Voice Recognition for Robotic Control

Project report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

In

Electrical and Electronics Engineering

Submitted by

Dhruv Garg
(1710110110)

Under supervision

of

Dr. Madan Gopal,
Distinguished Professor, Department of Electrical Engineering

SHIV NADAR UNIVERSITY

Department of Electrical Engineering

School of Engineering

Shiv Nadar University

(Spring 2021)

Candidate Declaration

I hereby declare that the thesis entitled "Voice Recognition for Robotic Control" is submitted for the B Tech Degree program. This thesis has written in my own words. I have adequately cited and referenced the original sources.

(Signature)

(Dhruv Garg)

(1710110110)

Date: 30/04/21

CERTIFICATE

It is certified that the work contained in the project report titled “Voice Recognition for Robotic Control,” by “Dhruv Garg,” has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Signature of Supervisor(s)
Name: Dr. Madan Gopal
Department: Electrical Engineering
School of Engineering
Shiv Nadar University
April 2021

Abstract

The term "robot" generally connotes some anthropomorphic (human-like) appearance. Many research pointed out some issues for developing a humanoid robot, and one of the significant research issues is to build a machine that has human-like perception. What is human-like perception? - The five classical human sensors - vision, hearing, touch, smell, and taste; by which they percept the surrounding world [1]. This project's primary goal is to develop a voice-recognition system for industrial robots to interact with humans through Automatic Speech Recognition (ASR). Automatic Speech Recognition (ASR) is commonly described as converting speech to text. Industrial Robots exhibit varying degrees of autonomy. Some robots are programmed to faithfully carry out specific actions repeatedly without variation and with a high degree of accuracy. These actions are determined by programmed routines that specify the direction, acceleration, velocity, deceleration, and distance of a series of coordinated motions. Other robots are much more flexible to the orientation of the object they are operating or even the task that has to be performed on the thing itself, which the robot may need to identify [2]. However, deep learning is becoming more prominent in industries for controlling various robotic dynamics. Companies are spending millions of dollars on research to upgrade the dynamics and precision of robots and make them more efficient. One of the research areas is to control the robot's activities through voice commands. This project aims to target the commands like " up ", "down", "left" and "right" and convert them to text through sequential numerical values using Mel Frequency Cepstral Coefficients. Open-source dataset is used for voice commands. The processed text will be given to Recurrent Neural network (RNN) and Long Short-Term Memory (LSTM) network, which will become the robot's commands. Long Short-Term Memory (LSTM) networks are a type of recurrent neural network proficient of learning order dependence in sequence prediction problems. LSTM has shown groundbreaking results in literature in voice recognition applications.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The presented approaches are only for academic purposes and should not be extended in automated applications without proper examination. Because of the simplicity of the educational project and the limitation of computational power, the discussion focuses on the commands 'up', 'down,' 'left,' 'right,' 'forward,' 'backward,' 'on,' 'off,' 'follow,' 'go,' 'stop' only. The approaches presented can be stretched to other datasets in a related manner which may contain many commands.

This work is done in the Python programming language. Python is chosen because of its versatility, expandability, and availability of numerous open-source packages that have been tested thoroughly by thousands of users over many years. Matplotlib library is used for the graphs and illustrations in this project. Python being a general-purpose programming language also gives the advantage of quick deploy ability in almost any working environment like Linux, Windows, Mac, without worrying about compatibility and deployment costs.

1.1 Motivation:

Voice recognition has been imagined about and worked on for decades—continuously surrounded by digital assistants like Amazon Alexa, Apple's Siri, Microsoft Cortana and Google Assistant. It is easy to take for granted how voice recognition technology works because the simplicity of speaking to digital assistants is misleading. Voice recognition is astonishingly complex, even now. Many voice assistants rely on keyword spotting to begin intercommunications. For example, an individual might say "HeyGoogle" or "Hey Siri" to begin a query. Once the device knows that the person wants to communicate, it is reasonable to send the audio to a web server for further processing. The initial detection of the start of communication is impossible to run as a cloud-based service since it will require transmitting audio data over the web from the robot or device all the time; this would be very expensive and would increase the privacy risks of the technology.

Instead, most voice interfaces run a recognition module locally on the phone or other device, it listens continuously to audio input from microphones, and rather than sending the data over the internet to a server, and they run models that listen for the desired trigger phrases. Once a likely trigger is heard, the transfer of the audio to a web service begins. Because the local model runs on hardware that is not under the web service provider's control, there are hard resource constraints that the on-device model must respect [3]. In this discussion, we want to use the same trigger for robotic devices that the industry uses to control robotic movements instantly. We cannot send the audio data over the web every time even if connected to the power supply because there would be a limit of dissipating the heat, and repeated movements may increase the processing time, and network latency can be highly variable depending on the environment.

1.2 Problem Statement:

Here, Google Voice Commands Dataset is used [4], which consists of various commands from which the 'up,' 'down,' 'left,' 'right,' 'forward,' 'backward,' 'on,' 'off,' 'follow,' 'go,' 'stop' commands is chosen for our application. There are 1500 - 3700 instances for each of these commands in the dataset. Each instance is stored as a .wav file sampled at S=16,000Hz and trimmed to exactly 1 second (total 16,000 samples per command). Due to the complexity of compensating for constant noise levels and the availability of a large enough dataset, we can skip the command signal's normalization.

We used Recurrent neural Network or LSTM, which are used in industry to tackle sequential model problems for deep learning architecture. The Recurrent Neural Networks are deliberately made to tackle problems having time dependencies. The transient nature makes these predicaments nearly related, and we will try to take their time-dependent nature into account using deep networks.

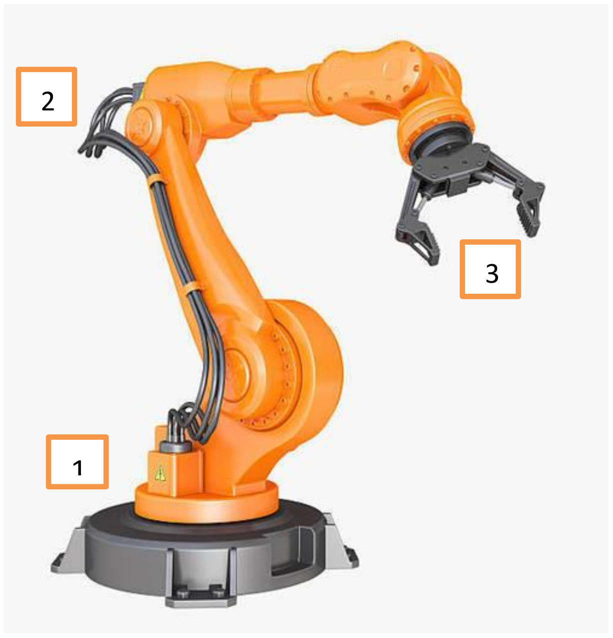


Figure 1.1: Schematic diagram of Proposed Robot

A robotic arm for mobile and stationary robots is proposed using voice command recognition, which can be directed to move things with high precision on the fly. The 'left' and 'right' commands can control the horizontally rotating servo motors at junction 1 in Figure 1.1. The 'go' and 'stop' commands can control the vertically moving servo motor at junction 2. Furthermore, the 'up' and 'down' commands can be used to manage the gripper at junction 3. 'Forward' and 'backward' can address the entire robot to move. We are focusing on these 11 commands due to computational time and power, but this proposal can be extended to any other dataset with a rich vocabulary command.

Chapter 2

Literature Review

A first indication of the capabilities of neural networks in tasks related to natural language was given by [24] with a neural language modeling task. In 2003 good results applying standard LSTM-RNN networks with a mix of LSTM and sigmoidal units to speech recognition tasks were obtained by [25, 26]. Better results comparable to Hidden-Markov-Model (HMM)-based systems [27] were achieved using bidirectional training with BLSTM [28, 29]. A variant named BLSTM-CTC [30] finally outperformed HMMs, with recent improvements documented in [31]. The performance of different LSTM-RNN architectures on large vocabulary speech recognition tasks was investigated by [33], with best results using an LSTM/HMM hybrid architecture. Comparable results were achieved by [33].

More recently LSTM was improving results using the sequence-to-sequence framework ([34]) and attention-based learning ([35]). In 2015 [36] introduced a specialized architecture for speech recognition with two functions, the first called 'listener' and the latter called 'attend and spell.' The 'listener' function uses BLSTM with a pyramid structure (pBLSTM), similar to clockwork RNNs introduced by [37]. The other function, 'attend and spell,' uses an attention-based LSTM transducer developed by [38] and [39]. Both functions are trained with methods introduced in the sequence-to-sequence framework [34] and in attention-based learning [38].

Chapter 3
Digital Signal Processing

Building machine learning models to classify, describe, or generate audio typically involves modelling tasks where the input data is obtained from audio samples through various approaches.

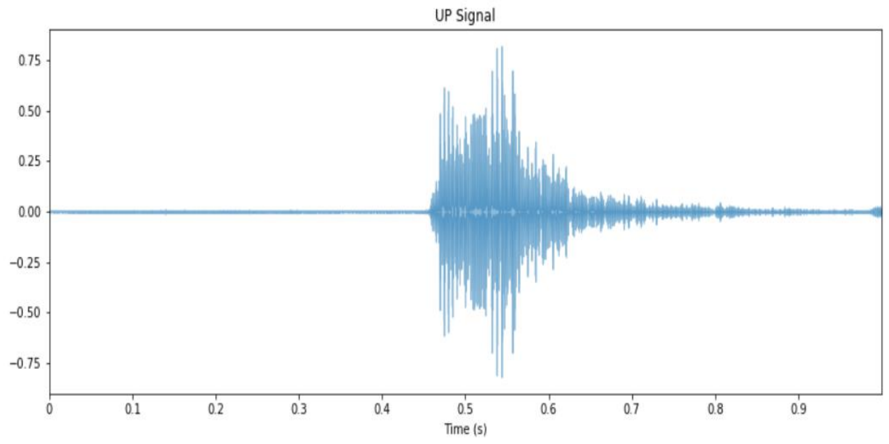
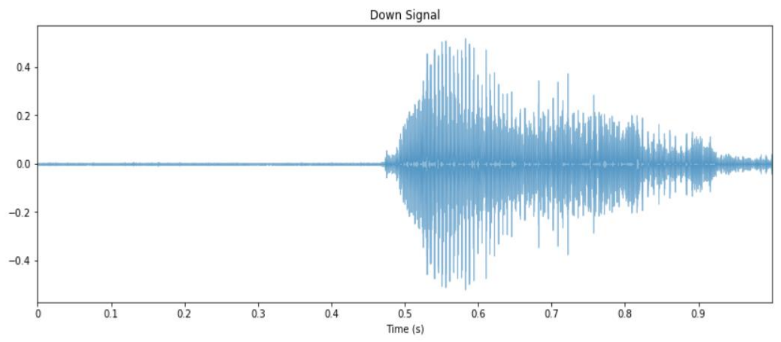


Figure 3.1: Time domain signal of one instance of "up"

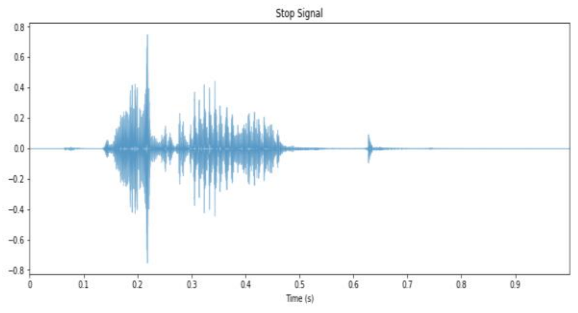
Example waveform of an audio dataset sample of an 'up' command from Google Speech Dataset [4].

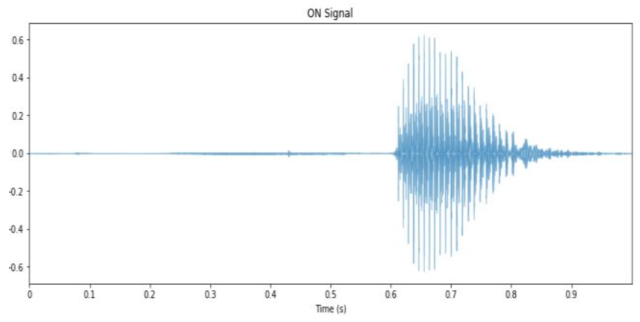
These audio samples are usually represented as time series, where the y-axis measurement is the amplitude of the waveform. The amplitude is usually measured as a function of the change in pressure around the microphone or receiver device that originally picked up the audio. Unless there is metadata associated with your audio samples, these time series signals will often be your only input data for fitting a model [5].

It is realized from the given samples given below that by merely observing a wave; we cannot assume that a particular wave belongs to any specific class.



(a)





(b) (c)

Figure 3.2: Time domain signal of one instance of (a) “down”, (b) “stop” and (c) “On”

One of the robust features for audio processing is Mel Frequency Cepstrum Coefficients which are widely used in automatic speech and speaker recognition. They were introduced by Davis and Mermelstein in the 1980s and have been state-of-the-art ever since.

3.1 Terminologies

3.1.1 Sampling and Sampling Frequency

Sampling rate or sampling frequency represents the number of samples per second taken from a continuous signal to make a discrete or digital signal. A high sampling frequency results in less information loss but a higher computational expense, and low sampling frequencies have higher information loss but are fast and cheap to compute [5].

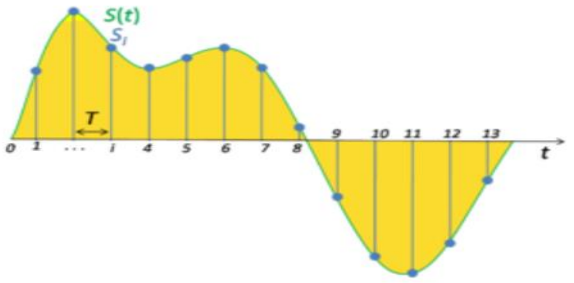


Figure 3.3: Sampling of a continuous signal into discrete steps

3.1.2 Fourier Transform

The Fourier Transform decomposes a function of time (signal) into constituent frequencies. Similarly, a musical chord can be expressed by its constituent notes' volumes and frequencies. A Fourier Transform of a function displays each frequency's amplitude (amount) present in the underlying signal.

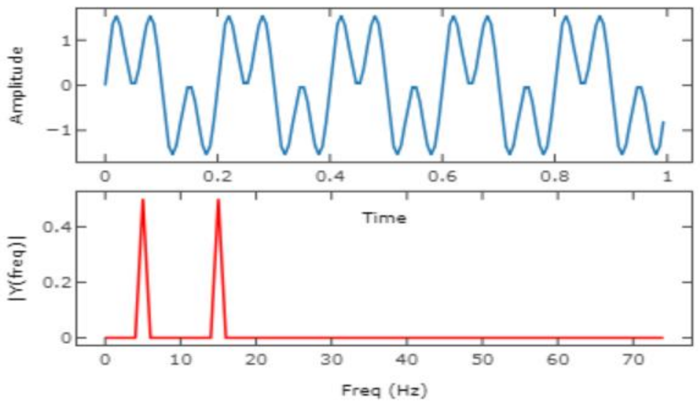


Figure 3.4: Fourier Transform of a signal from time domain to frequency domain.

The Fourier Transform variants include the Short-time Fourier transform, which is implemented in the Librosa library in Python and includes splitting an audio signal into frames and then taking the Fourier Transform of each frame [5].

3.1.3 Periodogram

In signal processing, a periodogram is an estimate of the spectral density of a signal. A periodogram is used to identify the dominant periods (or frequencies) of a time series [5]. This can help determine the dominant cyclical behavior in a series, mainly when the cycles are not related to the commonly encountered monthly or quarterly seasonality. The periodogram below shows the power spectrum of two sinusoidal basis functions of ~30Hz and ~50Hz. The output of a Fourier Transform can be thought of as being (not precisely) essentially a periodogram.

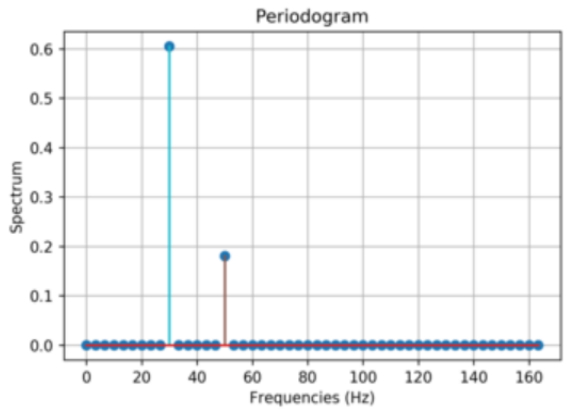


Figure 3.5: Periodogram

3.2 Signal Processing

Here, Mel Frequency Cepstral Coefficients are extracted for each frame of 25 ms of an audio signal with an overlap of about 15 ms between the adjacent frames and stacked horizontally.

