University of Padua

Department of Mathematics

Master of Science in Data Science

MASTER THESIS

DESIGN COMPACT AND EFFICIENT RECURRENT NEURAL NETWORKS FOR NATURAL LANGUAGE PROCESSING TASKS

Supervisor: Prof. GABRIELE TOLOMEI
Department of Mathematics
University of Padua (Padua, Italy)

Co-supervisor: Prof. EKATERINA CHERNYAK
Faculty of Computer Science
National Research University Higher School of Economics
(Moscow, Russia)

Student: WALTER GENCHI
Student N° 1178603

"When you grow up you tend to get told
that the world is the way it is and your
life is just to live your life inside the world.

Try not to bash into the walls too much.
Try to have a nice family life, have fun,
save a little money.

That's a very limited life. Life can be much
broader once you discover one simple fact:
Everything around you that you call life
was made up by people
that were no smarter than you.

And you can change it, you can influence it...
Once you learn that, you'll never be the same again."

Steve Jobs

Contents

Introduction

The present work takes into account the compactness and efficiency of Recurrent Neural Networks (RNNs) for solving Natural Language Processing (NLP) tasks.

RNNs are a class of Artificial Neural Networks (ANNs). Compared to Feedforward Neural Networks (FNNs), RNN architecture is cyclic, i.e. the connection between nodes form cycles. This subtle difference has actually a huge impact on solving sequence-based problems, e.g. NLP tasks.

In particular, the first advantage of RNNs regards their ability to model *long-range time dependencies*, which is a very desirable property for natural language data, where word's meaning is highly dependent on its context. The second advantage of RNNs is that are *flexible* and accept as input many different data types and representation. This is again the case of natural language data, which can come in different sizes, e.g. words with different lengths, and types, e.g. sequences or trees.

Having said that, it should be noted though that RNNs' efficacy and execution time are dependent on the *size* of the network. In particular, these models are often too large in size for deployment on *mobile devices* with *memory* and *latency* constraints. In practice, this means that many simple NLP tasks can be performed only by calling an external cloud server infrastructure, i.e. without the possibility to do the computation offline.

It goes without saying mobile devices are becoming increasignly pervasive in our everyday life, thanks to enabling technologies such as *Augmented and Virtual Reality* (*AR/VR*) and *Internet of Things* (*IoT*). Moreover, an always increasing number of deep learning applications on such mobile devices are required to run in *real-time*, e.g. pedestrian detection in an autonomous vehicle.

For these reasons, industry practitioners and academic researchers are asked to design neural network models which can be stored on device (*space-efficiency*) and produce fast predictions (*time-efficiency*) with very little performance loss (*effectiveness*). This has started a new research field named *model compression* by Bucilua et al. (2006), which has proposed in the recent years solutions coming from many disciplines, including but not limited to machine learning, optimization, computer architecture, data compression, and hardware design.

The first objective of this work is to understand and categorize the state-of-the-art neural network compression techniques. Most of these compression methods have been applied Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs), while only a small attention has been given to compressing RNN architectures when solving NLP tasks.

Therefore, the second objective is to apply two different compression methods on a multi-layer RNN architecture, which has been trained to perform a simple NLP task, i.e. Part-of-Speech Tagging (PoS Tagging). After evaluating a naive application of these compression methods, we will propose and evaluate more tailored compression methods, based on the distribution of redundancy in the LSTM architecture.

The present work is divided in two wide content areas. The first two chapters of the thesis are a gentle introduction to NLP and Neural Networks, while the last two chapters are entirely devoted to RNNs.

Chapter 1 gives an overview on the characteristics and challenges of natural language data when solving NLP tasks. Moreover, an example of the typical pipeline of feature design and feature embedding in a concrete NLP task will be provided. A final discussion about the reasons of the success of Deep Learning in NLP will close the chapter.

Chapter 2 provides an introduction to Supervised Learning and Feedforward architecture, which will be useful later when considering PoS Tagging as a classification problem and RNN architectures as a specialized neural network architecture. After discussing the advantages of Multi Layer Perceptron (MLP) over Linear Models in terms of representational power, the last part of this chapter is devoted to optimization and regularization strategies, gradient computations and best practices when training a neural network.

Chapter 3 presents the RNN high-level abstraction, graphical representations (recursive and unrolled) and architectural variations (bidirectional RNNs and deep RNNs). Finally, three concrete recurrent architectures (Simple RNN, LSTM and GRU) will be described in details.

Chapter 4 firstly conducts a literature review on existing neural network compression methods and metrics and eventually identifies four general approaches, i.e. Matrix Factorization, Parameter Pruning, Parameter Sharing and Quantization. After that, the focus goes on an extensive description of the Universal Dependencies (UD) dataset, which has been used for training and evaluating a multi-layer LSTM model on a PoS Tagging task. The final part of this chapter presents the conclusions and take-home messages, based on the results obtained from the different compression strategies implemented.

Mathematical Notation

For the sake of simplicity, we keep the same mathematical notation from Goldberg (2017):

- Matrices and vectors are represented with capital and lower case bold letters respectively, e.g. $\boldsymbol{W}$ for matrices and $\boldsymbol{x}$ for vectors.
- Vectors are assumed to be row vectors, i.e. the d-dimensional vector $\boldsymbol{x}$ has size $1 \times d$.
- The i-th element of vector $\boldsymbol{x}$ is indicated as $\boldsymbol{x}_{[i]}$.
- The element in the i-th row and j-th column of matrix $\boldsymbol{W}$ is indicated as w_{ij}.
- The j-th element of a sequence $w_{1:n} = w_1, w_2, \ldots, w_n$ is indicated as w_j.
- Vector concatenation of vectors $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ is written as $[\boldsymbol{x_1}; \boldsymbol{x_2}]$.

1 What is Natural Language Processing?

Natural language processing (NLP) is a subfield of computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human (natural) languages, in particular how to program computers to process and analyze large amounts of natural language data.

The first consideration regards the definition of NLP. Jurafsky and Martin (2008) claim that NLP cannot be defined as an independent scientific field, but it is rather an "**interdisciplinary** field with many names corresponding to its many facets, names like speech and language processing, human language technology, natural language processing, computational linguistics, and speech recognition and synthesis". In other words, the current NLP methods and algorithms are nothing but the result of a 50-year research involving experts from linguistics, computer science, electrical engineering and psychology. Given the strong linguistic component in NLP, in section 1.1 we will present the main linguistic challenges in common NLP tasks. These have been combined in more complex tasks and finally deployed by the industry in successful tools that are already part of our everyday life.

Secondly, NLP has the goal to provide computers with the "knowledge of language"(Jurafsky and Martin, 2008). The main obstacle is the nature of the input, i.e. natural language data, which is by definition **ambiguous** and **variable**. In section 1.2, we will examine in details the characteristics and challenges of processing natural language data. Moreover, we will introduce two design approaches when selecting not only the intrinsic features (single word), but also the extrinsic features (word in context). Note that such linguistic features have to be first encoded into an embedding vector, which can be then given as input to a machine learning model (in our case a RNN). This process is called feature embedding and it is described in details by using a concrete example on the Part-of-Speech Tagging task.

Finally, in this chapter we will talk about the advent of deep learning models for solving NLP task. Such models are indeed able to spot patterns and regularities in the training data and, eventually, generalize this ability to a set of previously unseen data. In section 1.3, we will see not only the reasons behind the success of deep learning in the NLP domain, but also the advantages of specific deep learning models, i.e. RNNs, for some specific NLP tasks. This will give the key ingredients for the next chapter, which will be focused on RNN models.

The reference literature for this chapter is Jurafsky and Martin (2008) regarding the linguistic challenges in NLP and Goldberg (2017) regarding the mathematical notation and specific terminology for NLP concepts, which will be also used in the following chapters.

1.1 NLP in everyday life

Nowadays it is not rare to start a conversation with some *virtual assistants* directly on our smartphones: "Hey Siri, what is the weather like today in Moscow?" could be one of the first questions in the morning. In order to be able to interact with the user, these complex systems perform several NLP tasks and have to overcome many linguistic challenges.

The first task these systems perform is called *Speech Recognition*, i.e. they should be able to traslate spoken language into a correct sequence of words. In order to do that, they should have knowledge about how words's sounds are pronounced in terms of sequences of sounds. This type of knowledge is called *phonetics* in linguistics.

Once the answer has been elaborated by the system, the task of *Speech Synthesis* consists in organizing the sounds for the answer, i.e. artificially traslate text to human speech. This ability is studied by *phonology* in linguistics.

Once the conversation has gone further, the user may ask "What are the main events in the weekend?". This type of question assumes that the dialogue system keeps track of previous parts of the discourse, i.e. we are interested in the events in the city of Moscow. In NLP this task is called *Coreference Resolution.*

Another very likely possibility is that the user's question may be ambiguous, i.e. there are many meanings which can be attributed to the same word. "Where can I eat a pizza with friends?" has a very different meaning from "Where can I eat a pizza with salami?", even tough they differ in only one word. Here lies the ambiguity of human language, which is solved by *Disambiguation* tasks. One example of lexical disambiguation in NLP is *Part-of-Speech (PoS)*

Tagging, which tags each word with a particular part of speech (e.g. noun, verb, adjective etc.). [1] We will see more about PoS Tagging task in section 1.2.
These are only some examples of the numerous NLP tasks currently studied by research and industry practitioners. A common pattern in NLP is that simple (but not trivial) tasks, e.g. PoS Tagging, are used to solve more complex tasks, e.g. Coreference Resolution. In the next sections we will explore the whole pipeline of natural language data preprocessing, which is a preliminary step in all NLP tasks.

1.2 Data Preprocessing in NLP

According to Goldberg (2017), human (natural) language data is very challenging from a computational point of view, since it is highly **discrete**, **compositional** and **sparse**.

The property of discreetness regards the nature of input (symbols), which are not necessary related to the meaning of the word itself. As an example, consider image data: there exist some filters (i.e. mathematical operators) to convert an image from color to greyscale. The same *continuous* relationship cannot be found between words RED and GREY.

Having said that, the meaning of individual words and the relationship between different words can be found by looking at the composition (sequence) of words in a sentence, in a paragraph or even in the whole document. This arises the issue of *long-range dependencies* between words in a document, which led to the success of RNNs for some specific tasks, e.g. *Language Modeling* (see section 1.3).

All in all, human language is sparse: it is highly variable, i.e. there are potentially infinite ways to combine words. Moreover, it is hard to generalize, which is the final goal of all machine learning models.

In the next section we will see firstly the design approaches to select the relevant features according to the NLP task. Moreover, it will be presented an overview of the mathematical methods used to convert linguistic features into an embedding vector x.

Feature Design for NLP Problems

The process of feature selection (also called *feature engineering*) is one the most crucial and, at the same time, delicate in any machine learning task. The NLP

[1]Hereafter we assume that the tokenization task, e.g. separating words from punctuation, has been already performed

community has tried to design relevant features for textual data, by looking at the properties of natural language data.

Disclaimer Before starting the discussion on feature design in NLP, here is an important disclaimer to the reader.

Textual data can come in different forms and sizes: the typical hierarchical structure is document, paragraph, sentence, word and finally character. A typical option is to consider word-level models, i.e. the input is a sequence of n words $(w_1, \ldots, w_n)$. In section 3.4, we will see an alternative character-level model for RNNs, i.e. the input is a sequence of N characters $(c_1, \ldots, c_N)$.

Intrinsic vs. Extrinsic Features The main idea for feature selection in NLP is to divide the source of information into two categories: *intrinsic features* (word-dependent) and *extrinsic features* (context-dependent).

Intrinsic features analyse each word individually, i.e. they do not carry any information about neighbor words. In PoS Tagging task, for example, some useful intrinsic features are prefixes, suffixes and orthographic shape. For example, if an English word's suffix is ING, then it is very likely a present-continuous verb.

However, as we explained in section 1.1, human natural language is by definition ambiguous and variable. This means that typically the meaning of a word is not exclusively word-dependent, but its interpretation is also context-dependent. Having said that, extrinsic features typically focus on the close neighbors of w_j, e.g. a window of k words to each side, centered in position j.

This window-approach is useful to model sequence data because it takes into account the relative position of the words. However, in NLP community is it well-known that "sentences in natural language have structures beyond the linear order of their words. The structure follows an intricate set of rules that are not directly observable to us"(Goldberg, 2017).

In particular, fixed-sized windows are not able to model long-range dependencies, e.g. a word at the beginning of a sentence is linked with a word at the end of the sentence. This behavior is actually very common in human language. Therefore, some alternative models have been proposed: for example, *Bidirectional Recurrent Neural Networks (biRNNs)* presented in section 3.4 provide a window with flexible and adaptive size.

Directly Observable vs. Inferred Linguistic Properties Another convenient feature categorization in NLP is between *directly observable* and *inferred linguistic* properties.

The first set of properties is derived *directly* from the input, by following some predefined algorithmic procedures, which typically involve counting operations, e.g. word frequency in the *bag-of-words (BOW)* approach (see section 1.2), and boolean operations, e.g. word position in a sentence. Other notable examples are *lemmatization* and *stemming*, which are both governed by linguistically-predefined rules and thus may not produce an appropriate output in all contexts.

The second set of properties is based on well-known concepts in linguistics, e.g. *word classes* (e.g. PoS tags), *morphology*, *syntax* and *semantics*. In practice, the results obtained from some of these *specialized* NLP tasks, e.g. PoS tags, are first combined and then used as predictors for solving more sophisticated classification tasks.

Human vs. Automatic Feature Design The process for getting such *human*-designed features can be subject to error and is time-consuming, since it requires specialized expertise in the linguistics field.

This approach to feature design is actually in contrast with the dominant trend nowadays in NLP. In fact, deep learning practitioners claim that neural network are able to *automatically* represent these linguistic concepts in the intermediate layers of the network. This aspect will be further discussed in section 1.3.

Feature Embedding: from Linguistic Features to Embedding Vectors

In the previous section we have discussed the popular approaches when deciding which linguistic features actually carry useful information for solving our specific NLP task.

The next step is to represent each **of the k liguistic features** f_i (sparse sequence of discrete symbols over a vocabulary V) into an **embedding vector** $v(f_i)$[2] (dense and with fixed size d).

The resulting embedding vectors $v(f_i)$ can be later combined (either by concatenation, summation or mix of both) into an **input vector $\boldsymbol{x}$** using a **feature function** $\phi(\cdot)$. Now $\boldsymbol{x}$ is a well-defined mathematical object and can be later given as input to other machine learning models, e.g. neural networks.

[2]Note that $v(\cdot)$ is a function operator. This is the reason why we will not use bold notation for embedding vectors, but instead refer to them using the functional notation $v(f_i)$.

One-hot Encodings

The first and most elementary way to econde textual data is the Bag-of-words (BOW) representation, which is widely used for document classification.

The main idea of BOW is to characterize a document by the words contained in it, i.e. think about *indicator functions* (which words are present in the document). The BOW representation does not take into account the order of the words.

For example, if we take the document s_1 ="My name is John Smith" over a vocabulary V of 10,000 items, s_1 will be represented by a very sparse 10,000-dimensional vector in which 5 elements have non-zero entries (in s_1 all words are different[3]).

Following this representation, if word NAME has for example position 7332 in V, then it is mapped into a 10,000-dimensional *one-hot* vector, where the only non-zero entry is in position 7332.

One of the weaknesses with such encoding scheme is that features are treated as *fully indipendent* from one another, e.g. the dissimilarity between words ORANGE and LEMON is the same as the one between ORANGE and NAME. Moreover, from a computational point of view, *high-dimensional* and *sparse* vectors are not easily handled by neural networks.

Dense Econdings

A more convenient representation of any linguistic feature f_i is the embedding vector $v(f_i)$, which is *dense* (not sparse) and has fixed size $d \ll |V|$.

It can be easily shown that this representation is actually related to the one-hot econding seen before. Let $\boldsymbol{f_i}$ be the one-hot encoding of the linguistic feature f_i, i.e. a $|V|$-dimensional vector. If we stack (vertically) the d-dimensional embedding vectors $v(f_i)$, we obtain the embedding matrix $\boldsymbol{E}$ with dimension $|V| \times d$. Now we can establish the equality $v(f_i) = \boldsymbol{f_i E}$, i.e. we select the i-th row of $\boldsymbol{E}$.

The main benefit of such dense representation is the *generalization power.*

Take as an example the *Named-entity Recognition (NER)* task. Our goal is to correctly identify and classify the named entities, e.g. LONDON, ROME, JAMES CAMERON, FEDERICO FELLINI, etc. It may happen that in our training set we have observed many times the words ROME and FEDERICO FELLINI, but only few times LONDON.
A *good embedding vector* will be able to tell us that LONDON is a city, so it is like ROME, and that JAMES CAMERON is a film director, so it is like FEDERICO

[3]Note that MY and MY are considered as different words if capitalization has been removed or if the capitalization status of word w_j has been included in the linguistic features.

FELLINI. Therefore, our model will still be able to generalize, i.e. LONDON will be correctly labeled as a city in the sentence "JAMES CAMERON HAS MADE A GREAT MOVIE IN LONDON".

Learn Embedding Vectors

In practice, we do not *learn embedding vectors* $v(\cdot)$ for every NLP task. Instead, we use *pre-trained embeddings*, which have been already trained on a huge quantities of unannotated text.

The motivation behind such an approach is that *words are similar if they appear in similar context*, i.e. according to the context, similar words will have similar embedding vectors. This assumption is called the *distributional hypothesis* in the NLP community.

Having said that, it should be noted that for some NLP tasks the pre-trained embeddings are not left fixed during the network training process, but instead they are further-tuned.

The most popular word embedding algorithms available are WORD2VEC and GLOVE.

Example: Part-of-Speech Tagging

Here is an example of data preprocessing required for the Part-of-Speech Tagging task, describing the whole *pipeline* from the feature design stage to the feature embedding stage. The example is taken from Goldberg (2017).

Just to refresh memory, PoS Tagging is a lexical Disambiguation task in NLP. The goal is to tag each word with a particular part of speech (e.g. noun, verb, adjective etc.). Following what has been done by Goldberg (2017), the tagset comes from the *Universal Treebank Project* and contains 17 tags.

Feature Design

We assume that when the classifier has to predict the tag for word w_j, it has access only to a small neighbor, e.g. w_{j-2},w_{j-1}, w_{j+1}, w_{j+2}, and not to the whole sentence $w_1, \ldots, w_n$.

For example, immagine we want to perform PoS Tagging task on the sentence "TONIGHT I WILL HAVE A PIZZA WITH MY FRIENDS".

For $j = 1, \ldots, n$, we take into account the following **INTRINSIC FEATURES** for word w_j:

- the **word** itself w_j

- 2 and 3-letter **prefix** of w_j, i.e. $pref(w_j, 2)$ and $pref(w_j, 3)$ respectively
- 2 and 3-letter **suffix** of w_j, i.e. $suf(w_j, 2)$ and $suf(w_j, 3)$ respectively
- vector of **boolean** variables $\boldsymbol{c_j}$ for w_j (yes=1, no=0): *word-is-capitalized*, *word-contains-hyphen* and *word-contains-digit*

We take into account the following **EXTRINSIC FEATURES** for word w_j:

- For $t = -2, -1, +1, +2$ (window of size 2)
  - the **neighbor word** w_{j+t}
  - 2 and 3-letter **prefix** of neighbour w_{j+t}, i.e. $pref(w_{j+t}, 2)$ and $pref(w_{j+t}, 3)$ respectively
  - 2 and 3-letter **suffix** of neighbour w_{j+t}, i.e. $suf(w_{j+t}, 2)$ and $suf(w_{j+t}, 3)$ respectively
  - vector of **boolean** variables $\boldsymbol{c_{j+t}}$ for neighbour w_{j+t}: *word-is-capitalized*, *word-contains-hyphen* and *word-contains-digit*
- For $t = -2, -1$ (previous 2 words)
  - **Predicted PoS** for word w_{j+t}, i.e. $\hat{p}_{j+t}$

Feature Embedding

The next step is to encode the linguistic features for word w_j into a vector $\boldsymbol{v_j}$, which summarizes the information coming from word w_j.

We use four different embedding functions with different dimensions[4]: $v_w(\cdot) \in \mathbb{R}^{d_w}$ for *words*, $v_p(\cdot) \in \mathbb{R}^{d_p}$ for *prefixes*, $v_s(\cdot) \in \mathbb{R}^{d_s}$ for *suffixes* and $v_t(\cdot) \in \mathbb{R}^{d_t}$ for *tags*.

Overall, $\boldsymbol{v_j}$ can be written as a *concatenation*

$$\boldsymbol{v_j} = \left[\underbrace{\boldsymbol{c_j}}_{BOOL}; \underbrace{v_w(w_j)}_{WORD}; \underbrace{v_s(suf(w_j,2)}_{2-SUFF}; \underbrace{v_s(suf(w_j,3)}_{3-SUFF}; \underbrace{v_p(pref(w_j,2)}_{2-PREF}; \underbrace{v_p(pref(w_j,3)}_{3-PREF} \right]$$

and has *dimension*

$$\boldsymbol{v_j} \in \mathbb{R}^{\overbrace{3}^{BOOL} + \overbrace{d_w}^{WORD} + \overbrace{2d_s}^{SUFF} + \overbrace{2d_p}^{PREF}}.$$

[4]This is an example where d can change for different feature types

For example, the **information** for the word PIZZA can be summarized in the vector

$$\boldsymbol{v}_{\text{PIZZA}} = \left[\underbrace{(0,0,0)}_{BOOL}; \underbrace{v_w(\text{PIZZA})}_{WORD}; \underbrace{v_s(\text{ZA})}_{2-SUFF}; \underbrace{v_s(\text{ZZA})}_{3-SUFF}; \underbrace{v_p(\text{PI})}_{2-PREF}; \underbrace{v_p(\text{PIZ})}_{3-PREF} \right]$$

Finally, the **input vector $\boldsymbol{x}$** will be the result of the *concatenation* of the embedding of extrinsic features, intrinsic features and previous tags

$$\boldsymbol{x} = \phi(s,j) = \left[\underbrace{\boldsymbol{v_{j-2}}; \boldsymbol{v_{j-1}}}_{EXTRINSIC}; \underbrace{\boldsymbol{v_j}}_{INTRINSIC}; \underbrace{\boldsymbol{v_{j+1}}; \boldsymbol{v_{j+2}}}_{EXTRINSIC}; \underbrace{v_t(\hat{p}_{j-2}); v_t(\hat{p}_{j-1})}_{PREVIOUS\ TAG} \right]$$

and dimension

$$\boldsymbol{x} \in \mathbb{R}^{\overbrace{5}^{WINDOW} \overbrace{(3 + d_w + 2d_s + 2d_p)}^{1\ EMBEDDING} + \overbrace{2d_t}^{TAG}}.$$

For example, in the sentence "TONIGHT I WILL HAVE A PIZZA WITH MY FRIENDS", the input vector $\boldsymbol{x}$ for the word PIZZA is

$$\boldsymbol{x} = \phi(s, \text{PIZZA}) = \left[\underbrace{\boldsymbol{v}_{\text{HAVE}}; \boldsymbol{v}_{\text{A}}}_{EXTRINSIC}; \underbrace{\boldsymbol{v}_{\text{PIZZA}}}_{INTRINSIC}; \underbrace{\boldsymbol{v}_{\text{WITH}}; \boldsymbol{v}_{\text{MY}}}_{EXTRINSIC}; \underbrace{v_t(\hat{p}_{\text{HAVE}}); v_t(\hat{p}_{\text{A}})}_{PREVIOUS\ TAG} \right].$$

Discussion

The first consideration when using such an approach regards the *computation* of the vectors $\boldsymbol{v_j}$.

Since we use the same embedding function $v_w(\cdot)$ for every word w_j, we actually have to learn only one embedding table, i.e. matrix $\boldsymbol{E}$. Therefore, the computation $v_w(w_j)$ is quite cheap, since it is performed *only once* and the result is reused for different positions j. Moreover, this approach stores the information about the *relative position* of the words in the sentence, i.e. some information is *shared* between different input vectors $\boldsymbol{x}$ and therefore their statistical strength in the model is increased.

The second consideration is about *word-capitalization.* In the previous example, the embedding vectors for the words TONIGHT and TONIGHT are different. Since we are already take into account the capitalization of word w_j in the boolean vector $\boldsymbol{c_j}$, this information will result to be *redundant* in the input vector $\boldsymbol{x}$. If we decide to keep the vector $\boldsymbol{c_j}$, then it is better to lower-case all words in our vocabulary and then compute the vectors $\boldsymbol{v_j}$.

Another source of redundancy may be the 2 and 3-letter prefix and suffix feature. In order to overcome this problem, one possibility is consider character-level models, instead of word-level models. As we will see in section 3.4 with character-level RNNs, this approach augments data granularity and is suitable for spotting specific patterns and regularities in the words.

Finally, we should always take into account that some words, e.g. PIZZA, may not be present in our vocabulary V. Such unkown words are called *out-of-vocabulary (OOV)* items and they are mapped to the special symbol UNK.

1.3 The Advent of Deep Learning in NLP

It was at the end of the 20th century that NLP experienced the so-called *statistical revolution* (Johnson, 2009). According to Jurafsky and Martin (2008), three main factors determined such a big change in the field:

1. Popularity of *probabilistic and data-driven models* over knowledge-based methods.

2. Steady increase of *computational power* allowed the deployment of commercial NLP tools.

3. The rise of the *Web* increased data availablity and imposed the need to develop language-based information retrieval.

Since the beginning of the 21st century, Deep Learning has dramatically changed again NLP. According to Goldberg (2017), two main factors have determined the success of deep learning over other machine learning models:

1. The use of the *embedding layer* to map *textual data* (discrete and sparse) to a *feature-vector* (continuous and low-dimensional).

2. The flexibility of RNNs to model sequence data and consequently the abandon of *markov assumption.*

Embedding Layer and Representational Learning

The concept of embedding layer in NLP is actually an instance of a much more general concept in machine learning: *representational learning* (Goodfellow et al., 2016). In general, this concept means that a machine learning algorithm learns not only to correctly predict the target output, but learns also how to correctly represent the data.

