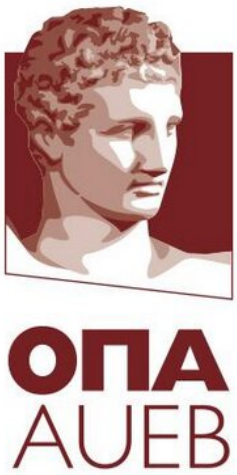
Natural Language Understanding for Dialogue Systems Using N-best Lists



Stratos Mansalis

Academic Supervisor: Prof. Ion Androutsopoulos

Capstone Project Supervisor: Dr. Themos Stafylakis

Master of Data Science
Athens University of Economics and Business

This dissertation is submitted for the degree of
Master of Science

Athens, November 2019

I would like to dedicate this thesis to my loving parents.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains appendices, bibliography, tables and equations and figures.

Stratos Mansalis
Athens, November 2019

Acknowledgements

I would like to express my sincere thanks to my supervisors Prof. Ion Androutsopoulos and Dr. Themos Stafylakis for their constant support and valuable guidance throughout the development of this thesis and especially for giving me, once again, the opportunity to explore an area of great interest. In addition, I would like to thank George Mastrapas for the fruitful meetings and the interesting discussions we had regarding this thesis.

Abstract

Dialogue systems are an important part of today's world Artificial Intelligence (AI) applications. Task-oriented dialogue systems have as main goal to help users to complete certain tasks more efficiently. Spoken language understanding (SLU), a key component of task-oriented dialogue systems, is the problem of extracting the meaning from speech utterances. It is typically addressed as a two-task procedure, where initially an Automatic Speech Recognition (ASR) model is employed to decode speech into text, followed by a Natural Language Understanding (NLU) model that takes as input the most likely hypothesis for the user's utterance in order to extract the meaning. Several challenges lead to errors in ASR that propagate to the NLU. Because misrecognition of a word may result in misunderstanding of the whole utterance, a technique in order these systems to be more robust is instead of using the most likely hypothesis (1-best) as input to the NLU module to use a list of most likely hypotheses (N-best lists). In this thesis, we examined if the SLU system performance for the problem of intent detection is improved by using N-best lists as input to NLU during training, as a form of data augmentation, compared to using only the 1-best hypothesis. We conducted experiments using a set of standard LSTM-based architectures and state-of-the-art transformers models using the recently introduced Fluent Speech Commands (FSC) dataset, where intents are formed in classes as combinations of three slots (action, object, and location).

Table of contents

List of figures xiii

List of tables xv

1 Introduction 1
- 1.1 Dialogue systems . . . 1
- 1.2 Spoken Language Understanding . . . 1
  - 1.2.1 Natural Language Understanding . . . 3
- 1.3 Outline . . . 4

2 Background 5
- 2.1 Natural Language Processing . . . 5
- 2.2 Neural Networks . . . 5
  - 2.2.1 Recurrent Neural Networks . . . 5
  - 2.2.2 RNN Extensions . . . 9
  - 2.2.3 LSTMs . . . 11
- 2.3 Transformers . . . 15

3 Experimental Setup 17
- 3.1 PyTorch . . . 17
  - 3.1.1 Transformers library . . . 17
- 3.2 Dataset . . . 17

4 Models Implemented and Results 21
- 4.1 LSTM Models . . . 21
- 4.2 Transformer Models . . . 23
- 4.3 Results . . . 24

5 Conclusions and Future Work 25

References 27

Appendix A LSTM Pytorch Code 31

List of figures

1.1 Conventional and End-to-End Spoken Language Understanding 2

2.1 Unrolled RNN 6
2.2 RNN input-output combinations 8
2.3 Backpropagation through time 9
2.4 RNN bidirectional 9
2.5 Deep (Bidirectional) RNNs 10
2.6 LSTM structure 11
2.7 LSTM: Cell state 12
2.8 LSTM: Forget gate 12
2.9 LSTM: Input gate 13
2.10 LSTM: Output gate 14
2.11 Encoder Transformer 15
2.12 BERT Masked Language Model 15
2.13 BERT: Next Sentence Prediction 16

3.1 Length of utterances 18

4.1 BiLSTM Model Architecture: 1-best 22
4.2 Tuning Learning Rate 22
4.3 BiLSTM Model Architecture: N-best 23

List of tables

1.1 An illustrative example of domain, intent, and slot recognition. 4

2.1 Comparison of the transformers models 16

3.1 Examples of wordings and their intents 18
3.2 Dataset Statistics 19

4.1 Hyperparameters 21
4.2 Transformers Hyper parameters 23
4.3 Test accuracy 24

Chapter 1

Introduction

1.1 Dialogue systems

Dialogue systems have been a prominent component in today's artificial intelligence applications. Nowadays, there is an increasing need of speech-driven dialogue systems that allow users to accomplish their desired tasks by interacting with virtual agents. This increasing need of robust speech recognition systems has attracted more and more the attention of dialogue system researchers both in research and industry. Dialogue systems are generally divided into two categories, according to different application purposes: task-oriented dialogue systems and non-task-oriented dialogue systems (also known as chatbots). **Task-oriented dialogue systems** are designed for a particular task. These systems work as personal assistants and aim to assist the user to complete certain tasks, for example to book a flight, to make a phone call or to find a restaurant. They support short conversations, from a single interaction to several interactions, and are based on a domain ontology, a knowledge structure representing the kinds of interactions the system needs to learn from these short conversations in order to complete the task. **Non-task-oriented dialogue systems** are considered to be open domain chat systems. These systems are designed for extended conversations that interact with humans to provide reasonable responses and entertainment. They are trying to mimic the unstructured conversational characteristics of human-human interaction on conversing for entertainment, rather than focused on a particular task.

1.2 Spoken Language Understanding

Spoken language understanding (SLU) is a key component of task-oriented dialogue systems. Formally, spoken language understanding is the problem of extracting the meaning from

a user's speech utterances. As Figure 1.1[1] shows SLU systems have two modules. The first module is the Automatic Speech Recognition (ASR) system which is the system of transcribing (decoding) the acoustic speech voice (signal) into text. It is considered to be a challenging task for real-world applications due to both inter-speaker variability (physiological differences and pronunciation differences of accents or dialects) and intra-speaker variability (various styles of speech). Moreover, other hardware and environmental factors such as channel distortion, or background noise pose more difficulties for achieving effective speech to text conversion. The second module is a Natural Language Understanding (NLU) system which extracts one or more semantic labels from a user's utterance. The main disadvantage of this approach is that each module of conventional SLU systems tends to be trained independently with training criteria that are specific to each module and may be different from the overall metric for the SLU system. ASR systems are typically evaluated based on word error rate (WER) but an ASR component with the lowest WER may not provide the correct best hypothesis for the user's utterance. For example, WER typically weights each word equally, despite the fact that not every word has the same impact. Likewise, the NLU module is trained on the output of the ASR, as a result errors in the ASR module will be propagated to NLU affecting its performance.

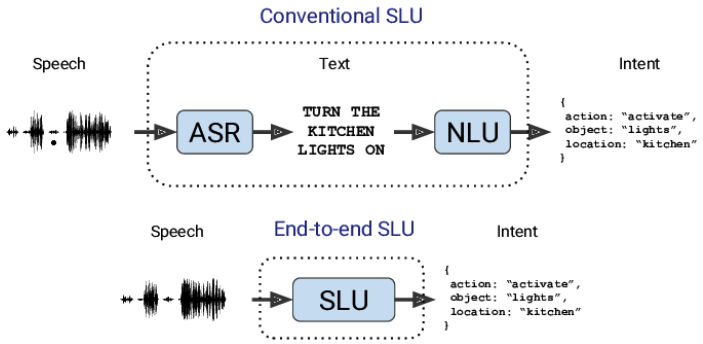


Fig. 1.1 Conventional and End-to-End Spoken Language Understanding

Most of the research efforts in SLU in the NLP community are based on conventional SLU processing, however, recent advances in technology and especially in Deep Learning have boosted the development of ASR-free end-to-end approaches [1–9] for SLU tasks showing promising results. End-to-end learning has been widely used in several areas, such as machine translation and image captioning. These methods as Figure 1.1 shows aim to learn SLU models from the acoustic signal without producing a text transcript, specifically a

[1] from Lugosch et al. [3].

single model maps speech audio directly to the corresponding task. In general, these methods require more training data in order to be efficient.

1.2.1 Natural Language Understanding

The goal of the Natural Language Understanding (NLU) module is to automatically identify the domain and intent of the user's utterance as expressed in natural language, and to extract associated arguments/slots. These tasks are structured hierarchically and there is inherent dependency between them. For example, a domain can be viewed as a group of intents which belong to the domain, and an intent consists of one or multiple slots/tags that define semantic keywords of this intent.