3.2.1 Pre-emphasis of an audio signal

The first step is to apply a pre-emphasis filter on the signal to amplify the high frequencies. A pre-emphasis filter is useful in several ways: (1) balance the frequency spectrum since high frequencies usually have smaller magnitudes compared to lower frequencies, (2) avoid numerical problems during the Fourier transform operation and (3) may also improve the Signal-to-Noise Ratio (SNR). The filter used is given by: -

$$y(t) = x(t) - \alpha x(t-1)$$

Here the filter coefficient (α) is 0.97. Now our resulting signal would be y(t).

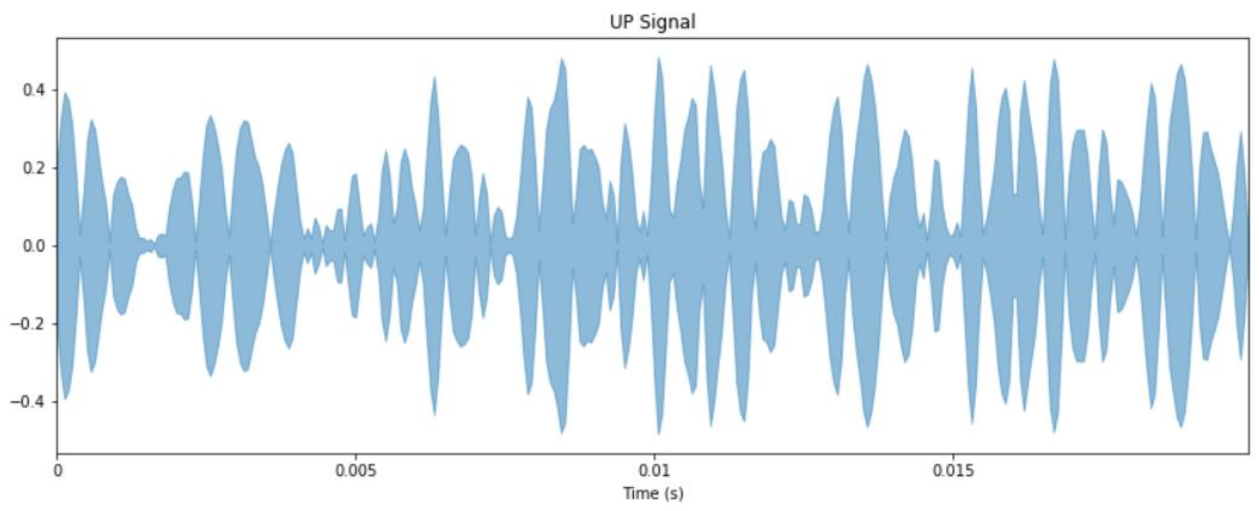


Figure 3.6: One frame of “up” command before pre-emphasis

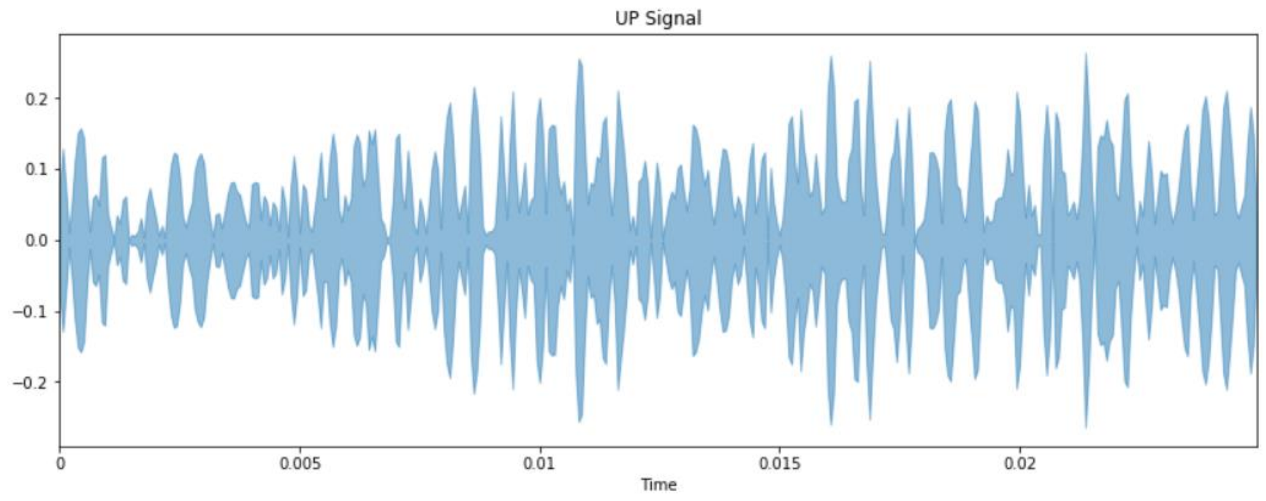


Figure 3.7: One frame of “up” command after pre-emphasis

The impact of pre-emphasis can be seen in Figure 3.6 and Figure 3.7. The high amplitude of low-frequency components has been modified to be more comparable to the high-frequency components.

3.2.2 Framing and Hamming Window

The speech signal is segmented into small duration blocks of 25 ms known as frames. Voice signal is divided into L samples, and M is separating adjacent frames (M<L). Framing is required as speech is a time-varying signal, but when it is examined over a sufficiently short period of time, its properties are relatively stationary. Therefore, the short-time spectral analysis is done. The first frame consists of the first N samples, and

the second frame starts M samples after the first sample; therefore, the first overlap will be started at the (L-M) th sample. An overlap between consecutive frames is required to avoid the discontinuity in signal parameters caused otherwise [6]. For frame length of 25ms sampled at Fs = 16,000 Hz:

$$L = \frac{25ms}{1000ms} \times 16000 = 400samples$$

Where L gives us the frame size in number of samples. The step size M for a step of 10ms is given by,

$$M = \frac{10}{1000} \times 16000 = 160samples$$

The rest of the frames are realized similarly. If enough samples are not available at the end to form the last frame, the signal is padded with the required number of trailing zeroes to make a complete frame.

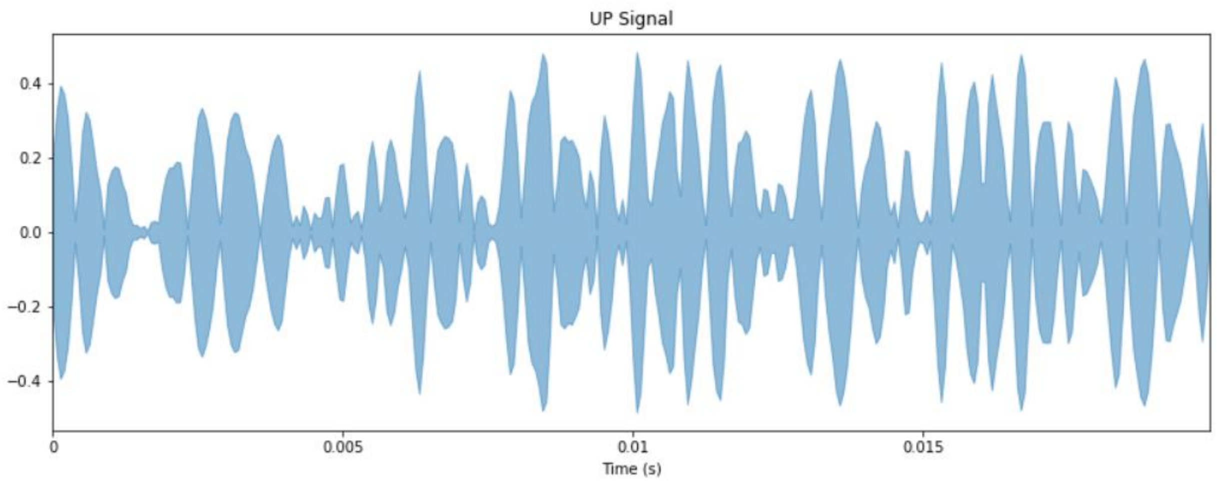


Figure 3.8: One frame of 25ms in time domain

Each of the above frames is multiplied with a hamming window to keep the continuity of the signal. So, to reduce this discontinuity, we apply a window function. The spectral distortion is minimized by using a window to taper the voice sample to zero at both the beginning and end of each frame [7], and higher importance is given to the center samples to lessen the amplitude of difference.

$$w(n) = 0.54 - 4.46cos\left(\frac{2\pi n}{L-1}\right) \qquad 0 \leq n \leq L-1,$$

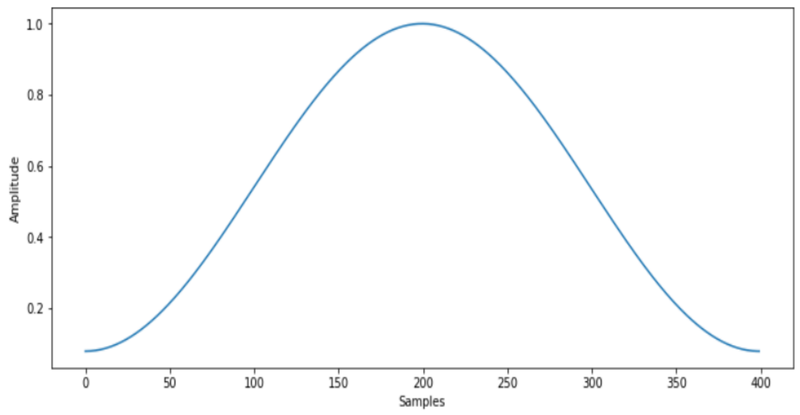


Figure 3.9: Hamming Window w(n) of 25ms length

Where w(n) is the window operation, n is the number of each sample, and L is the total number of speech samples. The windowed signal is given by:

$$y(n) = w(n) * x(n)$$

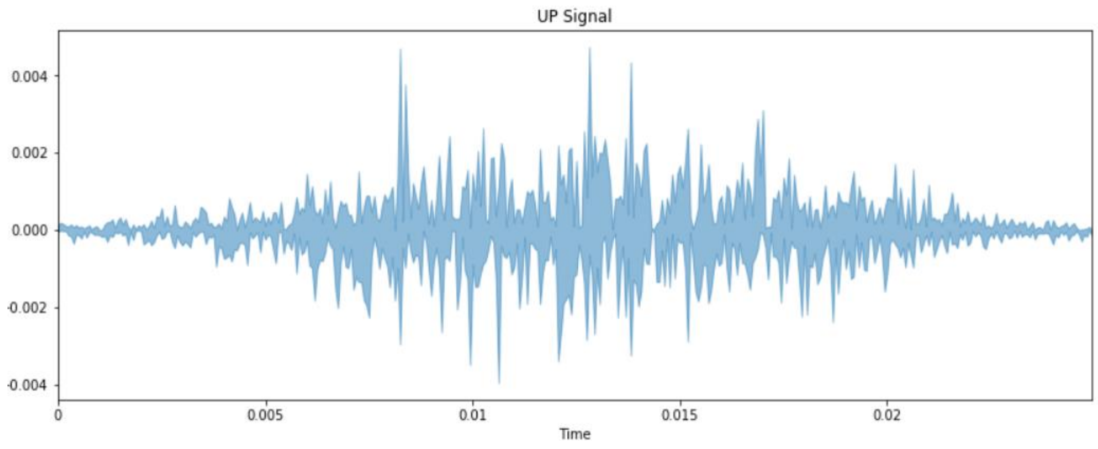


Figure 3.10: Windowed signal y(n) after multiplying w(n) with one frame x(n)

3.2.3 Discrete Fourier Transform

FFT is used for doing the conversion from the spatial domain to the frequency domain. Each frame having N_m samples are converted into the frequency domain. Fourier transformation is a fast algorithm to apply Discrete Fourier Transform (DFT) on N_m samples' given set below [8].

$$D_k = \sum_{m=0}^{N_m-1} D_m \, e^{\frac{-j2\pi km}{N_m}}$$

Where k= 0, 1, 2 ….. N_m-1

The definition of FFT and DFT is the same, which means that the transformation output will be the same; however, they differ in their computational complexity. In the case of DFT, each frame with N-M samples directly will be used as a sequence for Fourier transformation. On another, in the case of FFT, this frame will be divided into small DFT's, and then the computation will be done on this divided small DFT's as individual sequence; thus, the calculation will be faster and easier. Therefore, in digital processing or other areas, instead of directly using DFT, FFT is used for applying DFT. The discrete Fourier transform distributes the frequency components of the signal uniformly over [0,2π). The actual frequencies of the components are dependent on the sampling rate(S) of the signal, and the highest frequency component is of the Nyquist frequency [8].

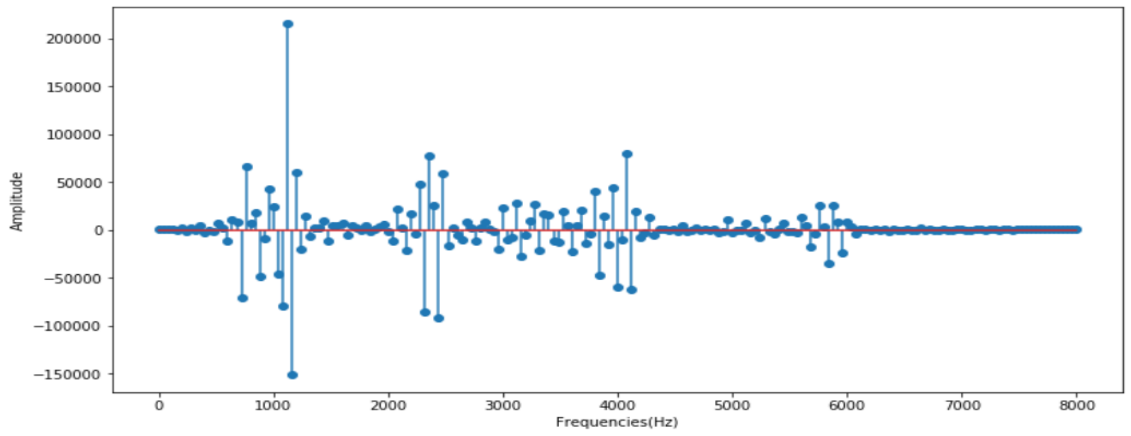


Figure 3.11: Real part of DFT of one frame of “up” command.

Commonly, D_k is the combination of real and imaginary numbers; thus, it represents the complex numbers but, merely absolute values (frequency magnitudes) are considered to carry out the further process. The obtained sequence can be interpreted as positive frequencies $0 \leq f < F_s/2$ correspond to values $0 \leq m \leq N_m/2 - 1$, while negative frequencies $-F_s/2 < f < 0$ correspond to values $N_m/2 + 1 \leq m \leq N_m - 1$, F_s is the sampling frequency. By calculating DFT, we can obtain the magnitude spectrum [8].

3.2.4 Power Spectral Density

The power spectral density is the quantity of power for each frequency component. It can be mathematically represented by the equation.

$$P(k) = \frac{1}{N}|X(k)|^2$$

Where N is the number of frequency components (N = 400). We take N = 400, even after though we are left with only 201 points after discarding half the DFT points obtained because we had initially taken the 400 points DFT of the frame.