In the deep learning domain, such representation is learned by successive approximations (from the early layers to the last layers) of the input data. In the last years, this feature has become very popular also in other domains and actually led to the development of **specialized** architectures for different tasks, e.g. Convolutional Neural Networks (CNNs) for Image Recognition tasks and Recurrent Neural Networks (RNNs) for NLP tasks.

Feedforward Neural Networks

The concept of representational learning can be easily understood when considering the first and simplest class of artificial neural network architectures, i.e. *Feedforward Neural Network*s.

As already mentioned, specialized recurrent architectures are actually a modification of the feedforward architecture. Therefore, the key intuitions behind the concept of representational learning can be first derived from feedfoward architecture and then generalized to the recurrent architecture. Having said that, the goal of Chapter 2 is to introduce the reader to the key concept of representational learning, by presenting the *Multiple-layer Perceptrons (MLPs)* abstraction and its advantages compared to the traditional linear models, e.g. *Logistic Regression*.

The first advantage of feedforward neural networks over linear models regards the *representation power*. In particular, it has been shown that MLP_1(MLP with a single hidden layer) is a universal approximator, i.e. any input can be mapped from any finite dimensional discrete space to another.

The second advantage is that MLP architecture implements a *trainable* nonlinear mapping function, which is a particularly useful feature when the linear-separability condition does not hold in the training data. This is definitely the case in NLP, where the input is discrete and highly sparse.

Reccurent Neural Networks

So far we have seen that MLPs are excellent alternatives to the standard linear models for classification tasks, given their ability to learn in principle any representations of the data, regardless the nonlinearity and complexity of the input. However, MLPs are **general-purpose** classification architectures and thus they are not tailored for specific NLP tasks, e.g. Language Modeling.

For this reason, in Chapter 3 we will discuss another class of artificial neural networks, i.e. RNN architecture. In particular, we will present the RNN high-level abstraction, graphical representations (recursive and unrolled) and architectural variations (*biRNNs* and *deep RNNs*). Finally, three concrete recurrent architectures (*Simple RNN*, *LSTM* and *GRU*) will be presented in details in sections 3.2 and 3.3.

Compact Recurrent Neural Networks

One typical problem of deep neural networks is that their efficacy and execution time are dependent on the *size* of the network. In particular, these models are often too large in size for deployment on mobile devices with memory and latency constraints.

In our NLP framework, this means that many simple NLP tasks can be performed only by calling an external cloud server infrastructure, i.e. without the possibility to do the computation offline.

The first objective of Chapter 4 is to understand and categorize the state-of-the-art compression techniques already available. The goal is firstly to understand the rationale behind each of these technique and secondly to go through the mathematical details and implementations of the methods. In the mean time, the second objective is to apply these compression techniques on some simple NLP tasks (Language Modeling, Part-of-speech Tagging, Named Entity Recognition, Chunking), in order to compare the compact model with the full model and eventually to spot the distribution of *redundancy* in the RNN architecture.

RNNs and Markov Assumption

One strong prerequisite for any NLP model is the ability to take into account the dependency structure in sequencial data, e.g. a sentence s made of $w_1, \dots, w_n$ words.

In the last decades, the traditional approach in NLP was to restrict the framework and simply to condition the word w_{t+1} on a fixed-size (e.g. with size k) past sequence $w_{t-k}, \dots, w_t$. This assumption is called k-th order *Markov assumption* in probability theory and generally refers to the memoryless property of stochastic processes.

The main issue with the Markov assumption is the nature of the input. As we have already discussed in section 1.2, textual data is highly ambiguous and variable. Ambiguity imposes to take into account the context of word w_{t+1} in a sentence, e.g. the last k words in the sequence. At first sight, Markov assumption fits the NLP framework. However, variability of textual data injects randomness in the input, which consequently makes it difficult to a-priori set a proper value for k.

RNNs solve this issue by allowing flexible and trainable windows of arbitrary size. This feature represented an incredible improvement in many important NLP tasks, e.g. Language Modeling.

Having said that, simple RNNs architectures have still some issues. In a sentence or in a longer text, the context and consequently the meaning of a word does not necessarily depend only on the past words. For example, in Handwriting Recognition task, where the goal is to convert handwritten input (noisy) to letter codes (bits), the prediction on the target letter can be improved by taking into account the letters coming after it.

Bidirectional Recurrent Neural Networks (biRNN) modify the training step of traditional RNNs, by presenting two version of the input sequence (original and reversed) to two separate RNNs. The final prediction is based on both RNNs outputs. The details are discussed in section 3.4.

Research Areas

Two promising research areas in the deep learning and NLP domains are Multitask Learning (MTL) and Semi-supervised Learning.

MTL has the goal to improve the performance in a given NLP task, by combining the information from other NLP tasks. This approach is motivated by the intimate relationship between NLP tasks, as already seen in section 1.1.

Semi-supervised Learning wants to improve the accuracy on one task, by exploting data which have been annotated or unannotated for other tasks. Data annotation is unfortunately a very delicate, costly and time-consuming process.

MTL and Semi-supervides Learning will not be futher discussed in this work.

2 Feedforward Neural Networks

Nowadays, it is extremely popular to see feedforward neural networks in many machine learning applications both at academic and industrial scale. As already mentioned in the previous chapter, this increase in popularity led to the development of specialized architectures for different tasks, e.g. Convolutional Neural Networks (CNNs) for Image Recognition tasks and Recurrent Neural Networks (RNNs) for NLP tasks.

The aim of this Chapter is to take a conceptual step back, before going into the details of RNNs and their applications to NLP tasks.

In section 2.1 we will review the key concepts behind supervised learning algorithms, which are one of the most used in several machine learning applications. The theorethical concepts of hypothesis class and inductive bias will be presented together with a simple example of parametric supervised learning algorithms, i.e. Linear Models. This will introduce the limits of linear models and so the need to perform nonlinear transformation on the input, when dealing with linearly non separable data. Three different startegies will be presented, which will eventualy lead to the choice of Multi Layer Perceptrons (MLPs) due to their univerisal representation power.

In section 2.2, after describing the differences between recurrent and feedforward architectures, we will mathematically define Artificial Neuron, Single Layer and Multi Layer architectures. Moreover, it will be introduced the concept of activation function and representation power in neural networks.

The last section of this Chapter is devoted to the design choices which have to be made when training a neural network. The largest difference between linear models and neural networks is that the neural network nonlinearity causes most interesting loss functions to become non-convex. Therefore, the optimization process is iterative and gradient-based and usually requires computing the gradients of complicated functions. The popular algorithm is Backpropagation and its implementation is usually based on the Computation Graph Abstraction. This algorithm will be extended to RNNs, where it will be called Backpropagation through time.

The neural network's notation and figures of this chapter are taken from Goldberg (2017). The exposition of the main ideas from machine learning and neural network's training are inspired by the work of Goodfellow et al. (2016).

2.1 Supervised Learning

Consider a functional dependency that maps points from an input space $X \in \mathbb{R}^{d_{in}}$ to an output space $Y \in \mathbb{R}^{d_{out}}$.

In a typical supervised learning task we are given a *training set* T of n input-target pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, i.e.

$$T = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : \boldsymbol{x}_i \in X, \boldsymbol{y}_i \in Y \text{ and } i = 1, \dots, n\}\,.$$

The goal of supervised learning is to define a mapping f and produce a prediction $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$, which correctly predicts the true output $\boldsymbol{y}$.

When dealing with *classification problems*, the input space X is divided into K subsets $X_1, \dots, X_K \in X$ such that $X_i \cap X_j = \emptyset$ for all $i, j = 1, \dots, K$ and $i \neq j$. Now the task is to assign a given input vector $\boldsymbol{x}$ to the subset it belongs to.

The basic form of any classification task is the *binary classification*, where there are two sets $X_1, X_2 \in X$ such that $X_1 \cap X_2 = \emptyset$ and we want to determine whether the input vector $\boldsymbol{x}$ belongs to X_1 or X_2. In this case, the training set is formaly defined as

$$T_{binary} = \{(\boldsymbol{x}_i, y_i) : \boldsymbol{x}_i \in X, y_i \in \{-1, +1\} \text{ and } i = 1, \dots, n\}$$

with the two subsets X_1 and X_2 labelled by $+1$ and -1, respectively.

Linear Model and Nonlinear Input

It goes without saying that the set of all possible functions f is extremely large. What is tipically done in practice is to restrict our search only to some families of functions f, called *hypothesis classes*, where all the elements in this family share some propreties, e.g. *parametric* supervised learning algorithms.

This is the case of Linear Models, which come in the form

$$f(\boldsymbol{x}; \Theta) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$$
$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \boldsymbol{b} \in \mathbb{R}^{d_{out}}$$
$$\Theta = \boldsymbol{W}, \boldsymbol{b}$$

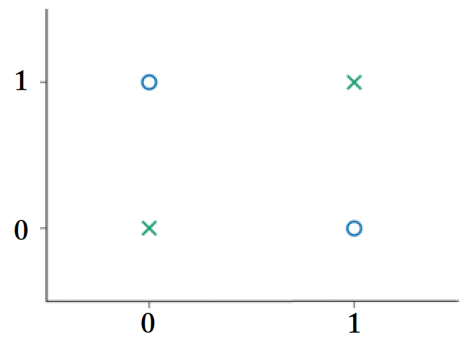


Figure 2.1: XOR function: green crosses belong to class -1, while blue circles belong to class +1.

and we assume that output $\boldsymbol{y}$ can be represented as a linear combination of inputs $\boldsymbol{x}$.

For example, in binary classification the linear model assumes that there exists a hyperplane which can perfectly separate the two set of points X_1 and X_2, i.e. the *linear separability condition* holds. In machine learning, this type of assumptions on the model are called *inductive bias*.

A simple example where the linear assumption is not verified is the XOR problem. The XOR function is defined as

$$\begin{aligned}\text{XOR}(0,0) &= -1\\ \text{XOR}(1,0) &= +1\\ \text{XOR}(0,1) &= +1\\ \text{XOR}(1,1) &= -1\end{aligned}$$

and is graphically represented in Figure 2.1.

In order to make linear models able to represent nonlinear functions of $\boldsymbol{x}$, one could apply the linear model to a trasformed input $\phi(\boldsymbol{x})$, where $\phi(\cdot)$ is a nonlinear function.

The question now is how to choose a suitable ϕ:

1. Use a very **generic** ϕ. This is the solution generally proposed by *kernel methods*, where $\boldsymbol{x}$ is projected into a high-dimensional (even infinite) space $\phi(\boldsymbol{x})$, where the linear model can fit well the training data. The main issue with such an approach is the poor generalization ability of the model.

2. Use a **manually-designed** ϕ. This approach belongs to the past and is definitely not suitable for modern machine learning applications, since it

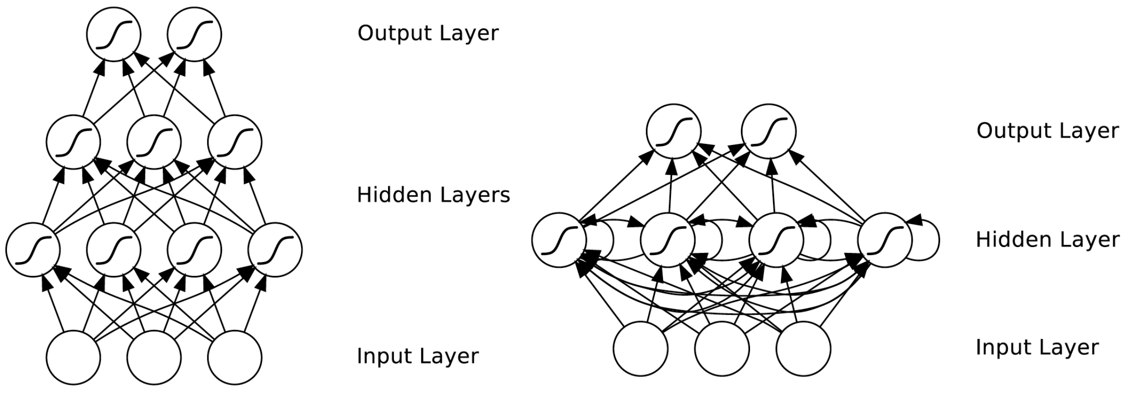


Figure 2.2: Feedforward (left) and Recurrent (right) architecture.

is highly dependent on the dataset and does not allow transfer learning between domains.

3. Use a **trainable** ϕ. This is the case of feedforward neural networks and it represents the core of representational learning. In this case the final prediction $\hat{\boldsymbol{y}}$ is based on two steps: the neural network first learns ϕ from a broad class of functions and then it maps $\phi(\boldsymbol{x})$ to the desired output $\hat{\boldsymbol{y}}$. As we saw in the previous chapter, representational learning means that a machine learning algorithm learns not only how to correctly predict the target output $\boldsymbol{y}$, but it learns also how to correctly represent the data through $\phi(\boldsymbol{x})$.

In the next sections we will see in detail the Multilayer Perceptron architecture and some general ideas about neural network training.

2.2 Feedforward Architecture

Artificial Neural Networks (ANNs) have appeared in the past in different architectures and related propreties, according to the task they were designed to solve. The main distinction in ANNs regards the presence or not of cycles in their graph.

ANNs *with cycles* (see Figure 2.2 on the right) are called Recurrent Neural Networks (RNNs) and are dealt with in Chapter 3. ANNs *without cycles* (see Figure 2.2 on the left) are referred to as feedforward neural networks (FNNs) and are explored in this section.

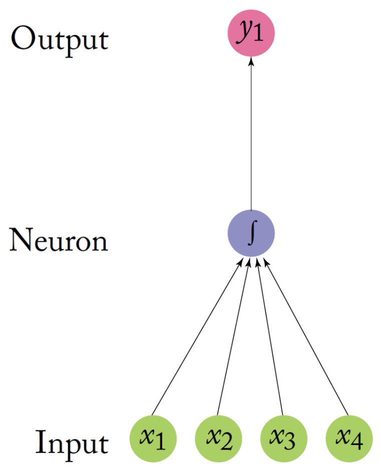


Figure 2.3: Single Artificial Neuron with 4 (scalar) inputs x_1, x_2, x_3, x_4.

Artificial Neuron

Using a metaphor from the biology and neuroscience field, an artificial *neuron* is nothing but the node of an ANN graph. In other words, a neuron represents the elementary computational unit of a neural network.

For example, the neuron in Figure 2.3 takes 4 scalars x_1, x_2, x_3, x_4 as input with associated weights w_1, w_2, w_3, w_4 (not shown in the picture, but graphically corresponding to the 4 arrows from inputs to the neuron).

The operations performed by the neurons are the following:

- Multiply each input x_i by its weight w_i
- Sum them up, i.e. $\sum_{i=1}^{4} x_i w_i$
- Apply a nonlinear function g (called activation function and shown as $\int$ in figure 2.3) to the previous result, i.e. $g(\sum_{i=1}^{4} x_i w_i)$
- Pass the priovious result to the output, i.e. $y_1 = g(\sum_{i=1}^{4} x_i w_i)$

In ANNs neurons are connected to each other, forming the so-called neural network.

Single Layer Perceptron

As a linear classifier, the simplest feedforward neural network is the single layer perceptron, which can be written in mathematical terms as

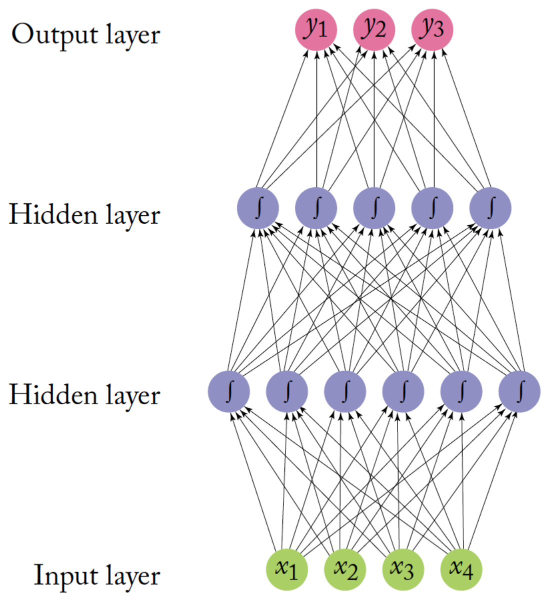


Figure 2.4: Multi Layer Perceptron with two hidden layers (MLP2).

$$\mathrm{NN}_{\text{Single Layer Perceptron}}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$$
$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \boldsymbol{b} \in \mathbb{R}^{d_{out}}$$
$$\Theta = \boldsymbol{W}, \boldsymbol{b}$$

and in fact coincides with the linear model introduced in Section 2.1 and therefore has the same limitations when dealing with nonlinear input.

Multi Layer Perceptron

The limits of linear functions can be overcome by adding one or more non-linear hidden layers in the network. The result is the Multi Layer Perceptron (MLP) architecture.

An example of feed-forward neural network with two hidden layers (MLP2) is represented in Figure 2.4.

More generally, a MLP2 can be mathemathically written as:

$$
\begin{gathered}
\mathrm{NN}_{\mathrm{MLP2}}(\boldsymbol{x}) = \boldsymbol{y} \\
\boldsymbol{h}^1 = g^1(\boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1) \\
\boldsymbol{h}^2 = g^2(\boldsymbol{h}^1\boldsymbol{W}^2 + \boldsymbol{b}^2) \\
\boldsymbol{y} = \boldsymbol{h}^2\boldsymbol{W}^3 \\
\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \boldsymbol{y} \in \mathbb{R}^{d_{out}} \\
\boldsymbol{W}^1 \in \mathbb{R}^{d_{in} \times d_1}, \boldsymbol{b}^1 \in \mathbb{R}^{d_1}, \boldsymbol{W}^2 \in \mathbb{R}^{d_1 \times d_2}, \boldsymbol{b}^2 \in \mathbb{R}^{d_2}, \boldsymbol{W}^3 \in \mathbb{R}^{d_2 \times d_{out}} \\
\Theta = \boldsymbol{W}^1, \boldsymbol{W}^2, \boldsymbol{W}^3, \boldsymbol{b}^1, \boldsymbol{b}^2
\end{gathered}
$$

Compared to the example of Figure 2.3, the network in MLP2 is now described in terms of vectors and matrices:

- Input $\boldsymbol{x}$ and output $\boldsymbol{y}$ are vectors with dimension d_{in} and d_{out} respectively.
- Weights and bias terms of layer l[1] are stored in matrices $\boldsymbol{W}^l$ and vectors $\boldsymbol{b}^l$ respectively.
- Weight w^l_{ij} represents the connection from the i-th neuron in layer l to the j-th neuron in layer $l+1$.
- The activation function g^l is applied element-wise.

In a multilayer perceptron architecture, neurons are arranged in layers, with connections feeding forward from one layer to the next. The length of this chain of connections gives the model's *depth*, from which comes the word deep learning.

Input patterns are presented to the input layer, then propagated through the hidden layers to the output layer. When layer $\boldsymbol{h}^l$ in the network is the result of a linear transformation of the input, then it is called *fully-connected* layer. Other types of layers are for example *convolutional* and *pooling* layers, which are particularly useful in immage recognition tasks.

The next parts of this section are devoted to some further discussion on the MLP architecture.

Activation Functions

In the MLP architecture, the activation function is applied to each neuron's value before passing it to the output. Figure 2.5 shows some common choices of activation functions g and their derivatives g'.

[1]The input layer has $l = 1$

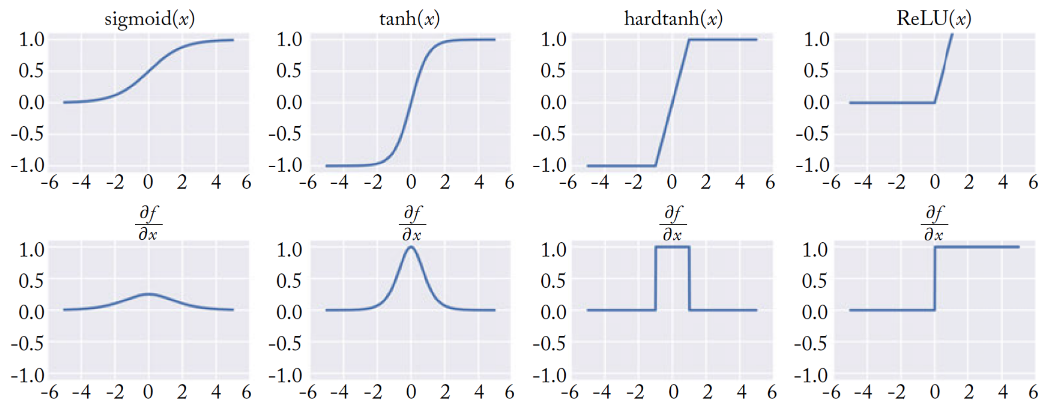


Figure 2.5: Four common Activation Functions (top) and their Derivatives (bottom).

The most popular activation functions are the hyperbolic tangent and the sigmoid functions, which are differentiable and thus allow the network to be trained with gradient descent, as we will see in Section 2.3.

Having said that, in general the choice of g is really task-dependent. For example, for binary classication tasks the standard configuration in the output layer is a single unit with a sigmoid activation function.

What these activation functions have in common is their nonlinearity. This is the reason why Multi Layer Perceptron is more powerful than Single Layer Perceptron, since it can succesfully deal with non-linear classication boundaries and model non-linear equations. Note that any MLP with linear hidden layers (i.e. g^l is linear for all l) is exactly equivalent to the Single Layer Perceptron, since any combination of linear operators is itself a linear operator. Here lies the advantage of using nonlinear activation functions: the neural network can first learn how to correctly represent the input $\boldsymbol{x}$ at successive hidden layers and then predict the output $\boldsymbol{y}$, based on this new representation of $\boldsymbol{x}$.

Representation Power

Going back to the beginning of this chapter, the goal of supervised learning is to define a mapping $\hat{\boldsymbol{y}}$ and produce a prediction $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$, which correctly predicts the true output $\boldsymbol{y}$.

A MLP can actually provide a wide range of different functions f to map from input $\boldsymbol{x}$ to the desired output $\boldsymbol{y}$. In particular, it has been proven that an MLP with a single hidden layer containing a sufficient number of nonlinear units can approximate "any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy" Hornik et al. (1989). This is the reason why MLPs are said to be *universal function approximators*.

From one hand, it seems that the simple MLP1 (only one hidden layer) is already the best architecture. From the other hand, it should be noted that this theorem states only the existence of such an universal function approximator and does not go into the exact configuration of such MLP network, e.g. how many units are in the hidden layer or how to set the network parameters.

In practice, when working with real-world large datasets, many computation and optimization problems arise when training a MLP1 neural network. Therefore, best practices in deep learning reccomend to build more complex and deep architectures.

2.3 Neural Network Training

The training step for a neural network does not differ so much from what is done in any other machine learning model. The parameter estimation in neural networks is also expressed as an optimization problem, which is solved using gradient descent. For this reason, the first part of this section will give a brief recap of the key concepts in optimization, i.e. *loss function*, *regularization* and *gradient-based optimization*.

The main difference in neural networks regards the computation of the gradient, which is more complicated but can still be done efficiently and exactly. The second part of this section will describe how to obtain the gradient using the *back-propagation algorithm* and how to implement it using the high-level *computational graph abstraction*, which is exploited nowadays by dedicated software libraries and APIs.

The largest difference between linear models and neural networks is the non-linearity. As we have seen in section 2.2, this represents the main advantage of using a neural network for modern deep learning applications. However, the non-linearity of a neural network causes the loss function to become non-convex. In optimization theory, convexity is used to mathematically proove the convergence of many optimization algorithms to a global minimum. When the same optimization algorithms, e.g. Stochastic Gradient Descent, are applied to non-convex loss functions, then there is no such convergence guarantee and the final result is sensitive to parameter initialization of the network. In the last part of this section, we will give an overview on the *best practices* for training neural networks, which have emerged among deep learning practitioners in the last years.

Training as Optimization

The majority of machine learning training algorithms can be seen as optimization problems. More specifically, an optimization problem consists of maximizing or minimizing a real function $f(x)$ by systematically choosing input values x from within an allowed set and computing the value of the function.

Formally, we consider problems of the following form:

$$\begin{gathered} \min f(x) \\ x \in \mathbb{R}^n \end{gathered}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a continuous function in $\mathbb{R}^n$.

We tipically define optimization problems by minimizing $f(x)$. Maximization of $f(x)$ can be obtained by applying the minimization algorithm on $-f(x)$.

Generally, f is called the *objective function.* In machine learning, f is usally indicated as L and is called *cost function*, *loss function*, or *error function.*

The *minimizer* is the value of x which minimizes or maximizes f and it is indicated as

$$\hat{x} = \text{argmin } f(x).$$

In deep learning x are the *parameters* of the network and are indicated as Θ.

Loss function

In machine learning, the loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ measures the quality of the prediction $\hat{\boldsymbol{y}}$ given the true expected output $\boldsymbol{y}$.

The loss function should be bounded from below, with the minimum attained only for cases where the prediction is correct. The parameters of the model Θ are then set in order to minimize the loss L over all training examples.

In practice, given the training set input-target pairs $(\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n})$, a mapping function $\hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i; \Theta)$ and a per-instance loss function $L(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)$, we tipically define the *overall loss*

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i) = \frac{1}{n} \sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i)$$

as the average of the loss functions over all training examples.

Now the goal of the training algorithm is to optimize the quantity $\mathcal{L}(\Theta)$ w.r.t. parameters Θ. In neural networks, the optimal value of the parameters will be therefore

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}}\ \mathcal{L}(\Theta). \tag{2.1}$$

In theory, the loss function can be any function mapping two vectors ($\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$) to a numerical score (scalar). However, in practice we choose loss functions for which the gradients can be easily computed.

We list here some convex loss functions that are commonly used for NLP classification problems with neural networks.

