NEURAL MACHINE TRANSLATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Minh-Thang Luong
August 2016

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christopher D. Manning) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dan Jurafsky)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Andrew Ng)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Quoc V. Le)

Approved for the Stanford University Committee on Graduate Studies

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> The Babel fish is small, yellow, leech-like, and probably the oddest thing in the universe. It feeds on brainwave energy ... if you stick a Babel fish in your ear, you can instantly understand anything in any form of language.
>
> *The Hitchhiker's Guide to the Galaxy*. Douglas Adams.

Human languages are diverse and rich in categories with about 6000 to 7000 languages spoken worldwide.[1] As civilization advances, the need for seamless communication and understanding across languages becomes more and more crucial. Machine translation (MT), the task of teaching machines to learn to translate automatically across languages, as a result, is an important research area. MT has a long history [26] from the original phiosophical ideas of universal languages in the seventeen century to the first practical instances of MT in the twentieth century, e.g., one proposal by Weaver [67]. Despite several excitement moments that led to hopes that MT will be solved "very soon", e.g., the 701 translator[2] developed by scientists at George Town and IBM in the 1950s or a simple vector-space transformation technique[3] proposed by Google researchers at the beginning of the twenty-first century, MT remains to be an extremely challenging problem.[4] To understand why MT is difficult, let us trace through one "evolution" path of MT which crosses

[1] http://www.linguisticsociety.org/content/how-many-languages-are-there-world
[2] http://www-03.ibm.com/ibm/history/exhibits/701/701_translator.html
[3] https://www.technologyreview.com/s/519581/how-google-converted-language-transl
[4] http://www.huffingtonpost.com/nataly-kelly/why-machines-alone-cannot-translati

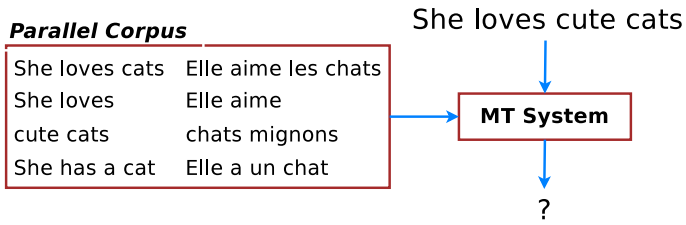

Figure 1.1: **Machine translation** (MT) – a general setup of MT. Systems build translation models from parallel corpora to translate new unseen sentences, e.g., "She loves cute cats".

through techniques that are used extensively in commercial MT systems.

1.1 Machine Translation Development

Modern statistical MT started out with a seminal work by IBM scientists [8]. The proposed technique requires minimal linguistic content and only needs a *parallel corpus*, i.e., a set of pairs of sentences that are translations of one another, to train machine learning algorithms to tackle the translation problem. Such a language-independent setup is illustrated in Figure 1.1 and remains to be the general approach for nowadays MT systems. For over twenty years since the IBM seminal paper, approaches in MT such as [9, 10, 13, 31, 32, 33, 50], are, by and large, similar according to the following two-stage process (see Figure 1.2). First, source sentences are broken into chunks which can be translated in isolation by looking up a "dictionary", or more formally a *translation model*. Translated target words and phrases are then put together to form coherent and natural-sounding sentences by consulting a *language model* (LM) on which sequences of words, i.e., n-*grams*, are likely to go with one another.

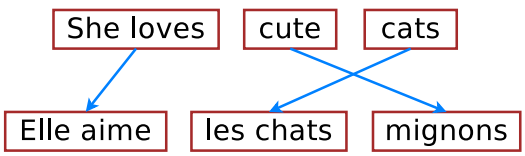

Figure 1.2: **Phrase-based machine translation** (MT) – example of how phrase-based MT systems translate a source sentence "She loves cute cats" into a target sentence "Elle aime les chats mignons": sentences are split into chunks and phrases are translated.

The aforementioned approach, while has been successfully deployed in many commercial systems, does not work very well and suffers from the following two major drawbacks. First, translation decisions are *locally determined* as we translate phrase-by-phrase and long-distance dependencies are often ignored. Second, it is slightly "strange" that language models (LMs), despite being a key component in the MT pipeline, utilize context information that is both short, consisting of only a handful of previous words, and target-only, never looking at the source words. These shortcomings in LMs gives rise to a new wave of *hybrid* systems which aim to empower phrase-based MT with neural network components, most notably neural probabilistic language models (NPLMs).