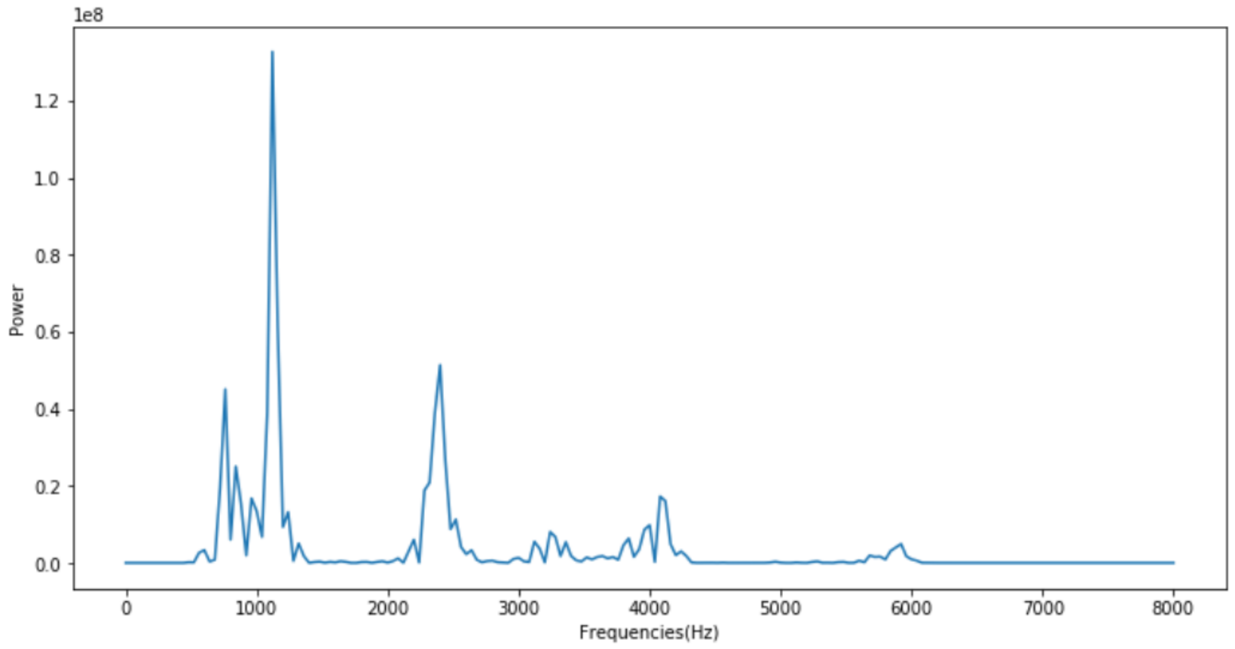


Figure 3.12: Power Spectral Density of 1 frame of “up” command.

3.3 Mel Frequency Cepstral Coefficients

MFCC's are based on the known variation of the human ear's critical bandwidths with frequency. Speech signal has been expressed in the Mel frequency scale in order to capture the important characteristic of phonetic in speech. This scale has a linear frequency spacing below 1000 Hz and a logarithmic spacing above 1000 Hz. Normal speech waveform may vary from time to time, depending on the physical condition of the speakers' vocal cord. Rather than the speech waveforms themselves, MFCCs are less susceptible to the said variations.[9]

3.3.1 Mel Scale

In this step, the above-calculated spectrums are mapped on the Mel scale to know the conjecture about the existing energy at each spot with the help of a Triangular overlapping window, also known as a triangular filter bank. This filter bank is a set of bandpass filters with spacing and bandwidth decided by steady Mel frequency time. Thus, the Mel scale helps space the given filter and calculate how much broader it should be because, as the frequency gets higher, these filters are also getting wider. For Mel- scaling mapping is needed to be done among the given real frequency scales (Hz) and the perceived frequency scale (Mels). During the mapping, when a given frequency value is up to 1000Hz, the Mel-frequency scaling is linear frequency spacing, but after 1000Hz, the spacing is logarithmic, as shown in Figure 3. The formula to convert frequency f hertz into Mel is given by [8]:

$$M(f) = 1127 \times ln\left(1 + \frac{f}{700}\right)$$

To go from Mel's to Hz, we can use the equation:

$$F(m) = 700 \times \left(e^{\frac{m}{1127}} - 1\right)$$

The Mel scale is defined by selecting the starting frequency as 20Hz to 8,000Hz and converting to Mel's, which gives 20Hz = 31.75Mels and 8,000Hz = 2840.04Mels. The scaling between Mel-Scale and Hz can be seen in the figure given below. The 201 points of DFT are uniformly distributed over the range of 0- 8,000Hz. Therefore, the gaps between consecutive points are 40Hz. But we need to start the first filter from 20Hz. Because of the unavailability of the point on 20Hz, we start the filter from 0Hz.

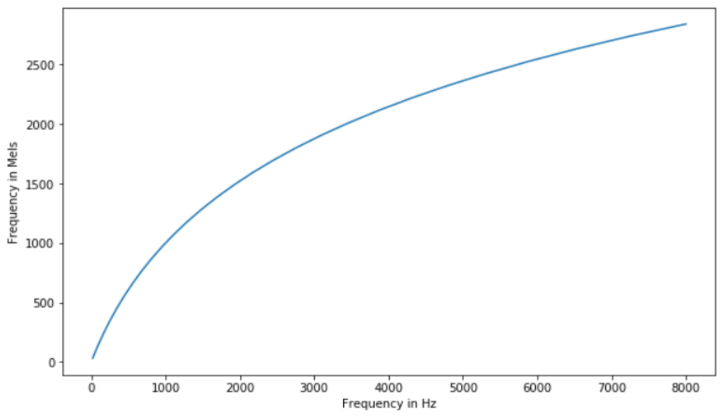


Figure 3.13: Plot of Mel-Scale vs the frequencies in Hz.

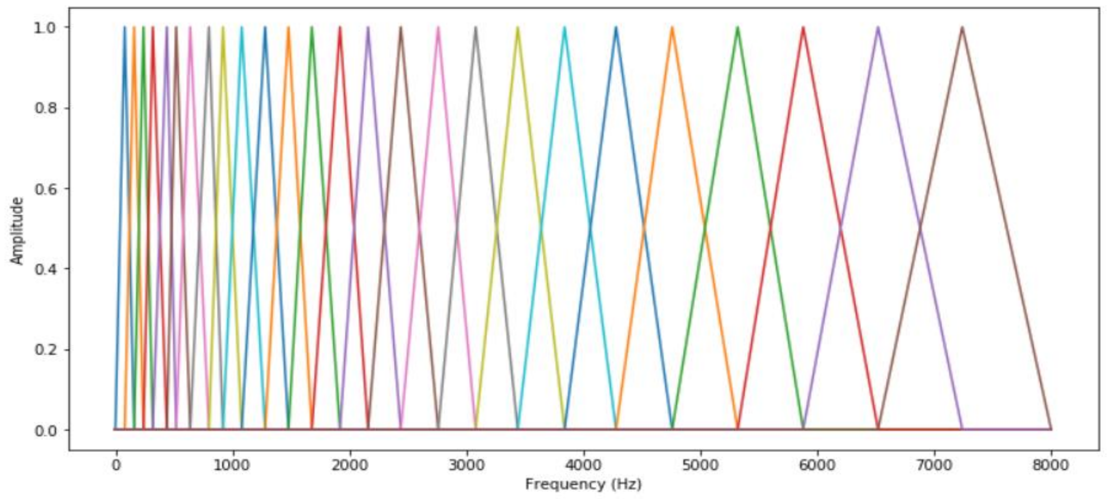


Figure 3.14: Mel Filter banks

26 Mel filter banks are made between the selected frequencies 20Hz to 8,000Hz. Twenty-six equally spaced points are selected between 31.75 Mels and 2840.04 Mels, giving us 28 filter bank frequencies in Mels. These are converted back to Hz, and a set of 3 consecutive frequencies gives us the 3 points for forming the triangular Mel filter bank. The obtained filters can be seen in the figure given above. Thus, with the help of a Filter bank with proper spacing done by Mel scaling, it becomes easy to get the estimation about the energies at each spot, and once these energies are estimated, then the log of these energies, also known as Mel spectrum, can be used for calculating first 13 coefficients using DCT. The increasing numbers of coefficients represent a faster change in the estimated energies and thus have less information to classify the given images. Hence, the first 13 coefficients are calculated using DCT, and higher are discarded [8].

3.3.2 Discrete Cosine Transform

This process of carrying out DCT is done to convert the log Mel spectrum back into the spatial domain. For this transformation, either DFT or DCT, both can be used for calculating Coefficients from the given log Mel spectrum as they divide a given sequence of finite length data into a discrete vector. However, DFT is generally used for spectral analysis, whereas DCT used for data compression as DCT signals have more information concentrated in a small number of coefficients, and hence, it is easy and requires less storage to represent Mel spectrum in a relatively small number of coefficients. This, instead of using DFT, DCT is desirable for the calculation of the coefficients as DCT outputs can contain substantial amounts of energy. The result after applying DCT is known as MFCC (Mel Frequency Cepstrum Coefficient) [8]. The DCT-II is used, and it is mathematically expressed as:

$$\Gamma(i) = \sum_{c=0}^{C-1} \gamma(c) cos \left[\frac{\pi}{C} \left(c + \frac{1}{2} \right) i \right] \qquad i = 0, 1, \ldots, C-1$$

where γ(c) are the log-energies and C(26) is the total number of energies.

3.3.3 Liftering

Liftering operation in the cepstral domain is like filtering in the frequency domain. The lifter gives the effect of increasing the magnitude of high-frequency DCT coefficients [10]. The lifter is given by:

$$z(i) = 1 + \frac{L}{2} sin \left(\frac{\pi i}{L} \right) \qquad i = 0, 1, ..., C-1$$

Where L = 22 is the liftering coefficient, and C = 26 is the number of cepstral coefficients. The final MFCCs are obtained by multiplying the lifter with the decorrelated log-energies obtained.

For each set of coefficients obtained for a frame, the first coefficient is replaced with the total energy contained in that frame. This gives the 26 cepstral coefficients of a single frame of the audio signal shown in Figure 3.15.

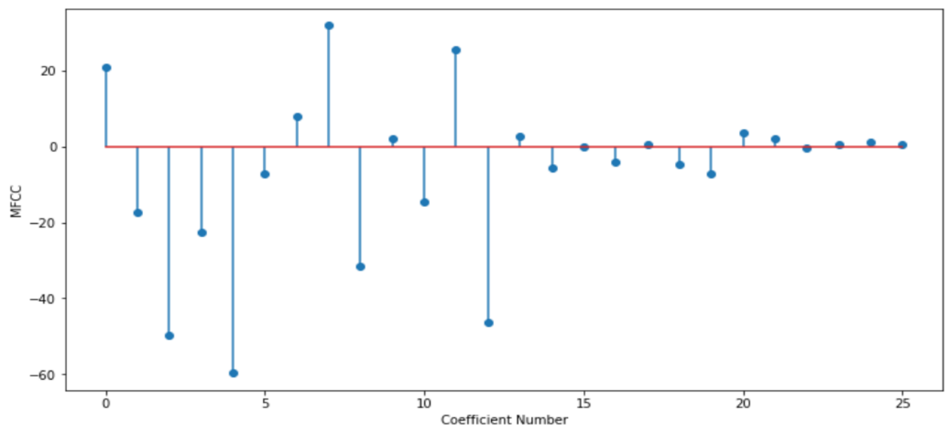


Figure 3.15: Mel Frequency Cepstral Coefficients

Chapter 4

Recurrent Neural Networks

A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time-series data. These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (NLP), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate. Like feedforward and convolutional neural networks (CNNs), recurrent neural networks utilize training data to learn. They are distinguished by their "memory" as they take information from prior inputs to influence the current input and output. While traditional deep neural networks assume that inputs and outputs are independent of each other, the output of recurrent neural networks depends on the sequence's prior elements. While future events would also help determine the output of a given sequence, unidirectional recurrent neural networks cannot account for these events in their predictions [11].

4.1 Architecture of RNNs

RNNs are also known as feedback networks in which connections between units form a directed cycle. RNNs can use their internal memory to process arbitrary sequences of inputs. In RNN, the signals travel both forward and backward by introducing loops in the network as shown in figure 4.1.

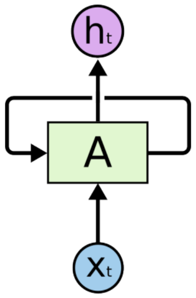


Figure 4.1: Basic RNN with feedback

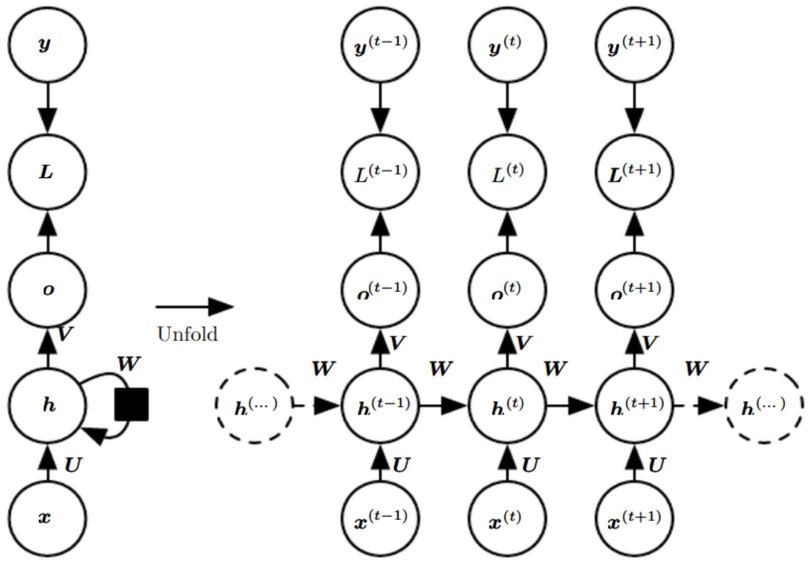


Figure 4.2: Unfolded RNN

The computational graph shown in above figure is computing the training loss of a recurrent network that maps an input sequence of *x* values to a corresponding sequence of output o values. A loss ***L*** measures how far each ***o*** is from the corresponding training target *y*. When using *SoftMax* outputs, we assume ***o*** is the unnormalized log probabilities. The loss ***L*** internally computes $\hat{y}$ = *SoftMax(o)* and compares this to the target ***y***. The RNN has input to hidden connections parametrized by a weight matrix ***U***, hidden-to-hidden recurrent connections parametrized by a weight matrix ***W***, and hidden-to-output connections parametrized by a weight matrix ***V***. (Left) The RNN and its loss has drawn with recurrent connections. (Right) The same is seen as a time- unfolded computational graph, where each node is now associated with one time instance. [12]

4.2 Working of RNNs

RNN takes input as a form of tensor or a 3-D matrix in which all the instances are stored row-wise. Instances get fed to the network (layer of neurons), and errors are calculated using the predicted output and actual output. This error is then backpropagated to update the weights by computing gradients.

4.2.1 Forward Propagation

We now develop the forward propagation equations for the RNN depicted in figure 4.2. The figure does not specify the choice of activation function for the hidden units. Here we assume the hyperbolic tangent activation function. Also, the figure does not specify precisely what form the output and loss function take. Here we assume that the output is discrete. A natural way to represent discrete variables is to regard the output ***o*** as giving the unnormalized log probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a post-processing step to obtain a vector ŷ of normalized probabilities over the output. Forward propagation begins with a specification of the initial state $h(0)$. Then, for each time step from $t = 1$ *to* $t = \tau$, we apply the following update equations:

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}$$

where the parameters are the bias vectors ***b*** and ***c*** along with the weight matrices ***U***, ***V*** and ***W***, respectively for input-to-hidden, hidden-to-output and hidden-to-hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of ***x*** values paired with a sequence of ***y*** values would then be just the sum of the losses over all the time steps. For example, if $\boldsymbol{L}^{(t)}$ is the negative log-likelihood of $\boldsymbol{y}^{(t)}$ given $\boldsymbol{x}^{(1)}$, . . . , $\mathbf{x}^{(t)}$, then,

$$\begin{aligned}
& L\left(\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}\}, \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(\tau)}\}\right) \\
&= \sum_t L^{(t)} \\
&= -\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\}\right)
\end{aligned}$$

States computed in the forward pass must be stored until they are reused during the backward pass [12].

4.2.2 Back Propagation through Time (BPTT)

For RNNs, we need to propagate information through the recurrent connections in-between steps. The most common and well-documented learning algorithms for training RNNs in temporal, supervised learning tasks are backpropagation through time (BPTT) and real-time recurrent learning (RTRL). In BPTT, the network is unfolded in the time to construct an FFNN. Then, the generalized delta rule is applied to update the weights. This is an offline learning algorithm because we first collect the data and then build the model from the system. In RTRL, the gradient information is forward propagated. Here, the data is collected online from the system, and the model is learned during collection. Therefore, RTRL is an online learning algorithm.[13].

The BPTT algorithm makes use of the fact that, for a finite period, there is an FFNN with identical behavior for every RNN. To obtain this FFNN, we need to unfold the RNN in time. Figure 4.3 shows a simple, fully recurrent neural network with a single two-neuron layer. The corresponding feed-forward neural network, shown in Figure 4.4, requires a separate layer for each time step with the same weights for all layers. If weights are identical to the RNN, both networks show the same behavior [13].