Hinge Loss Function In binary classification problem the ouput of the neural network is tipically the scalar $\tilde{y}$, but the target output y belongs to the set $\{-1, +1\}$. The prediction rule is then

$$\hat{y} = \operatorname{sign}(\tilde{y}) = \begin{cases} +1 & \text{if } \tilde{y} > 0 \\ 0 & \text{if } \tilde{y} = 0 \\ -1 & \text{if } \tilde{y} < 0 \end{cases}$$

and the hinge-loss function is defined as

$$L_{\text{hinge(binary)}}\left(\tilde{y}, y\right) = \max\left\{0, 1 - y \cdot \tilde{y}\right\} = \mid 1 - y \cdot \tilde{y} \mid_{+}.$$

Note that the loss is 0 when the prediction is correct, i.e. if y and $\tilde{y}$ share the same sign and $|\tilde{y}| > 1$. The loss is linear when the prediction is not correct. Compared to the *0-1 indicator function*, the hinge loss function provides a relative tight, convex upper bound.

When we deal with multi-class classification problems, $\boldsymbol{y}$ is a K-dimensional one-hot vector which contains the correct output class t among the K possible classes.

The output of the classifier is contained in the K-dimensional vector $\tilde{\boldsymbol{y}} = (\tilde{\boldsymbol{y}}_1, \tilde{\boldsymbol{y}}_2, \ldots, \tilde{\boldsymbol{y}}_K)$. The prediction rule is then

$$\hat{\boldsymbol{y}} = \underset{i \in \{1,\ldots,K\}}{\operatorname{argmax}}\ \tilde{\boldsymbol{y}}_i$$

and the hinge-loss function is defined as

$$L_{\text{hinge(multi-class)}}\left(\tilde{\boldsymbol{y}}, \boldsymbol{y}\right) = \max\left\{0, 1 - \left(\tilde{\boldsymbol{y}}_t - \tilde{\boldsymbol{y}}_k\right)\right\}$$

where t is the correct output class and $k = \underset{i \in \{1,\ldots,K\}: i \neq t}{\operatorname{argmax}}\ \tilde{\boldsymbol{y}}_i$ is the class with the highest predicted score such that $k \neq t$.

The hinge loss function is generally used for hard decision classification rule, i.e. when we are interested only in determining the most likely class, without quantifying our degree of belief on our decision.

Moreover, the binary hinge class is both continous and convex, but it is not differentiable at $\boldsymbol{y} \cdot \tilde{\boldsymbol{y}} = 1$. This does not allow the use of gradient-based optimization methods, which require differentiability over the entire domain.

Cross-Entropy Loss Function In classification problems, cross-entropy loss function is able to model class membership probability.

Let $\tilde{y}$ be the model's output and $y \in \{0, 1\}$ be the correct class[2]. We first apply the sigmoid function to the model's output $\tilde{y}$

$$z = \sigma(\tilde{y}) = \frac{1}{1 + e^{-\tilde{y}}}$$

where $z \in [0, 1]$. The transformation $\sigma(\tilde{y})$ can be interpreted as the conditional probability $\mathbf{P}(y = 1 \mid x)$.

The prediction rule is then

$$\hat{y} = \begin{cases} 0 & \text{if } z < 0.5 \\ 1 & \text{if } z \geq 0.5 \end{cases}$$

and the cross-entropy loss function is defined as

$$L_{cross-entropy(binary)}(z, y) = -y \log z - (1 - y) 1 \log(1 - z).$$

Similarly to the hinge loss, the cross-entropy loss function can also be applied to a multi-class classification task.

When we are interested in modeling the probability of the scores, $\boldsymbol{y}$ is assumed to be a K-dimensional vector which contains the true multinomial distribution among the K possible classes. In such case, we apply the softmax function to each element of the network's output $\tilde{\boldsymbol{y}} = (\tilde{\boldsymbol{y}}_1, \tilde{\boldsymbol{y}}_2, \ldots, \tilde{\boldsymbol{y}}_K)$

$$\boldsymbol{z}_i = \text{softmax}(\tilde{\boldsymbol{y}}_i) = \frac{e^{\tilde{\boldsymbol{y}}_i}}{\sum\limits_{j=1}^{K} e^{\tilde{\boldsymbol{y}}_j}}.$$

Now the vector $\boldsymbol{z}$ can be interpreted as a probability distribution, since its ele-

[2]This is just a transformation of the previous $y_{old} \in \{-1, 1\}$. Now $y_{new} = \frac{1 + y_{old}}{2} \in \{0, 1\}$

ments are positive and sum to 1. The cross-entropy loss function is then

$$L_{cross-entropy(multi-class)}(\boldsymbol{z}, \boldsymbol{y}) = -\sum_{i=1}^{K} \boldsymbol{y}_i \log(\boldsymbol{z}_i).$$

Regularization

The main issue of Equation 2.1 is that it minimizes the loss function by taking into account only training data. In the long run, this can cause the model to be very precise on training data (*training error*), but have poor performance on previously unseen data (*generalization error*). The larger the gap between training and generalization error, the more our model is *overfitting* training data.

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Back to Equation 2.1, we can modify it by adding the regularization term $R(\Theta)$, which quantifies the "complexity" of the model.

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}}\ \mathcal{L}(\Theta) + \lambda R(\Theta) \tag{2.2}$$

In the formula above, we have included also a hyper-parameter term λ, which controls the amount of regularization, i.e. how much simple models have to be preferred over complex ones. The value of λ is set manually and depends on the model performance and training/test set.

Now the optimization algorithm will choose the parameter values Θ which have not only low loss $\mathcal{L}(\Theta)$, but also low complexity $R(\Theta)$.

Talking about the neural network framework, what the regularizer R does in practice is to quantify the networks' complexity by first computing the norms of the weight matrices W and then choose parameters $\hat{\Theta}$ whose matrices have low norms. The typical regularization strategies involve L_2 norm, L_1 norm or *Elastic-Net*, which combines the previous two. Another common choice of regularization for neural networks in *dropout*, which will be discussed in Section 2.3.

Gradient-Based Optimization

The optimization methods which are tipically used to train any machine learning model, i.e. solve the optimization problem in equation 2.2, involve gradient computation.

Gradient-based optimization methods can be traced back to Cauchy and represent the simplest way to minimize a differentiable function g on $\mathbb{R}^n$. Mathematically speaking, the main idea behind gradient descent schemes is to find the local minimum of g by proportionally moving from the current point x_k towards the opposite directions of the gradient $\nabla g(x_k)$.

When applying gradient descent methods to neural network training, there exists many algorithmic variants which have been developed with the goal to speed up the training phase.

The common point of all these methods is that they compute just an *estimate* of the overall loss $\mathcal{L}$. Indeed, they differ on how error estimate is computed, and how update step is defined.

It should be noted that due to the nonlinearity of neural networks, the objective function $\mathcal{L}$ is *not convex* and so gradient-based methods may find solutions which are not global minima. Still, gradient-based methods represent most popular choice for neural network training.

Gradient Computation in Neural Networks

The main peculiarity for neural network training regards the gradient computation, which is done automatically and efficiently by the Backpropagation algorithm and is implemented in practice by the Computational Graph Abstraction. Indeed, this algorithm is nothing but a fancy name for methodically computing the derivatives of a complex expression using the chainrule, while caching intermediary results.

Computational Graph Abstraction

Theoretically, the gradient computations of thousands of parameters in a neural network can be first done by hand and then implemented in code. However, when deploying or testing a neural network for practical applications, it is much more convenient to use automatic tools, which minimally reduce the effort and the probability of errors.

This is the reason why the computational graph abstraction has become the standard way to build any neural network, evaluate prediction $\hat{\boldsymbol{y}}$ for given input $\boldsymbol{x}$ (*forward pass*), and compute gradient for parameters Θ with respect to arbitrary scalar loss $\mathcal{L}$ (*backward pass*).

A computation graph is nothing but a way to represent arbitrary mathematical computations as a graph. There exists many different ways of formalizing computations as graphs. Following what has been done by Goldberg (2017), in the present work a computation graph is a directed acyclic graph (DAG) and it is connected.

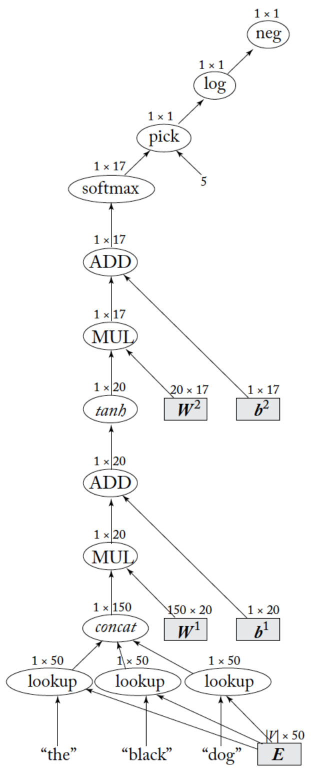


Figure 2.6: Computational Graph for a MLP1 taking three words as input.

Nodes correspond either to mathematical operations (ovals), e.g. SUM, either to model parameters (shaded rectangles), e.g. weight matrix. Nodes are connected by *edges* (arrows) and the overall graph structure determines the order of the computations. The *inputs* of the networks are considered as constants and are drawn without any surrounding node.

In Figure 2.6, we see the computation graph for a MLP with one hidden-layer and a softmax output transformation. This MLP takes as input three words (e.g. THE BLACK DOG), converts them into the embedding vector $\boldsymbol{x}$ (using the embetting matrix $\boldsymbol{E}$ as explained in the previous chapter in section 1.2) and predicts the part-of-speech tag for the third word (NOUN is the expected output for the word DOG). Formally the computation graph in Figure 2.6 is defined as

$$
\begin{gathered}
\mathrm{NN}_{\mathrm{MLP1}}(\boldsymbol{x}) = \boldsymbol{y} \\
\boldsymbol{h}^1 = tanh(\boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1) \\
\boldsymbol{y} = softmax(\boldsymbol{h}^1\boldsymbol{W}^2 + \boldsymbol{b}^2) \\
\boldsymbol{x} \in \mathbb{R}^{150}, \boldsymbol{y} \in \mathbb{R}^{17} \\
\boldsymbol{W}^1 \in \mathbb{R}^{150\times 20}, \boldsymbol{b}^1 \in \mathbb{R}^{20}, \boldsymbol{W}^2 \in \mathbb{R}^{20\times 17}, \boldsymbol{b}^2 \in \mathbb{R}^{17} \\
\Theta = \boldsymbol{W}^1, \boldsymbol{W}^2, \boldsymbol{b}^1, \boldsymbol{b}^2
\end{gathered}
$$

The computation graph also includes one more node *pick*, which is responsible for selecting the entry corresponding to the true part-of-speech tag (NOUN in this example) of the output vector $\boldsymbol{y}$, which contains the probability distribution over 17 part-of-speech tags (NOUN is in position 5).

Regardless the apparent complexity of this example, the process for building computation graph is actually very quick and easy by using dedicated software libraries and APIs.

Best Practices for Training Deep Models

Training a neural network is a very hard task from an optimization point view.

Firstly, the optimization problem is not convex and so optimization theory does not guarantee us convergence to a global minimum. What happens in practice is that the optimization algorithm produces different results for different (random and small) parameter initialization settings. For this reason it is common to train in parallel different neural networks with different parameter initializations and eventually choose the one which has the best performance of the development set. This procedure is known as *random restarts*. This implies that since different models have been trained, the final prediction on a specific task can be now based on the *model ensembles*, e.g. using the rule of majority vote.

Another issue with very deep models is the number hidden layers and parameters, which causes problems when computing the gradient at the first hidden layers. In this case the backpropagation algorithm suffers from either vanishing (almost 0) or exploding (very large value) gradient.

It can also happen that layers with tanh and sigmoid activations become saturated, i.e. these neurons produce output values which are all close to 1 and so the gradient becomes very small. This has a negative effect on the network training, since the saturated neuron is not contributing to the learning algorithm. At this point one could change activation function, e.g. ReLu, which prevents neurons from “saturating”, but this introduces the problem of “dead” neurons.

Finally, training deep models has become challenging also from a time and computational point of view. In modern deep learning applications not only it is crucial to choose a fast-convergence optimization algorithm, e.g. SGD, but also it is required fine-tune hyperparameters such as learning rate and minibatch size. A common practice used in the deep learning community to speed up neural network training is to use the parameter initialization of pre-trained models (e.g. word embeddings).

3 RNN Models

Recurrent Neural Networks (RNNs) are specialized neural network architectures for processing **sequential data $\boldsymbol{x_{1:n}}$**[1]. As already mentioned in Chapter 2, RNNs are a class of ANNs which present cycles in their compuational graph.

The main peculiarity of RNNs in the context of NLP is the ability to share parameters Θ across different parts of the network. Such **parameter sharing** scheme enhances not only the flexibility of the neural network, i.e. the model can be applied to examples of different lenghts, but also the generalization ability of the model, i.e. the model provides good results also for previously unseen instances.

If we think about the variability of natural language data, it is extremely important to have a flexible model which is able to recognize a particular information regardless its absolute position in the sentence. For example, "Yesterday, it was sunny" and "It was sunny yesterday" should be understood as sentences with the same meaning regardless the position of word yesterday in the sentence.

The first goal of this chapter (see section 3.1) is to define RNNs high-level abstraction and to visualize RNNs graphical representation in both folded and unfolded versions. This will then shed the light on the most popular concrete RNN architectures, i.e. Simple RNNs (section 3.2) and Gated RNNs (section 3.3), which have been developed with the aim both to avoid vanishing gradient problems and to model long-range dependencies. Section 3.4 will examine other two RNN architectural variations, i.e. bidirectional RNNs and deep RNNs, which have become popular for solving some specific NLP taks.

The high flexibility of RNNs made it possible to exploit RNNs' output as the input for other components in a bigger model pipeline architecture. At the end of section 3.4, we will provide an example of RNNs used as Feature Extractor for a PoS Tagging task.

The structure, the examples and related figures of this chapter are mostly taken from Goldberg (2017), in order to be coherent with the notation used in

[1]Keep in mind the difference of notation between $\boldsymbol{x}_{[1]}, \dots, \boldsymbol{x}_{[n]}$ (enumerating the n elements of vector $\boldsymbol{x}$) and $\boldsymbol{x_{1:n}}$ (sequence of n vectors $\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n}$).

the previous chapters. A useful source for visualizing and understanding the vanishing gradient problem in RNNs has been Graves (2012). For the sake of clarity for Chapter 4, the notation of gated architectures has slightly changed from Goldberg (2017).

3.1 RNN Abstraction

Let $\boldsymbol{x_{1:n}}$ be a sequence of n vectors $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$, where each $\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}$. Our goal is to produce an output vector $\boldsymbol{y_n} \in \mathbb{R}^{d_{out}}$ [2] using the RNN function

$$\boldsymbol{y_n} = \text{RNN}(\boldsymbol{x_{1:n}}) \tag{3.1}$$

which means that a vector $\boldsymbol{y_i}$ is produced for each prefix $\boldsymbol{x_{1:i}}$ of sequence $\boldsymbol{x_{1:n}}$. So, the output sequence $\boldsymbol{y_{1:n}}$ can be expressed through the RNN* function

$$\begin{aligned} \boldsymbol{y_{1:n}} &= \text{RNN}^*(\boldsymbol{x_{1:n}}) \\ &= \text{RNN}(\boldsymbol{x_1}), \text{RNN}(\boldsymbol{x_{1:2}}), \ldots, \text{RNN}(\boldsymbol{x_{1:i}}), \ldots, \text{RNN}(\boldsymbol{x_{1:n}}) \\ &= \boldsymbol{y_1}, \boldsymbol{y_2}, \ldots, \boldsymbol{y_i}, \ldots, \boldsymbol{y_n} \end{aligned}$$

where $\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}, \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}$ for all $i = 1, \ldots, n$.

Note that such formulation provides RNNs with a mathematical framework for conditioning on the entire history $\boldsymbol{x_{1:i}}$ when predicting $\boldsymbol{y_i}$. This is in contrast with the traditional **Markovian** assumption, which assumes a fixed-size window of past dependence and thus does not allow flexibility, which is actually required for many NLP tasks.

Looking more carefully at equation 3.1, the RNN function operates in two steps to produce output $\boldsymbol{y_i}$:

$$\boldsymbol{s_i} = R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i}) \tag{3.2}$$

$$\boldsymbol{y_i} = O(\boldsymbol{s_i}) \tag{3.3}$$

where the RNN first applies function $R(\cdot)$ to a *state vector* $\boldsymbol{s_{i-1}}$ (containing the history $\boldsymbol{x_{1:i-1}}$) and to the new input vector $\boldsymbol{x_i}$ and returns as output a new state vector $\boldsymbol{s_i}$[3]. The second step is to apply the function $O(\cdot)$ to the previously obtained state vector $\boldsymbol{s_i}$ and finally obtain the output $\boldsymbol{y_i}$, which is then used for further prediction.

[2]The subscript n means that $\boldsymbol{y_n}$ has been computed based on the whole sequence $\boldsymbol{x_{1:n}}$.
[3]Note that it is conventional to start the recursion with an initial state vector $\boldsymbol{s_0}$.

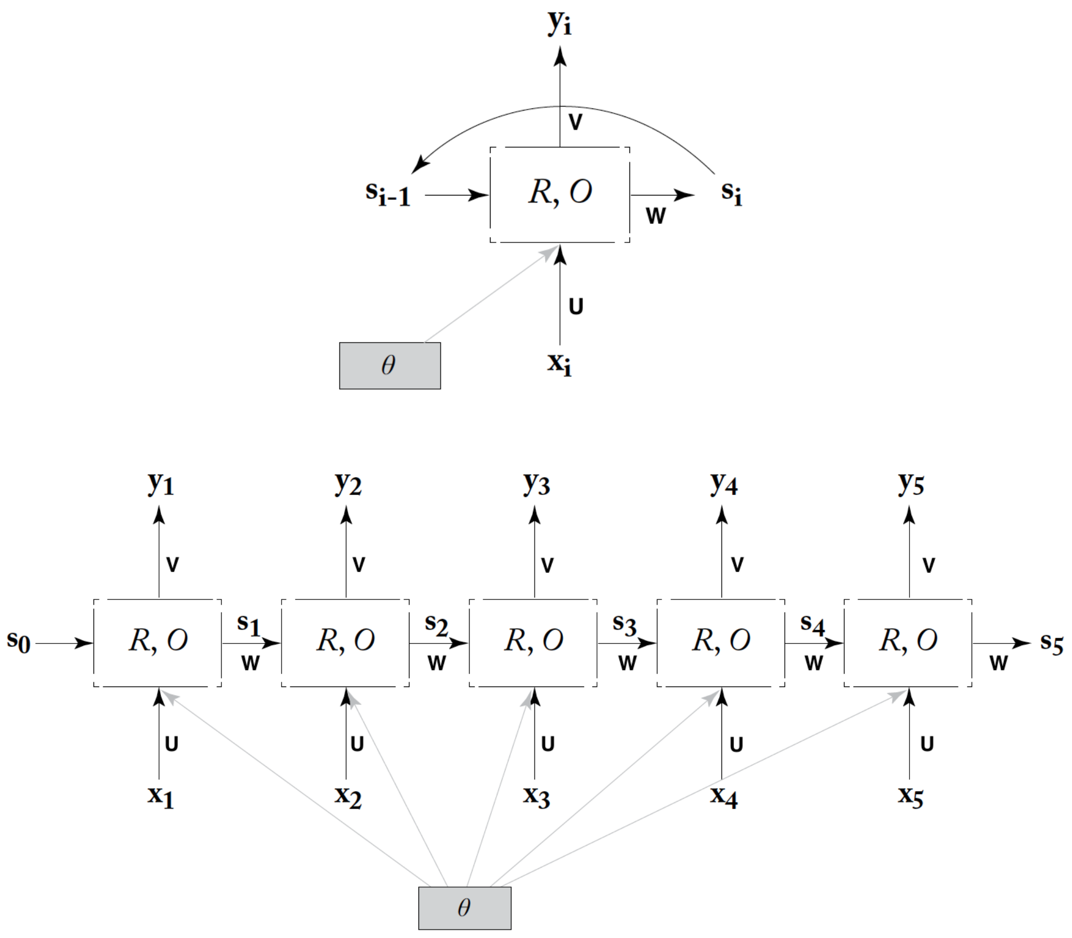


Figure 3.1: Recursive (top) and Unfolded (bottom) RNN abstraction.

Looking at equation 3.2, we can notice an interesting recursion

$$
\begin{aligned}
\boldsymbol{s_i} &= R(\boldsymbol{s_{i-1}}, \boldsymbol{x_i}) \\
&= R(\underbrace{R(\boldsymbol{s_{i-2}}, \boldsymbol{x_{i-1}})}_{\boldsymbol{s_{i-1}}}, \boldsymbol{x_i}) \\
&\vdots \\
&= R(R(\ldots(\underbrace{R(\boldsymbol{s_0}, \boldsymbol{x_1})}_{\boldsymbol{s_1}}, \boldsymbol{x_2})\ldots), \boldsymbol{x_i})
\end{aligned}
\tag{3.4}
$$

where each new state vector $\boldsymbol{s_i}$ is obtained as combination of the same function $R(\cdot)$ applied recursively to different inputs. Moreover, also the output $\boldsymbol{y_i}$ is obtained using always the same function $O(\cdot)$.

This recurrent formulation is the key part to understand how RNNs implement such *parameter sharing* mechanism at different time steps.

The *recursive formulation* of RNNs (equation 3.2) is graphically represented in

Figure 3.1 (top). On the other hand, in Figure 3.1 (bottom) we see the RNNs *unflolded formulation* (equation 3.4).

It goes without saying that network's parameters Θ are implicitely contained in the functions $R(\cdot)$ and $O(\cdot)$. As we will see in the next sections, the different types of RNNs actually depend on the choice of functions $R(\cdot)$ and $O(\cdot)$, i.e. on how we design the computations inside the RNN's node.

In a typical RNN architecture, parameters Θ and related computations can be divided into three blocks (as shown in Figure 3.1) at each network's node i:

1. **Block U (input-state)**: from the input vector $\boldsymbol{x_i}$ to the state vector $\boldsymbol{s_i}$
2. **Block W (state-state)**: from the previous state vector $\boldsymbol{s_{i-1}}$ to the next state vector $\boldsymbol{s_i}$
3. **Block V (state-output)**: from the state vector $\boldsymbol{s_i}$ to the output vector $\boldsymbol{y_i}$.

Using this notation, weight matrices in the following RNN architectures will be indicated as $\boldsymbol{U}$,$\boldsymbol{W}$ or $\boldsymbol{V}$ (except for subscripts and superscripts) according to the building block they belong to. This notation will be extremely useful for Chapter 4, where we will put our focus on making RNNs compact, i.e. operating on weight matrices with the aim to reduce the number of computations.

3.2 Simple RNN

The first concrete example of RNN architectures is the Simple RNN (S-RNN), also known as Elman or Vanilla Network (see Elman (1990)). Mathematically, S-RNN is defined as

$$\begin{aligned}
\boldsymbol{s_i} &= R_{S-RNN}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = g(\boldsymbol{a_i}) \\
\text{where } \boldsymbol{a_i} &= \boldsymbol{s_{i-1}}\boldsymbol{W} + \boldsymbol{x_i}\boldsymbol{U} + \boldsymbol{b} \qquad (3.5) \\
\\
\boldsymbol{y_i} &= O_{S-RNN}(\boldsymbol{s_i}) = \boldsymbol{s_i}
\end{aligned}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}, \boldsymbol{s_i}, \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}, \boldsymbol{U} \in \mathbb{R}^{d_{in} \times d_{out}}, \boldsymbol{W} \in \mathbb{R}^{d_{out} \times d_{out}}, \boldsymbol{b} \in \mathbb{R}^{d_{out}}$$
$$\Theta = \boldsymbol{U}, \boldsymbol{W}, \boldsymbol{b}$$

where $g(\cdot)$ is a nonlinear activation function.

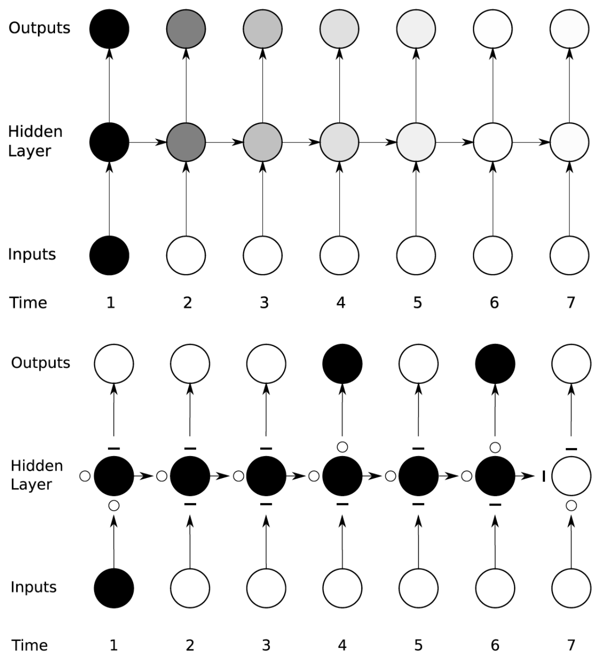


Figure 3.2: Vanishing Gradient in S-RNN (top) and LSTM (bottom).

Note first that the nonlinearity of function g allows the network to keep track of the order of the elements in $\boldsymbol{x_{1:n}}$. Note also that the output $\boldsymbol{y_i}$ is considered to be equal to the state $\boldsymbol{s_i}$, since further computations useful for solving the NLP tasks (e.g. linear transformations of MLP taking as input RNN's output) are not considered part of the RNN network and so are not explicitely mentioned in the RNN model's specification above.

3.3 Gated RNNs

We have seen in section 3.1 how RNNs are extremely flexible tools for mapping input and output sequences, since they do not make any Markovian assumption and thus take into account **contextual information** when predicting $\boldsymbol{y_i}$.

However, when using standard RNN architectures (e.g. S-RNN), such theoretical advantage does not exist in practice. What happens is that the influence of a given input $\boldsymbol{x_i}$ on the state vector $\boldsymbol{s_j}$, and so also on the network's output $\boldsymbol{y_i}$, either decays or blows up exponentially as it cycles around the network's recurrent connections. In other words, the sensitivity of node j decays as new inputs overwrite the memory state vector $\boldsymbol{s_j}$ and the network *forgets* input $\boldsymbol{x_i}$ as $j \gg i$.