Hierarchically, the first task is domain classification. The goal of domain classification is to identify the broad domain the user's utterance is talking about, e.g., airlines, movies, home-service. In case of a single-domain system, this 1-of-n domain classification task is unnecessary. The second task is the intent detection, which is often treated as an utterance semantic classification task. The goal of intent detection is to identify the specific task or the goal the user is trying to accomplish, e.g. book a flight, find a movie, switch off the lights. The final task is to extract relevant information required for accomplishing the user's intent. The latter task is known as slot filling. Slot filling can be treated as a sequence labelling problem that tags the input word sequence $x = (x_1, x_2, \ldots, x_T)$ with the slot labelling sequence $y^s = (x_1^s, x_2^s, \ldots, x_T^s)$. The input word sequence and slot labelling sequence are of the same length, and thus there is an explicit alignment. This three-tasks based approach has the advantage of flexibility, since specific changes to a domain or intent category can be implemented without requiring changes to other domains or intent categories. Finally, single domain models are more focused since the intent detection only needs to consider a relatively small set of intents and slot tags to determine and fill, respectively.

Table 1.1 illustrates an example of natural language understanding. The domain of the utterance is airline-travel tickets and the intent detection is to show/find a flight. The word "Athens" is labelled as the departure (origin) city and the word "Munich" as the arrival (destination) city. Many approaches have been proposed either for intent detection or slot filling. Moreover, joint models [17–21] for intent detection and slot filling have also been proposed in literature. Such joint models simplify the NLU system, as only one model needs to be trained and fine-tuned for the two tasks.

Sentence	*Show*	*me*	*morning*	*flights*	*from*	*Athens*	*to*	*Munich*	*on*	*Monday*
Slots	O	O	Time	O	O	Origin	O	Dest.	O	Date
Intent	Show/Find Flights									
Domain	Air-Travel Tickets									

Table 1.1 An illustrative example of domain, intent, and slot recognition.

N-best lists

The performance of ASR is determined by how accurately the spoken words are recognized. As we have already mentioned, there are several challenges that lead to errors in the output of ASR that propagate to NLU. Considering the following utterance: "What's the next flight from Athens to Berlin", in this utterance if an important entity, e.g. "Athens" is misrecognized by ASR the whole meaning of the utterance is changed and this affects the whole outcome of the SLU system. The recognition accuracy is vital in any application where a speech interface is required. Typically, SLU systems use the most likely hypothesis that ASR outputs (1-best hypothesis). Because it is very important for these systems to be robust to errors since misrecognition of a word may result in misunderstanding the whole utterance, a technique in order to improve the SLU performance is to use as input to NLU a list of hypotheses (N-best lists) for the user's utterance. The N-best list contains N-ranked hypotheses ordered by associated confidence scores for the user's speech, where the top entry is the engine's best hypothesis.

1.3 Outline

The rest of the thesis is organized as follows:

- Chapter 2 describes the background.
- Chapter 3 presents PyTorch and the dataset that used in this thesis.
- In Chapter 4, we discuss the models we implemented and their results.
- Chapter 5 concludes and proposes future work

Chapter 2

Background

2.1 Natural Language Processing

Natural Language Processing (NLP) is a field of artificial intelligence that aims to give machines the ability to 'understand' and generate natural language text. NLP aims to develop methods for solving practical problems involving language, such as information extraction, automatic speech recognition, machine translation, sentiment analysis, question answering, and summarization. Text data can be seen as either a sequence of characters or a sequence of words. It is very common to see text as a sequence of words for most problems.

2.2 Neural Networks

Deep learning is a class of machine learning techniques, where information is processed in hierarchical layers to understand representations and features from data in increasing levels of complexity. In practice, all deep learning algorithms are neural networks, which share some common basic properties. They all consist of interconnected neurons that are organized in layers. Where they differ is network architecture (or the way neurons are organized in the network), and sometimes in the way they are trained. Different classes of neural networks exist, the most common include multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), reccurent neural networks (RNNs).

2.2.1 Recurrent Neural Networks

Recurrent Neural Networks, also known as RNNs, are a kind of neural network that specialize in processing sequences. The word "sequences" imply that the elements of the sequence are related to each other and their order matters. In contrast with the traditional feed-forward

neural networks that take in a fixed amount of input data all at the same time and produce a fixed size output independently from the previous inputs, RNNs do not take all the input data at once. RNNs take it one at a time in a sequence and have a form of short-term "memory" which captures information about what has been calculated so far. As a result, the composite input at time t has some historical information from the inputs at time $T < t$. Furthermore, RNNs are networks with loops in them allowing information to be passed from one time-step of the network to the next, that's the reason why called recurrent because they perform the same task (function) for every element of a sequence recurrently, with the output depending on the previous computations. The ability to remember information using hidden states and connect it to the current task is what makes RNNs popular.

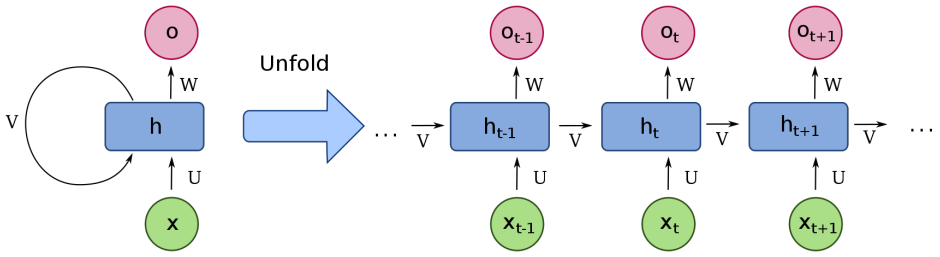


Fig. 2.1 RNN

The above Figure 2.1 shows a RNN being unrolled (or unfolded) into a full network. By unrolling we simply mean that we write out the network for the complete sequence. As the above figure shows, standard RNN is a network of neuron-like nodes organized into successive "layers", where each successive layer (input) is called a time-step. For example, if the sequence we care about is a sentence of 10 words, the network would be unrolled into a 10-layer neural network, one layer for each word. A recurrent neural network can be thought of as a multiple copies of the same network, each passing a message about the previous steps to a successor. RNNs share the same sets of parameters (U, V, W above) across all time-steps. That's the reason why a recurrent neural network can be thought of as a multiple copies of the same network, each passing a message about the previous steps to a successor. Specifically in RNNs there are the following weight-matrices:

- W maps the newly computed state h_t to the output o_t
- V transforms the previous state h_{t-1} to the current state h_t
- U transforms the input x_i to the state h

U,V and W apply linear transformation over their respective inputs.

RNN formula

The formula for the current state at time t of a RNN can be written as (recurrence relation):

$$h_t = f(h_{t-1}, x_t) \tag{2.1}$$

From the above equation it is obvious that in a RNN, each state h_i is dependent on all previous computations via this recurrence relation.

- x_t: is the input at time-step t. For example, x_1 could be a one-hot vector or a word embedding corresponding to the first word of a sentence.
- h_t: is the hidden state at time-step t, it's the "memory" of the network, h_t is calculated based on the previous hidden state h_{t-1} and the input x_t at the current step. The function f is a non-linearity such as ReLU, tahn, sigmoid and so on.

The formula for applying activation function at each time-step is: (where g is the activation function):

$$h_t = g(V\dot{h}_{t-1} + U\dot{x}_t + b_h) \tag{2.2}$$

and the output:

$$o_t = h_t \times W \tag{2.3}$$

RNN output

The output at each time-step t is calculated solely based on the memory at time-step t. Figure 2.1 has outputs at each time step, but depending on the task this may not be necessary. For example, for classification tasks we 'll need only one final output after passing all the input for the probabilities for each class. As this final output has already undergone calculations through all the previous cells, the context of all the previous inputs has been captured. This means that the final result is indeed dependent on all the previous computations and inputs. In another case, in text generation based on the previous character/word, we will need an output at every single time-step. Figure 2.2[1] shows some of the possible ways RNNs can be designed.

- **one-to-one:** This is non-sequential processing, such as feed-forward neural networks and convolution neural networks. There isn't much difference between a feed-forward network and applying a RNN to a single time-step.

[1] http://karpathy.github.io/2015/05/21/rnn-effectiveness/

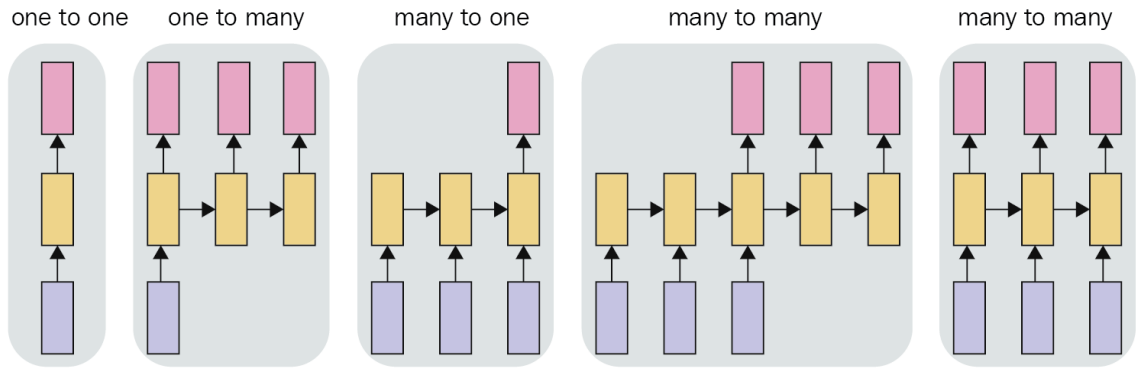


Fig. 2.2 Some of the possible ways of the input-output combinations

- **one-to-many:** This processing generates a sequence based on a single input. For example caption generation from an image.
- **many-to-one:** This processing outputs a single result based on a sequence, for example, sentiment analysis of a review as positive or negative.
- **many-to-many (indirect):** A sequence is encoded into a state vector, after which this state vector is decoded into a new sequence, for example, language translation.
- **many-to-many (direct):** This outputs a result for each input step, for example, frame phoneme labelling in speech recognition.