The unfolded network can be trained using the backpropagation algorithm. At the end of a training sequence, the network is unfolded in time. The error is calculated for the output units with existing target values using some chosen error measure. Then, the error is injected backwards into the network, and the weight updates for all time steps are calculated. The weights in the recurrent version of the network are updated with the sum of its deltas over all time steps. We calculate the error signal for a unit for all time steps in a single pass, using the following iterative backpropagation algorithm. We consider discrete-time steps $1, 2, 3...,$ indexed by the variable τ. The network starts at a point in time t' and runs until a final time t. This time frame between t' and t is called an **epoch**.

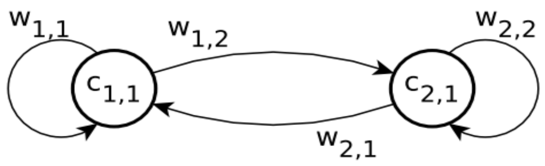


Figure 4.3: A single two neuron RNN layer

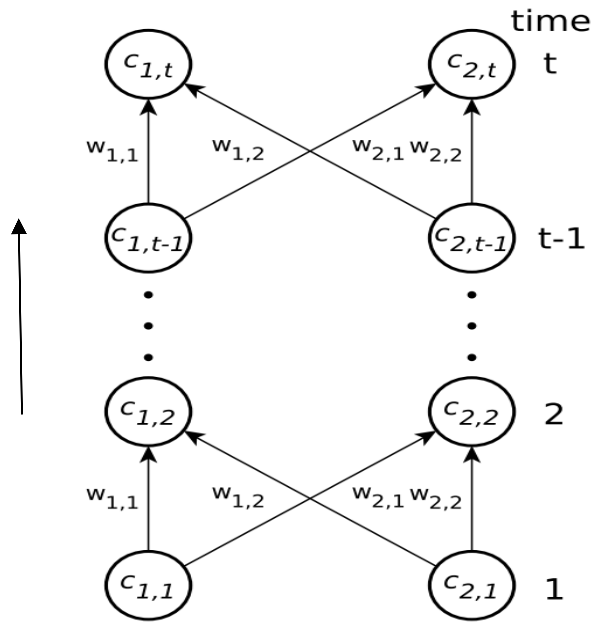


Figure 4.4: Unfolded network of figure 3.3

Let U be the set of non-input units, and let f_u be the differentiable, non-linear squashing function of the unit $u \in U$; the output $y_u(\tau)$ of u at time τ is given by:

$$y_u(\tau) = \mathtt{f}_u(z_u(\tau)) \tag{4.1}$$

with the weighted input

$$\begin{aligned} z_u(\tau+1) &= \sum_l W_{[u,l]} X_{[l,u]}(\tau+1), \quad \text{with } l \in \mathtt{Pre}\,(u) \\ &= \sum_v W_{[u,v]} y_v(\tau) + \sum_i W_{[u,i]} y_i(\tau+1) \end{aligned} \tag{4.2}$$

(Use Appendix for notations clarity.)

where $v \in U \cap Pre(u)$ and $i \in I$, the set of input units. Note that the inputs to u at time $\tau +1$ are of two types: the environmental input that arrives at time $\tau +1$ via the input units, and the recurrent output from all non-input units in the network produced at time τ. If the network is fully connected, then $U \cap Pre(u)$ is equal to the set U of non-input units. Let $T(\tau)$ be the set of non-input units for which, at time τ, the output value $y_u(\tau)$ of the unit $u \in T(\tau)$ should match some target value $d_u(\tau)$. The cost function is the summed error

$E_{total}(t',t)$ for the epoch $t', t'+1, \ldots, t$, which we want to minimize using a learning algorithm. Such total error is defined by:

$$E_{total}(t',t) = \sum_{\tau=t'}^{t} E(\tau), \qquad (4.3)$$

with the error $T(\tau)$ at time τ defined using the squared error as an objective function by

$$E(\tau) = \frac{1}{2}\sum_{u\in U}(e_u(\tau))^2, \qquad (4.4)$$

and with the error $e_u(\tau)$ of the non-input unit u at time τ defined by

$$e_u(\tau) = \begin{cases} d_u(\tau) - y_u(\tau) & \text{if } u \in T(\tau), \\ 0 & \text{otherwise.} \end{cases} \qquad (4.5)$$

To adjust the weights, we use the error signal $\vartheta_u(\tau)$ of a non-input unit u at time τ , which is defined by:

$$\vartheta_u(\tau) = \frac{\partial E(\tau)}{\partial z_u(\tau)}. \qquad (4.6)$$

When we unroll ϑ_u over time, we obtain the equality:

$$\vartheta_u(\tau) = \begin{cases} \mathtt{f}'_u(z_u(\tau))e_u(\tau) & \text{if } \tau = t, \\ \mathtt{f}'_u(z_u(\tau))\left(\sum_{k\in U} W_{[k,u]}\vartheta_k(\tau+1)\right) & \text{if } t' \le \tau < t \end{cases} \qquad (4.7)$$

After the backpropagation computation is performed down to time t' , we calculate the weight update $\Delta W[u,v]$ in the recurrent version of the network. This is done by summing the corresponding weight updates for all time steps [13]:

$$\Delta W_{[u,v]} = -\eta \frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}}$$

with,

$$\begin{aligned}\frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}} &= \sum_{\tau=t'}^{t} \vartheta_u(\tau)\frac{\partial z_u(\tau)}{\partial W_{[u,v]}} \\ &= \sum_{\tau=t'}^{t} \vartheta_u(\tau) X_{[u,v]}(\tau).\end{aligned}$$

BPTT is described in more detail in [14], [17].

4.2.3 Solving the Vanishing Error Problem

Standard RNN cannot bridge more than 5–10-time steps [18]. This is due to that back-propagated error signals tend to either grow or shrink with every time step. Over many time steps the error therefore typically blows-up or vanishes ([19, 20]). Blown-up error signals lead straight to oscillating weights, whereas with a vanishing error, learning takes an unacceptable amount of time, or does not work at all.

The explanation of how gradients are computed by the standard backpropagation algorithm and the basic vanishing error analysis is as follows: we update weights after the network has trained from time *t'* to time *t* using the formula [13]:

$$\Delta W_{[u,v]} = -\eta \frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}}$$

with,

$$\frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}} = \sum_{\tau=t'}^{t} \vartheta_u(\tau) X_{[u,v]}(\tau),$$

where the backpropagated error signal at time τ (with $t' \le \tau < t$) of the unit u is

$$\vartheta_u(\tau) = \mathtt{f}'_u(z_u(\tau)) \left(\sum_{v\in U} W_{vu}\vartheta_v(\tau+1) \right) \tag{4.8}$$

Consequently, given a fully recurrent neural network with a set of non-input units U, the error signal that occurs at any chosen output-layer neuron $o \in O$, at time-step τ , is propagated back through time for $t-t'$ time-steps, with $t' < t$ to an arbitrary neuron v. This causes the error to be scaled by the following factor:

$$\frac{\partial\vartheta_v(t')}{\partial\vartheta_o(t)} = \begin{cases} \mathtt{f}'_v(z_v(t'))W_{[o,v]} & \text{if } t-t'=1, \\ \mathtt{f}'_v(z_v(t'))\left(\sum_{u\in U}\frac{\partial\vartheta_u(t'+1)}{\partial\vartheta_o(t)}W_{[u,v]}\right) & \text{if } t-t'>1 \end{cases}$$

To solve the above equation, we unroll it over time. For $t' \le \tau \le t$, let u_τ be a non-input-layer neuron in one of the replicas in the unrolled network at time τ . Now, by setting $u_t = v$ and $u_{t'} = o$, we obtain the equation

$$\frac{\partial\vartheta_v(t')}{\partial\vartheta_o(t)} = \sum_{u_{t'}\in U}\cdots\sum_{u_{t-1}\in U}\left(\prod_{\tau=t'+1}^{t}\mathtt{f}'_{u_\tau}(z_{u_\tau}(t-\tau+t'))W_{[u_\tau,u_{\tau-1}]}\right) \quad (4.9)$$

Observing above Equation, it follows that if:

$$|\mathtt{f}'_{u_\tau}(z_{u_\tau}(t-\tau+t'))W_{[u_\tau,u_{\tau-1}]}| > 1 \quad (3.10)$$

for all τ , then the product will grow exponentially, causing the error to blow-up; moreover, conflicting error signals arriving at neuron v can lead to oscillating weights and unstable learning. If now

$$|\mathtt{f}'_{u_\tau}(z_{u_\tau}(t-\tau+t'))W_{[u_\tau,u_{\tau-1}]}| < 1 \quad (3.11)$$

for all τ , then the product decreases exponentially, causing the error to vanish, preventing the network from learning within an acceptable time period. Finally, the equation

$$\sum_{o\in O}\frac{\partial\vartheta_v(t')}{\partial\vartheta_o(t)}$$

shows that if the local error vanishes, then the global error also vanishes. A more detailed theoretical analysis of the problem with long-term dependencies is presented in [21]. The paper also briefly outlines several proposals on how to address this problem.

4.3 Stacked RNNs

A traditional RNN assumes that $h_t = f(x_t, h_{t-1})$ introduces a recurrent structure. Many previous works show that by stacking multiple RNNs on top of each other, the performance of classification or regression can be further boosted. We denote x_t as input at time t and denote h_t^k as an output of hidden nodes in the kth layer at time t. σ_b is the nonlinear activation function parameterized by b. Mathematically, the stacked RNN (sRNN) can be written as follows [16] :

$$h_t^{(k)} = \begin{cases} \sigma_b(W^{(1)} h_{t-1}^{(1)} + V x_t), & k = 1, \\ \sigma_b(W^{(k)} h_{t-1}^{(k)} + U^{(k)} h_t^{(k-1)}), & k > 1. \end{cases}$$

The first layer accepts the last moment output at the same layer h_{t-1}^1 and the current moment input x_t as its inputs. Similarly, the rest of the stacked layers accept the last moment output h_{t-1}^k at the same layer and the previous layer output h_t^{k-1} at the same moment as their inputs[16].

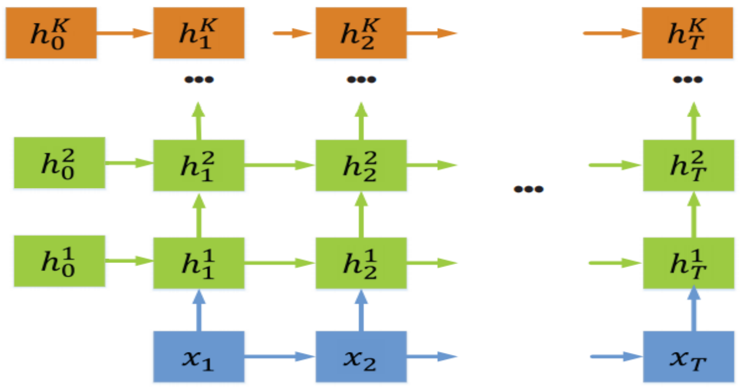


Figure 4.5 -: Vanilla Stacked RNN.

Chapter 5

Long Short-Term Memory Networks

It is challenging for RNN models to recollect the information from the initial states for long-term time dependencies due to vanishing gradients explained in chapter 3.

One solution that addresses the vanishing error problem is a gradient-based method called long short-term memory (LSTM). LSTM can learn how to bridge minimal time lags of more than 1,000 discrete time steps. The solution uses constant error carousels (CECs), which enforce a constant error flow within special cells. Access to the cells is handled by multiplicative gate units, which learn when to grant access [13].

5.1 Constant Error Carousel

Suppose that we have only one unit u with a single connection to itself. The local error back flow of u at a single time-step τ follows from Equation 4.8 and is given by:

$$\vartheta_u(\tau) = \mathtt{f}'_u(z_u(\tau))W_{[u,u]}\vartheta_u(\tau+1).$$

From Equations 4.10 and 4.11 we see that, in order to ensure a constant error flow through u, we need to have:

$$\mathtt{f}'_u(z_u(\tau))W_{[u,u]} = 1.0$$

and by integration we have

$$\mathtt{f}_u(z_u(\tau)) = \frac{z_u(\tau)}{W_{[u,u]}}.$$

From this, we learn that f_u must be linear, and that u*'s* activation must remain constant over time, i.e.,

$$y_u(\tau+1) = \mathtt{f}_u(z_u(\tau+1)) = \mathtt{f}_u(y_u(\tau)W_{[u,u]}) = y_u(\tau)$$

This is ensured by using the identity function $f_u = id$, and by setting $W_{[u,u]} = 1.0$. This preservation of error is called the constant error carousel (CEC), and it is the central feature of LSTM, where short-term memory storage is achieved for extended periods of time. Clearly, we still need to handle the connections from other units to the unit u, and this is where the different components of LSTM networks come into the picture.

5.2 Memory Blocks

In the absence of new inputs to the cell, we now know that the CEC's backflow remains constant. However, as part of a neural network, the CEC is not only connected to itself, but also to other units in the neural network. We need to take these additional weighted inputs and outputs into account. Incoming connections to neuron u can have conflicting weight update signals because the same weight is used for storing and ignoring inputs. For weighted output connections from neuron u, the same weights can be used to both retrieve u's contents and prevent u's output flow to other neurons in the network.

To address the problem of conflicting weight updates, LSTM extends the CEC with input and output gates connected to the network input layer and to other memory cells. This results in a more complex LSTM unit, called a memory block; its standard architecture is shown in Figure 5.2.

The input gates, which are simple sigmoid threshold units with an activation function range of [0, 1], control the signals from the network to the memory cell by scaling them appropriately; when the gate is closed, activation is close to zero. Additionally, these can learn to protect the contents stored in u from disturbance by irrelevant signals. The activation of a CEC by the input gate is defined as the cell state. The output gates can learn how to control access to the memory cell contents, which protects other memory cells from disturbances originating from u. So, we can see that the basic function of multiplicative gate units is to either allow or deny access to constant error flow through the CEC[13].

5.3 Training LSTM-RNNs - the Hybrid Learning Approach

In order to preserve the CEC in LSTM memory block cells, the original formulation of LSTM used a combination of two learning algorithms: BPTT to train network components located after cells, and RTRL to train network components located before and including cells. The latter units work with RTRL because there are some partial derivatives (related to the state of the cell) that need to be computed during every step, no matter if a target value is given or not at that step. For now, we only allow the gradient of the cell to be propagated through time, truncating the rest of the gradients for the other recurrent connections. We define discrete time steps in the form $\tau = 1, 2, 3, ...$. Each step has a forward pass and a backward pass; in the forward pass the output/activation of all units are calculated, whereas in the backward pass, the calculation of the error signals for all weights is performed [13].

5.3.1 The Forward Pass

Let M be the set of memory blocks. Let m_c be the c-th memory cell in the memory block m, and $W_{[u,v]}$ be a weight connecting unit u to unit v. In the original formulation of LSTM, each memory block m is associated with one input gate in_m and one output gate out_m. The internal state of a memory cell m_c at time $\tau + 1$ is updated according to its state s$m_c(\tau)$ and according to the weighted input Z$m_c(\tau +1)$ multiplied by the activation of the input gate y$in_m(\tau +1)$. Then, we use the activation of the output gate $Zout_m(\tau +1)$ to calculate the activation of the cell y$m_c(\tau +1)$.