NPLMs were first proposed by Bengio et al. [5] as a way to combat the "curse" of dimensionality suffered by traditional LMs. In traditional LMs, one has to explicitly store and handle all possible n-grams occurred in a training corpus, the number of which quickly becomes enormous. As a result, existing MT systems often limit themselves to use only short, e.g., 5-gram, LMs [23], which capture little context and cannot generalize well to unseen n-grams. NPLMs address these concerns by using distributed representations of words and not having to explicitly store all enumerations of words. As a result, many MT systems, [37, 56, 65], inter alia, start adopting NPLMs alongside with traditional LMs. To make NPLMs even more powerful, recent work [1, 12, 57, 59] propose to condition on source words beside the target context to lower uncertainty in predicting next words (see Figure 1.3).[5]

These hybrid MT systems with NPLM components, while having addressed shortcomings of traditional phrase-based MT, still translate locally and fail to capture long-range dependencies. For example, in Figure 1.3, the source-conditioned NPLM does not see the word "stroll", or any other words outside of its fixed context windows, which can be useful in deciding that the next word should be "bank" as in "river bank" rather "financial bank". More problematically, the entire MT pipeline is already complex with different components needed to be tuned separatedly, e.g., translation models, language models, reordering models, etc.; now, it becomes even worse as different neural components are incorporated. Neural Machine Translation to the rescue!

[5] In [12], the authors have constructed a model that conditions on 3 target words and 11 source words, effectively building a 15-gram LM.

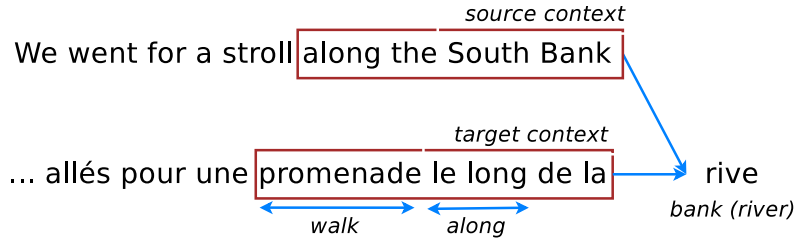

Figure 1.3: **Source-conditioned neural probabilistic language models** (NPLMs) – example of a source-conditioned NPLM proposed by Devlin et al. [12]. To evaluate a how likely a next word "rive" is, the model not only relies on previous target words (context) "promenade le long de la" as in traditional NPLMs [5], but also utilizes source context "along the South Bank" to lower uncertainty in its prediction.

Neural Machine Translation (NMT) is a new approach to translating text from one language into another that captures long-range dependencies in sentences and generalizes better to unseen texts. The core of NMT is a single deep neural network with hundreds of millions of neurons that learn to directly map source sentences to target sentences [11, 29, 63]. This is often referred as the sequence-to-sequence or encoder-decoder approach.[6] NMT is appealing since it is conceptually simple and can be trained end-to-end. NMT translates as follows: an *encoder* reads through the given source words one by one until the end, and then, a *decoder* starts emitting one target word at a time until a special end-of-sentence symbol is produced. We illustrate this process in Figure 1.4.

Such simplicity leads to several advantages. NMT requires minimal domain knowledge: it only assumes access to sequences of source and target words as training data and learns to directly map one into another. NMT beam-search decoders that generate words from left to right can be easily implemented, unlike the highly intricate decoders in standard MT [31]. Lastly, the use of recurrent neural networks (RNNs) allow NMT to generalize well to very long word sequences while not having to explicitly store any gigantic phrase tables or language models as in the case of standard MT.

[6]Forcada and Neco [17] wrote the very first paper on sequence-to-sequence models for translation!