Training Recurrent Neural Networks

Training a RNN is similar to training a traditional Neural Network but with a little change since the parameters are shared by all time steps in the network. The gradient at each output depends not only on the calculations of the current time-step, but also the previous time steps. Recurrent networks rely on an extension of back-propagation called back-propagation through time, or BPTT.

As Figure 2.3[2] below shows, back-propagation through time is the typical algorithm we use to train recurrent networks. The main difference between regular back-propagation and back-propagation through time is that the recurrent network is unfolded through time for a certain number of time steps. Once the unfolding is complete, we end up with a model that is quite similar to a regular multilayer feed-forward network.

[2]https://www.sciencedirect.com/science/article/pii/S0959438818302009

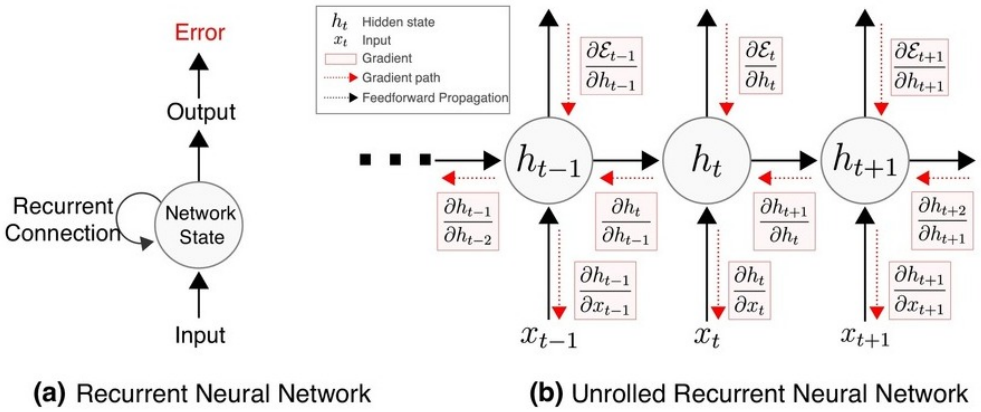


Fig. 2.3 Recurrent neural networks and BPTT. (Backpropagation through time)

2.2.2 RNN Extensions

More advanced types of the vanilla RNNs in order to solve shortcomings have been proposed in literature. As Figure 2.4[3] below shows Bidirectional RNNs are based on the idea that the output at time t may not only depend on the previous elements in the sequence, but also future elements. For example, to predict in a language model the next word you may want to look at both the left and the right context.

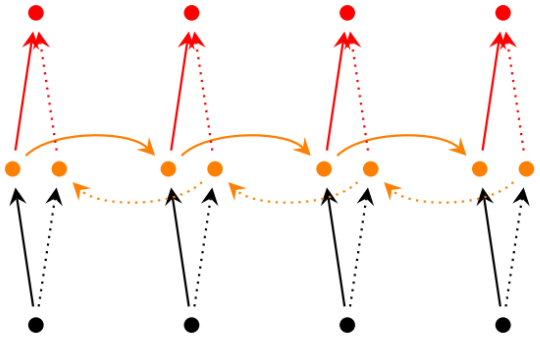


Fig. 2.4 Bidirectional

[3] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

Deep (Bidirectional) RNNs as Figure 2.5[4] shows are similar to Bidirectional RNNs, but we now have multiple layers per time step. In practice this gives us a higher learning capacity.

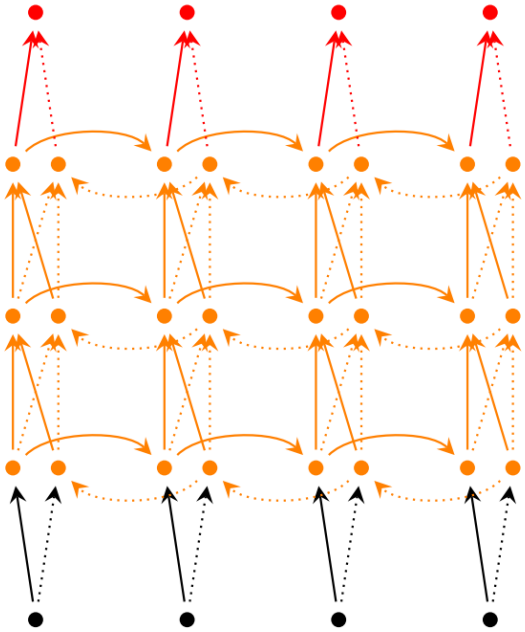


Fig. 2.5 Deep (Bidirectional) RNNs.

Limitations of RNNs

Recurrent neural networks are old. The main reason why standard RNNs are rarely used is due to the vanishing and exploding gradient problem. During training the gradients are being propagated back in time to the initial layer. Because the information flowing though neural nets passes many stages of multiplication, derivatives are susceptible to **vanishing** or **exploding**. Specifically, if the gradients have small values (< 1) as they approach the earlier layers, they shrink exponentially until they vanish and make it impossible for the model to learn. This is the vanishing gradient problem. Similarly, if the gradients have large values (> 1) they get larger and eventually blow up and crash the model. This is the exploding gradient problem (any quantity multiplied frequently by an amount slightly greater than one can become immeasurably large).

[4] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

2.2.3 LSTMs

Long Short-Term Memory (LSTMs), a variant of RNN, were introduced by Hochreiter and Schmidhuber [10] in 1997 to address the problem of vanishing gradients. LSTMs are capable of learning long-term dependencies and remembering information for long periods of time. In standard RNNs at each time-step t, the model takes the input x_t, the previous hidden state h_{t-1} and using a non-linear function obtains the output o_t and a new hidden state h_t. In LSTMs as figure 2.6 shows[5] there are four layers, interacting in order to remove or add information to the cell state. Specifically, in LSTMs at each time-step, the LSTM cell takes in the following three different pieces of information: (i) the current input x_t, (ii) the short term memory h_t, similar to hidden states in RNNs and (iii) the long term memory, known as cell state c_t.

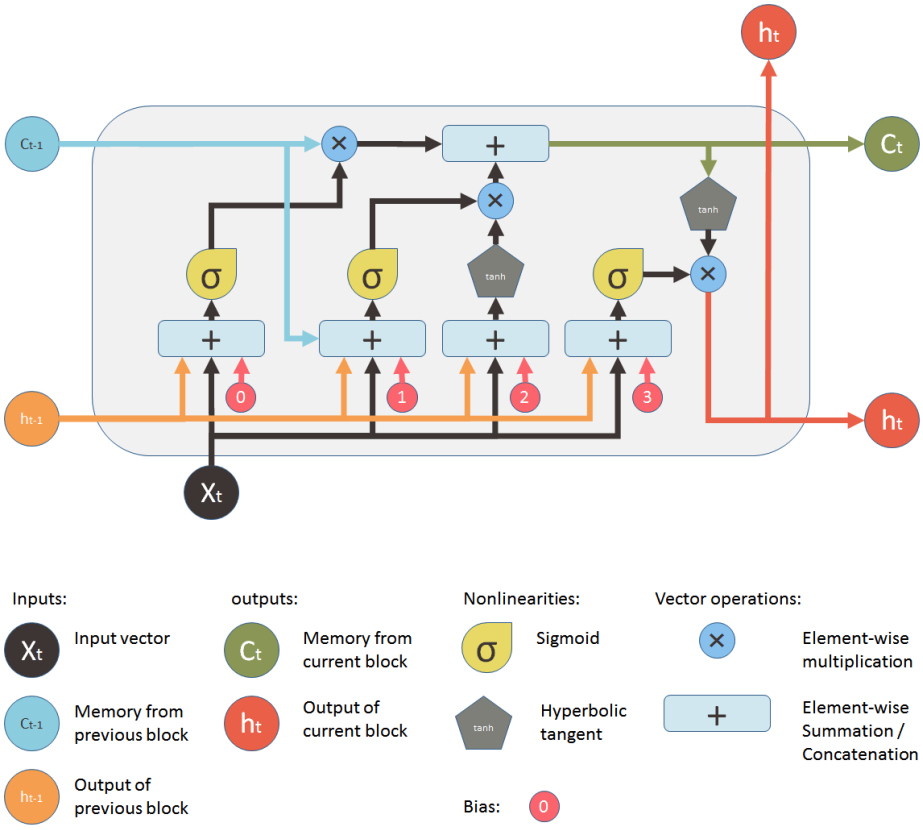


Fig. 2.6 The structure of the Long Short-Term Memory (LSTM) neural network.