The activation yin_m of the input gate in_m is computed as

$$y_{\mathtt{in}_m}(\tau+1) = \mathbf{f}_{\mathtt{in}_m}(z_{\mathtt{in}_m}(\tau+1)) \tag{5.1}$$

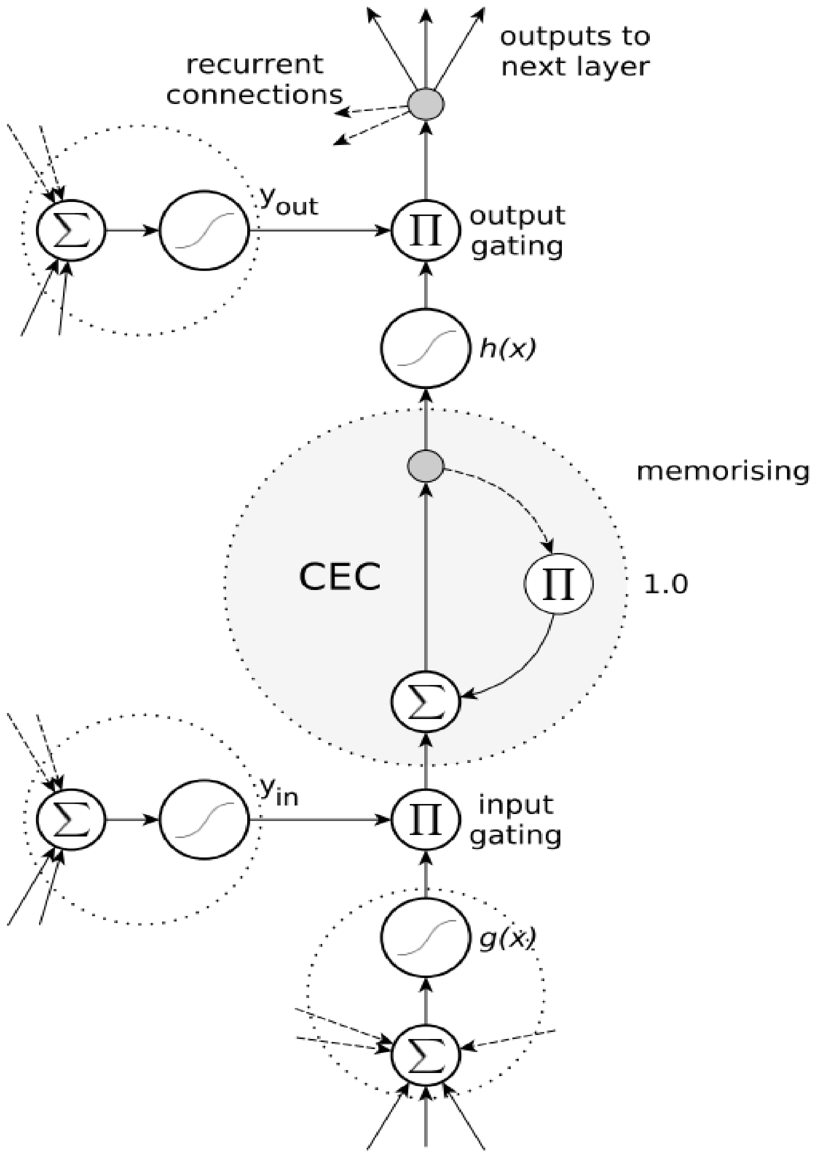


Figure 5.1: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of ‘1’. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, y_{in}, and the output gate, y_{out}. The internal cell state is calculated by multiplying the result of the squashed input, g, by the result of the input gate, y_{in}, and then adding the state of the last time step, $s_c(t-1)$. Finally, the cell output is calculated by multiplying the cell state, s_c, by the activation of the output gate, y_{out}.

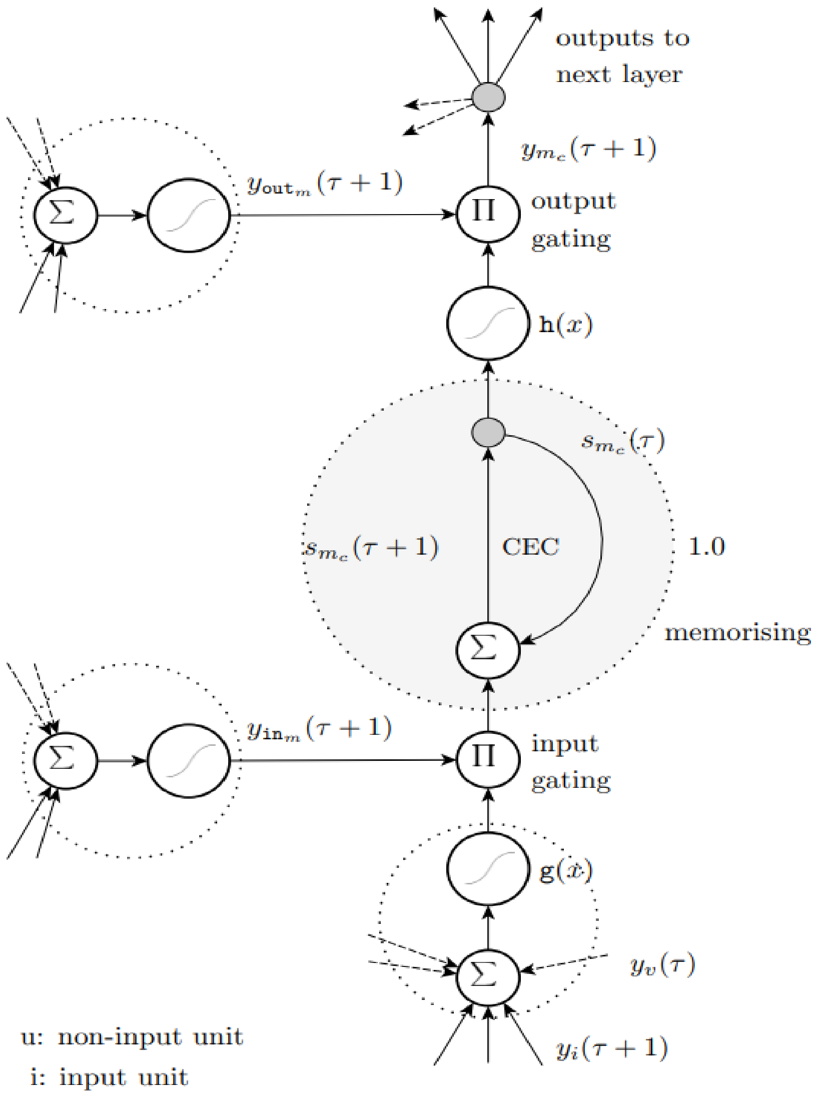


Figure 5.2: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of ‘1’. The state of the cell is denoted as s_c . Read and write access is regulated by the input gate, y_{in} , and the output gate, y_{out} . The internal cell state is calculated by multiplying the result of the squashed input, $g(x)$, by the result of the input gate and then adding the state of the current time step, $sm_c(\tau)$, to the next, $sm_c(\tau+1)$. Finally, the cell output is calculated by multiplying the cell state by the activation of the output gate.

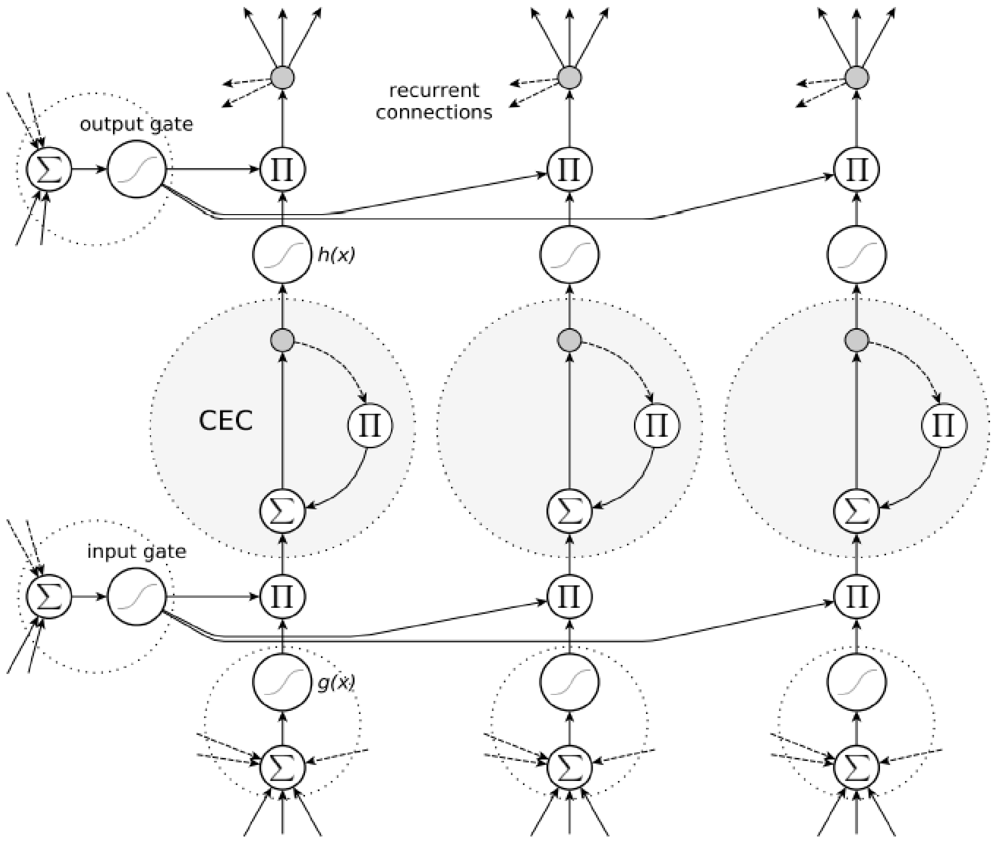


Figure 5.3 : A three cell LSTM memory block with recurrent self-connections

with the input gate input

$$
\begin{aligned}
z_{\mathtt{in}_m}(\tau+1) &= \sum_u W_{[\mathtt{in}_m,u]} X_{[u,\mathtt{in}_m]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\,(\mathtt{in}_m) \\
&= \sum_{v \in U} W_{[\mathtt{in}_m,v]} y_v(\tau) + \sum_{i \in I} W_{[\mathtt{in}_m,i]} y_i(\tau+1).
\end{aligned}
\tag{5.2}
$$

The activation of the output gate *out*$_m$ is:

$$
y_{\mathtt{out}_m}(\tau+1) = \mathtt{f}_{\mathtt{out}_m}(z_{\mathtt{out}_m}(\tau+1)) \tag{5.3}
$$

with the output gate input

$$
\begin{aligned}
z_{\mathrm{out}_m}(\tau+1) &= \sum_u W_{[\mathrm{out}_m,u]} X_{[u,\mathrm{out}_m]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\,(\mathrm{out}_m) \\
&= \sum_{v\in U} W_{[\mathrm{out}_m,v]} y_v(\tau) + \sum_{i\in I} W_{[\mathrm{out}_m,i]} y_i(\tau+1).
\end{aligned} \tag{5.4}
$$

The results of the gates are scaled using the non-linear squashing function $fin_m = fout_m = f$, defined by

$$
\mathtt{f}(s) = \frac{1}{1+e^{-s}} \tag{5.5}
$$

so that they are within the range [0, 1]. Thus, the input for the memory cell will only be able to pass if the signal at the input gate is sufficiently close to ‘1’.

For a memory cell m_c in the memory block m, the weighted input $zm_c(\tau+1)$ is defined by:

$$
\begin{aligned}
z_{m_c}(\tau+1) &= \sum_u W_{[m_c,u]} X_{[u,m_c]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}\,(m_c)\,. \\
&= \sum_{v\in U} W_{[m_c,v]} y_v(\tau) + \sum_{i\in I} W_{[m_c,i]} y_i(\tau+1).
\end{aligned} \tag{5.6}
$$

As we mentioned before, the internal state $sm_c(\tau+1)$ of the unit in the memory cell at time $(\tau+1)$ is computed differently; the weighted input is squashed and then multiplied by the activation of the input gate, and then the state of the last time step $sm_c(\tau)$ is added. The corresponding equation is:

$$
s_{m_c}(\tau+1) = s_{m_c}(\tau) + y_{\mathrm{in}_m}(\tau+1)\mathtt{g}(z_{m_c}(\tau+1)) \tag{5.7}
$$

with $sm_c(0) = 0$ and the non-linear squashing function for the cell input

$$
\mathtt{g}(z) = \frac{4}{1+e^{-z}} - 2 \tag{5.8}
$$

which, in this case, scales the result to the range [−2, 2].

The output ym_c is now calculated by squashing and multiplying the cell state sm_c by the activation of the output gate $yout_m$:

$$y_{m_c}(\tau+1) = y_{\mathtt{out}_m}(\tau+1)\mathtt{h}(s_{m_c}(\tau+1)) \tag{5.9}$$

with the non-linear squashing function

$$\mathtt{h}(z) = \frac{2}{1+e^{-z}} - 1 \tag{5.10}$$

with range [−1, 1].

Assuming a layered, recurrent neural network with standard input, standard output and hidden layer consisting of memory blocks, the activation of the output unit o is computed as

$$y_o(\tau+1) = \mathtt{f}_o(z_o(\tau+1)) \tag{5.11}$$

$$z_o(\tau+1) = \sum_{u \in U-G} W_{[o,u]} y_u(\tau+1). \tag{5.12}$$

where G is the set of gate units, and we can again use the logistic sigmoid in Equation 5.5 as a squashing function f_o.

5.3.2 Forget Gates

The self-connection in a standard LSTM network has a fixed weight set to ‘1’ in order to preserve the cell state over time. Unfortunately, the cell states s_m tend to grow linearly during the progression of a time series presented in a continuous input stream. The main negative effect is that the entire memory cell loses its memorizing capability and begins to function like an ordinary RNN network neuron.

By manually resetting the state of the cell at the beginning of each sequence, the cell state growth can be limited, but this is not practical for continuous input where there is no distinguishable end, or subdivision is very complex and error prone.

To address this problem, [18] suggested that an adaptive forget gate could be attached to the self-connection. Forget gates can learn to reset the internal state of the memory cell when the stored information is no longer needed. To this end, we replace the weight ‘1.0’ of the self-connection from the CEC with a multiplicative, forget gate activation y_ϕ, which is computed using a similar method as for the other gates:

$$y_{\varphi_m}(\tau+1) = \mathtt{f}_{\varphi_m}(z_{\varphi_m}(\tau+1) + b_{\varphi_m}) \tag{5.13}$$

where f is the squashing function from Equation 5.5 with a range [0, 1], $b_{\phi m}$ is the bias of the forget gate, and

$$\begin{aligned} z_{\varphi_m}(\tau+1) &= \sum_u W_{[\varphi_m,u]} X_{[u,\varphi_m]}(\tau+1), \quad \text{with } u \in \mathtt{Pre}(\varphi_m). \\ &= \sum_{v \in U} W_{[\varphi_m,v]} y_v(\tau) + \sum_{i \in I} W_{[\varphi_m,i]} y_i(\tau+1). \end{aligned} \tag{5.14}$$

Originally, $b_{\phi m}$ is set to 0, however, following the recommendation by [22], we fix $b_{\phi m}$ to 1, in order to improve the performance of LSTM. The updated equation for calculating the internal cell state sm_c is

$$s_{m_c}(\tau+1) = s_{m_c}(\tau)\underbrace{y_{\varphi_m}(\tau+1)}_{\substack{=1 \text{ without} \\ \text{forget gate}}} + y_{\mathtt{in}_m}(\tau+1)\mathtt{g}(z_{m_c}(\tau+1)) \tag{5.15}$$

with $sm_c(0) = 0$ and using the squashing function in Equation 5.8, with a range [−2, 2]. The extended forward pass is given simply by exchanging Equation 5.7 for Equation 5.15.

The bias weights of input and output gates are initialized with negative values, and the weights of the forget gate are initialized with positive values. From this, it follows that at the beginning of training, the forget gate activation will be close to '1.0'. The memory cell will behave like a standard LSTM memory cell without a forget gate. This prevents the LSTM memory cell from forgetting before it has actually learned anything.

4.3.3 Backward Pass

LSTM incorporates elements from both BPTT and RTRL. Thus, we separate units into two types: those units whose weight changes are computed using a variation of BPTT (i.e, output units, hidden units, and the output gates), and those whose weight changes are computed using a variation of RTRL (i.e., the input gates, the forget gates and the cells).