This effect is graphically represented in Figure 3.2, which shows two unfolded RNNs with S-RNN (top) and LSTM (bottom) architectures. The shade of the network's nodes graphically indicates the different nodes' sensitivity $\boldsymbol{y_j}$ and $\boldsymbol{s_j}$

($j = 1, \dots 7$) to the first input $\boldsymbol{x_1}$. So, the black nodes are maximally sensitive and the white nodes are entirely insensitive. In the picture, the gate is *binary*, i.e. it is either opened ◯ or closed −. This is a simplicistic assumption, since gated RNNs discussed below have instead *differentiable* gates, which make possible for the neural network to learn them.

In the literature, this neural network's behavior is called **vanishing gradient problem** and it has been already mentioned in section 2.3 for Feed Forward architectures.

The main intuition behind gated architectures is to provide a more controlled access to the memory state vector $\boldsymbol{s_i}$, by introducing a *gating* mechanism, which is automatically *learned* by the neural network. In practice, as we will see in the following subsections, gated RNN architecture get rid of the repeated multiplication of a single matrix $\boldsymbol{W}$ in S-RNN (see equation 3.5), which is the main responsible for the vanishing (or exploding) gradient effect.

Long Short-Term Memory (LSTM)

The Long Short-Term Memory (LSTM) architecture has been historically the first attempt to solve the vanishing gradient problem and to design the idea of gating mechanism (see Hochreiter and Schmidhuber (1997)). Mathematically, the LSTM architecture is defined as

$$
\begin{aligned}
\boldsymbol{s_i} &= R_{LSTM}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = [\boldsymbol{c_i}; \boldsymbol{h_i}] \\
\text{where } \underbrace{\boldsymbol{c_i}}_{MEMORY} &= \underbrace{\boldsymbol{\Gamma_u}}_{UPDATE} \odot \underbrace{\tilde{\boldsymbol{c_i}}}_{CANDIDATE} + \underbrace{\boldsymbol{\Gamma_f}}_{FORGET} \odot \boldsymbol{c_{i-1}} \qquad (3.6) \\
&\quad \tilde{\boldsymbol{c_i}} = \tanh\left(\boldsymbol{x_i U} + \boldsymbol{h_{i-1} W} + \boldsymbol{b}\right) \\
&\quad \boldsymbol{\Gamma_u} = \sigma\left(\boldsymbol{x_i U^u} + \boldsymbol{h_{i-1} W^u} + \boldsymbol{b^u}\right) \\
&\quad \boldsymbol{\Gamma_f} = \sigma\left(\boldsymbol{x_i U^f} + \boldsymbol{h_{i-1} W^f} + \boldsymbol{b^f}\right) \\
\text{and } \boldsymbol{h_i} &= \underbrace{\boldsymbol{\Gamma_o}}_{OUTPUT} \odot \tanh\left(\boldsymbol{c_i}\right) \\
&\quad \boldsymbol{\Gamma_o} = \sigma\left(\boldsymbol{x_i U^o} + \boldsymbol{h_{i-1} W^o} + \boldsymbol{b^o}\right) \\
\boldsymbol{y_i} &= O_{LSTM}(\boldsymbol{s_i}) = \boldsymbol{h_i}
\end{aligned}
$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}, \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}, \boldsymbol{s_i} \in \mathbb{R}^{2 \cdot d_{out}}$$
$$\boldsymbol{U}, \boldsymbol{U^u}, \boldsymbol{U^f}, \boldsymbol{U^o} \in \mathbb{R}^{d_{in} \times d_{out}}, \boldsymbol{W}, \boldsymbol{W^u}, \boldsymbol{W^f}, \boldsymbol{W^o} \in \mathbb{R}^{d_{out} \times d_{out}}, \boldsymbol{b}, \boldsymbol{b^u}, \boldsymbol{b^f}, \boldsymbol{b^o} \in \mathbb{R}^{d_{out}}$$
$$\Theta = \boldsymbol{U}, \boldsymbol{U^u}, \boldsymbol{U^f}, \boldsymbol{U^o}, \boldsymbol{W}, \boldsymbol{W^u}, \boldsymbol{W^f}, \boldsymbol{W^o}, \boldsymbol{b}, \boldsymbol{b^u}, \boldsymbol{b^f}, \boldsymbol{b^o}$$

where $\odot$ indicates the Hadamard Product (element-wise product) and $\boldsymbol{\Gamma}$ indicates a gate.

Note that the state vector $\boldsymbol{s_i}$ has been splitted into two d_{out}-dimensional vectors $\boldsymbol{c_i}$ (*memory cell*) and $\boldsymbol{h_i}$ (*output cell*).

The memory cell $\boldsymbol{c_i}$ is designed to keep the information from the past $\boldsymbol{x_{1:i-1}}$. At each step, the new memory cell $\boldsymbol{c_i}$ is computed given the new input $\boldsymbol{x_i}$ as written down in equation 3.6:

- A new candidate update $\boldsymbol{\tilde{c}_i}$ is proposed.

- The update gate $\boldsymbol{\Gamma_u}$ controls how much of the candidate update $\boldsymbol{\tilde{c}_i}$ to keep.

- The forget gate $\boldsymbol{\Gamma_f}$ controls how much of the previous memory cell $\boldsymbol{c_{i-1}}$ to keep.

Finally, the output cell $\boldsymbol{h_i}$ is computed based on the on the new $\boldsymbol{c_i}$ and the output gate $\boldsymbol{\Gamma^o}$.

From a practical point of view, the first advantage of LSTM architecture is that they avoid the vanishing gradient effect by introducing a gating mechanism. When applying LTSM models for solving NLP tasks, this architecture has also the advantage to allow the neural network to be more flexible in terms of modeling succesfully **long-range dependencies** between words.

Note that there exists many variants of LSTM architecture, which slightly differ from our definition above, e.g. *peephole connection* and gate-typing.

Gated Recurrent Unit (GRU)

A simplification of the LSTM architecture is the Gated Recurrent Unit (GRU) architecture, which has been more recently introduced by Cho et al. (2014). Mathematically speaking, the GRU architecture is defined as

$$\boldsymbol{s_i} = R_{GRU}(\boldsymbol{x_i}, \boldsymbol{s_{i-1}}) = \boldsymbol{h_i}$$

$$\text{where } \boldsymbol{h_i} = \boldsymbol{c_i} = \underbrace{\boldsymbol{\Gamma_u}}_{UPDATE} \odot \underbrace{\boldsymbol{\tilde{c}_i}}_{CANDIDATE} + (1 - \boldsymbol{\Gamma_u}) \odot \boldsymbol{c_{i-1}}$$

$$\boldsymbol{\tilde{c}_i} = \tanh\left(\boldsymbol{x_i}\boldsymbol{U} + \left(\underbrace{\boldsymbol{\Gamma_r}}_{RESET} \odot \boldsymbol{h_{i-1}}\right)\boldsymbol{W} + \boldsymbol{b}\right)$$

$$\boldsymbol{\Gamma_r} = \sigma\left(\boldsymbol{x_i}\boldsymbol{U^r} + \boldsymbol{h_{i-1}}\boldsymbol{W^r} + \boldsymbol{b^r}\right)$$

$$\boldsymbol{\Gamma_u} = \sigma\left(\boldsymbol{x_i}\boldsymbol{U^u} + \boldsymbol{h_{i-1}}\boldsymbol{W^u} + \boldsymbol{b^u}\right)$$

$$\boldsymbol{y_i} = O_{GRU}(\boldsymbol{s_i}) = \boldsymbol{h_i}$$

$$\boldsymbol{x_i} \in \mathbb{R}^{d_{in}}, \boldsymbol{y_i} \in \mathbb{R}^{d_{out}}, \boldsymbol{s_i} \in \mathbb{R}^{d_{out}}$$

$$\boldsymbol{U}, \boldsymbol{U^u}, \boldsymbol{U^r} \in \mathbb{R}^{d_{in} \times d_{out}}, \boldsymbol{W}, \boldsymbol{W^u}, \boldsymbol{W^r} \in \mathbb{R}^{d_{out} \times d_{out}}, \boldsymbol{b}, \boldsymbol{b^u}, \boldsymbol{b^r} \in \mathbb{R}^{d_{out}}$$

$$\Theta = \boldsymbol{U}, \boldsymbol{U^u}, \boldsymbol{U^r}, \boldsymbol{W}, \boldsymbol{W^u}, \boldsymbol{W^r}, \boldsymbol{b}, \boldsymbol{b^u}, \boldsymbol{b^r}$$

The first simplification in GRU regards the absence of a separate memory cell $\boldsymbol{c_i}$ (in GRU $\boldsymbol{c_i} = \boldsymbol{h_i}$) for storing past information $\boldsymbol{x_{1:i-1}}$, i.e. the output is only one state vector $\boldsymbol{s_i} = \boldsymbol{h_i} = \boldsymbol{c_i}$ as in S-RNN.

Secondly, GRU architecture reduces the number of gates: the update of state $\boldsymbol{s_i}$ is now entirely governed by gate $\boldsymbol{\Gamma_u}$, i.e. there is not a forget gate $\boldsymbol{\Gamma_f}$ for the previous state $\boldsymbol{s_{i-1}}$.

Nevertheless, the fully-gated variant introduces a new reset gate $\boldsymbol{\Gamma_r}$, that controls which parts of the previous state $\boldsymbol{s_{i-1}}$ will be used to compute the next candidate state $\boldsymbol{\tilde{c}_i}$. This introduces an additional nonlinear effect in the relationship between past and future state.

Finally, if we compare the two presented gated architectures from the point of view of number of parameters Θ, we find out that the LSTM architecture has

$$4 \times \left(\underbrace{d_{in} \cdot d_{out}}_{U} + \underbrace{d_{out}^2}_{W} + \underbrace{d_{out}}_{b}\right)$$

parameters, while the (fully-gated) GRU architecture has

$$3 \times \left(\underbrace{d_{in} \cdot d_{out}}_{U} + \underbrace{d_{out}^2}_{W} + \underbrace{d_{out}}_{b} \right)$$

parameters. In other words, the GRU architecture has 25% less parameters than the LSTM architecture, which makes GRU much more preferable from a computational point of view.

3.4 Architectural Variations

Up to this point, our focus has been put only on modifying the computations inside the S-RNN node, but keeping the network structure as the one in Figure 3.1. However, in RNN literature there exists many *architectural variations*, which propose alternative connections between input, state and output vectors.

In this section we explore *deep-RNN* (dRNN) and *bidirectional-RNN* (biRNN) architectures. We will also provide a concrete example of a character-level biRNNs used for solving a PoS-Tagging task, which has been already explored in section 1.2.

Deep RNNs

The analogous of MLP for feedforward architectures is deep-RNN for recurrent architectures.

The idea of dRNNs is to stack k RNNs ($\text{RNN}_1, \ldots, \text{RNN}_k$) such that the input of RNN_1 is $\boldsymbol{x_{1:n}}$, while the input of RNN_j ($j = 2, \ldots, k$) is the output of the previous RNN, i.e. $\boldsymbol{y^{j-1}_{1:n}}$ [4]. The final output is then $\boldsymbol{y^{k}_{1:n}}$.

Many architectural design of dRNNs exist in the literature and usually are task-specific. For example, in the end of this section we will see an example of a deep bidirectional RNN (deep-biRNN), which is used in the intermediate step of a pipeline for solving PoS tagging task.

The main advantage of using a dRNN is the **representation power**: we can think of the lower layers in the hierarchy shown in Figure 3.3 as playing a role in transforming the raw input $\boldsymbol{x_{1:n}}$ into a representation that is more appropriate, at the higher levels of the hidden state.

[4] The superscript indicates the j-th layer.

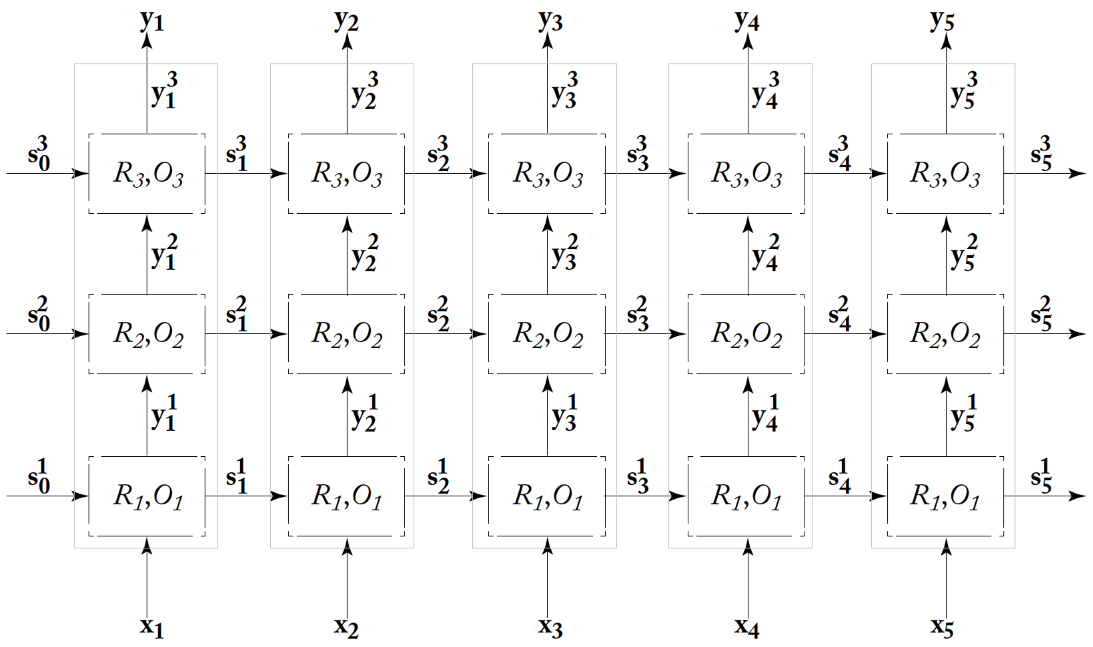

y_1 y_2 y_3 y_4 y_5
y_1^3 y_2^3 y_3^3 y_4^3 y_5^3
s_0^3 s_1^3 s_2^3 s_3^3 s_4^3 s_5^3
R_3,O_3
y_1^2 y_2^2 y_3^2 y_4^2 y_5^2
s_0^2 s_1^2 s_2^2 s_3^2 s_4^2 s_5^2
R_2,O_2
y_1^1 y_2^1 y_3^1 y_4^1 y_5^1
s_0^1 s_1^1 s_2^1 s_3^1 s_4^1 s_5^1
R_1,O_1
x_1 x_2 x_3 x_4 x_5

Figure 3.3: Deep-RNN with 3 layers.

Bidirectional RNNs

All RNN architectures we have considered so far process past information $\boldsymbol{x_{1:i-1}}$ and present input $\boldsymbol{x_i}$ to predict output $\boldsymbol{y_i}$. However, there are many sequence labelling tasks where it may be useful to access both past and future information. For example, in the handwriting recognition task, the correct classification of the current written letter may depend not only on the previous letters, but also on those coming after it.

Having said that, the main idea behind biRNNs is to present each training example $\boldsymbol{x_i}$ forwards and backwards to two separate recurrent hidden layers, both of which are connected to the same output layer. This structure provides the output layer with **complete past and future context** for every point in the input sequence, without displacing the inputs from the relevant targets.

In practice, for every input position i, biRNNs keep in memory two separate states $\boldsymbol{s_i^f}$ (*forward state*) and $\boldsymbol{s_i^b}$ (*backward state*), which are generated by two separate RNNs, i.e. RNN^f and RNN^b respectively. Note that the two RNNs run forward and backward indipendently of each other.

Similarly to the high-level definition of RNNs (see equation 3.1), the output

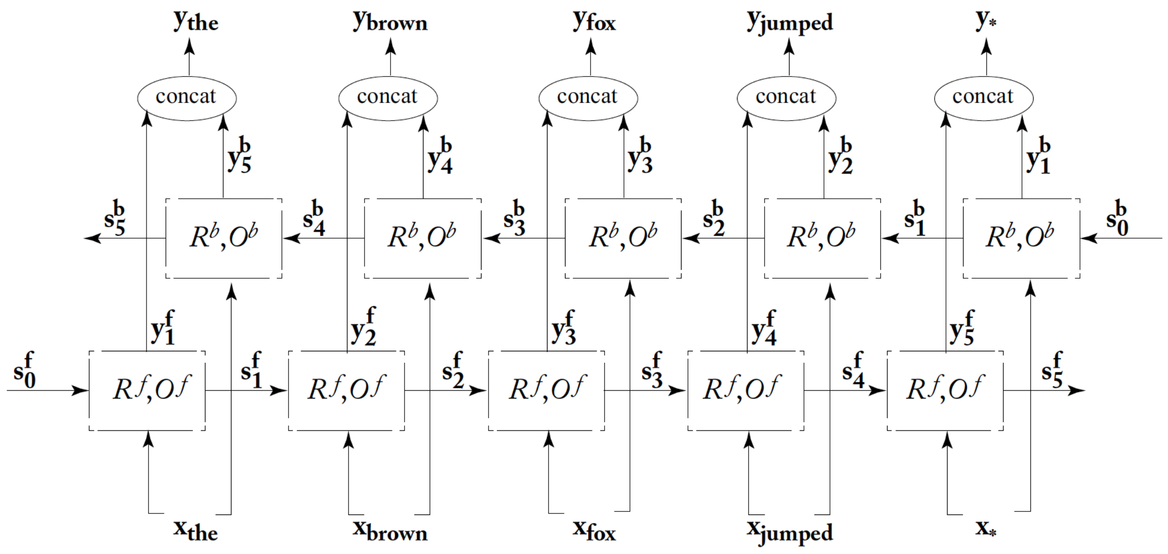


Figure 3.4: Bidirectional RNN for $s =$ "THE BROWN FOX JUMPED.".

$\boldsymbol{y_i} \in \mathbb{R}^{2\times d_{out}}$ is obtained in biRNNs as

$$\begin{aligned} y_i &= \text{biRNN}\left(\boldsymbol{x_{1:n}}, i\right) \\ &= \left[\text{RNN}^f\left(\boldsymbol{x_{1:i}}\right); \text{RNN}^b\left(\boldsymbol{x_{n:i}}\right)\right] \\ &= \left[\boldsymbol{y_i^f}; \boldsymbol{y_i^b}\right] \\ &= \left[O^f\left(\boldsymbol{s_i^f}\right); O^b\left(\boldsymbol{s_i^b}\right)\right] \end{aligned} \tag{3.7}$$

where the output $\boldsymbol{y_i}$ is the concatenation of the vectors $\boldsymbol{y_i^f}$ and $\boldsymbol{y_i^b}$, which are obtained from the output functions O^f and O^b of the two RNNs.

The entire sequence $\boldsymbol{y_{1:n}}$ is mathematically defined using the biRNN* function as

$$\begin{aligned} \boldsymbol{y_{1:n}} &= \text{biRNN}^*\left(\boldsymbol{x_{1:n}}\right) \\ &= \text{biRNN}\left(\boldsymbol{x_{1:n}}, 1\right), \ldots, \text{biRNN}\left(\boldsymbol{x_{1:n}}, n\right) \\ &= [\boldsymbol{y_1^f}, \boldsymbol{y_1^b}], \ldots, [\boldsymbol{y_n^f}, \boldsymbol{y_n^b}] \\ &= \boldsymbol{y_1}, \ldots, \boldsymbol{y_n} \end{aligned}$$

which is graphically represented in Figure 3.4.

Character-level biRNN for PoS Tagging

This example is taken from Goldberg (2017), but it is very useful to extend the example on PoS tagging (see section 1.2) where we discussed about the Feature

Design and Feature Embedding stages.

Here we introduce a *character-level* model, which uses RNNs used as automatic feature extractors and thus replaces the manually-designed features (e.g. suffixes, prefixes, capitalization) required by word-level models.

We assume that each sentence s is a sequence of $w_{1:n}$ words and each word w_i $(i = 1, \dots, n)$ is made of N characters $c_1, \dots, c_N$.

Feature Extraction and Embedding: from Characters to Word Our first goal is to map each character c_j $(j = 1, \dots, N)$ of word w_i to a corresponding *character embedding vector* $\boldsymbol{c_j}$, which is then given as input to a biRNN. The output of the biRNN is then contatenated to the word embedding vector $\boldsymbol{v_i}$ (see section 1.2) and the result is the *feature vector*

$$\begin{aligned}
\boldsymbol{x_i} = \phi(s, i) = \phi(w_{1:n}, i) &= [\boldsymbol{v_i}; \text{biRNN}(\boldsymbol{c_{1:N}})] \\
&= \left[\boldsymbol{E_{[w_i]}}; \text{RNN}^f(\boldsymbol{c_{1:N}}); \text{RNN}^b(\boldsymbol{c_{N:1}})\right]
\end{aligned}$$

where $\boldsymbol{E_{[w_i]}}$ is the row of the embedding matrix $\boldsymbol{E}$ corresponding to word w_i.

Note also that here we use a forward (RNN^f) and backward (RNN^b) model each reading the *entire sequence* $\boldsymbol{c_{1:N}}$ in different directions. Plank et al. (2016) call *sequence biRNN* this type of biRNN, compared to the *context biRNN*, which has been defined in equation 3.7.

The main advantage of using bidirectional RNN in this problem setting is the *automatic* (learnable) and *granular* (character-level) feature extraction process, which was manual and subjective as defined in section 1.2.

Here the forward-running RNN focuses on detecting suffixes, while the backward-running RNN focuses on prefixes. The biRNN model overall is able to capture also capitalization, hypens and other word features.

Contextual Information Extraction: from Words to Sentence At this point the feature vector $\boldsymbol{x_i}$ contains information at word-level, but lacks contextual information about word w_i in sentence s.

For this reason, an intermediate step is to feed the feature vector $\boldsymbol{x_i}$ into a biRNN and obtain a *context vector*

$$\boldsymbol{y_i} = \text{biRNN}(\boldsymbol{x_{1:n}}, i)$$

for each word w_i. Note that Goldberg (2017) propose a deep-biRNN architecture with 3 hidden layers. The mathematical details are not provided by the author.

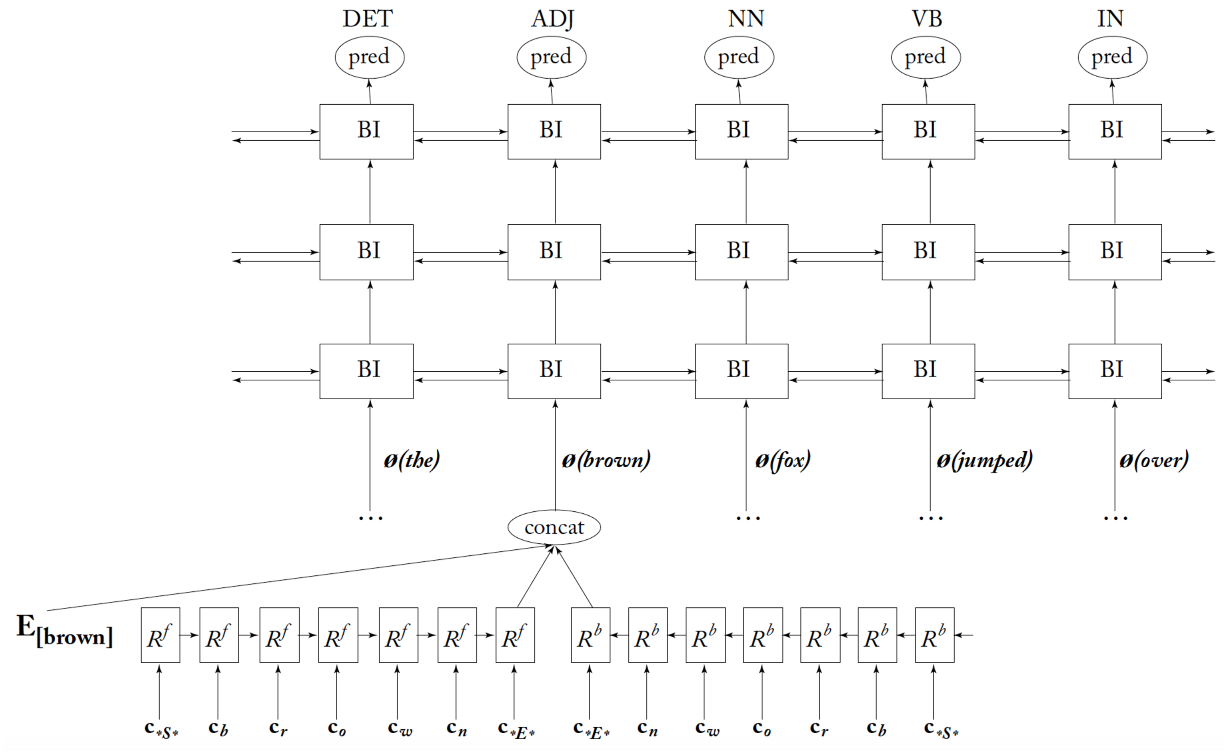

DET
ADJ
NN
VB
IN
pred
BI
ø(the)
ø(brown)
ø(fox)
ø(jumped)
ø(over)
concat
E[brown]
R^f
R^b
c_{*S*} c_b c_r c_o c_w c_n c_{*E*}
c_{*E*} c_n c_w c_o c_r c_b c_{*S*}

Figure 3.5: Example of an RNN Pipeline Architecture for PoS Tagging.

Tag Prediction The next step is to use the new feature representation $\boldsymbol{y_i}$ for word w_i as input for a predicting network (MLP [5] followed by a softmax layer). The final output is the probability that word w_i is associated with tag k [6]

$$\mathbf{P}\left(t_i = k | w_1, \dots, w_n\right) = \text{softmax}\left(\text{MLP}\left(\boldsymbol{y_i}\right)\right)_{[k]} .$$

The whole pipeline is graphically represented in Figure 3.5, starting from character embedding vectors $\boldsymbol{c_{1:N}}$, passing through feature vectors $\boldsymbol{x_{1:n}}$ and context vectors $\boldsymbol{y_{1:n}}$, ending with the tag prediction t_i for word w_i.

Another possible way to predict the tag for word w_i is to use the information on how MLP classified either previous k words $w_{i-k:i-1}$ or *all* previous words $w_{1:i-1}$.