[5]LSTM figures from https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714 and https://blog.floydhub.com/long-short-term-memory-from-zero-to-hero-with-pytorch/

The key idea in LSTMs as Figure 2.7 shows is the cell state, the horizontal line running through the top of the diagram which works as a conveyor and through these three gates has the ability to remove or add information to the cell state.

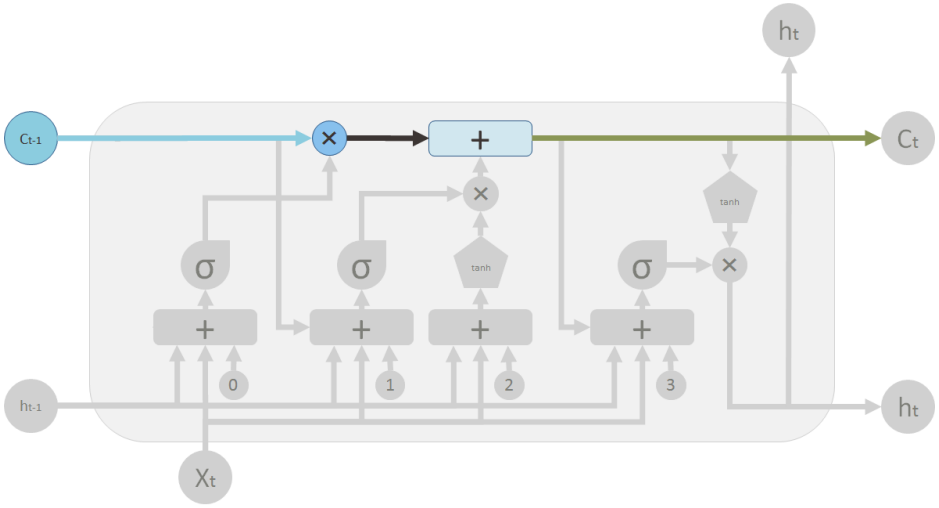


Fig. 2.7 LSTM: Cell state

The **forget gate** as Figure 2.8 shows is a selective filter layer. It decides which information from the cell state should be kept or discarded by multiplying the incoming long-term memory c_{t-1} (cell state) with a forget vector. The forget vector is generated by a simple one layer neural network which takes as input the current input x_t, the incoming short-term memory h_{t-1} as well as a bias vector b_0 and using a sigmoid function as activation outputs values between 0 (discard) and 1 (keep) for each element in the cell state.

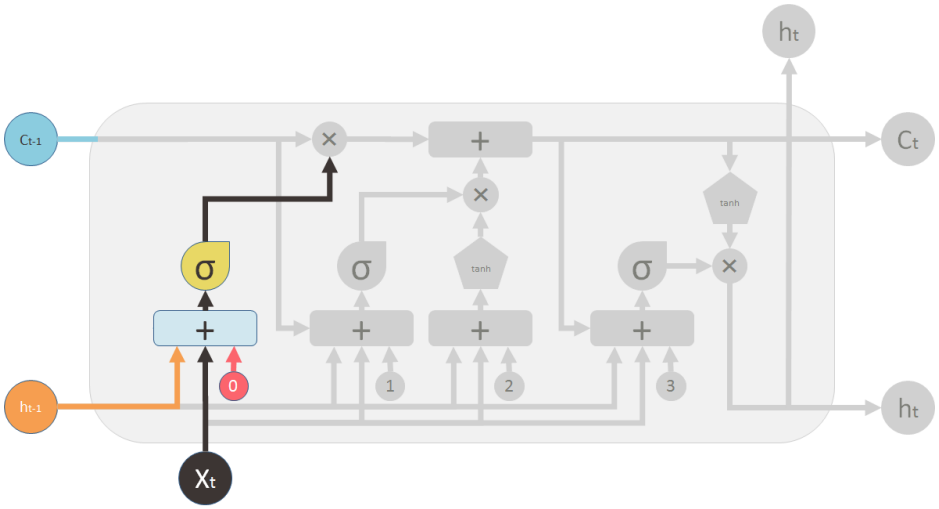


Fig. 2.8 Forget Gate

Equation 2.4 shows how the forget gate is computed:

$$f = \sigma(W_{forget}(h_{t-1}, x_t) + b_{forget}) \tag{2.4}$$

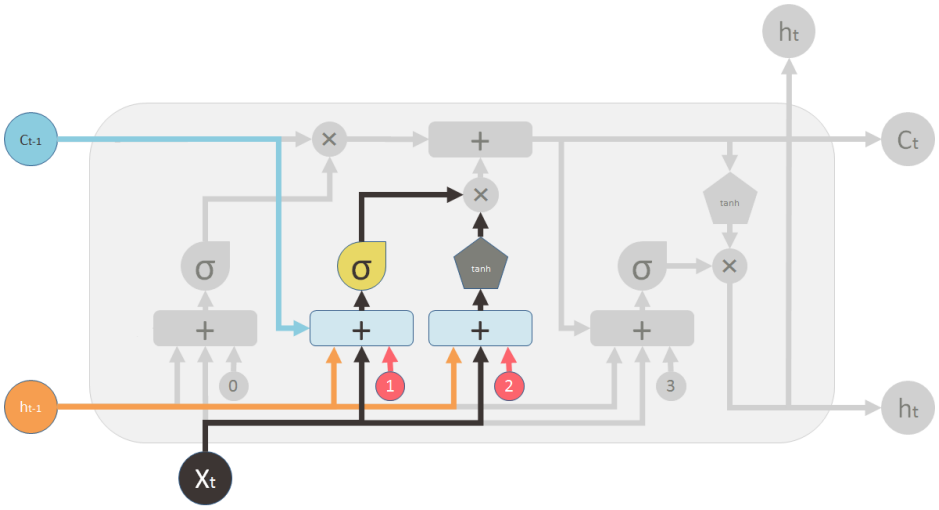


Fig. 2.9 Input Gate

The **input gate** decides what new information will be stored in the long-term memory. It only works with the information from the current input x_t and the short-term memory h_{t-1} from the previous time step. Therefore, it has to filter out the information from these variables. This is achieved using 2 layers.

$$i_1 = \sigma(Wi_1(h_{t-1}, x_t) + b_{i_1}) \tag{2.5}$$

$$i_2 = tahn(Wi_2(h_{t-1}, x_t) + b_{i_1}) \tag{2.6}$$

The first layer as Figure 2.9 above shows can be seen as the filter which selects what information can pass through it and what information to be discarded. To create this layer, we pass the short-term memory and current input into a sigmoid function (Eq. 2.5). The sigmoid function will transform the values to be between 0 and 1, with 0 indicating that part of the information is unimportant, whereas 1 indicates that the information will be used. This helps to decide the values to be kept and used, and also the values to be discarded. The second layer takes the short term memory and current input and passes it through an activation function, usually the *tanh* function (Eq. 2.6), to regulate the network. The outputs from these 2 layers are then multiplied, and the final outcome represents the information to be kept in the long-term memory and used as the output. $i_{input} = i_1 \times i_2$

The outputs from the Input gate and the Forget gate will undergo a point-wise addition (Eq. 2.7) to give a new version of the long-term memory, which will be passed to the next cell.

$$c_t = f_t * c_{t-1} + i_t \tag{2.7}$$

Output gate The output gate will take the current input, the previous short-term memory, and the newly computed long-term memory to produce the new short-term memory/hidden state which will be passed on to the cell in the next time step.

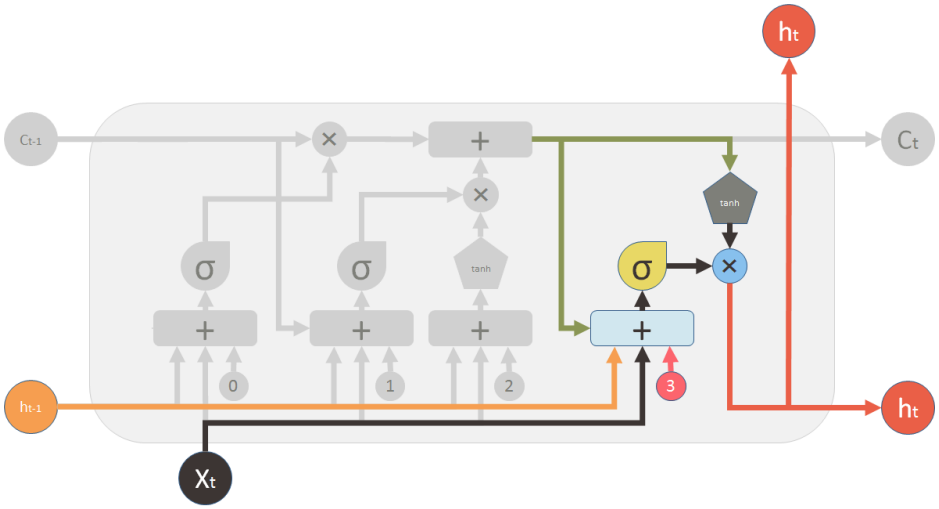


Fig. 2.10 Output Gate

The previous short-term memory and current input will be passed into a sigmoid (Eq. 2.8) function for 3rd time to create another filter. Then, the new long-term memory pass through an activation tahn function (Eq. 2.9)

$$O_1 = \sigma(W_{O_1}(h_{t-1}, x_t) + b_{O_1}) \tag{2.8}$$

$$O_2 = tahn(W_{O_2}(h_{t-1}, x_t) + b_{O_2}) \tag{2.9}$$

The output of these 2 processes will be multiplied (Eq. 2.10) to produce the new short-term memory. The output of each time step can be obtained from the short-term memory, also known as the hidden state.

$$h_t, O_t = O_1 \times O_2 \tag{2.10}$$

2.3 Transformers

Although LSTMs based models have produced state-of-the-art results in many NLP tasks, recent advancements in attention-based models and Transformers have produced even better results. The Transformer in NLP is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease and was proposed in [22].