Following the notation used in previous sections, and using Equations 4.3 and 4.5, the overall network error at time step τ is:

$$E(\tau) = \frac{1}{2} \sum_{o \in O} (\underbrace{d_o(\tau) - y_o(\tau)}_{e_o(\tau)})^2.$$

Let us first consider units that work with BPTT. We define the notion of individual error of a unit u at time τ by

$$\vartheta_u(\tau) = -\frac{\partial E(\tau)}{\partial z_u(\tau)},$$

where z_u is the weighted input of the unit. We can expand the notion of weight contribution as follows:

$$\begin{aligned} \Delta W_{[u,v]}(\tau) &= -\eta \frac{\partial E(\tau)}{\partial W_{[u,v]}} \\ &= -\eta \frac{\partial E(\tau)}{\partial z_u(\tau)} \frac{\partial z_u(\tau)}{\partial W_{[u,v]}}. \end{aligned}$$

The factor $\partial z_u(\tau)/\ \partial W_{[u,v]}$ corresponds to the input signal that comes from the unit v to the unit u. However, depending on the nature of u, the individual error varies. If u is equal to an output unit o, then:

$$\vartheta_o(\tau) = \mathtt{f}'_o(z_o(\tau))(d_o(\tau) - y_o(\tau))$$

thus, the weight contribution of output units is

$$\Delta W_{[o,v]}(\tau) = \eta \vartheta_o(\tau) X_{[v,o]}(\tau).$$

Now, if u is equal to a hidden unit h located between cells and output units, then:

$$\vartheta_h(\tau) = \mathtt{f}'_h(z_h(\tau)) \left(\sum_{o \in O} W_{[o,h]} \vartheta_o(\tau) \right)$$

where O is the set of output units, and the weight contribution of hidden units is:

$$\Delta W_{[h,v]}(\tau) = \eta \vartheta_h(\tau) X_{[v,h]}(\tau)$$

Finally, if u is equal to the output gate out_m of the memory block m, then:

$$\vartheta_{\mathtt{out}_m}(\tau) \overset{\mathrm{tr}}{=} \mathtt{f}'_{\mathtt{out}_m}(z_{\mathtt{out}_m}(\tau)) \left(\sum_{m_c \in m} \mathtt{h}(s_{m_c}(\tau)) \sum_{o \in O} W_{[o,m_c]} \vartheta_o(\tau) \right)$$

where (tr) means the equality only holds if the error is truncated so that it does not propagate "too much"; that is, it prevents the error from propagating back to the unit via its own feedback connection. Finally, the weight contribution for output gates is :

$$\Delta W_{[\mathtt{out}_m, v]}(\tau) = \eta \vartheta_{\mathtt{out}_m}(\tau) X_{[v, \mathtt{out}_m]}(\tau)$$

Let us now consider units that work with RTRL. In this case, the individual errors of the input gate and the forget gate revolve around the individual error of the cells in the memory block. We define the individual error of the cell m_c of the memory block m by

$$\begin{aligned}
\vartheta_{m_c}(\tau) &\overset{\mathrm{tr}}{=} -\frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} + \underbrace{\vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1)}_{\text{recurrent connection}} \\
&\overset{\mathrm{tr}}{=} \frac{\partial y_{m_c}(\tau)}{\partial s_{m_c}(\tau)} \left(\sum_{o \in O} \frac{\partial z_o(\tau)}{\partial y_{m_c}(\tau)} \left(-\frac{\partial E(\tau)}{\partial z_o(\tau)} \right) \right) + \vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1) \\
&\overset{\mathrm{tr}}{=} y_{\mathtt{out}_m}(\tau) \mathtt{h}'(s_{m_c}(\tau)) \left(\sum_{o \in O} W_{[o,m_c]} \vartheta_o(\tau) \right) + \vartheta_{m_c}(\tau+1) y_{\varphi_m}(\tau+1).
\end{aligned}$$

Note that this equation does not consider the recurrent connection between the cell and other units, propagating back in time only the error through its recurrent connection (accounting for the influence of the forget gate). We use the following partial derivatives to expand the weight contribution for the cell as follows:

$$\begin{aligned}\Delta W_{[m_c,v]}(\tau) &= -\eta \frac{\partial E(\tau)}{\partial W_{[m_c,v]}} \\ &= -\eta \frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}} \\ &= \eta \vartheta_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}}\end{aligned}$$

and the weight contribution for forget and input gates as follow:

$$\begin{aligned}\Delta W_{[u,v]}(\tau) &= -\eta \frac{\partial E(\tau)}{\partial W_{[u,v]}} \\ &= -\eta \sum_{m_c \in m} \frac{\partial E(\tau)}{\partial s_{m_c}(\tau)} \frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}} \\ &= \eta \sum_{m_c \in m} \vartheta_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}}.\end{aligned}$$

Now, we need to define what is the value of $\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[u,v]}}$. As expected, these also depend on the nature of the unit u. If u is equal to the cell m_c, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[m_c,v]}} \overset{\text{tr}}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}} y_{\varphi_m}(\tau+1) + \mathbf{g}'(z_{m_c}(\tau+1)) \mathtt{f}_{\mathtt{in}_m}(z_{\mathtt{in}_m}(\tau+1)) y_v(\tau).$$

Now, if u is equal to the input gate $\mathtt{in}_m$, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[\mathtt{in}_m,v]}} \overset{\text{tr}}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[\mathtt{in}_m,v]}} y_{\varphi_m}(\tau+1) + \mathbf{g}(z_{m_c}(\tau+1)) \mathtt{f}'_{\mathtt{in}_m}(z_{\mathtt{in}_m}(\tau+1)) y_v(\tau).$$

Finally, if u is equal to a forget gate φ_m, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[\varphi_m,v]}} \overset{\text{tr}}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[\varphi_m,v]}} y_{\varphi_m}(\tau+1) + s_{m_c}(\tau) \mathtt{f}'_{\varphi_m}(z_{\varphi_m}(\tau+1)) y_v(\tau).$$

with $s_{m_c}(0) = 0$. A more detailed version of the LSTM backward pass with forget gates is described in [18]

5.3.4 Strengths and limitations of LSTM-RNNs

According to [23], LSTM excels on tasks in which a limited amount of data must be remembered for a long time. This property is attributed to the use of memory blocks. Memory blocks are interesting constructions: they have access control in the form of input and output gates, which prevent irrelevant information from entering or leaving the memory block. Memory blocks also have a forget gate which weights the information inside the cells, so whenever previous information becomes irrelevant for some cells, the forget gate can reset the state of the different cell inside the block. Forget gates also enable continuous prediction, because they can make cells completely forget their previous state, preventing biases in prediction. Like other algorithms, LSTM requires the topology of the network to be fixed a priori. The number of memory blocks in networks does not change dynamically, so the memory of the network is ultimately limited. Moreover, [23] point out that it is unlikely to overcome this limitation by increasing the network size homogeneously and suggest that modularization promotes effective learning. The process of modularization is, however, “not generally clear”.

5.3.5 Stacked LSTMs

LSTM-RNN permits many different variants and topologies. They are problem specific. Here we are using Stacked LSTM.

A stacked LSTM , as its name suggests, stacks LSTM layers on top of each other in order to increase capacity. At a high level, to stack N LSTM networks, we make the first network have X_1 as defined in Equation (5.16) , but we make the i-th network have X_i defined by

$$X_i = \begin{bmatrix} \vec{y_h}_{i-1} \\ \vec{y_h}_i \end{bmatrix} \tag{5.16}$$

instead, replacing the input signals **x** with the hidden signals from the previous LSTM transform, effectively “stacking” them.

5.4 Input to LSTMs

Inputs to an LSTM network are supposed to be given in the form of a tensor or a 3-D matrix. In this 3-D matrix, the rows are total instances; columns are the number of time-steps that your data traces back to or depends on, and width is the content of each time-step. Therefore, in our case, the shape of our matrix came out to be (25760, 101, 13) for 101 frames and 13 MFCCs per frame. We can see an example of features for one instance in table 5.1, where Γ is a single coefficient, and n denotes nth frame from a total of 101 frames. We can imagine the whole dataset as all the tables like this for different instances stacked on top of each other.

$x_n^{(i)}$	1	2	…….	101
1	1st Γ of 1st n	1st Γ of 2nd n		1st Γ of 101st n
2	2nd Γ of 1st n	2nd Γ of 2nd n		2nd Γ of 101st n
3	3rd Γ of 1st n	3rd Γ of 2nd n		3rd Γ of 101st n
4	4th Γ of 1st n	4th Γ of 2nd n		4th Γ of 101st n
5	5th Γ of 1st n	5th Γ of 2nd n		5th Γ of 101st n
6	6th Γ of 1st n	6th Γ of 2nd n		6th Γ of 101st n
7	7th Γ of 1st n	7th Γ of 2nd n		7th Γ of 101st n
8	8th Γ of 1st n	8th Γ of 2nd n		8th Γ of 101st n
9	9th Γ of 1st n	9th Γ of 2nd n		9th Γ of 101st n
10	10th Γ of 1st n	10th Γ of 2nd n		10th Γ of 101st n
11	11th Γ of 1st n	11th Γ of 2nd n		11th Γ of 101st n
12	12th Γ of 1st n	12th Γ of 2nd n		12th Γ of 101st n
13	13th Γ of 1st n	13th Γ of 2nd n		13th Γ of 101st n

Table 5.1 : Description of MFCCs for one instance of command

For the purpose of traditional machine learning we need to convert all the features of one particular command instance into one feature vector. To do this, we have flattened the individual matrices as described in Table 5.1 by concatenating each set of 13 MFCCs one after another to get individual feature vectors of length (13 × 101) each for each command instance. The new feature vector can be represented as $z_k^{(j)}$, where $k = 1, 2, . . . , 1313$ is the feature number and i tells us which command instance the particular feature vector belongs to. Then, to get the complete feature matrix, we append each such feature vector one below the other to get the complete data matrix. After this a column indicating the classes of instances is appended horizontally with the obtained matrix. The ‘up’ command is assigned ‘0’, ‘down’ command is assigned ‘1’ and ‘stop’ command is assigned ‘2’ and so on.

Chapter 6
Model Discussion

Using the extracted MFCCs, we need to make a feature matrix of the MFCCs for use with deep learning architectures.

6.1 Using 13 MFCCs

After the decorrelation procedure through taking DCT of the log energies, the higher-order coefficients contain information about the pitch of the sound. For our purpose of speech recognition, however, we do not need the pitch details, and we can get good results using only the first 13 coefficients. We have decided to use the first 13 MFCCs for our purpose, which will give us good accuracy because it contains information about the formants of the sound. Also, increasing the number of MFCCs will not improve the accuracy in a tangible way; rather, it will increase the complexity of the system.

6.2 Data Matrix

To illustrate this, we are taking from the google voice commands dataset [4] the commands 'up,' 'down,' 'left,' 'right,' 'forward,' 'backward,' 'on,' 'off,' 'follow,' 'go,' 'stop' with a total of 25,760 voice samples. All these voice commands are of fixed 1-second length. After preprocessing, we obtain 101 frames of each voice sample and 13 MFCCs of each frame. Each following frame is dependent on the previous frame, which is termed time dependency or temporal nature. Thus, for each voice sample, we have a 2-Dimensional data matrix where rows represent each frame's content, i.e. 13 MFCCs for that frame, and columns represent the number of dependent frames on each other. We obtain a matrix of size 13x101, as shown in table 5.1, for each instance of a voice sample.

LSTM or rather RNN’s structure as shown in previous section demands data in a 3-Dimensional tensor form. In this 3-D tensor, the rows are total instances, columns are the number of time-steps that your data traces back to or depends on and width is the content of each time-step(in our case frame's MFCCs). What we've shown in table 5.1 is just for one instance of voice command i.e., whole data matrix which is to be fed to the network can be imagined as many tables just like table 5.1 stacked on top of each other just like in figure 6.1 in which rows correspond to instances, columns to time-steps/frames and depth to content of each time-step/frame (MFCCs). Therefore, in our case, the shape of our 3D matrix came out to be (25760, 101, 13) for 101 frames and 13 MFCCs per frame.

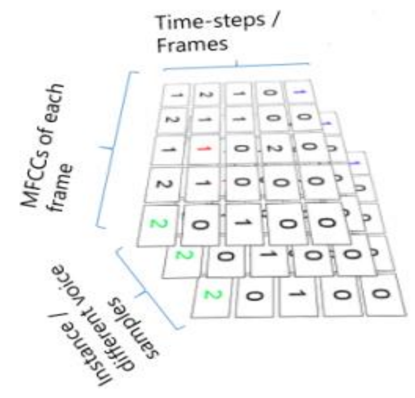
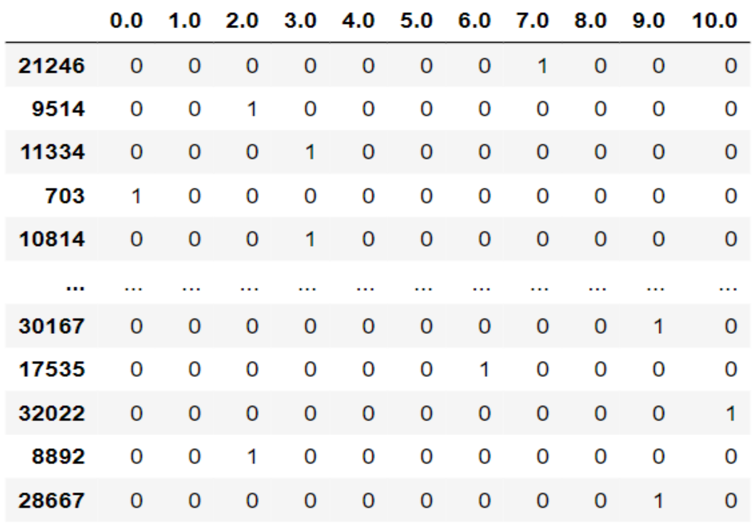


Figure 6.1: A 3-D tensor depicting rows, columns and depth

For the sake of clarity, table 5.1 corresponds to just one voice command where rows are MFCCs, and columns are frames in sequential order. Whereas figure 6.1 is essentially the whole dataset; a collection of tables stacked on top of each other where now the depth is equivalent to what table's rows(MFCCs) corresponded and columns are equivalent to table's columns(Frames) and additionally the rows correspond to all the instances of voice commands.

From table 5.1, all the columns(frames) are in a sequential order i.e., frame 2 is dependent on frame 1, frame 3 on frame 2 and so on. This dependency is both due to the temporal nature of voice and due to overlapping of voice windows when going through the process of generating Mel Frequency Cepstral Coefficients.

We used One hot encoding for assigning every example the class (here command) It belongs. The first column in the figure 6.2 shows the random example number take from the dataset:



	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
21246	0	0	0	0	0	0	0	1	0	0	0
9514	0	0	1	0	0	0	0	0	0	0	0
11334	0	0	0	1	0	0	0	0	0	0	0
703	1	0	0	0	0	0	0	0	0	0	0
10814	0	0	0	1	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
30167	0	0	0	0	0	0	0	0	0	1	0
17535	0	0	0	0	0	0	1	0	0	0	0
32022	0	0	0	0	0	0	0	0	0	0	1
8892	0	0	1	0	0	0	0	0	0	0	0
28667	0	0	0	0	0	0	0	0	0	1	0

Figure 6.2: Classes representing into binary form

6.3 Hyperparameters:

As a part of every machine learning task, tuning of hyperparameters is extremely important in order to achieve desired accuracy. We are using LSTM layers from keras [32] which gives flexibility of many hyperparameters so that we can tune them as per our requirement. The task of hyperparameter tuning is highly selective and value of every hyperparameter changes according to the type of data with us.

6.3.1 Optimizer

The optimizer is responsible for minimizing the objective function of the neural network. A commonly selected optimizer is stochastic gradient descent (SGD), which has proved itself as an efficient and effective optimization method for a large number of published machine learning systems. However, SGD can be quite sensitive towards the selection of the learning rate. Choosing a too large rate can cause the system to diverge in terms of the objective function and choosing a too low-rate results in a slow learning process. Further, SGD has trouble navigating ravines and at saddle points. To eliminate the shortcomings of SGD, other gradient based optimization algorithms have been proposed. Namely Adagrad, Adadelta , RMSProp and Adam. Adam optimizer has given promising results in literature and thus we will use this as our optimizer while showing a comparison with others.

6.3.2 Dropout

Dropout is a popular method to deal with overfitting for neural networks [40]. Three most commonly used dropout variants are no dropout, naive dropout, and variational dropout. In Naive Dropout, we apply a randomly selected dropout mask for each LSTM output. The mask changes from time step to time step while the recurrent connections are not dropped. As noted by Gal and Ghahramani [41], this form of dropout is suboptimal for recurrent neural networks. They [41] proposed to use Variational Dropout instead in which they propose to use the same dropout mask for all timesteps of the recurrent layer, i.e., at each timestep the same positions of the output are dropped. They also propose to use the same strategy to drop the recurrent connections. The fraction p of dropped dimensions is a hyperparameter and is selected randomly from the set 0.0, 0.05, 0.1, 0.25, 0.5 but can be any float value from 0 to 1 as mentioned in keras documentation [32]. Note, for variational dropout, the fraction p is selected independently for the output units as well as for the recurrent units.

6.3.3 Number of Layers

Decision of Number of LSTM and RNN layers or rather depth of our model has a tradeoff between computational complexity and fine tuning of model. It is possible that we achieve higher accuracy with increase in the depth of our model, but this increases the time required for our model to train substantially. There are some studies which also shows that more than 3 stacked layers can hamper the model’s performance. In general, 2 stacked LSTM layers are commonly used [42].