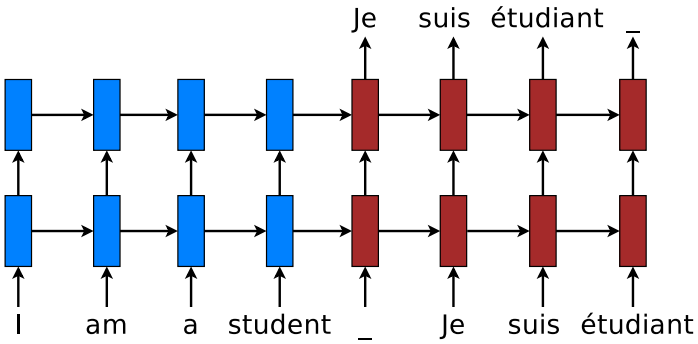

Figure 1.4: **Neural machine translation** – example of a deep recurrent architecture proposed by Sutskever et al. [63] for translating a source sentence "I am a student" into a target sentence "Je suis étudiant". Here, "_" marks the end of a sentence.

1.2 Thesis Outline

Despite all the aforementioned advantages and potentials, the early NMT architecture [11, 63] still has many drawbacks. In this thesis, I will highlight three problems pertaining to the existing NMT model, namely the *vocabulary size*, the *sentence length*, and the *language complexity* issues. Each chapter is devoted to solving each of these problems in which I will describe how I have pushed the limits of NMT, making it applicable to a wide variety of languages with state-of-the-art performance such as English-French [39], English-German [35, 38], and English-Czech [36]. Towards the *future* of NMT, I answer two questions: (1) whether we can improve translation by jointly learning from a wide variety of sequence-to-sequence tasks such as parsing, image caption generation, and auto-encoders or skip-thought vectors [40]; and (2) whether we can compress NMT for mobile devices [58]. In brief, this thesis is organized as follows. I start off by providing background knowledge on RNN and NMT in Chapter 2. The aforementioned three problems and approaches for NMT future are detailed in Chapters 3, 4, 5, and 6 respectively, which we will go through one by one next. Chapter 7 wraps up and discusses remaining challenges in NMT research.

Copy Mechanisms

A significant weakness in conventional NMT systems is their inability to correctly translate very rare words: end-to-end NMTs tend to have relatively small vocabularies with a single

<unk> symbol that represents every possible out-of-vocabulary (OOV) word. In Chapter 3, we propose simple and effective techniques to address this *vocabulary size* problem through teaching NMT to "copy" words from source to target. Specifically, we train an NMT system on data that is augmented by the output of a word alignment algorithm, allowing the NMT system to emit, for each OOV word in the target sentence, the position of its corresponding word in the source sentence. This information is later utilized in a post-processing step that translates every OOV word using a dictionary. Our experiments on the WMT'14 English to French translation task show that this method provides a substantial improvement of up to 2.8 BLEU points over an equivalent NMT system that does not use this technique. With 37.5 BLEU points, our NMT system is the first to surpass the best result achieved on a WMT'14 contest task.

Attention Mechanisms

While NMT can translate well for short- and medium-length sentences, it has a hard time dealing with long sentences. An attentional mechanism was proposed by Bahdanau et al. [2] to address that *sentence length* problem by selectively focusing on parts of the source sentence during translation. However, there has been little work exploring useful architectures for attention-based NMT. Chapter 4 examines two simple and effective classes of attentional mechanism: a *global* approach which always attends to all source words and a *local* one that only looks at a subset of source words at a time. We demonstrate the effectiveness of both approaches on the WMT translation tasks between English and German in both directions. With local attention, we achieve a significant gain of 5.0 BLEU points over non-attentional systems that already incorporate known techniques such as dropout. Our ensemble model using different attention architectures yields a new state-of-the-art result in the WMT'15 English to German translation task with 25.9 BLEU points, an improvement of 1.0 BLEU points over the existing best system backed by NMT and an n-gram reranker.

Hybrid Models

Nearly all previous NMT work has used quite restricted vocabularies, perhaps with a subsequent method to patch in unknown words such as the copy mechanisms mentioned earlier.

While effective, the copy mechanims cannot deal with all the complexity of human languages such as rich morphology, neologisms, and informal spellings. Chapter 5 presents a novel word-character solution to that *language complexity* problem towards achieving open vocabulary NMT. We build hybrid systems that translate mostly at the *word* level and consult the *character* components for rare words. Our character-level recurrent neural networks compute source word representations and recover unknown target words when needed. The twofold advantage of such a hybrid approach is that it is much faster and easier to train than character-based ones; at the same time, it never produces unknown words as in the case of word-based models. On the WMT'15 English to Czech translation task, this hybrid approach offers an addition boost of $+2.1-11.4$ BLEU points over models that already handle unknown words. Our best system achieves a new state-of-the-art result with 20.7 BLEU score. We demonstrate that our character models can successfully learn to not only generate well-formed words for Czech, a highly-inflected language with a very complex vocabulary, but also build correct representations for English source words.