[5]The number of hidden layers is not specified by the author.
[6]The Universal Treebank Project contains 17 tags, i.e. $k = 1, \dots, 17$.

4 Compact RNNs

When considering performance just as a matter of prediction accuracy, it goes without saying that deep neural networks have achieved outstanding results in a wide range of machine learning tasks, including for example object classification Krizhevsky et al. (2012) and speech recognition (Hinton et al. (2012)). However, it should be noted that the architectures proposed in these and similar works rely not only on deep networks with millions or even billions of parameters, but also on the availability of GPUs with very high computation capability. This might be not the case when deploying deep learning systems on portable devices with limited resources, for example in terms of memory, CPU, energy and bandwidth.

For these reasons, in recent years both academia and industry have recognized the serious drawbacks of applying *traditional* deep neural networks in emerging fields, such as Augmented and Virtual Reality (AR/VR), Internet of Things (IoT) and Smart Wearable Devices. This has started a new research field named **model compression** by Bucilua et al. (2006), which has proposed in the recent years solutions coming from many disciplines, including but not limited to machine learning, optimization, computer architecture, data compression, indexing, and hardware design.

In particular, two research trends have emerged in the model compression literature.

The first trend originates from the field of *Continual Learning*, i.e. "the ability to continually learn over time by accommodating new knowledge while retaining previously learned experiences" Parisi et al. (2019). Without going into the details of this new field, such (desired) ability implicitly makes training and updating an iterative process and thus pose the question of how an ongoing series of updates can be performed robustly and efficiently on mobile devices with limited hardware resources and power budgets. This is where model compression techniques can help to design neural network models which are compact *a priori*, i.e. during *training* phase.

The second research trend is motivated by *Real-Time Deep Learning* applications, e.g. pedestrian detection in an autonomous vehicle (Lane and Georgiev

(2015)). The goal here is for example *fast inference*, i.e. minimize the network's end-to-end response time (latency) required to predict an output $\hat{y}$, e.g. the probability that the self-driving car has encountered a pedestrian, given a new input x, e.g. a real-time image of a pedestrian. In other words, these model compression techniques are focused on making compact *a posteriori* very large neural network models, which have been trained for long time on powerful clusters and have a huge number of parameters.

In this chapter we will present in detail compression methods belonging to both research trends, but practical experiments will be conducted using only the compression methods belonging to the *a posteriori* approach.

The chapter is organized as follows. Section 4.1 gives an overview of the main methods for neural networks compression, including Matrix Factorization, Parameter Pruning, Parameter Sharing and Quantization. For each compression scheme, we provide a rigorous definition of the method, together with the current state-of-the-art results and applications in the deep learning community, with a focus on RNN architecture and NLP tasks.

Section 4.2 contains the empirical results of the experiment of PoS Tagging conducted on the Universal Dependencies dataset using a 3-layer LSTM architecture. The first part of the section will introduce the dataset and the data preprocessing step. After that, we will give a formal definition of PoS Tagging task and define in details the proposed LSTM architecture. Finally, the performance of the traditional LSTM model will be compared with the compact LSTM model, together with a discussion on the trade-off between accuracy and model size.

4.1 Neural Network Compression Methods

Neural network compression methods and approaches are somehow all motivated by the consideration that there is a large amount of **redundancy** in the parameters of a neural network. One of the first works in this research field is by Bucilua et al. (2006), who showed that the output distribution learned by a larger neural network can be approximated by a neural network with fewer parameters by training the smaller network to directly predict the outputs of the larger network. More recently, Hinton et al. (2015) proposed a similar approach named *knowledge distillation*.

There have been many solutions proposed to compress such large, over-parameterized neural networks including Matrix Factorization, Parameter Pruning, Parameter

Sharing and Quantization. In the literature, most of these approaches have been applied to Feed-forward Neural Networks and Convolutional Neural Networks, while only a small attention has been given to compressing LSTM architectures and even less in NLP tasks.

For the sake of simplicity, in this chapter compression methods will focus on matrix $\boldsymbol{W}$ of size $m \times n$. The goal of these compression schemes is to reduce the number of parameters of $\boldsymbol{W}$, namely mn.

Matrix Factorization

In a matrix-factorization scheme, we assume that matrix $\boldsymbol{W}$ has rank r. The *rank factorization* result from linear algebra guarantees that there exists a (non-unique) factorization $\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V}$, where $\boldsymbol{U}$ is a full-rank matrix of size $m \times r$ and $\boldsymbol{V}$ is a full-rank matrix of size $r \times n$. For example, if $\boldsymbol{W}$ is a 6×4 matrix, the rank factorization with $r = 2$ is

$$\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V} = \underbrace{\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \\ u_{31} & u_{32} \\ u_{41} & u_{42} \\ u_{51} & u_{52} \\ u_{61} & u_{62} \end{bmatrix}}_{6\times 2} \times \underbrace{\begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \end{bmatrix}}_{2\times 4}. \tag{4.1}$$

Note that the number of parameters of $\boldsymbol{W}$ will be reduced by a fraction p if

$$\begin{aligned} mr + rn &< pmn \\ r &< p\frac{mn}{m+n}. \end{aligned}$$

A priori matrix factorization The first strategy in matrix factorization is to learn $\boldsymbol{U}$ and $\boldsymbol{V}$ during training, i.e. *a priori.* From the point of view of the performances, this approach firstly reduces the number of network's parameters and consequently speeds up the training phase.

This approach was applied to a 5-layer DNN architecture in acoustic and language modeling by Sainath et al. (2013). The motivation was the extremely slow training phase, which was due to a large number of parameters, especially in the final layer matrix, which had $m = 1024$ and $n = 2220$. Authors successfully applied matrix factorization to the matrix of the final layer and found that the number of parameters and also training time could be reduced by a fraction

p =30-50% with no signficant loss in accuracy, by setting $r = 128$.

The same approach has been applied by Lu et al. (2016) to multi-layer S-RNN and LSTM architecture for a speech recognition task. The factorization regarded input-state matrices $\mathbf{U}$ and state-state matrices $\mathbf{W}$.

In the one-layer S-RNN model, equation 3.5 becomes

$$\boldsymbol{h_i} = \boldsymbol{s_{i-1}} \boldsymbol{W_a} \boldsymbol{W_b} + \boldsymbol{x_i} \boldsymbol{U_a} \boldsymbol{U_b} + \boldsymbol{b}$$

and in the one-layer LSTM case, equation 3.6 is now

$$\begin{aligned}
\boldsymbol{\tilde{c}_i} &= \tanh\left(\boldsymbol{x_i}\, \underbrace{\boldsymbol{U_a U_b}} + \boldsymbol{h_{i-1}}\, \underbrace{\boldsymbol{W_a W_b}} + \boldsymbol{b}\right) \\
\boldsymbol{\Gamma_u} &= \sigma\left(\boldsymbol{x_i}\, \underbrace{\boldsymbol{U^u_b U^u_b}} + \boldsymbol{h_{i-1}}\, \underbrace{\boldsymbol{W^u_a W^u_b}} + \boldsymbol{b^u}\right) \\
\boldsymbol{\Gamma_f} &= \sigma\left(\boldsymbol{x_i}\, \underbrace{\boldsymbol{U^f_a U^f_b}} + \boldsymbol{h_{i-1}}\, \underbrace{\boldsymbol{W^f_a W^f_b}} + \boldsymbol{b^f}\right) \\
\boldsymbol{\Gamma_o} &= \sigma\left(\boldsymbol{x_i}\, \underbrace{\boldsymbol{U^o_a U^o_b}} + \boldsymbol{h_{i-1}}\, \underbrace{\boldsymbol{W^o_a W^o_b}} + \boldsymbol{b^o}\right)
\end{aligned}$$

where $\boldsymbol{W_a} \in \mathbb{R}^{d_{out} \times r}, \boldsymbol{W_b} \in \mathbb{R}^{r \times d_{out}}$ and $\boldsymbol{U_a} \in \mathbb{R}^{d_{in} \times r}, \boldsymbol{U_b} \in \mathbb{R}^{r \times d_{out}}$ for all gates.

The hyperparameter r controls the level of compression, e.g. authors report that when $r = 5$, the number of parameters is reduced by $p = 40\%$. Researchers found out that the best compression was obtained by using matrix factorization (*moderate* compression) in the top-layers and structured matrices[1] (*aggressive* compression) in the bottom layers.

Authors show that the number of parameters in the final LSTM model was reduced by $p = 75\%$ with a slight decrease in the prediction performances. Moreover, they find that compressing input-state matrices $\mathbf{U}$ or state-state matrices $\mathbf{W}$ in gates $\boldsymbol{\Gamma_u}$, $\boldsymbol{\Gamma_f}$ or $\boldsymbol{\Gamma_o}$ does not have a significant impact on the prediction performance. However, compressing the candidate $\boldsymbol{\tilde{c}_i}$ has a negative impact on the prediction performance.

In a multi-layer RNN architecture, Sak et al. (2014) proposed a projection model where low-rank matrices are shared across layers of the recurrent architecture.

An extension of the matrix factorization scheme in 4.1 is Tensor-Train decomposition, which has been applied by Grachev et al. (2019) for Language Modeling task. We refer the reader to Grachev et al. (2019) for more details.

[1]See Paragraph 4.1

A posteriori matrix factorization The second approach is to apply low-rank matrix factorization as a *post-processing* compression method. In other words, we are given the full matrix $\boldsymbol{W}$ and we would like to compute $\boldsymbol{U}$ and $\boldsymbol{V}$ by solving the following optimization problem

$$\min_{\boldsymbol{U},\boldsymbol{V}} \| \boldsymbol{W} - \boldsymbol{U}\boldsymbol{V} \|_F \tag{4.2}$$

where $\| \cdot \|_F$ is the Frobenius norm, which is essentially the generalization of the Euclidean norm for matrices.

There exist many constrained version of the optimization problem in 4.2. For example, assuming that $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal matrices and that $\boldsymbol{W}$ can be reconstructed as $\boldsymbol{USV}$, where $\boldsymbol{S}$ is a diagonal matrix, then we have an analytical solution to the problem

$$\min_{\boldsymbol{U},\boldsymbol{S},\boldsymbol{V}} \| \boldsymbol{W} - \boldsymbol{USV} \|_F \ \text{ s.t. } \boldsymbol{U}, \boldsymbol{V} \text{ orthogonal and } \boldsymbol{S} \text{ diagonal}$$

in terms of Singular Value Decomposition (SVD) of $\boldsymbol{W}$. In other words, the optimal values $\boldsymbol{U_r}$,$\boldsymbol{S_r}$ and $\boldsymbol{V_r}$ for $\boldsymbol{U}$,$\boldsymbol{S}$ and $\boldsymbol{V}$ are obtained by taking the top r singular values from the diagonal matrix $\boldsymbol{S}$ and the corresponding singular vectors from $\boldsymbol{U}$ and $\boldsymbol{V}$.

For example, if $\boldsymbol{W}$ is again a 6×4 matrix, its reduced form SVD is expressed as

$$\boldsymbol{W} = \boldsymbol{USV} = \underbrace{\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{21} & u_{22} & u_{23} & u_{24} \\ u_{31} & u_{32} & u_{33} & u_{34} \\ u_{41} & u_{42} & u_{43} & u_{44} \\ u_{51} & u_{52} & u_{53} & u_{54} \\ u_{61} & u_{62} & u_{63} & u_{64} \end{bmatrix}}_{6\times4} \times \underbrace{\begin{bmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & s_4 \end{bmatrix}}_{4\times4} \times \underbrace{\begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \\ v_{31} & v_{32} & v_{33} & v_{34} \\ v_{41} & v_{42} & v_{43} & v_{44} \end{bmatrix}}_{4\times4}$$

and the compact matrix $\boldsymbol{\tilde{W}_2}$, i.e. setting $r = 2$, is obtained by Truncated SVD as

$$\boldsymbol{W} \simeq \tilde{\boldsymbol{W}}_{\boldsymbol{2}} = \boldsymbol{U_2 S_2 V_2} = \underbrace{\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \\ u_{31} & u_{32} \\ u_{41} & u_{42} \\ u_{51} & u_{52} \\ u_{61} & u_{62} \end{bmatrix}}_{6\times 2} \times \underbrace{\begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix}}_{2\times 2} \times \underbrace{\begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ v_{21} & v_{22} & v_{23} & v_{24} \end{bmatrix}}_{2\times 4}.$$

The SVD reduction has been applied to DNNs by Xue et al. (2013) on two large vocabulary continuous speech recognition (LVCSR) tasks. After applying the SVD reduction (more or less aggresively) on different weight matrices in the network, authors also fine-tune the reduced model, in order to get back the lost accuracy. The final model has been reduced in size by 73% and has less than 1% relative accuracy loss.

The experimental analysis will focus on SVD reduction on the weight matrices in the LSTM architecture.

Parameter Pruning

Given a trained neural network, we can visualize the distribution of the weights using a histogram. For example, in Figure 4.11 we can clearly notice that the majority of these weights are really close to zero. In other words, these weights do not to provide a valuable contribution for producing the final neural network's output and thus can be *pruned* from the network. For example, in a feedforward architecture, pruning consists in removing the connections between neurons from one layer to another.

The main point of pruning is how to decide which weights have to be set to zero. In literature there have been proposed many approaches which can be more or less expensive from a computational point of view.

The first approach is the naive *magnitude-based* approach, which first sets a threshold q_α and then removes all the weights whose absolute value is below that threshold. The quantity q_α is usually obtained as the α-th percentile of the distribution of the absolute values of the weights in matrix $\boldsymbol{W}$, i.e. the size of the network is reduced by a fraction $p = 1 - \alpha$. Finally, the network is fine-tuned, so that the remaning parameter weights are adjusted with the new network sparse architecture.

Other popular pruning techniques are the *hessian-based* approaches (LeCun et al. (1990), Hassibi and Stork (1993)) and thus are computationally more expensive, especially for deep learning tasks. The pruning strategy consists in computing the Hessian matrix of the loss function and then derive a *saliency* value for each parameter in the network: parameters whose saliency is low are removed and finally the network is fine-tuned.

Most of the applications of pruning techniques in deep learning use the magnitude-based approach, due to the high computational cost of the hessian-based approaches.

In particular, successful application regard mainly computer vision tasks and thus involve CNNs. For example in the ImageNet task, Collins and Kohli (2014) and Han et al. (2015b) report that 75% and 89% of the parameters in AlexNet has been pruned with a small loss in accuracy, respectively.

Pruning methods have been applied to RNNs by See et al. (2016), who extended the approach of Han et al. (2015b) to an LSTM architecture for Neural Machine Translation (NMT) task, by implementing three different magnitude-based pruning schemes.

The first result by See et al. (2016) regards the amount of compression which can be applied to an LSTM network. From one hand, authors show that without fine-tuning 40% of the parameter weights can be pruned with a small loss in performance. On the other hand, if the pruning strategy also includes a retraining step, then up to 80% of the parameters can be removed from the network, with again no loss in performance.

Their second result regards the distribution of redundancy across layers in the neural network. In particular, authors find that lower layers, e.g. embedding layer, carry a lot of redundancy, while higher layers, e.g. output layer, provide a valuable contribution in terms of performance. This is result is actually very similar with the one from Lu et al. (2016), who chose to use aggressive compression for bottom layers (more need to sparsify $\boldsymbol{W}$) and moderate compression for higher layers (less need to sparsify $\boldsymbol{W}$).

Parameter Sharing

Another possibility to shrink the number of parameters in matrix $\boldsymbol{W}$ (mn) is to allow some forms of parameter-sharing mechanisms, so that the compressed matrix has k unique parameters.

Hashing The first possibility is to exploit a (cheap) hash function that uniformly randomly groups parameters w_{ij} and then maps each group to one of the k hash buckets.

Formally, each $w_{ij} = v_{h(i,j)}$, where

$$h(i,j) : \mathbb{N} \times \mathbb{N} \to \{1, \dots, k\}$$

is a predefined hashing function and so v is a k-dimensional vector.

For example, if matrix $\boldsymbol{W}$ has size 4×4, one possible random weight sharing initialization can be

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix}$$

where the colours determine which parameters w_{ij} share the same value. For example, $w_{11} = w_{22} = w_{32} = w_{43} = w_{34} = v_1$.

Note that the bucket assignment is kept during both feedforward and back-propagation.

This compression approach has been named HashedNets by Chen et al. (2015) and applied for computer vision tasks involving DNN architecture.

Structured Matrices Structured matrices represent another mathematical trick to enable parameter sharing mechanism is our neural network model. The main idea is to exploit the structure of well-known matrices in linear algebra with desirable propreties. For example, Sindhwani et al. (2015) introduced the use of *Toeplitz* matrices for achieving model compression when performing speech recognition on mobile devices.

For example, a 4×4 Toeplitz matrix is defined as

$$\begin{bmatrix} t_0 & t_{-1} & t_{-2} & t_{-3} \\ t_1 & t_0 & t_{-1} & t_{-2} \\ t_2 & t_1 & t_0 & t_{-1} \\ t_3 & t_2 & t_1 & t_0 \end{bmatrix}.$$

The first property of such matrix structure regards the *efficiency* in linear algebraic operation.

For example, consider the matrix-vector multiplication $\boldsymbol{x}\boldsymbol{W}$, where $\boldsymbol{x}$ is a n-dimensional vector and $\boldsymbol{W}$ is a $n \times n$ square matrix.

If $\boldsymbol{W}$ is a dense matrix, i.e. it has n^2 parameters, the cost for computing matrix-vector multiplication is $O(n^2)$. On the other hand, if $\boldsymbol{W}$ is a structured matrix, e.g. Toeplitz, the matrix-vector multiplication costs $O(n \log n)$.

This means that using structured matrices in our neural network model can significantly speed up both training and inference phase, since n is usually very large in deep learning.

The second property is that any Toeplitz matrix $\boldsymbol{T}$ can be *linearly* transformed into matrices of rank less than or equal to 2, by using a suitable transformation called *displacement operator*.

Since our goal is to find a trade-off between compression rate and model's performance, we would like to approximate matrix $\boldsymbol{T}$ with matrics of rank $r > 2$. For this reason, Sindhwani et al. (2015) introduced *Toeplitz-like* matrices, which are a generalization of Toeplitz matrices. The main differences is that with Toeplitz-like it is allowed to perform *nonlinear* transformations, e.g. inversion and product, on Toepliz matrices $\boldsymbol{T}$.

In their experiment on speech recognition, Lu et al. (2016) applied Toeplitz-like structured matrices on input-state matrices $\mathbf{U}$ and state-state matrices $\mathbf{W}$ in the bottom layers of a 3-layer RNN and LSTM architecture. Authors report that this approach provided a much more aggressive compression than hashing and low-rank schemes.

Quantization

Another popular approach in neural network's compression literature is quantization. Quantization methods aim to reduce the size of the network *a posteriori* by reducing the number of bits required to represent each weight w_{ij}.

From engineering point of view, each weight is represented in memory as floating-point value with a fixed precision, e.g. 64-bit. Quantization techniques pack each weight into a lower-space, e.g. 8-bit. The extreme case is the binary weight neural networks, which have a 1-bit representation of each weight.

Han et al. (2015a) propose a three stage pipeline called "deep compression". First they perform *Parameter Pruning* (with fine-tuning) on the network, in order to learn only the important connections. After that, they operate *Quantization* (from 32-bit to 5-bit representation) on the remaining weights. Finally, they apply *Huffman coding*, which is a well-known compression algorithm in information theory.

The compression rate gradually increases in the proposed compression pipeline.

Considering the average results obtained with different deep learning models using the ImageNet dataset, the compression rate is at first between 9× and 13× (pruning), then between 27× and 31× (quantization) and finally between 35×

and 49× (huffman coding).

Specifically, AlexNet was overall compressed by 35×, from 240MB to 6.9MB, without loss of accuracy; VGG-16's size has been reduced by 49×, from 552MB to 11.3MB, again with no loss of accuracy.

Metrics

In general, it is common to compare compression methods in terms of **effectiveness** and **efficiency** (see Cheng et al. (2017)).
The former is a measure of the performance of our compressed neural network model, compared to the performance of the full model. Therefore, model's performance really depends on the nature of the task at hand.
The latter is usually considered in the dimensions of *space* (memory footprint) and *time* (latency).

Space-Efficiency Space-efficiency takes into account the amount of memory footprint of the compressed model, e.g. *number of parameters*. If we assume that $|\Theta|$ is the number of the parameters in the full model M and $|\Theta^*|$ is that of the compressed model M^* , then the *compression rate* $CR\,(M, M^*)$ of M^* over M is

$$CR\,(M, M^*) = \frac{|\Theta|}{|\Theta^*|}$$

and for example $CR\,(M, M^*) = 1.5$ is usually indicated as 1.5×.

Time-Efficiency Time-efficiency deals with model's ability to perform fast computations either in training, either in inference phase. In neural network architectures, the bottleneck is usually represented by matrix-vector multiplications. As we have seen in section 4.1, Structured Matrices have desirable properties in terms of fast algebraic operations.

If we consider only inference phase, one typical metric for time-efficiency is *inference time*, i.e. the latency s required by the neural network M to perform a forward pass given a new input. Therefore, we measure the improvement in time using the *speedup rate*

$$SR\,(M, M^*) = \frac{s}{s^*}$$

which is usually computed as an average of multiple runs.

Chosen metrics In our experiment we measure compression effectiveness in terms of **accuracy** and **f1-score**, since PoS Tagging is essentially a multi-class classification problem.

Moreover, we consider only compression methods which have an impact on space-efficiency and thus we consider the **number of parameters**.

In particular, Matrix Factorization using Truncated SVD reduction (4.1) reduces the number of parameters in matrix $\boldsymbol{W}$ by a fraction

$$p_{SVD} = r\frac{m+n}{m \times n}$$

by taking only the top r singular values in matrix $\boldsymbol{S}$.

Moreover, magnitude-based Pruning (4.1) reduces the number of parameters in $\boldsymbol{W}$ by a fraction

$$p_{PRUN} = 1 - \alpha$$

by setting to zero the weights in $\boldsymbol{W}$ whose absolute value is below q_α, which is obtained as the α-th percentile of the distribution of the absolute values of the weights in $\boldsymbol{W}$.

4.2 Experiment

This aim of this section is to present the main results obtained when applying Truncated SVD and Magnitude-based Pruning compression methods on a multi-layer LSTM architecture, trained on the Universal Dependencies dataset for performing PoS Tagging.

The first part of this section will provide a description and some summary statistics on the dataset. After that, we will describe in detail the whole model's pipeline (see Figure 4.1), i.e. preprocessing sentences and tags (4.2), passing them through the network and finally ending with a prediction (4.2). In order to help the reader, the description of all steps will be combined with a proof-of-concept visual representation.

The whole experiment has been conducted using Pytorch, given its ease of use compared to other deep learning frameworks, especially when dealing with RNN architectures and NLP taks.

Dataset

The dataset used is the Universal Dependencies - English Dependency Treebank dataset, a corpus of sentences annotated using Universal Dependencies annotation. In particular we use v2 (17 Part-of-Speech Tags) with the given data splits

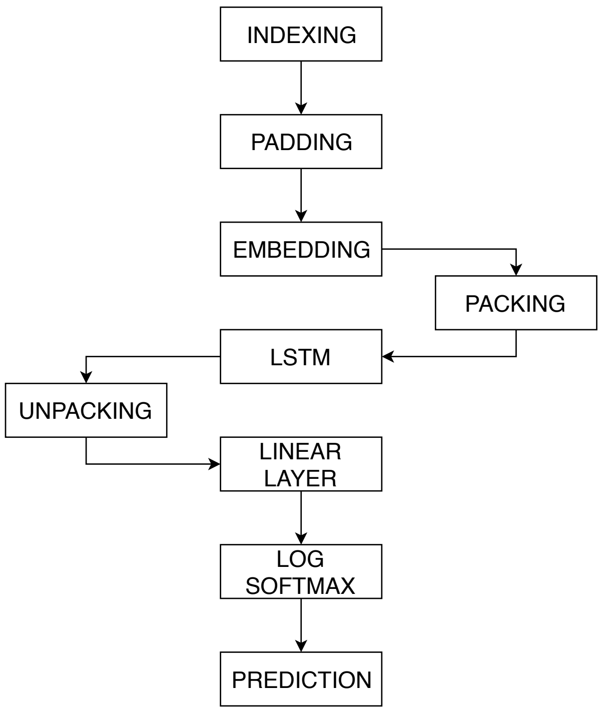


Figure 4.1: Model Pipeline.

[2].

The whole corpus comprises 254,850 words and 16,622 sentences, taken from various web media including weblogs, newsgroups, emails, reviews, and Yahoo! answers. The training set comprises 204,605 words[3] and 12,543 sentences. The dev set comprises 25,148 words and 2002 sentences. The test set comprises 25,097 words [4] and 2077 sentences.

The histogram in Figure 4.2 shows the relative frequency of the 17 PoS Tags in the corpus. The tags most present in the corpus are NOUN, PUNCT and VERB, but overall the dataset does not show any serious class unbalance.

Figure 4.3 shows the distribution of sentence lengths for the three sets in the corpus. We notice that it highly asymmetric, i.e. the majority of sentences has few words. The average sentence length is equal to 16 words for the training set and 12 for the dev and test set.

Another interesting statistics on the corpus regards the number of lemmas, i.e. the canonical form, dictionary form, or citation form of a set of words For

[2]See the Pytorch class torchnlp.datasets.ud_pos.