BERT [11] introduced by Google, stands for Bidirectional Encoder Representations from Transformers. It is based on the transformer architecture. The general transformer uses an encoder and a decoder network, however, as BERT is a pre-training model, it only uses the encoder (Figure 2.11) to learn a latent representation of the input text. In particular, BERT has been pre-trained in two tasks the masked Language Modelling and Next Sentence Prediction using Wikipedia and Books Corpus. Language Modelling (Figure 2.12[6]) is the task of predicting the next word given a sequence of words. In masked language modelling instead of predicting every next token, a percentage of input tokens is masked at random and only those masked tokens are predicted. Next sentence prediction task (Figure 2.13) is a binary classification task in which, given a pair of sentences, it is predicted if the second sentence is the actual next sentence of the first sentence.

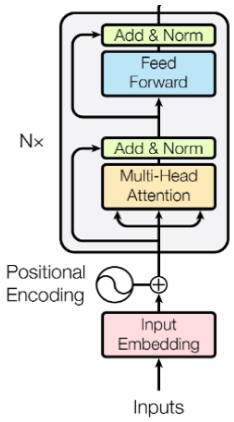


Fig. 2.11 Encoder Transformer

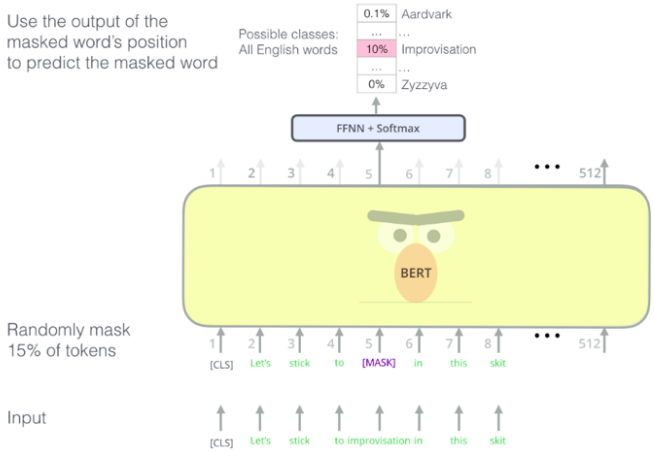


Fig. 2.12 BERT: Masked Language Model

RoBERTa [12] (Robustly optimized BERT) introduced by Facebook, is a retraining of BERT with improved training methodology with more data. To improve the training procedure, RoBERTa removes the Next Sentence Prediction (NSP) task from BERT's pre-training and introduces dynamic masking so that the masked token changes during the training epochs.

[6] http://jalammar.github.io/illustrated-bert/

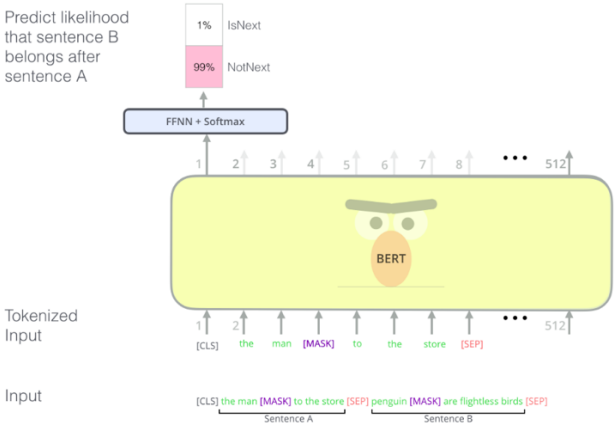


Fig. 2.13 BERT: Next Sentence Prediction

XLNet [14] is a large bidirectional Transformer that uses improved training methodology, more data and more computational power to achieve better than BERT prediction metrics on 20 language tasks. To improve the training, XLNet introduces permutation language modelling, where all tokens are predicted but in random order. **DistilBERT** [13] learns a distilled (approximate) version of BERT without token-type embeddings. DistilBERT uses a technique called distillation, which approximates the Google's BERT, i.e. the large neural network by a smaller one. The idea is that once a large neural network has been trained, its full output distributions can be approximated using a smaller network. This is in some sense similar to posterior approximation. The key optimization functions used is the Kulback Leiber divergence. Table 2.1 summarizes the basic characteristics of the transformer models we used in this thesis.

	BERT [11]	**RoBERTa** [12]	**DistilBERT** [13]	**XLNet** [14]
Size Parameters (in millions)	Base: 110 Large: 340	Base: 110 Large: 340	66	Base: 110 Large: 340
Data	16 GB (Book corpus + wikipedia)	160 GB (BERT plus 144 GB)	16 GB BERT 3.3 billion words	Base: 16 GB BERT
Method	Bidirectional Transformer with NSP and MLM	BERT without NSP, different masking	BERT distillation	Bidirectional Transformer permutation based modeling

Table 2.1 Comparison of the transformers models

Chapter 3

Experimental Setup

3.1 PyTorch

PyTorch [15] is a Python based scientific computing package, but with the added power of GPUs. Moreover, PyTorch is a deep learning framework that provides maximum flexibility and speed during implementing and building deep neural network architectures. A model can be defined in PyTorch in two steps by sub-classing the *torch.nn.Module* class specifying the parameters of the model initially and describing then how they are applied to the inputs. The biggest advantages of PyTorch are the dynamic computation graphing and its imperative programming style that performs computations as it goes through each line of the written code.

3.1.1 Transformers library

Transformers [16] provides state-of-the-art general-purpose architectures (BERT, GPT-2, RoBERTa, XLM, DistilBert, XLNet) for Natural Language Understanding (NLU) and Natural Language Generation (NLG) with over 32+ pretrained models in 100+ languages and deep interoperability between PyTorch.

3.2 Dataset

The Fluent Speech Commands dataset was used for this thesis. It contains utterances from 97 speakers. It is recorded as 16 KHz single-channel .wav files each containing a single utterance used for controlling smart-home appliances or virtual assistant, for example, “Play music”. Each audio is labeled with three slots (action, object, and location) that take on one of multiple values: for instance, the “location” slot can take on the values “none”, “kitchen”,

"bedroom", or "washroom". We refer to the combination of slot values as the intent of the utterance. For each intent, there are multiple possible wordings: for example, the intent action: "activate", object: "lights", location: "none" can be expressed as "turn on the lights", "lights on", etc.

Utterance	**Action**	**Object**	**Location**
Turn up the bathroom temperture	increase	temperature	washroom
Make it cooler	decrease	heat	none
Switch the bedroom lights off	deactivate	lights	bedroom
Play music	activate	music	none
Decrease the heating in the kitchen	decrease	heat	kitchen
Language settings	change language	none	none

Table 3.1 Examples of wordings and their intents

Table 3.1 above shows several examples of wordings and their intents. The dataset contains 248 phrasing mapping to 31 unique intents. The utterances are randomly divided into train, valid, and test splits in such a way that no speaker appears in more than one split. For the purpose of this thesis Kaldi [23] a speech recognition framework was used in order to obtain the N-best lists using a pre-trained model that accompanies Kaldi.

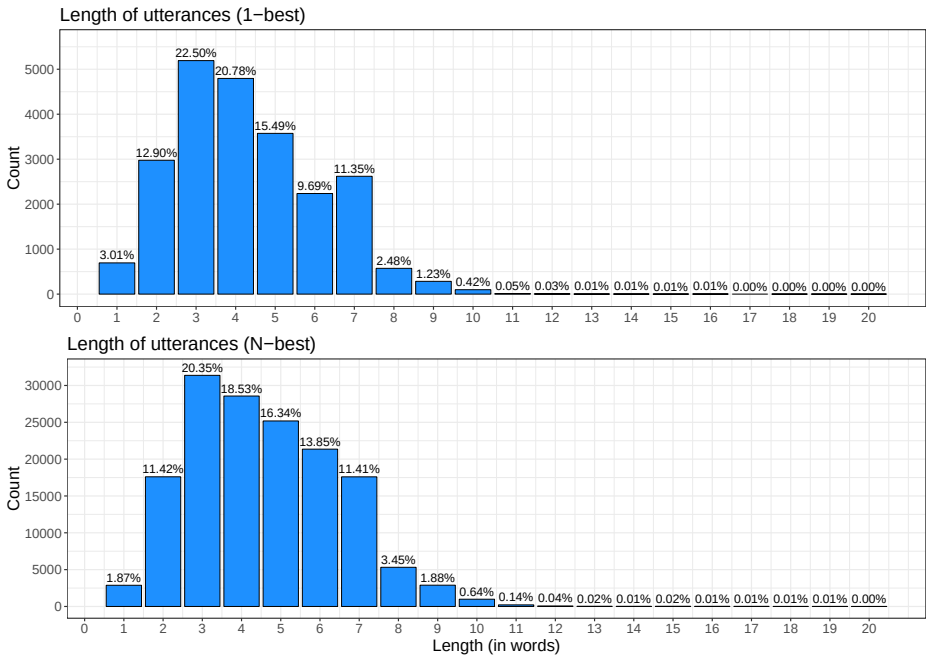


Fig. 3.1 Length for 1-best and N-best utterances

Each combination of the three tasks corresponds to one intent. The total number of intents due to the constraints between the tasks is 31 intents. For each original utterance of the dataset, we obtained the N-best transcribed utterances produced by the ASR in a data augmentation form. We treat all transcribed utterances as training examples of the classifier, with the same class label as of the original utterance. As Figure 3.1 shows the utterances for both 1-best and N-best are small sentences. Table 3.1 summarizes dataset statistics.