NMT Future

Chapter 6 answers the two aforementioned questions for the future of NMT: whether we can utilize other tasks to improve translation and whether we can compress NMT models.

For the first question, we examine three multi-task learning (MTL) settings for sequence to sequence models: (a) the *one-to-many* setting – where the encoder is shared between several tasks such as machine translation and syntactic parsing, (b) the *many-to-one* setting – useful when only the decoder can be shared, as in the case of translation and image caption generation, and (c) the *many-to-many* setting – where multiple encoders and decoders are shared, which is the case with unsupervised objectives and translation. Our results show that training on a small amount of parsing and image caption data can improve the translation quality between English and German by up to 1.5 BLEU points over strong single-task baselines on the WMT benchmarks. Rather surprisingly, we have established a new *state-of-the-art* result in constituent parsing with 93.0 F_1 by utilizing translation data. Lastly, we reveal interesting properties of the two unsupervised learning objectives, autoencoder and skip-thought, in the MTL context: autoencoder helps less in terms of perplexities but more on BLEU scores compared to skip-thought.

For the second question, we examine three simple magnitude-based pruning schemes to compress NMT models, namely *class-blind*, *class-uniform*, and *class-distribution*, which differ in terms of how pruning thresholds are computed for the different classes of weights in the NMT architecture. We demonstrate the efficacy of weight pruning as a compression technique for a state-of-the-art NMT system. We show that an NMT model with over 200 million parameters can be pruned by 40% with very little performance loss as measured on the WMT'14 English-German translation task. This sheds light on the distribution of redundancy in the NMT architecture. Our main result is that with *retraining*, we can recover and even surpass the original performance with an 80%-pruned model.

Chapter 2

Background

> For neural machine translation, it all started from language modeling.
>
> Thang Luong.

Language modeling plays an indispensable role in ensuring that machine translation systems produce fluent target sentences and has always been an active area of research. Despite much effort in improving traditional n-gram language models [16, 23, 24, 52, 54, 60, 64], traditional LMs inherently can only handle short contexts of a few words. Approaches to building neural probabilistic language models (NPLMs) using feed-forward networks such as those initiated by Bengio et al. [5] and enhanced by others [3, 47, 48, 49] have addressed that drawback to model longer contexts. Still, NPLMs can only capture fixed-length contexts and is incapable of handling variable-length sequences, which is the case for sentences. Recurrent neural networks (RNNs) come in handy as a powerful and expressive architecture to handle sequential data and have successfully been applied to the language modeling task [43, 44, 45]. By viewing RNNs as generative models [62] that can produce texts and by pushing another step towards conditioning RNNs on source sentences, recent works [11, 29, 63] have started a new line of resesarch in machine translation, namely Neural Machine Translation (NMT). NMT is technically a source-conditioned NPLM that can be trained end-to-end.

In this chapter, we provide background knowledge on two main topics, RNN and NMT.

We first go through the basics of RNNs, explaining how they can be used to model sentences. Then, we delve into details of one particular type of RNNs, the Long Short-term Memory, that makes training RNNs easier. Given RNNs as a building block, we discuss NMT together with tips and tricks for better training and testing NMT.

2.1 Recurrent Neural Network

Recurrent Neural Network (RNNs) [15] are models that help understand the temporal aspect as well as build up representations for sequential data using a dynamic memory structure. At the surface form, an RNN takes as input a sequence of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ and processes them one by one. For each new input $\boldsymbol{x}_i$, an RNN updates its memory to produce a hidden state $\boldsymbol{h}_i$ which one can think of as a representation for the partial sequence $\boldsymbol{x}_{\overline{1,i}}$. The beauty of RNNs lies in the fact that it can capture the dynamics of an arbitrarily long sequence without having to increase its modeling capacity unlike the case of feedforward network which can only model relationship within a fixed-length sequence. The key secret sauce is in the recurrence formula of an RNN that defines how its hidden state is updated. At its simplest form, a "vanilla" RNN defines its recurrence function as:

$$\boldsymbol{h}_t = f\left(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}\right) \tag{2.1}$$