[3]The number of words in the training set is 204,585 according to https://github.com/UniversalDependencies/UD_English-EWT/blob/master/stats.xml

[4]The number of words in the test set is 25,096 according to https://github.com/UniversalDependencies/UD_English-EWT/blob/master/stats.xml

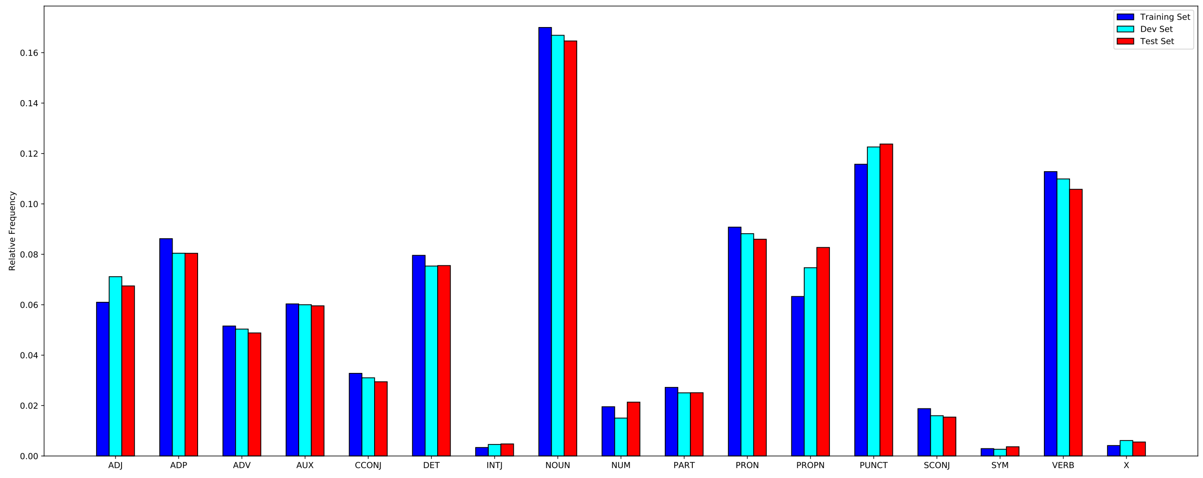


Figure 4.2: Distribution of the 17 PoS Tags in the corpus.

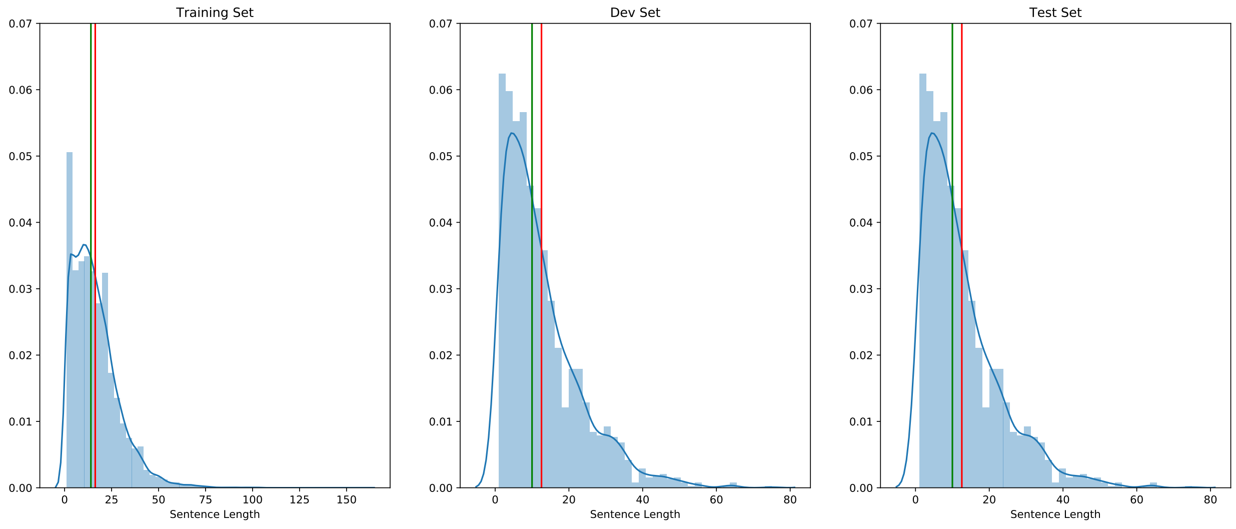


Figure 4.3: Sentence Lengths Distribution, Mean (red) and Median (green).

	# sentences	# words	# unique words	# unique lemmas
Total Corpus	16,622	254,850	30,797	24,796
Train	12,543	204,605	19,672	15,594
Dev	2,002	25,148	5,495	4,482
Test	2,077	25,097	5,630	4,720

Table 4.1: Summary Statistics on Universal Dependencies dataset.

example, in the training set, the number of (unique) words is equal to 19,672, while 15,594 is the number of (unique) lemmas[5]. This suggest that many words belong to the same lemma, e.g. RUN, RUNS, RAN and RUNNING are forms of the same lemma RUN. This means that when performing PoS Tagging on words, the relevant information in most of the cases is contained in the lemma.

Table 4.1 summarizes the statistics about the whole UD dataset and its three data splits.

Data Preprocessing

The steps of data preprocessing can be summarized as follows:

1. Create Vocabulary of Words and PoS Tags
2. Map Words and PoS Tags to Index Vectors
3. Create Mini-Batches

Create Dictionary of Words and PoS Tags

Our training set has N sentences $S_1, S_2, \ldots, S_N$ and N sequences of tags $T_1, \ldots, T_N$. Each sentence has $w_1, w_2, \ldots, w_{n_i}$ words tagged with correspondent tags $t_1, \ldots, t_{n_i}$, for $i = 1, \ldots, N$.

We would like now to create a dictionary that maps each (unique) word w^j to an index, i.e.

$$vocabulary = \left\{w^0 : 0, w^1 : 1, \ldots w^{|V|} : |V|\right\}$$

where $|V|$ represents the size of vocabulary V.

In our training set, after lower-casing the sentences, the vocabulary size is 16,654 and the vocabulary is

$$\{\texttt{<PAD>} : 0, \texttt{al} : 1, \texttt{-} : 2, \ldots, \texttt{ordeal} : 16652, \texttt{apathetic} : 16653, \texttt{<UNK>} : 16654\}.$$

[5]Lemmatization was performed using the Spacy Lemmatizer.

Note that element w^0 is set to be the *padding symbol* `<PAD>`, which will be useful for creating mini-batches of equal sizes.

The vocabulary has been created using only the training set and we assume that it is *not exhaustive* for representing also the dev and test set, i.e. not all words in the dev and test set have appeared (at least once) in the training set. For this reason, we added the special symbol `<UNK>` at the end of the vocabulary, which represents *out-of- vocabulary* (OOV) items. In our case, there are 1709 (∼6%) and 1882 (∼7%) unkown words in the dev and test set respectively.

Another dictionary is created also for tags, so that each tag is assigned an index, i.e.

$$\{\texttt{<PAD>} : 0, \texttt{ADJ} : 1, \texttt{ADP} : 2, \ldots, \texttt{VERB} : 16, \texttt{X} : 17\}$$

where the padding symbol is used here for assigning a tag to the elements which have been padded.

Map Words and PoS Tags to Index Vectors

The next step of data preprocessing pipeline is **Indexing**, i.e. map each sentence S_i and tag sequence T_i to index vectors v_i and z_i respectively.

For example, given our vocabulary, the sentence

$$\begin{aligned} S_4 &= \text{TWO OF THEM WERE BEING RUN BY 2 OFFICIALS} \\ &\quad \text{OF THE MINISTRY OF THE INTERIOR !} \end{aligned}$$

is mapped into the index vector

$$\boldsymbol{v_4} = [54, 18, 55, 56, 57, 58, 59, 60, 61, 18, 12, 62, 18, 12, 63, 64]\,.$$

The same operation is performed with tags, e.g. T_4 is mapped to index vector $\boldsymbol{z_4}$.

Create Mini-Batches

The last step is to create mini-batches, which will be finally given as input to the neural network.

As mentioned before, mini-batches are obtained by concatenating a fixed amount sentences, which may different in length, i.e. in number of words.

Our goal is to make all sequences in the mini-batch have the same length by performing **Padding**.

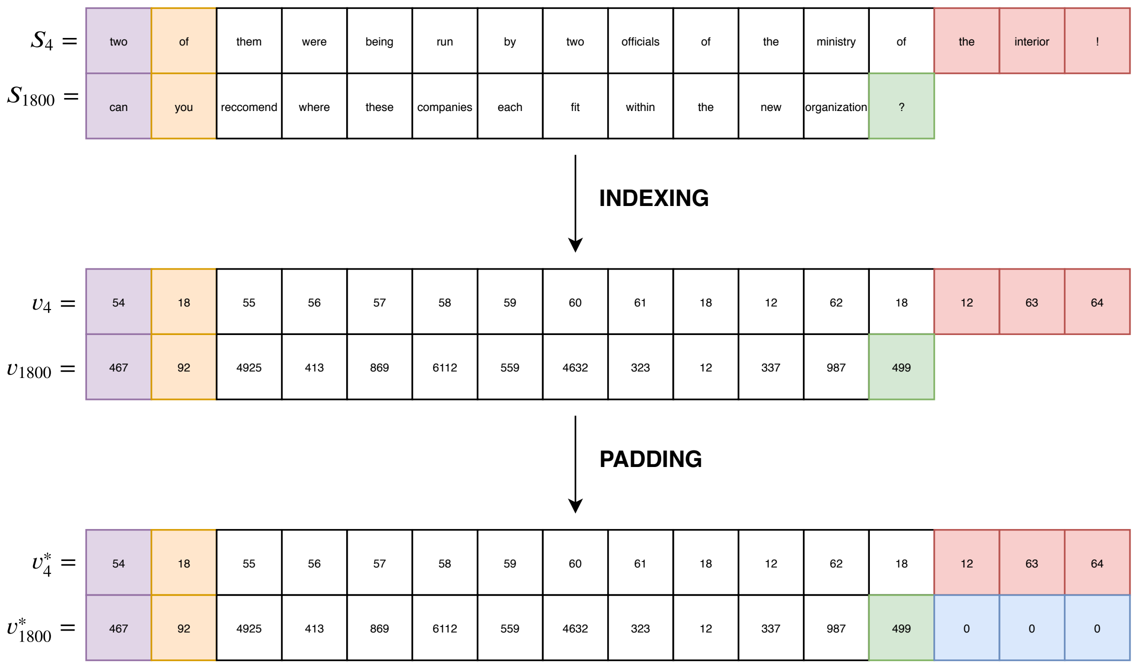


Figure 4.4: Indexing and Padding Steps.

For instance, the old index vector $\boldsymbol{v_4}$ (see above) becomes the padded index vector

$$\boldsymbol{v_4^*} = [54, 18, \dots, 64, 0, 0, \dots, 0]$$

where the length of every $\boldsymbol{v_i^*} \in batch$ is equal to $\max_{j \in batch} length(\boldsymbol{v_j})$.

The same operation is done with index vectors $\boldsymbol{z_i}$ that turn into $\boldsymbol{z_i^*}$.

After that, we have to sort the index vectors in the batch by their length, so that they can be packed later.

Figure 4.4 summarizes the data preprocessing steps taking as example two sentences in the corpus. Colors will be useful for further visualizations.

Architecture

Part-of-Speech (PoS) Tagging is a lexical Disambiguation task in NLP. The goal is to tag each word with a particular part of speech (e.g. noun, verb, adjective etc.), i.e. to predict tag t or a given input word w. In machine learning, this corresponds to a multi-class (17 classes) classification problem.

The prediction in our experiment is performed using an RNN architecture with 3 hidden layers and a Long Short-Term Memory (LSTM) recurrent unit.

Mathematically, the network is the same as the one defined in 3.4.

The network's input $\boldsymbol{x_i}$ is obtained from the padded index vector $\boldsymbol{v_i^*}$ using an **Embedding** matrix $\boldsymbol{E}$ of size $|V| \times d_{in}$, e.g. the first element of the padded

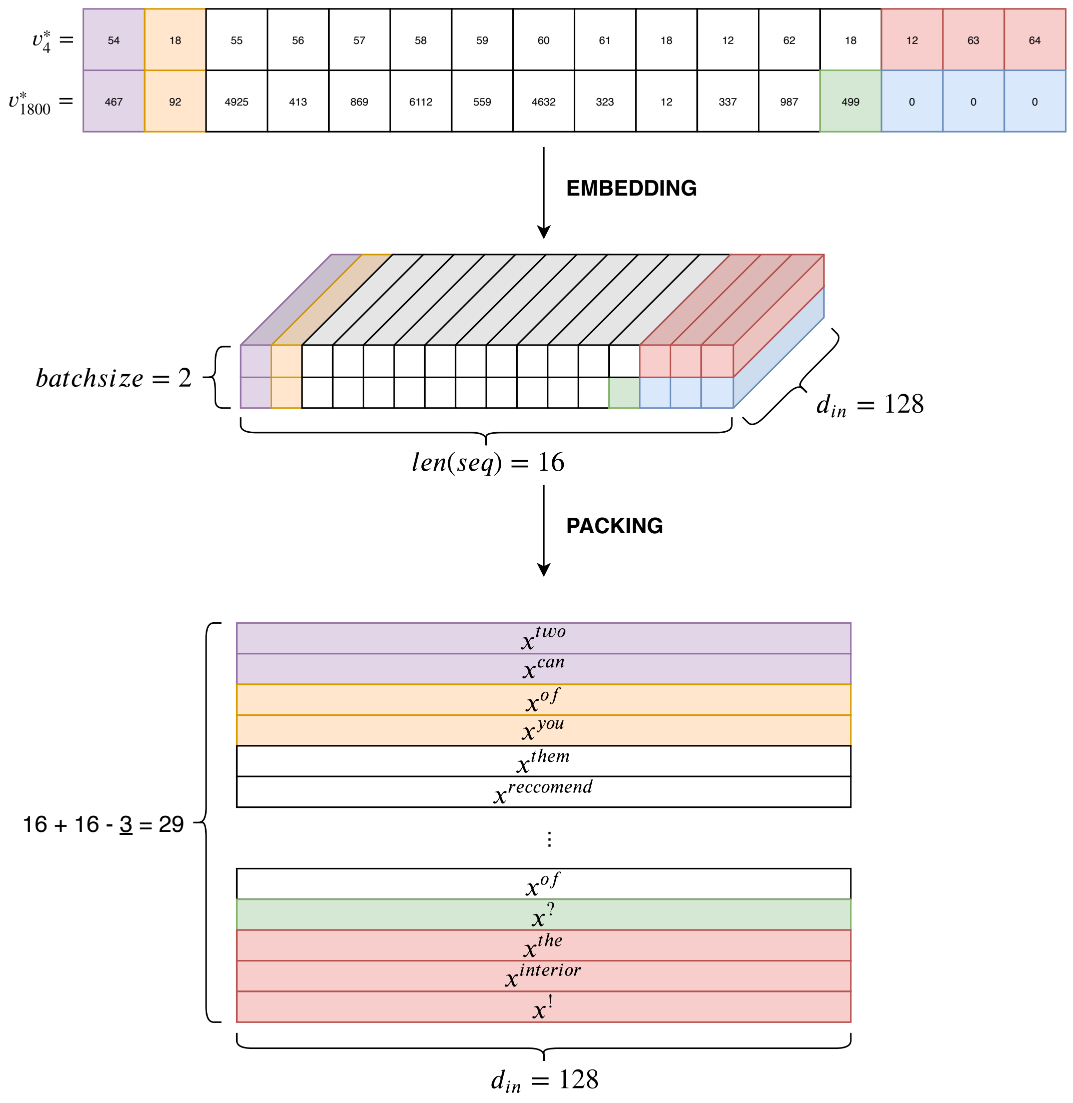


Figure 4.5: Embedding and Packing Steps.

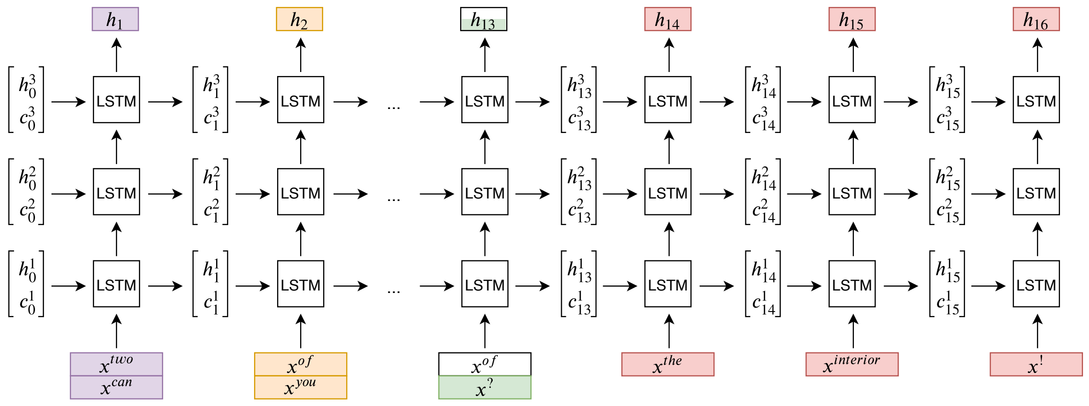


Figure 4.6: LSTM Network.

index vector $\boldsymbol{v_4^*}$ (54) is mapped into a d_{in}-dimensional vector, corresponding to the 54-th row of matrix $\boldsymbol{E}$.

After passing through the **LSTM** network by performing the **Packing** and **Unpacking** operations, the output $\boldsymbol{o_i} \in \mathbb{R}^{d_{hidd}}$ [6] is passed first to a **Linear Layer** and then to a **LogSoftmax Layer**. This transforms $\boldsymbol{o_i}$ into $\boldsymbol{\tilde{y}_i} \in \mathbb{R}^{18}$, where the sum of the elements of $\boldsymbol{\tilde{y}_i}$ sum to 1.

The final network's **prediction** $\boldsymbol{\hat{y}_i}$ is obtained by taking the tag with the highest predicted probability in $\boldsymbol{\tilde{y}_i}$.

Figures 4.5, 4.6 and 4.7 summarize the whole LSTM architecture by using the same two sentences as in Figure 4.4 and taking $d_{in} = 128$, $d_{hidd} = 100$ and batch size equal to 2.

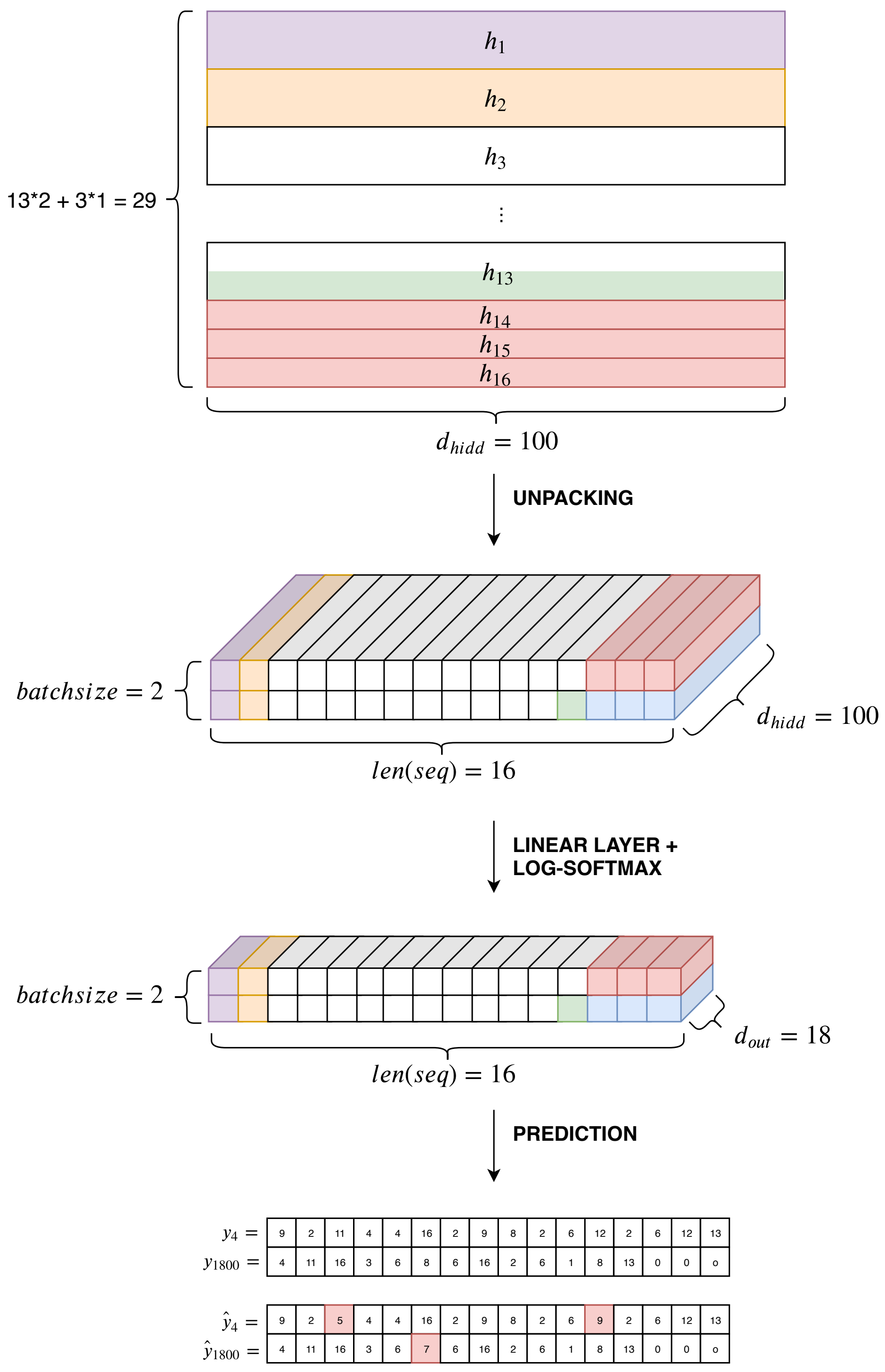
From a compression point of view, the goal is to make the neural network compact by reducing the number of parameters Θ , which consist of the embedding matrix (denoted with letter $\boldsymbol{E}$), 4×3 input-state matrices (denoted with letter $\mathbf{U}$), 4×3 state-state matrices (denoted with letter $\mathbf{W}$) and 8×3 bias vectors [7](denoted with letter $\mathbf{b}$).

Experimental Setup

The aim of the experiment is to validate the efficacy of 2 Compression Methods (Matrix Factorization and Pruning) on an RNN architecture (3-layer LSTM) with 3 different sizes (Small, Medium, Large) when performing an NLP task (PoS Tagging). We compare full and compact models in terms of two quality metrics (see

[6]Note that in Chapter 3 the output of the RNN was indicated as $\boldsymbol{y_i} \in \mathbb{R}^{d_{out}}$. Here the output of the RNN is $\boldsymbol{o_i} \in \mathbb{R}^{d_{hidd}}$, which becomes after $\boldsymbol{\tilde{y}_i} \in \mathbb{R}^{d_{out}}$, where $d_{out} = 18$.

[7]Note that the LSTM definition in 3.6 has 4 bias terms, while the Pytorch implementation has 8 bias terms.



$y_4 =$	9	2	11	4	4	16	2	9	8	2	6	12	2	6	12	13
$y_{1800} =$	4	11	16	3	6	8	6	16	2	6	1	8	13	0	0	o

$\hat{y}_4 =$	9	2	5	4	4	16	2	9	8	2	6	9	2	6	12	13
$\hat{y}_{1800} =$	4	11	16	3	6	7	6	16	2	6	1	8	13	0	0	o

Figure 4.7: Upacking, Linear Layer, LogSoftmax and Prediction steps.

section 4.1): accuracy (effectiveness) and number of parameters (space-efficiency).

Below we motivate the experimental choices.

Compression Methods Among the neural network compression approaches discussed in section 4.1, we successfully implemented a posteriori Matrix Factorization using Truncated SVD reduction (4.1) and magnitude-based Parameter Pruning (4.1). A fine-tuning step has been implemented for both compression methods.

Architecture Investigating the compressed architectures proposed in the neural network compression literature for solving various NLP tasks, we chose an LSTM architecture. The majority of works on neural network compression in NLP tasks has been performed using LSTM architectures on Speech Recognition (Lu et al. (2016) and Sak et al. (2014)), Machine Translation (See et al. (2016)) and Language Modeling (Grachev et al. (2019)). A typical starting architecture in many works on LSTM compression is the 3-layer LSTM proposed by Merity et al. (2017). Therefore, we decide to keep also this multi-layer configuration.
At best of our knowledge no work has considered recurrent architectures for solving simpler NLP tasks, e.g. PoS Tagging. Therefore, the architecture choice has been based more on the compression litterature, instead of the PoS Tagging litterature. The approaches tailored for PoS Tagging exploit more complex architectures (Heinzerling and Strube (2019)) and thus will lead us out of the scope of the present work, i.e. evaluate compression methods on RNN models.

Having said that, we should remark that our compression methods are *agnostic* to the particular architecture configuration. Therefore, our results could be potentially generalized to any feedforward and recurrent architecture.

Network Size Since our main focus is to reduce the number of parameters in the neural network, we noticed that parameter density in the network varies in a multiplicative way. In particular, given the large vocabulary size $|V|$, the largest number of parameters is present in the embedding matrix $\boldsymbol{E}$. However, when increasing the input dimension d_{in} and the hidden size d_{hidd}, an increasingly larger number of parameters is present in the input-state matrices $\mathbf{U}$ and state-state matrices $\mathbf{W}$. In other words, we should keep in mind that the compression rate p could change for different matrices in the model, according either to the layer they belong to, either to the relative number of parameters they occupy in the model.
Therefore, in our experiment we choose to evaluate compression methods on three

Model	$\boldsymbol{E}$	**W**	**U**
Full 100 × 100	87%	6%	6%
Full 150 × 150	82%	8.8%	8.8%
Full 200 × 200	77%	11%	11%

Table 4.2: Relative number of parameters with different $d_{in} \times d_{hidd}$ configurations.

identical LSTM networks with different values for d_{in} and d_{hidd}. Such approach can represent a reasonable use-case where high compression rate or aggressive compression methods are applied to large models, which hold perhaps a higher redundancy in the parameters.
In particular, in our experiment we consider Small (100×100), Medium (150×150) and Large (200 × 200) LSTM models, where $d_{in} \times d_{hidd}$ defines the size of the model. The range of values for d_{in} and d_{hidd} is chosen emperically. From one hand, models larger than 200 do not provide a significant benefit to the performance, actually leading to overfitting training data. On the other hand, models smaller than 100 do not have enough parameters, i.e. they underfit training data.
Table 4.2 summarizes the relative number of parameters occupied by the main parameter blocks in our model, i.e. the embedding matrix $\boldsymbol{E}$, the input-state matrices **U** and the state-state matrices **W**.