	1-best	**N-best**
Training (utterances)	23132	154266
Unique intents (classes)	31	31
Vocabulary Size (in words)	1843	7187
Unique tasks Actions	6	6
Unique tasks Objects	14	14
Unique tasks Locations	4	4

Table 3.2 Dataset Statistics

Chapter 4

Models Implemented and Results

4.1 LSTM Models

In this chapter we describe the models that we used in our experiments and the corresponding results. Initially, LSTM based models were developed, including a simple LSTM model, a stacked LSTM model and a bidirectional stacked LSTM model. The output vectors of all these LSTM models are sent through a series of dense layers using max pooling and average pooling initially and then concatenation to a softmax layer to build a sentence classifier. Figure 4.1 shows the training process for the 1-best hypothesis and Figure 4.3 shows for the N-best lists as a form of data augmentation, where all transcribed utterances were used as training examples with the same label as the original utterance. As Figure 4.2 shows the learning rate is set to 0.001, the hidden size to 512 and the embedding dimension to 200. (Tables 4.1 summarizes the chosen hyper-parameters). Also a dropout layer is applied before the softmax layer. The models are trained using cross-entropy loss and the Adam optimizer for 20 epochs on both 1-best and N-best training sets and evaluated on the validation split in order to choose the model with the lowest loss. Finally, the models are evaluated on the test (unseen) data as reported in Table 4.3. As the results show for all models N-best lists outperform 1-best hypothesis.

Hyperparameter	**LSTM**	**LSTM II**	**BiLSTM**
batch size	400	400	400
epochs	20	20	20
learning rate	0.001	0.001	0.001
dropout (linear layer)	0	0.3	0.3
embedding dim	200	200	200
hidden size	512	512	512