6.3.4 Number of Units

This is a parameter that varies from problem to problem and can only be set with domain knowledge. For our dataset of Mel Frequency Cepstral Coefficients, as we are accounting for 101-time instances, we will be using 101 recurrent units. As we go deeper into the network, incrementing recurrent units is forbidden [42] and therefore we will be keeping

101 units in our stacked LSTM layer also.

6.3.5 Epochs

Number of epochs are defined as the number of times we are training our model or the number of times we are going back and reuse the dataset to update weights once the whole training is done. It is advised not to keep the number very high as it causes the model to overfit and costs poor testing accuracy. The value should not be too less either as we do not want our learning to stop while our model has not converged yet. This parameter is rather an empirical one and is selected based on hit and trial methods and can be reduced once for high validation accuracy, you start getting lower testing accuracy. Here we are employing early stopping, which is widely used in LSTM and RNN applications to remove overfitting.

6.3.6 Early Stopping

Early stopping is an optimization technique used to reduce overfitting without compromising on model accuracy. The main idea behind early stopping is to stop training before a model starts to overfit [43,44].

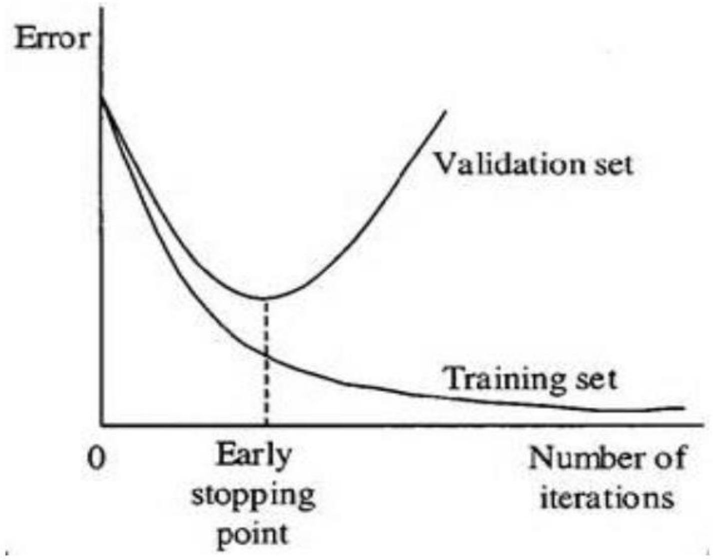


Figure 6.3: Early Stopping Point

Early Stopping Approaches

There are three main ways early stopping can be achieved. Let us look at each of them:

Training model on a pre-set number of epochs

This method is a simple, but naive way to early stop. By running a set number of epochs, we run the risk of not reaching a satisfactory training point. With a higher learning rate, the model might possibly converge with fewer epochs, but this method requires a lot of trial and error. Due to the advancements in machine learning, this method is obsolete.

Stop when the loss function update becomes small:

This approach is more sophisticated than the first as it is built on the fact that the weight updates in gradient descent become significantly smaller as the model approaches minima. Usually, the training is stopped when the update becomes as small as 0.001, as stopping at this point minimizes loss and saves computing power by preventing any unnecessary epochs. However, overfitting might still occur.

Validation set strategy:

This clever technique is the most popular early stopping approach. To understand how it works, it is important to look at how training and validation errors change with the number of epochs (as in the figure 6.3). The training error decreases exponentially until increasing epochs no longer have such a large effect on the error. The validation error, however, initially decreases with increasing epochs, but after a certain point, it starts increasing. This is the point where a model should be early stopped as beyond this the model will start to overfit.

Although the validation set strategy is the best in terms of preventing overfitting, it usually takes many epochs before a model begins to overfit, which could cost a lot of computing power. A smart way to get the best of both worlds is to devise a hybrid approach between the validation set strategy and then stop when the loss function update becomes small. For example, the training could stop when either of them is achieved.

6.4 Structure of inputs fed to RNN and outputs obtained:

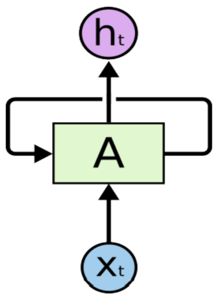
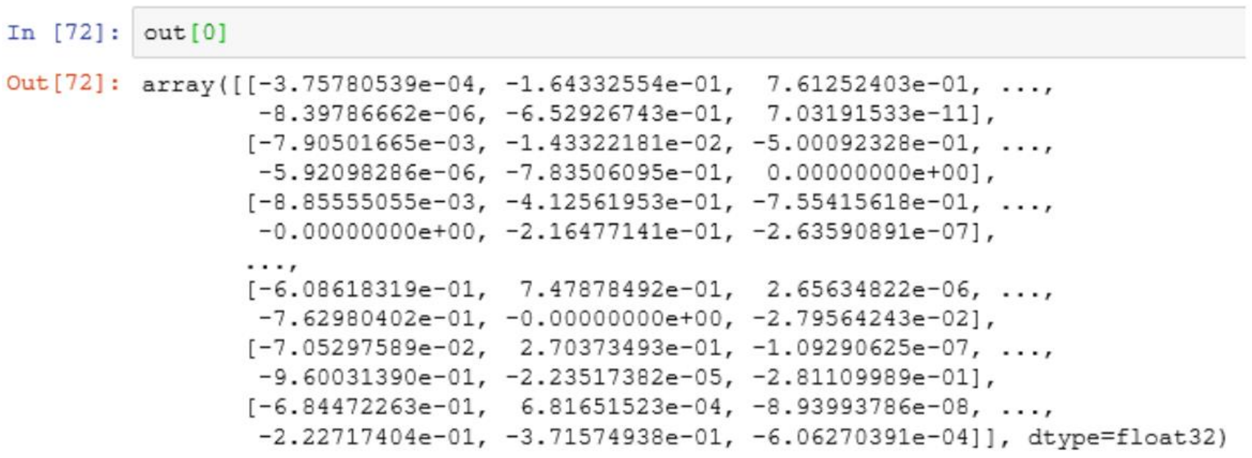


Figure 6.4: Feeding I/Ps to RNNs

To get an idea of how the input is fed to the Recurrent Neural Networks, consider the figure 6.4. In this figure, we provide inputs time-step wise i.e., all the frames in an order. The input x_1 is frame 1 column from table 5.1, input x_1 is frame 2 column and so on. But from the definition of RNN, the outputs from each cell (h_o, h_1, etc.) also gets fed to the next cell. This way RNNs consider all previous inputs along with current input. Finally, after all the inputs are fed, from the last cell, we get a final output which is supposed to be the probability of labels for that voice command. This completes our one epoch. Now, the error is calculated and is minimized using gradient descent algorithm.

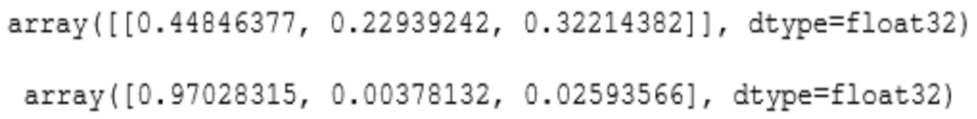
In our case, we have 101 frames for each input instance and thus, we have chosen 101 recurrent cells. For each input frame, we will get an output which will then be fed to next input cell and so on and so forth. To visualize this, consider figure 6.5 which shows a matrix of size 13x101 (13 for MFCCs and 101 for 101 recurrent cells) which is a collection of outputs from each cell carried over to the next cell.



```
In [72]: out[0]

Out[72]: array([[-3.75780539e-04, -1.64332554e-01,  7.61252403e-01, ...,
                 -8.39786662e-06, -6.52926743e-01,  7.03191533e-11],
                [-7.90501665e-03, -1.43322181e-02, -5.00092328e-01, ...,
                 -5.92098286e-06, -7.83506095e-01,  0.00000000e+00],
                [-8.85555055e-03, -4.12561953e-01, -7.55415618e-01, ...,
                 -0.00000000e+00, -2.16477141e-01, -2.63590891e-07],
                ...,
                [-6.08618319e-01,  7.47878492e-01,  2.65634822e-06, ...,
                 -7.62980402e-01, -0.00000000e+00, -2.79564243e-02],
                [-7.05297589e-02,  2.70373493e-01, -1.09290625e-07, ...,
                 -9.60031390e-01, -2.23517382e-05, -2.81109989e-01],
                [-6.84472263e-01,  6.81651523e-04, -8.93993786e-08, ...,
                 -2.22717404e-01, -3.71574938e-01, -6.06270391e-04]], dtype=float32)
```

Figure 6.5: LSTM Hidden cell outputs

Finally, after one epoch, we get outputs in the form of probabilities that the current input belongs to which class. Figure 6.6's upper part shows the probabilities of the input sample for each class after just one epoch (first three voice commands). This output is then compared with the one-hot output labels ([1, 0, 0] for this case). After each epoch, model tries to maximize the probability of the class to which the voice sample belongs to using gradient descent optimizer. This maximized and much improved probabilities after completion of training can be seen in the figure 6.6's lower part. We can see that probability that the voice sample belongs to class 1 has increased from 0.45 to 0.97 and all others have decreased (effectively minimizing the difference from actual output (1,0,0)).



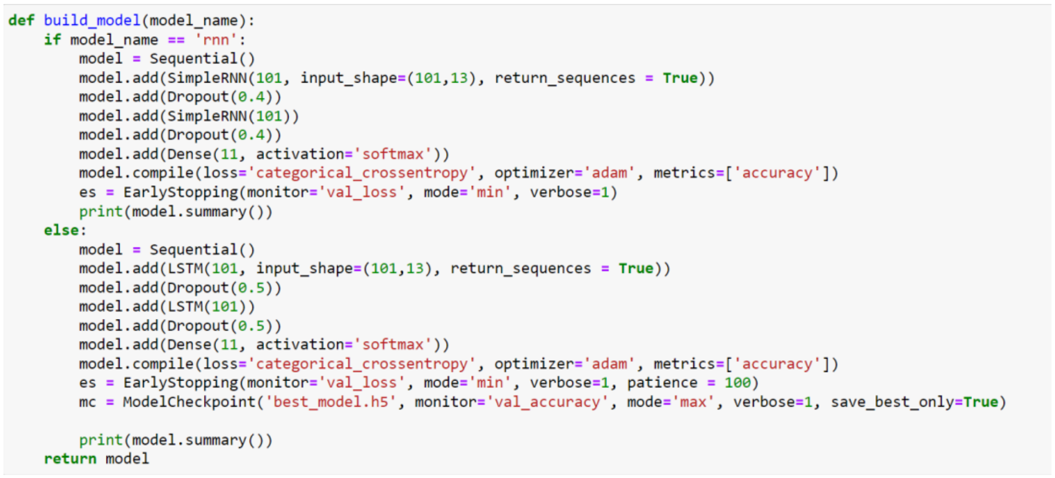
```
array([[0.44846377, 0.22939242, 0.32214382]], dtype=float32)

 array([0.97028315, 0.00378132, 0.02593566], dtype=float32)
```

Figure 6.6: Probabilities of sample 1 being in each class.

Chapter 7
Results and Conclusion

This case study compares results obtained from Stacked Vanilla RNN and LSTM on the google voice commands dataset [4]. As a part of hyperparameter tuning, we referred to Nils et al. [45] and tuned on the number of epochs, RNN layers, and dropout values. We kept the optimizer as Adam with loss function as categorical cross-entropy, which calculates the difference of predicted probabilities and actual output and then considering one as ideal output probability for correct class and 0 as ideal for the incorrect class, it maximizes these probabilities, i.e., minimizing the difference between predicted and actual probabilities and shown in figure 6.6. We kept the number of recurrent units as 101 for the input layer and each hidden layer as we have 101-time steps in our data or our data traces back 101 steps as 101 frames for each voice command instance. The activation function in the output layer is SoftMax responsible for the outputs in the form of probability.

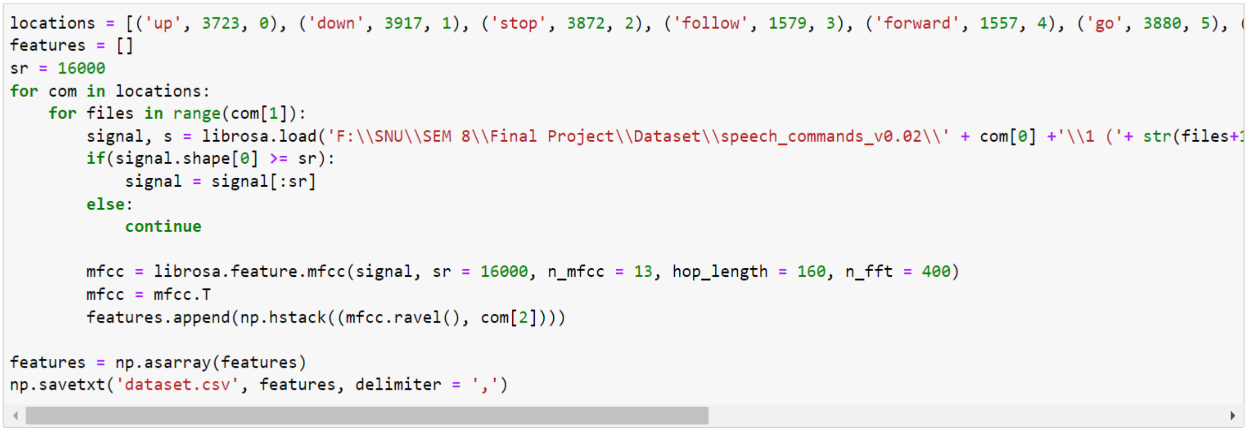
Here we have used 25,760 audio files in which 50% is used for training, 30% is used for validating, and 20% is used for testing. Keras Library of TensorFlow framework is employed for generating the required stacked RNN and stacked LSTM model. Given below is the function which we used for generating the required models [46].



```python
def build_model(model_name):
    if model_name == 'rnn':
        model = Sequential()
        model.add(SimpleRNN(101, input_shape=(101,13), return_sequences = True))
        model.add(Dropout(0.4))
        model.add(SimpleRNN(101))
        model.add(Dropout(0.4))
        model.add(Dense(11, activation='softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
        es = EarlyStopping(monitor='val_loss', mode='min', verbose=1)
        print(model.summary())
    else:
        model = Sequential()
        model.add(LSTM(101, input_shape=(101,13), return_sequences = True))
        model.add(Dropout(0.5))
        model.add(LSTM(101))
        model.add(Dropout(0.5))
        model.add(Dense(11, activation='softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
        es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience = 100)
        mc = ModelCheckpoint('best_model.h5', monitor='val_accuracy', mode='max', verbose=1, save_best_only=True)

        print(model.summary())
    return model
```

Figure 7.1: Python function for generating LSTM and RNN models.

For extracting MFCC directly, we have used librosa [47], a python package for music and audio analysis. It provides the building blocks necessary to create music information retrieval systems. Figure 7.2; shows the code with the required parameters for extracting the features.



```
locations = [('up', 3723, 0), ('down', 3917, 1), ('stop', 3872, 2), ('follow', 1579, 3), ('forward', 1557, 4), ('go', 3880, 5), (
features = []
sr = 16000
for com in locations:
    for files in range(com[1]):
        signal, s = librosa.load('F:\\SNU\\SEM 8\\Final Project\\Dataset\\speech_commands_v0.02\\' + com[0] +'\\1 ('+ str(files+1
        if(signal.shape[0] >= sr):
            signal = signal[:sr]
        else:
            continue

        mfcc = librosa.feature.mfcc(signal, sr = 16000, n_mfcc = 13, hop_length = 160, n_fft = 400)
        mfcc = mfcc.T
        features.append(np.hstack((mfcc.ravel(), com[2])))

features = np.asarray(features)
np.savetxt('dataset.csv', features, delimiter = ',')
```

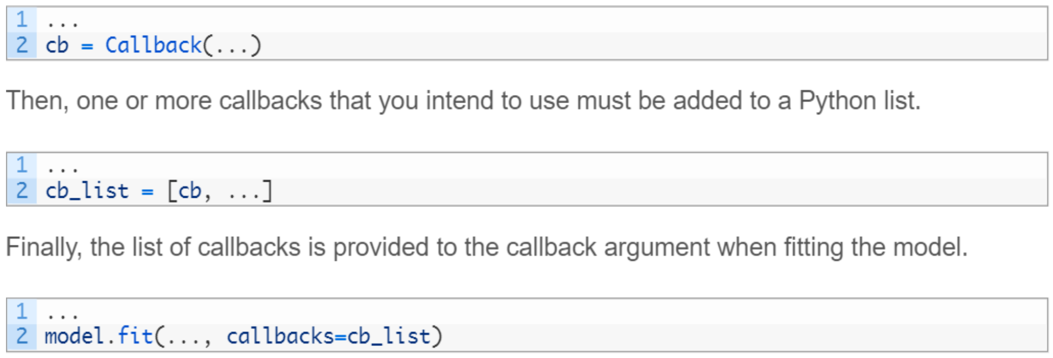
Figure 7.2: Python code for extracting MFCC from librosa

We have used Early stopping, HDF5[48] callbacks for stopping and saving the best model. An HDF5 file is a container for two kinds of objects: datasets, array-like collections of data, and groups, which are folder-like containers that hold datasets and other groups. For more details, read [48].