Optimization In order to obtain the best accuracy on the full models, we perform training using the Adagrad optimizer with batch size equal to 16 and default starting learning rate (0.01). Adagrad automatically adapts the learning rate for each parameter according to its frequency. Therefore, this optimizer is well-suited for application with sparse data, which is typical when dealing with natural language data.

The same optimization setting is used also for the fine-tuning step.

Model overfitting is prevented by applying early stopping, i.e. the training procedure is stopped when the validation accuracy starts to decrease. No additional regularization techniques, such as dropout, are applied for training the model.

Full Model Analysis

After estimating the 3 full models, we conduct an empirical analysis of them. The goal of this section is to investigate the distribution of redundancy among different sizes, layers and matrices.

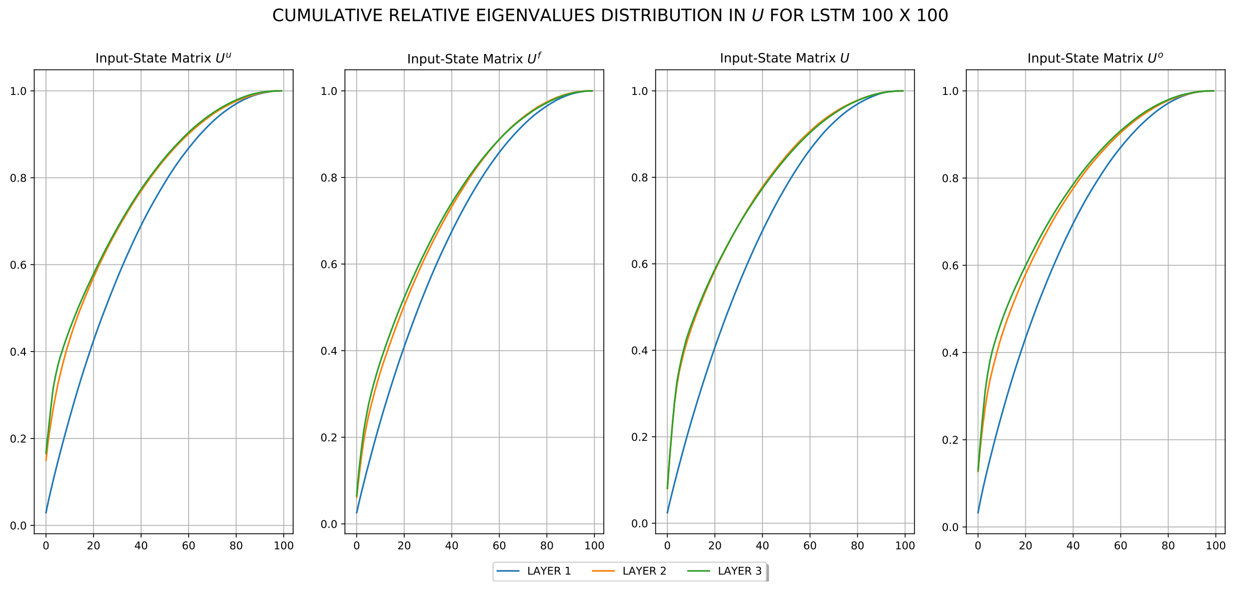


Figure 4.8: Eigenvalue Distribution for input-state matrices **U** across layers.

Singular Values Distribution

Before applying SVD reduction on our full models, it is convenient to visualize the cumulative distribution of singular values in the different matrices, which have been estimated in our models. Our goal is to spot the distribution of redundancy among matrices and layers.

Figures 4.8 and 4.9 show for model 100×100 the eigenvalues cumulative distributions of **U** and **W** respectively. We notice that for matrices **U** and **W** around $15-20\%$ of singular values contribute $40-50\%$ of total values, and around 40% of singular values contribute $70-80\%$ of total values. So, if we set those small values to 0, this will not considerably change the values of elements in these matrices.

Moreover, we notice an interesting behavior in matrices **U**: the cumulative distribution of singular values for the first layer is lower. In other words, input-state matrices in the first hidden layer store more relevant information than input-state matrices in the upper layers. This suggests that compression rate should be higher in the matrics U located in the upper layers.

Looking at Figure 4.10, the SVD is not able to capture any distribution of redundancy in the embedding matrix E.

These patterns are suprisingly identical also in the other two full models.

Weight Distribution

Another useful analysis consists in visualizing the distribution of weights in matrices $\boldsymbol{E}$, **U** and **W**.

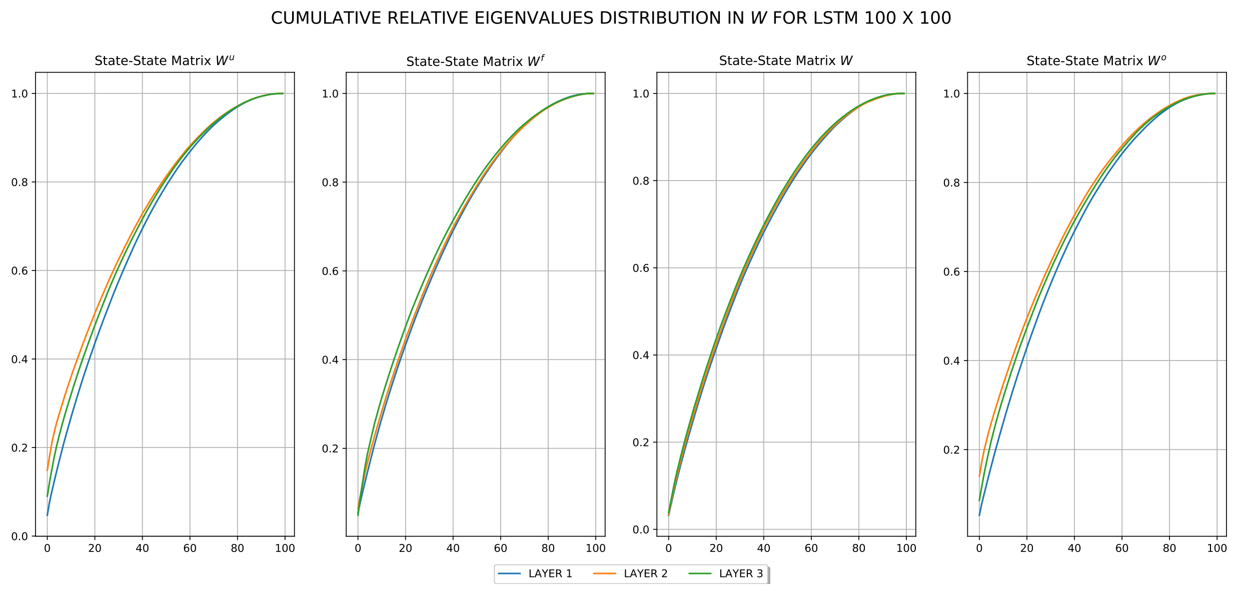


Figure 4.9: Eigenvalue Distribution for state-state matrices $\mathbf{W}$ across layers.

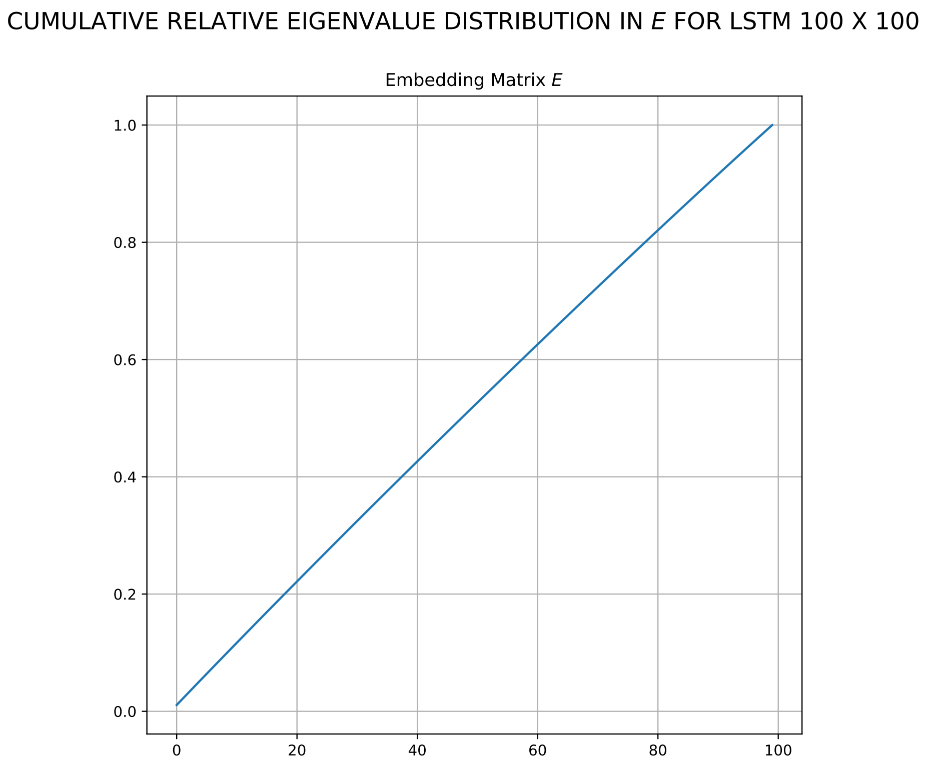


Figure 4.10: Eigenvalue Distribution for embedding matrix $\boldsymbol{E}$.

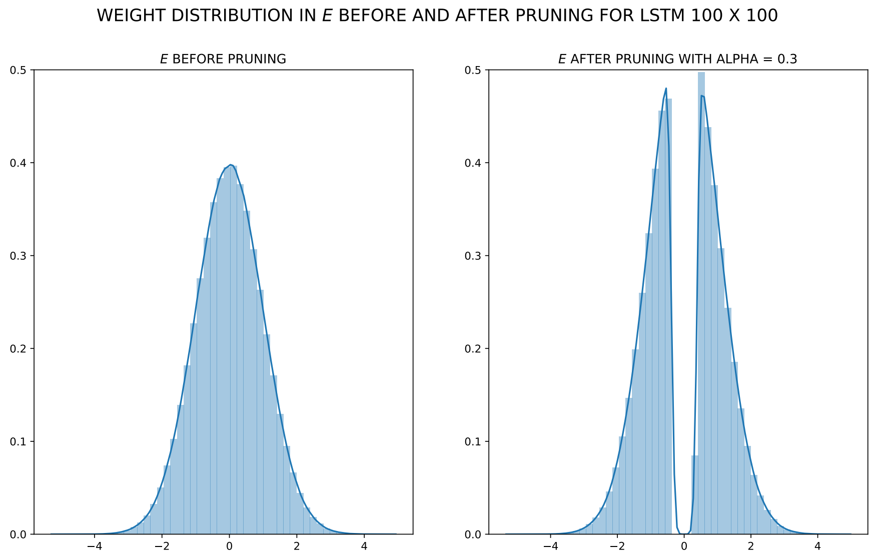


Figure 4.11: Weight Distribution for the embedding matrix $\boldsymbol{E}$ Before (Top) and After (Bottom) Pruning.

Figures 4.12 and 4.13 show for model 100 × 100 the weight distribution in $\mathbf{U}$ and $\mathbf{W}$ among different layers and gates, respectively. The first consideration is that state-state matrices $\mathbf{U}$ at the bottom layer have more weights close to zero, i.e. they are more sparse. This pattern for $\mathbf{U}$ is present also in the other full models. However, looking at the weight distribution in $\mathbf{W}$, we notice that there is no such clear pattern among layers and model sizes.

Figure 4.11 shows the distribution of the weights for the embedding matrix $\boldsymbol{E}$. Compared to the eigenvalue distribution in 4.10, here the distribution of redundancy is much clearer. As we will see in the next section, SVD Matrix Factorization will be improved by applying Pruning to the embedding matrix $\boldsymbol{E}$, instead of the Truncated SVD reduction.

The bottom parts of Figures 4.12, 4.13 and 4.11 show also the distribution of weights after pruning them with $\alpha = 0.3$. The pruning technique used here is the magnitude-based approach, which has been explained in detail in section 4.1.

Results and Discussion

In a neural network compression framework, there are two main variables involved in the process of experimental design:

- The **Method** chosen for reducing the size of the i-th parameter block[8]. This means that a neural network model can be compressed by using one

[8]Keep in mind that in our discussion the minimal level of granularity is matrix.

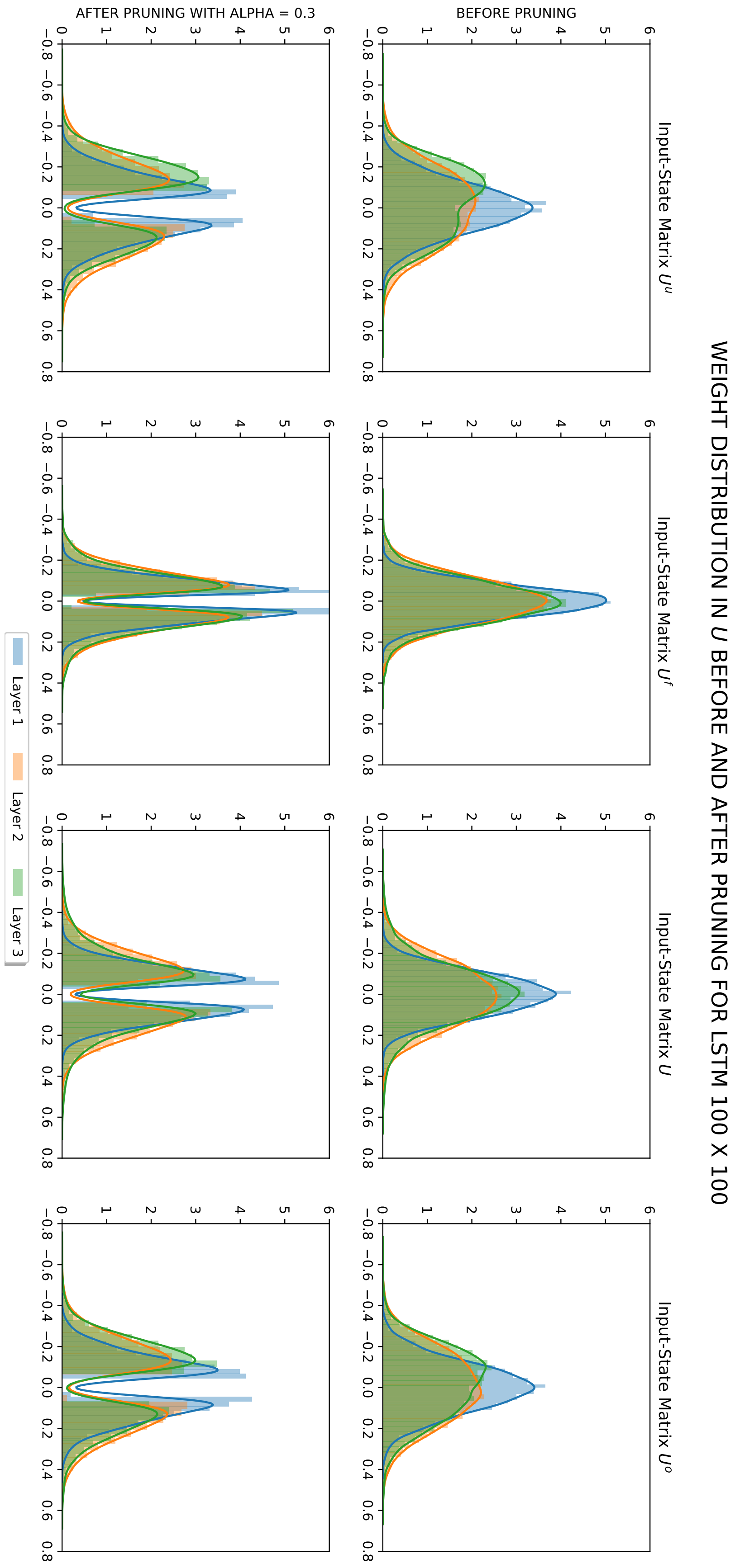


Figure 4.12: Weight Distribution for input-state matrices **U** across layers Before (Top) and After (Bottom) Pruning.

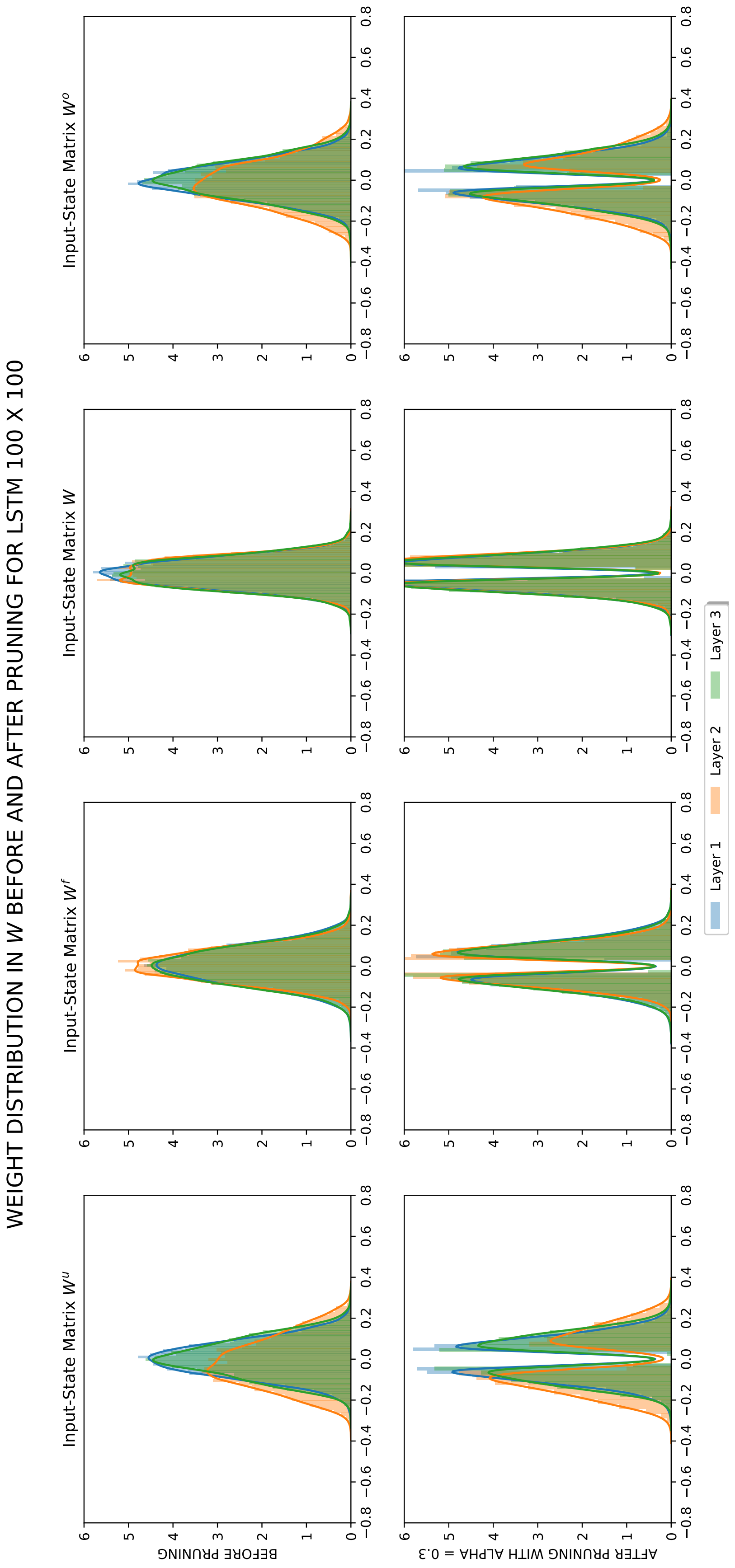


Figure 4.13: Weight Distribution for state-state matrices **W** across layers Before (Top) and After (Bottom) Pruning.

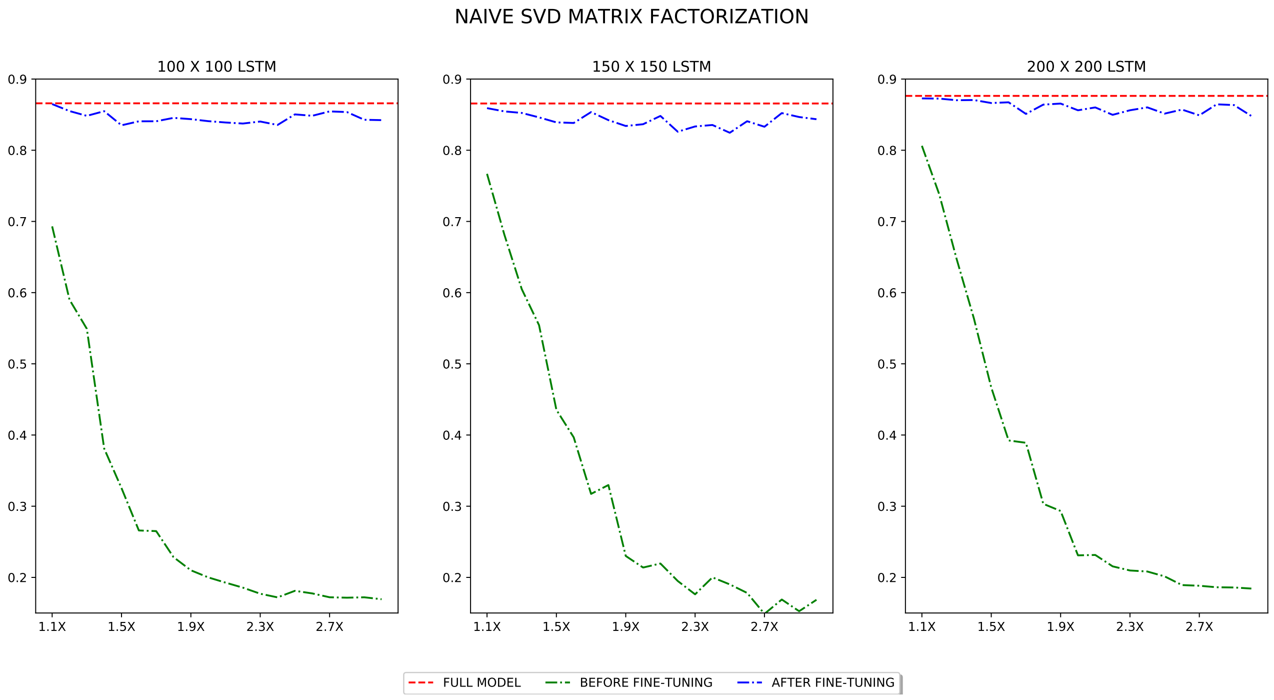


Figure 4.14: Compression Rate vs. Test Accuracy in Naive SVD model.

or multiple methods for different parts of the model, according to some properties of the model or of the method.

- The **Amount** of compression applied to the i-th parameter block. In other words, each matrix in the model is assigned a value $p_i \in (0, 1)$, which determines how many parameters will be left after the compression.

Having said that, we have experimented three different scenarios, by testing model performance (test accuracy) under different compression rates (from 1.1× to 3×).

Baseline

SVD Matrix Factorization and Pruning have been applied *separately* to the neural network model, by keeping the amount of compression p_i *equal* for all matrices $\boldsymbol{E}$, **U** and **W**. We called this types of reductions **Naive SVD Matrix Factorization** and **Naive Pruning**.

Figures 4.14 and 4.15 show model's performance (accuracy) without compression (red line), before fine-tuning (blue line) and after fine-tuning (green line).

Firstly, we notice that there is a huge difference in terms of performance between before and after fine-tuning. It should be noted that fine-tuning is a very cheap operation in our case, i.e. only 1 epoch is performed.

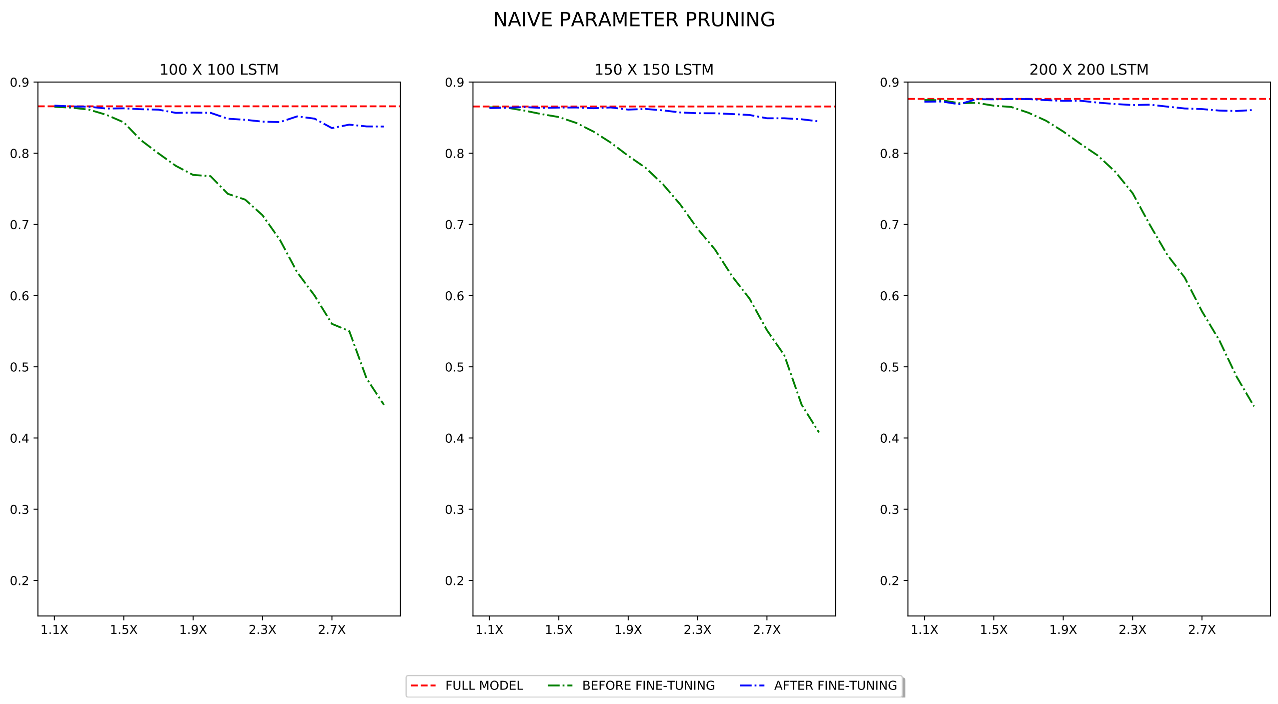


Figure 4.15: Compression Rate vs. Test Accuracy in Naive Pruning model.