Table 4.1 Hyperparameters

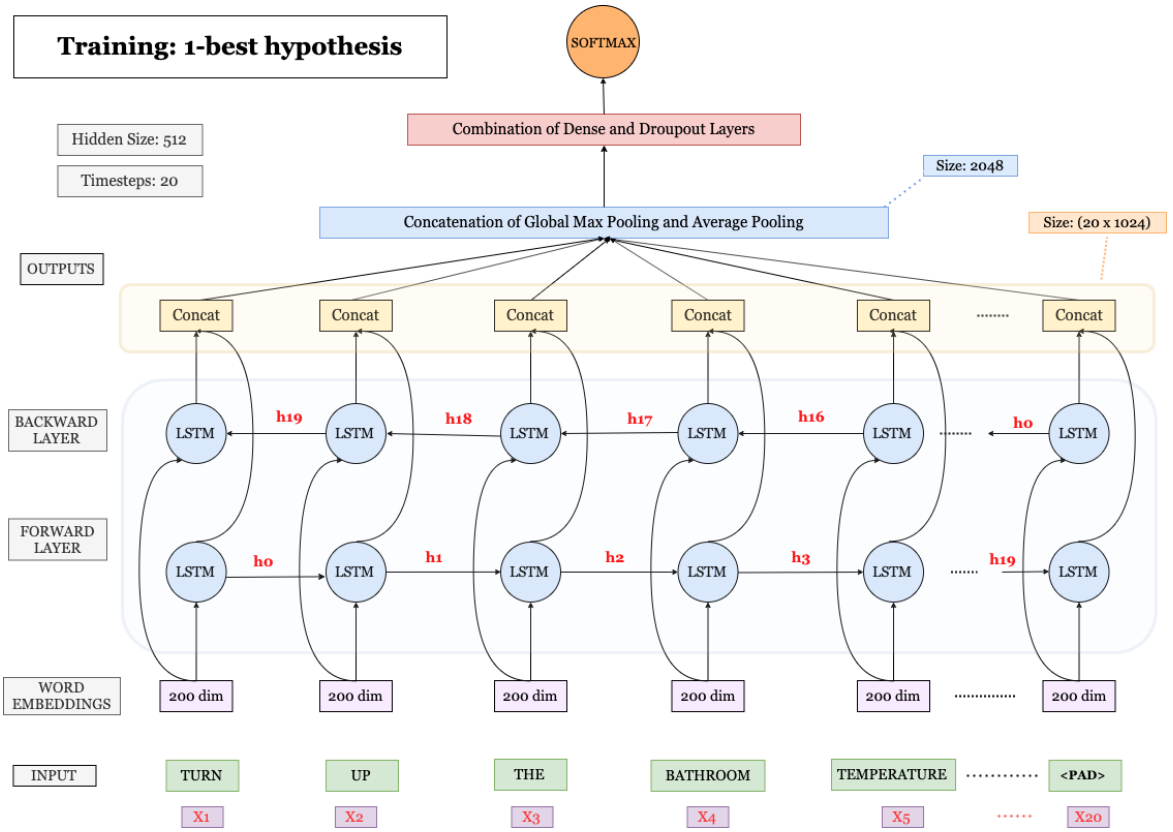


Fig. 4.1 BiLSTM Architecture: Training 1-best

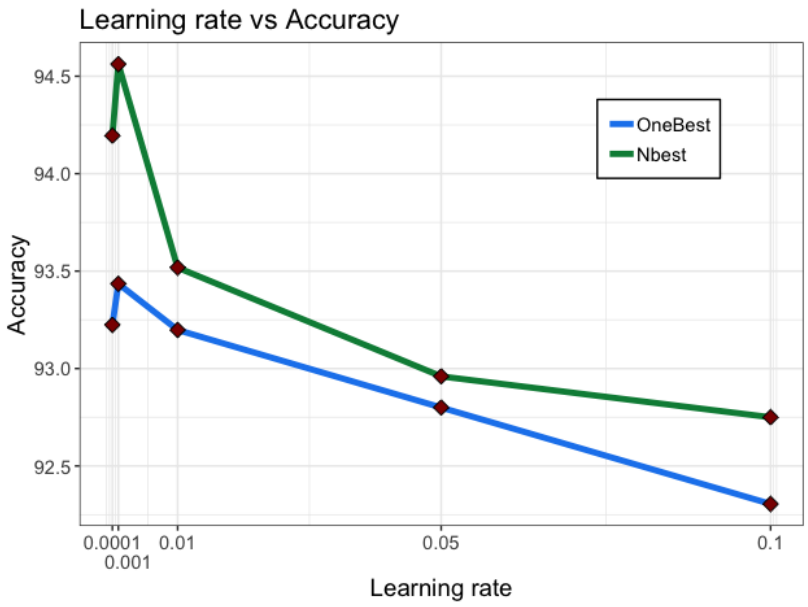


Fig. 4.2 Learning rate tuning

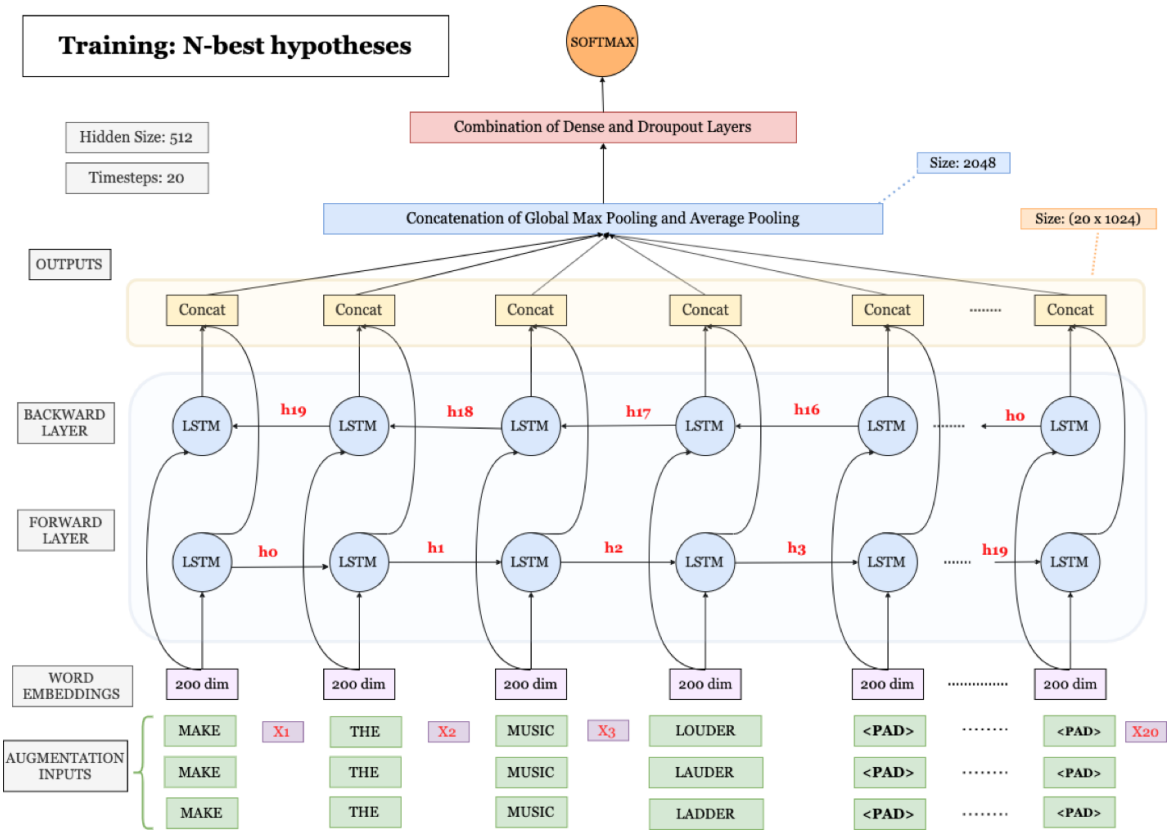


Fig. 4.3 BiLSTM Architecture: Training N-best

4.2 Transformer Models

Our experiments were conducted using the transformers library which provides state-of-the-art general-purpose architectures requiring only tuning the models. Table 4.2 shows the hyperparameters for our transformers models.

Hyperparameter	BERT	RoBERTa	XLNet	DistilBERT
epochs	2	2	4	4
batch size	32	32	32	32
learning rate	4e-5	4e-5	4e-5	4e-5
adam (epsilon)	1e-8	1e-8	1e-8	1e-8
weight decay	0	0	0	0
logging steps	50	50	50	50
save steps	1000	1000	1000	1000

Table 4.2 Transformers Hyper parameters

4.3 Results

The results of each model on the test set can be found in the following table 4.3. As the table shows all models that used the N-best lists for training outperform the models that used 1-best hypothesis. In addition, all Transformer models outperform LSTM based models. This was expected since Transformers are the state of the art models. Furthermore, BERT and RoBERTa seem to achieve slightly worse performance for both 1-best and N-best lists in contrast with DistiBERT and XLNet. It's worth pointing out that the N-best augmentation seems to have a more noticeable effect in the LSTM-based models, compared to the BERT-based models. This may be due to the fact that the BERT-based models are already pre-trained on huge datasets. XLNet seems to be an exception, in the sense that it is also Transformer-based and pre-trained on huge datasets, but still benefits from the N-best data augmentation. Finally, DistilBERT achieves the best results for both cases.

Model	**1-best**	**N-best**
LSTM	0.9362	0.9445
LSTM (2 layers)	0.9351	0.9447
BiLSTM	0.9343	0.9436
BERT (base)	0.9631	0.9638
RoBERTa (base)	0.9613	0.9666
DistilBERT	0.9836	0.9856
XLNet (base)	0.9762	0.9854

Table 4.3 Test accuracy

Chapter 5

Conclusions and Future Work

In this thesis, we discussed and compared methods to tackle the problem of the errors occurring in the output of ASR and affect the performance of the SLU system. Specifically, we examined instead of using the 1-best hypothesis as input to the NLU module to use a list of N-best hypotheses (N-best lists) for the user's utterance for the problem of intent detection. For each original utterance of the dataset, we obtained the N-best transcribed utterances in a data augmentation form produced by the ASR. We treat all these training examples of the classifier with the same label as the original utterance. We explored our problem by developing several LSTM based models. In particular, we developed a simple LSTM model, stacked and bidirectional LSTM models. Furthermore, we leveraged the transformers library for conducting experiments using state of the art models including BERT, RoBERTa, XLNet and DistilBERT. Our experiments conducted using the recently introduced Fluent Speech Commands (FSC) dataset which contains utterances for controlling smart-home appliances, where intents are formed in classes as combinations of three slots (action, object and location). We observed that all models that use N-best lists as input to NLU outperform the models that use 1-best hypothesis for the problem of intent detection.

In future work, we plan to experiment with more datasets as well as to experiment with different types of input data to the NLU module. Specifically, instead of using only the 1-best hypothesis or N-best hypotheses to use lattice or word confusion networks representations. Lattices are an efficient and effective method to encode ambiguity of ASR systems in NLU, for example to handle multiple speech recognition hypotheses. Previous work [25–28] has extended recurrent neural networks to model lattice inputs and achieved improvements in many tasks. Moreover, an idea for future work is using the same dataset to solve the problem as a multi-task classification problem where each of the three tasks correspond to one classifier. Finally, one could use more datasets for comparing 1-best and N-best lists for the problem of joint intent detection and slot filling.

References

[1] Y. Chen and R. Price and S. Bangalore (2018). Spoken Language Understanding without Speech Recognition. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2018)*, 6189–6193.

[2] Ghannay, Sahar and Caubrière, Antoine and Estève, Yannick and Camelin, Nathalie and Simonnet, Edwin and Laurent, Antoine and Morin, Emmanuel (2018). End-to-end named entity and semantic concept extraction from speech. *IEEE Spoken Language Technology Workshop*.

[3] Loren Lugosch and Mirco Ravanelli and Patrick Ignoto and Vikrant Singh Tomar and Yoshua Bengio (2019). Speech Model Pre-training for End-to-End Spoken Language Understanding. *arXiv (1904.03670)*.

[4] Haghani, P., et al. (2018). From audio to semantics: Approaches to end-to-end spoken language understanding. preprint arXiv:1809.09190.

[5] Y. Qian and R. Ubale and V. Ramanaryanan and P. Lange and D. Suendermann-Oeft and K. Evanini and E. Tsuprun (2017). Exploring ASR-free end-to-end modelling to improve spoken language understanding in a cloud-based dialogue system. IEEE Automatic Speech Recognition and Understanding Workshop (ASRU 2017), 569–576.

[6] Elisavet Palogiannidi and Ioannis Gkinis and George Mastrapas and Petr Mizera and Themos Stafylakis. End-to-end architectures for ASR-free spoken language understanding. arXiv: 1910.10599.

[7] D. Serdyuk and Y. Wang and C. Fuegen and A. Kumar and B. Liu and Y. Bengio (2018). Towards End-to-end Spoken Language Understanding. 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 5754–5758.

[8] Prashanth Gurunath Shivakumar and Naveen Kumar and Panayiotis Georgiou and Shrikanth Narayanan (2019). Incremental Online Spoken Language Understanding. preprint arXiv:1910.10287.

[9] Tomashenko, Natalia and Caubrière, Antoine and Estève, Yannick and Laurent, Antoine and Morin, Emmanuel (2019). Recent Advances in End-to-End Spoken Language Understanding. Statistical Language and Speech Processing, Springer, 44–55.

[10] Hochreiter, Sepp and Schmidhuber, Jürgen (1997). Long Short-Term Memory. *Neural Comput.*, 9.8:1735–1780.

[11] Jacob Devlin and Ming-Wei Chang and Kenton Lee and Kristina Toutanova (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding *CoRR.*, abs/1810.04805.

[12] Yinhan Liu and Myle Ott and Naman Goyal and Jingfei Du and Mandar Joshi and Danqi Chen and Omer Levy and Mike Lewis and Luke Zettlemoyer and Veselin Stoyanov (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach preprint arXiv:1907.11692.

[13] Victor Sanh and Lysandre Debut and Julien Chaumond and Thomas Wolf (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter preprint arXiv:1910.01108.

[14] Zhilin Yang and Zihang Dai and Yiming Yang and Jaime Carbonell and Ruslan Salakhutdinov and Quoc V. Le (2019). XLNet: Generalized Autoregressive Pretraining for Language Understanding preprint arXiv:1906.08237.

[15] Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam (2017). Automatic Differentiation in PyTorch NIPS Autodiff Workshop.

[16] Thomas Wolf and Lysandre Debut and Victor Sanh and Julien Chaumond and Clement Delangue and Anthony Moi and Pierric Cistac and Tim Rault and R'emi Louf and Morgan Funtowicz and Jamie Brew (2019). HuggingFace's Transformers: State-of-the-art Natural Language Processing preprint arXiv:1910.03771.

[17] Liu, Bing and Lane, Ian (2016). Attention-Based Recurrent Neural Network Models for Joint Intent Detection and Slot Filling Interspeech.

[18] Goo, Chih-Wen and Gao, Guang and Hsu, Yun-Kai and Huo, Chih-Li and Chen, Tsung-Chieh and Hsu, Keng-Wei and Chen, Yun-Nung (2018). Slot-Gated Modeling for Joint Slot Filling and Intent Prediction Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies., 753–757.

