May 2017

[7] Official Torch webpage, *http://torch.ch/*, May 2017

[8] GitHub repository of nn, *https://github.com/torch/nn*, May 2017

[9] GitHub repository of Element-Research, *https://github.com/Element-Research/rnn*, May 2017

[10] GitHub repository of cutorch, *https://github.com/torch/cutorch*, May 2017

[11] Nvidia Developers, *https://developer.nvidia.com/cudnn*, May 2017

[12] Wikipedia article on GitHub, *https://en.wikipedia.org/wiki/Git*, May 2017

[13] Github article on neural networks, *http://cs231n.github.io/assets/nn1/neural_net.jpeg*, May 2017

[14] The data scienc blog, *http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/11/Screen-Shot-2015-11-07-at-7.26.20-AM.png*, May 2017

[15] WildML, *http://d3kbpzbmcynnmx.cloudfront.net/wp-content/uploads/2015/09/rnn.jpg*, May 2017

[16] Wikipedia article on LTSM, *https://en.wikipedia.org/wiki/File:Long_Short_Term_Memory.png*, May 2017