7.1 Using Callbacks in Keras:

Call backs provide a way to execute code and interact with the training model process automatically.

Call backs can be provided to the *fit()* function via the “*callbacks*” argument.
First, call-backs must be instantiated [49].



```
...
cb = Callback(...)
```

Then, one or more callbacks that you intend to use must be added to a Python list.

```
...
cb_list = [cb, ...]
```

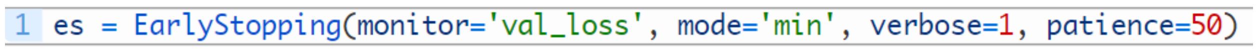
Finally, the list of callbacks is provided to the callback argument when fitting the model.

```
...
model.fit(..., callbacks=cb_list)
```

Keras supports the early stopping of training via a callback called EarlyStopping. This callback allows us to specify the performance measure to monitor, the trigger, and once triggered, it will stop the training process. The EarlyStopping callback is configured when instantiated via arguments. The "monitor" allows you to specify the performance measure to monitor in order to end training. The calculation of measures on the validation dataset will have the 'val_' prefix, such as 'val_loss 'for the loss on the validation dataset. Here we are monitoring 'val_loss' so as not to overfit our desired model.

Based on the choice of the performance measure, the "mode" argument will need to be specified as to whether the objective of the chosen metric is to increase (maximize or 'max') or to decrease (minimize or 'min'). We would seek a minimum for validation loss, and a minimum for validation mean squared error, whereas we would seek a maximum for validation accuracy. Here we are going to monitor min for val_loss and max for val_accuracy. By default, the mode is set to 'auto' and knows that you want to minimize loss or maximize accuracy. Training will stop when the chosen performance measure stops improving. To discover the training epoch on which training was stopped, the "*verbose*" argument can be set to 1. Once stopped, the callback will print the epoch number.

Often, the first sign of no further improvement may not be the best time to stop training. This is because the model may coast into a plateau of no improvement or even get slightly worse before getting much better. We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the "patience" argument. The exact amount of patience will vary between models and problems. Reviewing plots of performance measures can be very useful to understand how noisy the optimization process for our model on our data may be.
By default, any change in the performance measure, no matter how fractional, will be considered an improvement. For more read at [50].



```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
```

The EarlyStopping callback will stop training once triggered, but the model at the end of training may not be the model with the best performance on the validation dataset. An additional callback is required that will save the best model observed during training for later use. This is the ModelCheckpoint callback.

The ModelCheckpoint callback is flexible in how it can be used, but we will use it only to save the best model observed during training as defined by a chosen performance measure on the validation dataset. Saving and loading models require that HDF5 support has been installed on the workstation. The callback will save the model to file, which requires that a path and filename be specified.

The preferred loss function to be monitored can be specified via the monitor argument, in the same way as the EarlyStopping callback—for example, loss on the validation dataset (the default).

Also, as with the EarlyStopping callback, we must specify the "mode" to minimize or maximize the performance measure. We are only interested in the best model observed during training, rather than the best compared to the previous epoch, which might not be the best overall if training is noisy. This can be achieved by setting the "save_best_only" argument to True. That is all needed to ensure the model with the best performance is saved when using early stopping or in general. The saved model can then be loaded and evaluated at any time by calling the load_model() function. For more read at [50].

We have used the following callbacks and parameters for our model:

```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience = 100)
mc = ModelCheckpoint('best_model.h5', monitor='val_accuracy', mode='max', verbose=1, save_best_only=True)
```

7.2 Accuracy:

After thoroughly training and testing, we have achieved an average of **94.3% accuracy** on the test set on LSTM architecture and 42% accuracy on RNN architecture, which can further be improved by adjusting hyperparameters. This analysis sums up our point of discussion that LSTM are better in solving vanishing gradient problems.

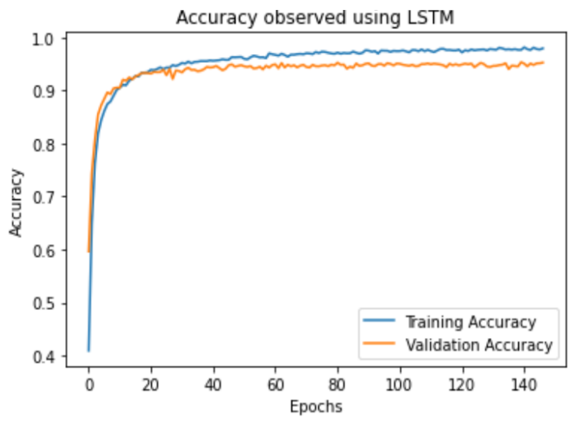


Figure 7.3: Comparison of training and validation accuracy of LSTM

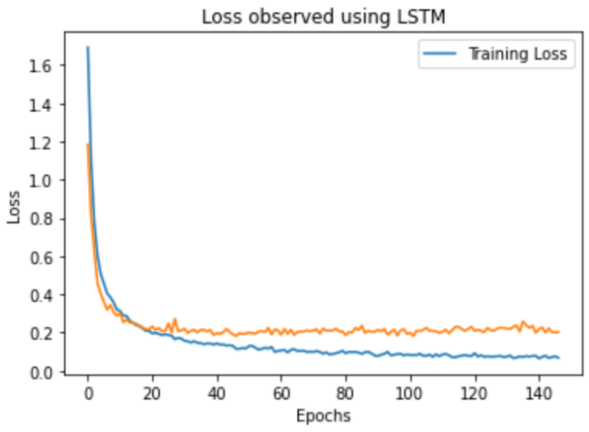


Figure 7.4: Comparison of training (Blue) and validation (Orange) loss of LSTM

Chapter 6
Future Prospects

The above study can further be extended to any voice command dataset with a rich vocabulary for more precise movements of industrial robots. Different topologies of LSTMs can be applied to increase the accuracy of the model. A combination of CNNs+RNNs can be applied, and instead of choosing MFFCs as features, spectrograms could be used for better analysis. With further improvement and research, this model can be deployed in real-world scenarios.

Appendix

- The learning rate of the network is η.
- A time unit is τ. Initial times of an epoch are denoted by t' and final times by t.
- The set of units of the network is N, with generic (unless stated otherwise) units $u, v, l, k \in N$.
- The set of input units is I, with input unit $i \in I$.
- The set of output units is O, with output unit $o \in O$.
- The set of non-input units is U.

- The output of a unit u (also called the activation of u) is y_u, and unlike the input, it is a single value.
- The set of units with connections to a unit u; i.e., its predecessors, is $\mathtt{Pre}\,(u)$
- The set of units with connections from a unit u; i.e., its successors, is $\mathtt{Suc}\,(u)$
- The weight that connects the unit v to the unit u is $W_{[v,u]}$.
- The input of a unit u coming from a unit v is denoted by $X_{[v,u]}$
- The weighted input of the unit u is z_u.
- The bias of the unit u is b_u.
- The state of the unit u is s_u.
- The squashing function of the unit u is $\mathtt{f}_u$.
- The error of the unit u is e_u.
- The error signal of the unit u is ϑ_u.
- The output sensitivity of the unit k with respect to the weight $W_{[u,v]}$ is p^k_{uv}.

References

[1] **Kibria, Shafkat** (2005) Speech Recognition for Robotic Control. 10.13140/RG.2.2.20304.05129.

[2] https://en.wikipedia.org/wiki/Industrial_robot

[3] Speech Commands: A Dataset for Limited – Vocabulary Speech Recognition by Pete Warden

[4] **Warden P. Speech Commands**: A public dataset for single-word speech recognition, 2017. Available from http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz

[5] https://towardsdatascience.com/how-to-apply-machine-learning-and-deep-learning-methods-to-audio-analysis-615e286fcbbc

[6] **S. Paulose, D. Mathew, and A. Thomas,** "Performance evaluation of different modeling methods and classifiers with mfcc and ihc features for speaker recognition,"Procedia Computer Science, vol. 115, p. 55–62, 2017

[7] An Approach to Extract Feature using MFCC by **Parwinder Pal Singh, Pushpa Rani**

[8] Feature Extraction Using MFCC by **Shikha Gupta, Jafreezal Jaafar, Wan Fatimah wan Ahmad and Arpit Bansal.**

[9] **L. R. Rabiner and B.-H. Juang**, Fundamentals of speech recognition. Pearson Education, 2005.

[10] **Paliwal and K. K**., "Decorrelated and liftered filter-bank energies for robust speech recognition," EUROSPEECH '99, pp. 85–88, 1999

[11] What are Recurrent Neural Networks? | IBM

[12] **Ian Goodfellow, Yoshua Bengio, Aaron Courville,** "Deep Learning", 1st edition, Chapter 10, The MIT Press, 2016

[13] **Ralf C. Staudemeyer, Eric Rothstein Morris**. "– Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks", September 23, 2019

[14] **Paul J. Werbos, "**Backpropagation through time: What it does and how to do it". Proc. of the IEEE, 78(10):1550–1560, 1990.

[15] Understanding LSTM Networks -- colah's blog

[16] **Weixin Luo, Wen Liu, Shenghua Gao, "** A Revisit of Sparse Coding Based Anomaly Detection in Stacked RNN Framework."

[17] **Ronald J. Williams and David Zipser**, "Gradient-based learning algorithms for recurrent networks and their computational complexity. In Backpropagation: Theory, Architectures and Applications", pages 1–45. L. Erlbaum Associates Inc., jan 1995

[18] **Felix A. Gers, J¨urgen Schmidhuber, and Fred Cummins**, "Learning to Forget: Continual Prediction with LSTM". Neural Computation, 12(10):2451– 2471, oct 2000.

[19] **Yoshua Bengio, Patrice Simard, Paolo Frasconi, and Paolo Frasconi Yoshua Bengio, Patrice Simard, "**Learning long-term dependencies with gradient descent is difficult." IEEE trans. on Neural Networks / A publication of the IEEE Neural Networks Council, 5(2):157–66, jan 1994.

[20] **Sepp Hochreiter and J¨urgen Schmidhuber,** "LSTM can solve hard long time lag problems". Neural Information Processing Systems, pages 473–479, 1997

[21**] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and J¨urgen Schmidhuber**, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies." A Field Guide to Dynamical Recurrent Neural Networks, page 15, 2001.

[22] **Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever,** "An empirical exploration of Recurrent Network Architectures". In Proc. of the 32nd Int. Conf on Machine Learning, pp. 23422350, 2015, pages 2342—-2350, 2015

[23] **Felix A. Gers, Nicol N. Schraudolph, and J¨urgen Schmidhuber**. "Learning precise timing with LSTM recurrent networks". Journal of Machine Learning Research (JMLR), 3(1):115–143, 2002.

[24] **Yoshua Bengio, R´ejean Ducharme, Pascal Vincent, and Christian Janvin**, "A Neural Probabilistic Language Model." The Journal of Machine Learning Research, 3:1137–1155, 2003.

[25] **Alex Graves, Nicole Beringer, and J¨urgen Schmidhuber.** "A comparison between spiking and differentiable recurrent neural networks on spoken digit recognition." In Proc. of the 23rd IASTED Int. Conf. on Modelling, Identification, and Control, Grindelwald, 2003.

[26] **Alex Graves, Nicole Beringer, and J¨urgen Schmidhuber**. "Rapid retraining on speech data with LSTM recurrent networks." Technical Report IDSIA 09-05, IDSIA, 2005.

[27] **Herv´e A. Bourlard and Nelson Morgan**. “Connectionist Speech Recognition - a hybrid approach.” Kluwer Academic Publishers, Boston, MA, 1994

[28] **N Beringer, A Graves, F Schiel, and J Schmidhuber**. “Classifying unprompted speech by retraining LSTM Nets”. In W Duch, J Kacprzyk, E Oja, and S Zadrozny, editors, Artificial Neural Networks: Biological Inspirations (ICANN), volume 3696 LNCS, pages 575–581. Springer-Verlag Berlin Heidelberg, 2005

[29] **Alex Graves and J¨urgen Schmidhuber**, “Framewise phoneme classification with bidirectional LSTM networks.” In Proc. of the Int. Joint Conf. on Neural Networks, volume 18, pages 2047–2052, Oxford, UK, UK, jun 2005. Elsevier Science Ltd.

[30] **Alex Graves, Santiago Fern´andez, Faustino Gomez, and J¨urgen Schmidhuber**. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In Proc. of the 23rd Int. Conf. on Machine Learning (ICML’06), number January, pages 369–376, New York, New York, USA, 2006. ACM Press.

[31] **Emanuel Indermuehle, Volkmar Frinken, Andreas Fischer, and Horst Bunke**. Keyword spotting in online handwritten documents containing text and non-text using BLSTM neural networks. In Int. Conf. on Document Analysis and Recognition (ICDAR), pages 73–77. IEEE, 2011.

[32] **F. Chollet et al**., “Keras.” https://keras.io, 2015.

[33] **T Geiger, Zixing Zhang, Felix Weninger, Gerhard Rigoll, J¨urgen T Geiger, Zixing Zhang, Felix Weninger, Bj¨orn Schuller, and Gerhard Rigoll**. Robust speech recognition using long short-term memory recurrent neural networks for hybrid acoustic modelling. Proc. of the Ann. Conf. of International Speech Communication Association (INTERSPEECH 2014), (September):631–635, 2014.

[34] **Ilya Sutskever, Oriol Vinyals, and Quoc V Le.** Sequence to Sequence learning with neural networks. Advances in Neural Information Processing Systems (NIPS’14), pages 3104–3112, sep 2014

[35] **Jan Chorowski, Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio.** End-to-end continuous speech recognition using attention based recurrent NN: first results. Deep Learning and Representation Learning Workshop (NIPS 2014), pages 1–10, dec 2014.

[36] **William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals**. Listen, Attend and

Spell. arXiv preprint, pages 1–16, aug 2015.

[37] **Jan Koutnik, Klaus Greff, Faustino Gomez, and J¨urgen Schmidhuber.** A Clockwork RNN. In Proc. of the 31st Int. Conf. on Machine Learning (ICML 2014), volume 32, pages 1863–1871, 2014.

[38] **Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio**. "Neural machine translation by jointly learning to align and translate." In Proc. of the Int. Conf. on Learning Representations (ICLR 2015), volume 26, page 15, sep 2015.

[39] **Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio.** "Attention-based models for speech recognition". In C Cortes, N D Lawrence, D D Lee, M Sugiyama, and R Garnett, editors, Advances in Neural Information Processing Systems 28, pages 577–585. Curran Associates, Inc., jun 2015

[40] **N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov**, "Dropout: a simple way to prevent neural networks from overfitting," The journal of machine learning research, vol. 15, no. 1, pp. 1929–1958, 2014.

[41] **Y. Gal and Z. Ghahramani**, "A theoretically grounded application of dropout in recurrent neural networks," in Advances in neural information processing systems, pp. 1019–1027, 2016

[42] **N. Reimers and I. Gurevych**, "Optimal hyperparameters for deep lstmnetworks for sequence labeling tasks," arXiv preprint arXiv:1707.06799, 2017.

[43] **Prechelt, Lutz.** (2000). Early Stopping - But When?. 10.1007/3-540-49430-8_3.

[44] What is early stopping? (educative.io)

[45] **N. Reimers and I. Gurevych,** "Optimal hyperparameters for deep lstmnetworks for sequence labeling tasks," arXiv preprint arXiv:1707.06799, 2017

[46] **Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng**. "TensorFlow: Large-scale machine learning on heterogeneous systems, 2015." Software is available from tensorflow.org.

[47] **McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric**

Battenberg, and Oriol Nieto. “librosa: Audio and music signal analysis in python.” In Proceedings of the 14th python in science conference, pp. 18-25. 2015.

[48] Licenses and legal info — h5py 3.2.1 documentation