[19] Qian Chen and Zhu Zhuo and Wen Wang (2019). BERT for Joint Intent Classification and Slot Filling preprint arXiv:1902.10909

[20] D. Guo and G. Tur and W. Yih and G. Zweig (2014). Joint semantic utterance classification and slot filling with recursive neural networks *IEEE Spoken Language Technology Workshop (SLT)*, 554–559.

[21] P. Xu and R. Sarikaya (2013). Convolutional neural network based triangular CRF for joint intent detection and slot filling. IEEE Workshop on Automatic Speech Recognition and Understanding.

[22] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N. and Kaiser, Lukasz and Polosukhin, Illia (2017). Attention is All You Need. Proceedings of the 31st International Conference on Neural Information Processing Systems.

[23] Povey, Daniel and Ghoshal, Arnab and Boulianne, Gilles and Burget, Lukas and Glembek, Ondrej and Goel, Nagendra and Hannemann, Mirko and Motlicek, Petr and Qian, Yanmin and Schwarz, Petr and Silovsky, Jan and Stemmer, Georg and Vesely, Karel (2011). The Kaldi Speech Recognition Toolkit. IEEE 2011 Workshop on Automatic Speech Recognition and Understanding.

[25] Sperber, Matthias and Neubig, Graham and Pham, Ngoc-Quan and Waibel, Alex (2019). Self-Attentional Models for Lattice Inputs. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics., 1185–1197.

[25] Sperber, Matthias and Neubig, Graham and Pham, Ngoc-Quan and Waibel, Alex (2019). Self-Attentional Models for Lattice Inputs. Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics., 1185–1197.

[26] Sperber, Matthias and Neubig, Graham and Niehues, Jan and Waibel, Alex (2017). Neural Lattice-to-Sequence Models for Uncertain Inputs. Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing., 1380–1389.

[27] Jinsong Su and Zhixing Tan and Deyi Xiong and Rongrong Ji and Xiaodong Shi and Yang Liu (2016). Lattice-Based Recurrent Neural Network Encoders for Neural Machine Translation. preprint arXiv:1609.07730

[28] Buckman, Jacob and Neubig, Graham (2018). Neural Lattice Language Models. Transactions of the Association for Computational Linguistics

Appendix A

LSTM Pytorch Code

Preprocessing

```
# Dictionary that will map a word to the number of times it appeared in all the training sentences
words = Counter()
for i, sentence in enumerate(train_sentences):
    # The sentences will be stored as a list of words/tokens
    train_sentences[i] = []
    for word in nltk.word_tokenize(sentence):  # Tokenizing the words
        words.update([word.lower()])  # Converting all the words to lowercase
        train_sentences[i].append(word)
    if i%20000 == 0:
        print(str((i*100)/x) + "% done")
print("100% done")
```

Listing A.1 Vocabualry

```
words = sorted(words, key=words.get, reverse=True)
```

Listing A.2 Words

```
# Dictionaries to store the word to index mappings and vice versa
word2idx = {o:i for i,o in enumerate(words)}
idx2word = {i:o for i,o in enumerate(words)}
```

Listing A.3 Creating Dictionary

```
for i, sentence in enumerate(train_sentences):
    train_sentences[i] = [word2idx[word] if word in word2idx else 0 for word in sentence]

for i, sentence in enumerate(test_sentences):
    # For test sentences
    test_sentences[i] = [word2idx[word.lower()] if word.lower() in word2idx else 0 for word in nltk.word_tokenize(sentence)]
```

```
7
8  for i, sentence in enumerate(val_sentences):
9      # For validation sentences
10     val_sentences[i] = [word2idx[word.lower()] if word.lower() in word2idx else 0↘
           for word in nltk.word_tokenize(sentence)]
```

Listing A.4 Mapping dictionary

```
1  def pad_input(sentences, seq_len):
2      features = np.zeros((len(sentences), seq_len),dtype=int)
3      for ii, review in enumerate(sentences):
4          if len(review) != 0:
5              features[ii, -len(review):] = np.array(review)[:seq_len]
6      return features
```

Listing A.5 Padding

```
1  from torch.utils.data import TensorDataset, DataLoader
2  import torch.nn as nn
3
4  train_data = TensorDataset(torch.from_numpy(train_sentences), torch.from_numpy(↘
       train_labels))
5  val_data = TensorDataset(torch.from_numpy(val_sentences), torch.from_numpy(↘
       val_labels))
6  test_data = TensorDataset(torch.from_numpy(test_sentences), torch.from_numpy(↘
       test_labels))
7
8  batch_size = 400
9
10 train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
11 val_loader = DataLoader(val_data, shuffle=True, batch_size=batch_size)
12 test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
```

Listing A.6 Dataset creation

```
1  # torch.cuda.is_available() checks and returns a Boolean True if a GPU is ↘
       available, else it'll return False
2  is_cuda = torch.cuda.is_available()
3
4  # If we have a GPU available, we'll set our device to GPU. We'll use this device ↘
       variable later in our code.
5  if is_cuda:
6      device = torch.device("cuda")
7  else:
8      device = torch.device("cpu")
```

Listing A.7 Setting device

Bi-LSTM

```
1
2  class Multiclass(nn.Module):
3      def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim,
           n_layers, drop_prob=0.3):
4          super(Multiclass, self).__init__()
5          self.output_size = output_size
6          self.n_layers = n_layers
7          self.hidden_dim = hidden_dim
8
9          self.embedding = nn.Embedding(vocab_size, embedding_dim)
10         self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=
               drop_prob, batch_first=True, bidirectional=True,)
11         self.linear = nn.Linear(self.hidden_dim*4 , 64)
12         self.relu = nn.ReLU()
13         self.dropout = nn.Dropout(drop_prob)
14         self.out = nn.Linear(64, 31)
15
16     def forward(self, x, hidden):
17         batch_size = x.size(0)
18         x = x.long()
19         embeds = self.embedding(x)
20         lstm_out, hidden = self.lstm(embeds, hidden)
21         avg_pool = torch.mean(lstm_out, 1)
22         max_pool, _ = torch.max(lstm_out, 1)
23         conc = torch.cat(( avg_pool, max_pool), 1)
24         conc = self.relu(self.linear(conc))
25         conc = self.dropout(conc)
26         out = self.out(conc)
27         return out, hidden
28
29     def init_hidden(self, batch_size):
30         weight = next(self.parameters()).data
31         hidden = (weight.new(self.n_layers*2, batch_size, self.hidden_dim).zero_
               ().to(device),
32                   weight.new(self.n_layers*2, batch_size, self.hidden_dim).
                       zero_().to(device))
33         return hidden
```

Listing A.8 Bidirectional Model

```
1 vocab_size = len(word2idx) + 1
2 output_size = 31
3 embedding_dim = 200
4 hidden_dim = 512
5 n_layers = 2
```

Listing A.9 Parameters

```
1 model = Multiclass(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
2 model.to(device)
```

Listing A.10 Setting device

```
lr=0.0005
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

Listing A.11 Tuning

```
model.train()
for i in range(epochs):
    print('**NEW EPOCH**')
    h = model.init_hidden(batch_size)

    for inputs, labels in train_loader:
        if((inputs.shape[0],inputs.shape[1]) != (batch_size,seq_len)):
            print('Next')
            continue
        counter += 1
        #print('step:',counter)
        h = tuple([e.data for e in h])
        inputs, labels = inputs.to(device), labels.to(device)
        model.zero_grad()
        output, h = model(inputs, h)
        loss = criterion(output.squeeze(), labels.long())
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()
        train_loss.append(loss.item())

        if counter%100 == 0:
            val_h = model.init_hidden(batch_size)
            val_losses = []
            model.eval()
            for inp, lab in val_loader:
                if((inp.shape[0],inp.shape[1]) != (batch_size,seq_len)):
                    print('Next')
                    continue
                val_h = tuple([each.data for each in val_h])
                inp, lab = inp.to(device), lab.to(device)
                out, val_h = model(inp, val_h)
                val_loss = criterion(out.squeeze(), lab.long())
                val_losses.append(val_loss.item())
            model.train()
            print("Epoch: {}/{}...".format(i+1, epochs),
              "Step: {}...".format(counter),
              "Loss: {:.6f}...".format(loss.item()),
              "Val Loss: {:.6f}".format(np.mean(val_losses)))
```

Listing A.12 Training

```

test_losses = []
num_correct = 0
h = model.init_hidden(batch_size)

for inputs, labels in test_loader:
```

```
    if((inputs.shape[0],inputs.shape[1]) != (batch_size,seq_len)):
            print('Next')
            continue
    h = tuple([each.data for each in h])
    inputs, labels = inputs.to(device), labels.to(device)
    output, h = model(inputs, h)
    test_loss = criterion(output.squeeze(), labels.long())
    pred = torch.argmax(output, dim=1)
    test_losses.append(test_loss.item())
    correct_tensor = pred.eq(labels.long().view_as(pred))
    correct = np.squeeze(correct_tensor.cpu().numpy())
    num_correct += np.sum(correct)

print("Test loss: {:.3f}".format(np.mean(test_losses)))
test_acc = num_correct/len(test_loader.dataset)
print("Test accuracy: {:.3f}%".format(test_acc*100))
```

Listing A.13 Testing