Secondly, Pruning technique provides much better predictive performances than Matrix Factorization, especially without applying fine-tuning. Without re-training the model, we can compress the Naive Pruning model up to 1.5× without any significant loss in accuracy.

After fine-tuning, model's performance is restored and we can compress up to 2× without losing any accuracy. Even a large compression (3×) on the 200 × 200 model leads to a loss of less than 2% of accuracy for both Naive compression techniques.

These two compressed models represent a baseline for the other two customized compression approches.

Hybrid Compression

Given the results emerged from the full model analysis (see section 4.2), we have tried to use Parameter Pruning only for the embedding matrix $\boldsymbol{E}$, and keeping the SVD Matrix Factorization for $\mathbf{U}$ and $\mathbf{W}$ matrices. In other words, we have found a suitable *combination* of the compression methods, keeping the amount of compression p_i *equal* for all matrices. This compression method was named **Hybrid Compression.**

Figure 4.16 shows the results of this hybrid approach. This compression method has a significant approach on model's performance before fine-tuning, which

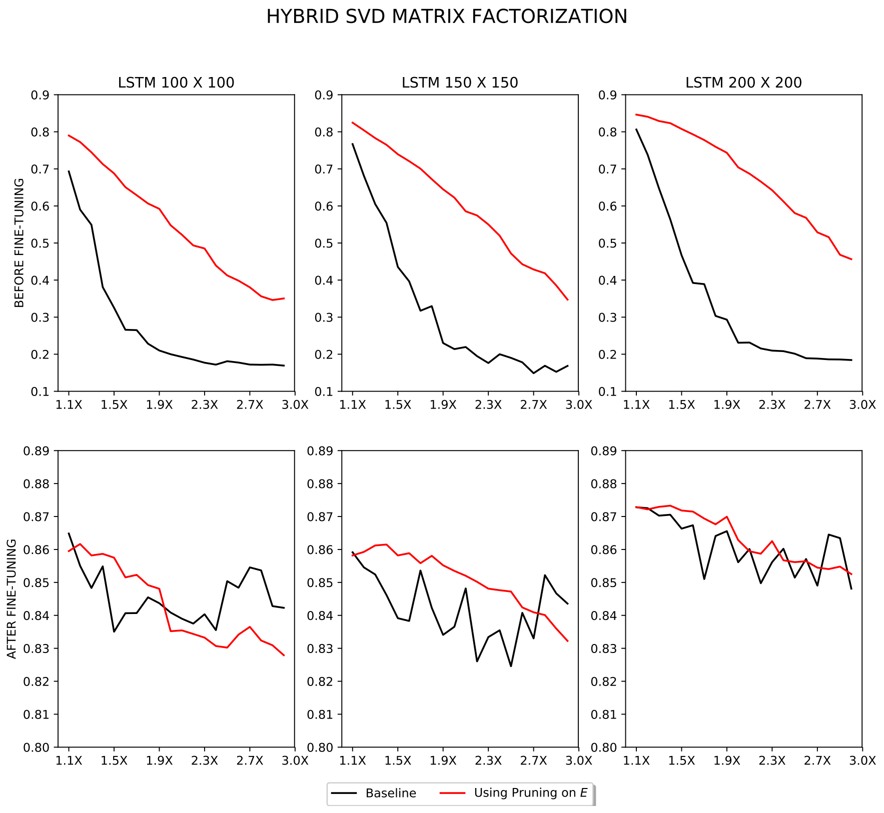


Figure 4.16: Compression Rate vs. Test Accuracy in Hybrid Compression.

was the main weakness of Naive SVD MF. The best result before fine-tuning is achieved by the 200×200 model, which can keep test accuracy above 0.80 even with a $1.5\times$ compression rate.

There are small improvements also on model's performance after fine-tuning, but it is not evident any clear pattern.

Tailored Compression

The third and final compression approach has the goal to improve the performance of Naive Pruning compression method. Looking at the weight distributions of state-state matrices $\mathbf{U}$, it has been noted in section 4.2 that matrices in the bottom layer ($\mathbf{U^1}$) had a larger density of parameters close to zero, compared to matrices in the second and third layer ($\mathbf{U^2}$ and $\mathbf{U^3}$).

This empirical information has been validated by trying to compress matrices $\mathbf{U^1}$ *more aggressively* (between 10% and 30% more) than $\mathbf{U^2}$, $\mathbf{U^3}$ and other matrices in the model. Looking at Figure 4.17 we can see the results of this compression strategy, which has been named **Tailored Compression**.

We note that this strategy is not helpful without fine-tuning the model, i.e. aggressiveness requires a retraining step for obtaining more trustful predictions.

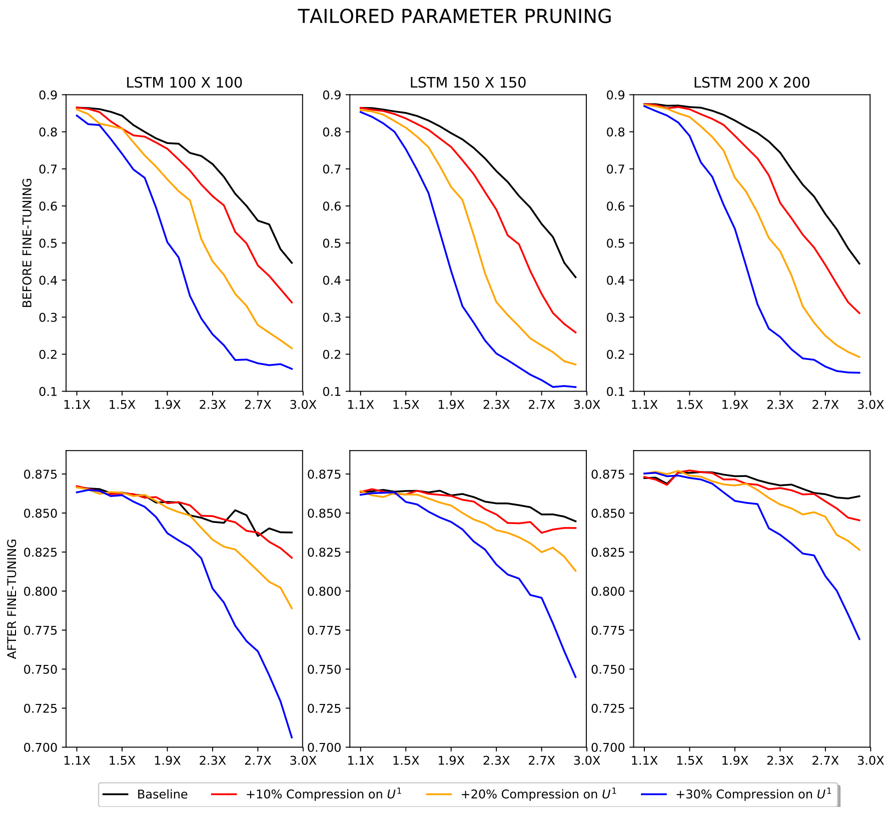


Figure 4.17: Compression Rate vs. Test Accuracy in Tailored Pruning.

In fact, after fine-tuning with moderate aggressiveness (below 20%), model's accuracy is very similar to the baseline and even higher for the 100×100 model for moderate compression rate (around $2\times$).

Summary of Results

Overall, Table 4.3 summarizes the results obtained using different compression approaches. In particular, we compare compressed and full models in terms of test accuracy and f1-score (effectiveness) and number of parameters (space-efficacy). For the sake of simplicity, the summary results regard only the Large LSTM 200×200. However, looking at the plots of Compression Rate vs. Test Accuracy the pattern observed on the Large model is very similar also for the Small and Medium sized models.

Best Compression Method When trying to score the best compression methods, it goes without saying that Pruning and its variants produce the most effective (accuracy and f1-score) and efficient (number of parameters). In particular, the Tailored approach is able to save additional 10% parameters with a decrease of only 1% in accuracy, compared to the Naive Pruning. This approach can ac-

	Model	# parameters (Milions)	test accuracy	f1-score (weighted average)
FULL	Full 100×100	1.909	86.59	86.17
	Full 150×150	3.044	86.57	86.02
	Full 200×200	4.299	87.64	87.34
BEFORE FINE-TUNING	$1.5\times$ Naive Pruning 200×200	2.866 (67%)	**87.16**	86.78
	$1.5\times$ Hybrid SVD 200×200	2.866 (67%)	80.76	80.58
	$1.5 \times +10\%$ Tailored Pruning 200×200	2.566 (59%)	86.08	85.88
AFTER FINE-TUNING	$3\times$ Naive Pruning 200×200	1.433 (33%)	86.08	85.93
	$3\times$ Hybrid SVD 200×200	1.433 (33%)	85.23	84.87
	$3 \times +10\%$ Tailored Pruning 200×200	1.133 (26%)	84.65	84.23
	$3 \times +30\%$ Tailored Pruning 200×200	833 (19%)	**82.53**	81.64

Table 4.3: Best compression results before and after fine-tuning. Number in brackets indicated the percentage of remaining size, compared to the full model.

tually aggressively compress the 200 $\times$ 200 model to approximately 833 Milion parameters, which is 19% of the original model size.

Best Compression Strategy On one hand, for high compression rates, e.g. 3$\times$, both these methods require a fine-tuning step, in order to obtain satisfactory performance results, especially for the Naive SVD method. On the other hand, when applying medium compression rage, e.g. 1.5$\times$, the fine-tuning step could be avoided and the Naive pruning strategy can be the best result among other compression methods.

5 Conclusions

The rise of new technological trends such as *AR/VR* and *IoT* is promising to revolutionize our user-experience by providing us with a set of incredible smart devices, which will assist us everywhere and at any time. From a data science point of view, this characteristics of ubiquity and pervasivity of technology sets industry practitioners and academic researchers both effectiveness and efficiency challenges.

From one hand, it is required to design machine learning models with high predictive performances (*effectiveness*) on a given set of tasks. On the other hand, these models are required to be stored on device (*space-efficiency)* and produce fast predictions (*time-efficiency*).

In the recent years, Neural Networks have proven to be extremely powerful and flexible tools for solving an incredible wide range of machine learning tasks. However, neural network models are often too large in size for deployment on mobile devices with memory and latency constraints. In practice, this means that many tasks from the user can actually be performed only by calling an external cloud server infrastructure, i.e. without the possibility to do the computation offline.

On this premise, the present work had the goal to first provide an overview of the state-of-the-art results and methods on Neural Network Compression. After that, the second goal was to have hands-on experience of two of these compression methods, i.e. *SVD Matrix Factorization* and *Parameter Pruning.* In particular, we chose to compress an RNN architecture which has been trained for solving an NLP task, i.e. PoS Tagging, using the Universal Dependencies dataset.

Compression results are based on three compression strategies, which apply different *methods* with a different *amount* of compression on different parts of the model.

The first compression scheme is a *naive* implementation of the methods on all matrices. The second and third compression schemes instead adopt *data-driven* compression approaches, based on simple but effective visualizations of eigenvalues and parameter distribution across different model sizes, layers and matrices. The goal of such data-driven approach is first to spot the distribution of redundancy among the bilions of parameters inside the neural network, and then to operate a selective compression only on these redudant parameter blocks.

Empirical results validate the effectiveness of such approach. The size of a multi-layer LSTM network with more than 4 milions of parameters can be compressed by 1.5× (*moderate compression*) with no significant loss in accuracy. Adopting a 3× compression rate (*aggressive compression*), the size of the full model can be compressed by more than 80%, with only a 5% loss in test accuracy.

6 Acknowledgements

I believe that reflecting on our past gives us the opportunity to look at the present with greater awareness and to imagine the future better.

It is even more important to recall in the mind and heart all the people who have been part of our growth path. In particular, here I want to mention the people who have left an indelible mark on me, the people that I will carry with me forever...

Let's start with who gave me life and that is my two parents **Gioacchino** and **Tania**. Thank you for passign down to me the ideals of justice, legality and perseverance with your daily actions. Thank you for always trusting me and for always leaving me free to pursue my dreams, even thousands of miles from home! I believe there is no better feeling for a child!

Fortunately, 1547 km far from home my beautiful grandmother **Edda** and my legendary grandfather **Miro** were waiting for me. I tell you thank you for having welcomed me during both childhood and university period, for having been a second family ... in the North! Without you I wouldn't be the person I am today!

A little closer, 92 km away from home, in the Madonie park, another family was waiting for me on the weekends and on the hot Sicilian summer days! Thanks to my aunt **Antonella**, my uncle **Mimmo**, **Vincenzo**, **Cinzia** and **Elvis**! You have been extraordinary fans in these years, you have made me (re) discover the beauty of people, food and Sicilian territories. You have planted the seed of "Sicilianity" in me and I feel that this seed will grow in the future!

Much more distant, almost 5000 km away from my beloved Palermo, lies the charming town of Togliatti. Thank you **Inna**, **Vladimir** and **Jana** for welcoming me into your family at the beginning of that fantastic adventure in 2012. Thanks to my Russian teacher **Rimma Anatolevna**, who allowed me to master a language so complex and in some ways "magical". Thanks to my second family **Natasha**, **Jurij**, **Sasha** and **Pasha**. Returning to Togliatti after 6 years I felt at home again, especially after putting on my old slippers again!

Living away from home has allowed me to meet fantastic people, who are some of my greatest friends today, my second, third, fourth, fifth family. It is these people who remind me of the different periods of my life, I remember myself when I met them and so I think of myself as I am now.

I want to start with the most recent friendships, here in Russia in the dormitory No. 5 of Kibalchicha Ulitsa 7. Thanks **Erica**, **Angelica**, **Andrea**, **Eleonora**, **Chantal**, **Francesca**, **Vasilia** and **Diana**! Living with cockroaches, babushkas, Albano and Country Roads we didn't go so badly!

Also in Russia I met a friend again, who also recently took the habit of traveling the world. Thanks **Anatoly** for letting me discover the submerged parts of the iceberg of Russian culture and therefore also of the Italian one. Thanks for still being patient with my grammatical errors in Russian!

First in Russia, but then in Cisterna di Latina, life introduced me to **Matteo**, a companion of adventures, dreams and personal growth. From you I learned that the limits exist only in our head and therefore they are not real! Never stop dreaming big, my friend!

Let's go back to the North ... in the land of suricati! Thank you **Riccardo** and **Stefano** for sharing unforgettable moments and always bringing also in the kitchen Sicilian, Friulian, Venetian and European pride! Thanks also to you **Alessandro** for showing me the richness inside you and your passion for *finlandssvenska*!

The best gift that I had from the experience in Helsinki, however, was to meet you **Giulia**! Life goes fast, but we run faster, hand in hand, like everything that comes in our future together! Our hearts will be our compass to realize our dreams and life projects together. I can't wait!

What should I say about the Sgorgi group! Thanks **Ale Jr**, **Ale Sr**, **Peppe** and **Tizi**! You who have seen me grow from elementary school until now, you know how much I care about you!

With you **Tiziano**, our friendship has grown even more in the last years. Together we have shared millions of situations and together every day we try to find solutions to the great puzzle of life! I want to thank you for the critical and rebellious point of view you brought to my life, it was necessary!

Thanks to all the **friends of the University of Padua** and in particular to **Salvatore**, who shared with me in 26 m^2 over 3 years of burrata, evening conversations and many transformations in our lives! The heaters didn't always work, but we carried the Sicilian heat inside ourselves! We kindly remind the reader that Frederick II, the last King of Sicily, once stated "*I do not envy God's paradise because I am so satisfied to live in Sicily*!".

I also want to thank the volunteer groups with whom I have had the opportunity to collaborate in these years. Thanks to **Intercultura** and all the **Padua Volunteer Chapter**. Experiencing intercultural dialogue day by day with students, parents, teachers, volunteers and ... captains of an intercontinental flight ... was the greatest gift I could receive after the experience in Russia! Thanks to **Erasmus Student Network Padova** and in particular to **Marco**, **Giovanni** and **Matteo**!

Thanks to those "foolish and brave" guys from **Contamination Lab Veneto** and above all thanks to the **Konica Minolta team**. Thanks to you I discovered the value and the daily difficulties of being an entrepreneur. I will keep in mind all the precious lessons learned for my future!

One day I read a sentence that has remained with me, like all the people I have mentioned so far.

> " I have learned to judge the goodness of a man not so much by the results he achieves in life, but by the results of the people he comes into contact with ".

This remains and will always remain the main objective of my personal, relational and professional growth path. We're only at the beginning!

Moscow, 30 June 2019

7 Ringraziamenti

Credo che riflettere sul nostro passato ci dia l'opportunità di guardare con maggiore coscienza al presente e ad immaginare meglio il futuro.

Ancora più importante è ripercorrere nella mente e nel cuore tutte le persone che hanno fatto parte del nostro percorso di crescita. In particolare, qui voglio soffermarmi su quelle persone che hanno lasciato in me un'impronta indelebile, che porterò per sempre con me...

Cominciamo da chi mi ha dato la vita e cioè i miei due genitori **Gioacchino** e **Tania**. Grazie per avermi trasmesso con le vostre azioni quotidiane gli ideali di giustizia, legalità e perseveranza. Grazie per esservi sempre fidati di me e per avermi lasciato sempre libero di inseguire i miei sogni, anche a migliaia di chilometri da casa! Credo che non ci sia sensazione più bella per un figlio!

Per fortuna a 1547 km da casa si trovava la mia splendida nonna **Edda** e il mio mitico nonno **Miro**. A voi dico grazie per avermi accolto sia da piccolo che durante l'università, per essere stati una seconda famiglia... al Nord! Senza di voi non sarei la persona che sono oggi!

Un po' più vicino, a 92 km nel parco delle Madonie mi aspettava un'altra famiglia nei fine settimana e nelle torride giornate estive siciliane! Grazie **Zia Antonella**, **Zio Mimmo**, **Vincenzo**, **Cinzia** ed **Elvis**! Siete stati dei tifosi straordinari in questi anni, mi avete fatto (ri)scoprire la bellezza delle persone, del cibo e dei territori siciliani. Avete piantato dentro di me il semino della sicilianità e sento che crescerà nel futuro!

Ben più distante, a quasi 5000 km di distanza dalla mia amata Palermo sorge la ridente cittadina Togliatti. Grazie **Inna**, **Vladimir** e **Jana** per avermi accolto nella vostra famiglia all'inizio di quella fantastica avventura nel 2012. Grazie alla mia insegnante di russo **Rimma Anatolevna**, che mi ha permesso di padroneggiare una lingua così complessa e per certi versi "magica". Grazie alla mia seconda famiglia **Natasha**, **Jurij**, **Sasha** e **Pasha**. Tornando a Togliatti dopo 6 anni mi sono sentito di nuovo a casa, soprattutto dopo avere indossato di nuove le mie vecchie ciabatte!

Vivere lontano da casa mi ha permesso di incontrare persone fantastiche, che oggi sono i miei amici più grandi, la mia seconda, terza, quarta, quinta famiglia. Sono queste persone che mi ricordano i diversi periodi della mia vita, ricordo me quando li ho conosciuti e quindi penso a me come sono adesso.

Voglio cominciare dalle amicizie più fresche, qui in Russia nel dormitorio N.5 di Kibalchicha Ulitsa 7. Grazie **Erica**, **Angelica**, **Andrea**, **Eleonora**, **Chantal**, **Francesca**, **Vasilia** e **Diana**! Tra scarafaggi, babushke, Albano e Country Roads non ce la siamo passati poi così male!

Sempre in Russia ho incontrato di nuovo un amico, che anche lui ultimamente ha preso il vizio di girare il mondo. Grazie **Anatoly** per avermi fatto scoprire i lati sommersi dell'iceberg della cultura russa e quindi anche di quella italiana. Grazie per essere ancora paziente con i miei errori grammaticali in russo!

Prima in Russia, ma poi a Cisterna di Latina, la vita mi ha presentato **Matteo**, compagno di avventure, sogni e crescita personale. Da te ho imparato che i limiti esistono solo nella nostra testa e quindi non sono reali! Non smettere mai di sognare in grande, amico mio!

Torniamo a Nord... nella terra dei suricati! Grazie **Riccardo** e **Stefano** per avere condiviso momenti indimenticabili ed avere sempre portato alto l'orgoglio siciliano, friulano, veneto ed europeo anche ai fornelli! Grazie anche a te **Alessandro** per avermi mostrato la ricchezza che porti dentro di te e la tua passione *finlandssvenska*!

Il dono più bello che mi ha fatto l'esperienza ad Helsinki però è stato incontrare te **Giulia**! La vita va veloce, ma noi corriamo più velocemente, mano nella mano, come tutto ciò che verrà nel nostro futuro insieme! I nostri cuori saranno la nostra bussola per realizzare insieme i nostri sogni e progetti di vita. Non vedo l'ora!

Che dire invece del gruppo Sgorgi! Grazie **Ale Jr**, **Ale Sr**, **Peppe** e **Tizi**! Voi che mi avete visto crescere dalle scuole elementari fino ad ora, sapete bene quanto tengo a voi!

Con te **Tiziano** il legame si è rafforzato ancora di più negli ultimi anni. Insieme abbiamo condiviso milioni di situazioni e ogni giorno insieme cerchiamo di trovare soluzioni al grande puzzle della vita! Voglio ringraziarti per il punto di vista critico e ribelle che hai portato alla mia vita, era necessario!

Grazie a tutti gli **amici dell'università di Padova** e in particolare a **Salvatore**, che con me ha condiviso in 26 m^2 ben 3 anni di burrate, conversazioni serali e tante trasformazioni nelle nostre vite! I riscaldamenti non funzionavano sempre, ma tanto noi portavamo dentro il calore siciliano! Ricordiamo al lettore che Federico II, ultimo Re di Sicilia, affermò "*Non invidio a Dio il Paradiso, perché sono ben soddisfatto di vivere in Sicilia!*".

Voglio ringraziare anche i gruppi di volontariato con i quali ho avuto l'opportunità di collaborare in questi anni. Grazie ad **Intercultura** e a tutto il **Centro Locale di Padova**. Sperimentare giorno per giorno il dialogo interculturale con studenti, genitori, docenti, volontari e... capitani di un volo intercontinentale... è stato il più grande regalo che potessi ricevere dopo l'esperienza in Russia! Grazie ad **Erasmus Student Network Padova** ed in particolare a **Marco**, **Giovanni** e **Matteo**!

Grazie a quei "pazzi furiosi" del **Contamination Lab Veneto** e soprattuto grazie al **team di Konica Minolta**. Grazie a voi ho scoperto il valore e le difficoltà quotidiane di essere imprenditore. Terrò bene a mente tutte le preziose lezioni apprese per il mio futuro!

Un giorno ho letto una frase che mi è rimasta impressa, come tutte le persone che ho ricordato finora.

> " Ho imparato a giudicare la bontà di un uomo non tanto dai risultati che lui stesso raggiunge nella vita, ma da quelli delle persone con cui entra in contatto ".

Questo rimane e resterà per sempre l'obiettivo principale del mio percorso di crescita personale, relazionale e professionale. Siamo solo all'inizio!

Mosca, 30 Giugno 2019

Bibliography

Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 535–541.

Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*. 2285–2294.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

Maxwell D Collins and Pushmeet Kohli. 2014. Memory bounded deep convolutional networks. *arXiv preprint arXiv:1412.1442* (2014).

Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14, 2 (1990), 179–211.

Yoav Goldberg. 2017. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies* 10, 1 (2017), 1–309.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Artem M Grachev, Dmitry I Ignatov, and Andrey V Savchenko. 2019. Compression of Recurrent Neural Networks for Efficient Language Modeling. *arXiv preprint arXiv:1902.02380* (2019).

Alex Graves. 2012. *Supervised Sequence Labelling with Recurrent Neural Networks.* Studies in Computational Intelligence, Vol. 385. Springer. `https://doi.org/10.1007/978-3-642-24797-2`

Song Han, Huizi Mao, and William J Dally. 2015a. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

Song Han, Jeff Pool, John Tran, and William Dally. 2015b. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems.* 1135–1143.

Babak Hassibi and David G Stork. 1993. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems.* 164–171.

Benjamin Heinzerling and Michael Strube. 2019. Sequence Tagging with Contextual and Non-Contextual Subword Representations: A Multilingual Evaluation. *arXiv preprint arXiv:1906.01569* (2019).

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. 2012. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine* 29 (2012).

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

Mark Johnson. 2009. How the statistical revolution changes (computational) linguistics. In *Proceedings of the EACL 2009 Workshop on the Interaction between Linguistics and Computational Linguistics: Virtuous, Vicious or Vacuous?* Association for Computational Linguistics, 3–11.

Daniel Jurafsky and James H Martin. 2008. Speech and Language Processing: An introduction to speech recognition, computational linguistics and natural language processing. *Upper Saddle River, NJ: Prentice Hall* (2008).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

Nicholas D Lane and Petko Georgiev. 2015. Can deep learning revolutionize mobile sensing?. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. ACM, 117–122.

Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.

Zhiyun Lu, Vikas Sindhwani, and Tara N Sainath. 2016. Learning compact recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 5960–5964.

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* (2019).

Barbara Plank, Anders Søgaard, and Yoav Goldberg. 2016. Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss. *arXiv preprint arXiv:1604.05529* (2016).

Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. 2013. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 6655–6659.

Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128* (2014).

Abigail See, Minh-Thang Luong, and Christopher D Manning. 2016. Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274* (2016).

Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. 2015. Structured transforms for small-footprint deep learning. In *Advances in Neural Information Processing Systems*. 3088–3096.

Jian Xue, Jinyu Li, and Yifan Gong. 2013. Restructuring of deep neural network acoustic models with singular value decomposition.. In *Interspeech*. 2365–2369.