In the above formula, f is an abstract function that computes a new hidden state given the current input $\boldsymbol{x}_t$ and the previous hidden state $\boldsymbol{h}_{t-1}$. The starting state $\boldsymbol{h}_0$ is often set to $\mathbf{0}$ though it can take any value as we will see later in the context of NMT decoders. A popular choice of f is provided below with σ being a non-linear function such as $\mathrm{sigmoid}$ or $\tanh$.[1]

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}) \tag{2.2}$$

At each timestep t, an RNN can (optionally) emit an output symbol y_t which can either be discrete or real-valued. For the discrete scenario, which is often the case for languages,

[1] There could also be an optional bias term in Eq. (2.2).

a probability distribution $\boldsymbol{p}$ over a set of output classes Y is derived as follows[2]:

$$\boldsymbol{s}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t \tag{2.3}$$

$$\boldsymbol{p}_t = \text{softmax}(\boldsymbol{s}_t) \tag{2.4}$$

Here, we introduce a new set of weights $\boldsymbol{W}_{hy} \in \mathbb{R}^{|Y| \times d}$, with d being the dimension of the RNN hidden state, to compute a score vector $\boldsymbol{s}_t$, or *logits*, over different individual classes. Often, with a large output set Y, the matrix-vector multiplication in Eq. (2.3) is a major computational bottleneck in RNNs, which results in several challenges for neural language modeling and machine translation that we will address in later chapters. The softmax function transforms the score vector $\boldsymbol{s}_t$ into a probability vector $\boldsymbol{p}_t$, which is defined for each specific element $y \in Y$ as below. For convenience, we overload our notations to use $\boldsymbol{p}_t(y)$ and $\boldsymbol{s}_t(y)$ to refer to entries in the vectors $\boldsymbol{p}_t$ and $\boldsymbol{s}_t$ that correspond to y.

$$\boldsymbol{p}_t(y) = \frac{\mathrm{e}^{\boldsymbol{s}_t(y)}}{\sum_{y' \in Y} \mathrm{e}^{\boldsymbol{s}_t(y')}} \tag{2.5}$$

With the above formulas, we have completely defined the RNN weight set $\boldsymbol{\theta}$ which consists of *input* connections $\boldsymbol{W}_{xh}$, *recurrent* connections $\boldsymbol{W}_{hh}$, and *output* connections $\boldsymbol{W}_{hy}$. These weights are shared across timesteps as illustrated in Figure 2.1 Draw a picture on general RNNs, which enables RNNs to handle arbitrarily long sequences.

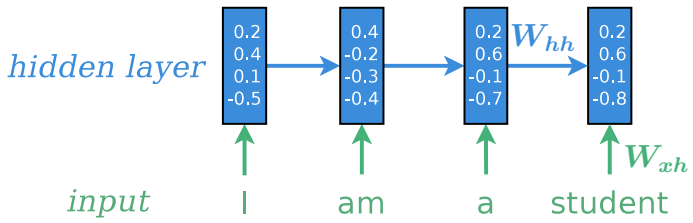

Figure 2.1: **Recurrent neural networks** – example of a recurrent neural network that processes a sequence of input words "I am a student" to build up hidden representations as input symbols are consumed. The recurrent $\boldsymbol{W_{hh}}$ and feed-forward $\boldsymbol{W_{xh}}$ weights are shared across timesteps.

[2]For the real-valued case, we refer readers to mixture density models [7] which have been applied to RNN training, e.g., for hand-writing synthesis [19].

Next, we discuss the training and testing phases of RNNs from a slightly more focused angle, the language learning aspect. For more details on RNNs, we refer readers to the following resources [30, 42, 61].

2.1.1 Recurrent Language Models

To apply RNNs to sentences in languages, or generally sequences of discrete symbols, one can consider one-hot representations $\boldsymbol{x}_i \in \mathbb{R}^{|V|}$, with V being the vocabulary considered. However, for a large vocabulary V, such a representation choice is problematic as it results in a large weight matrix $\boldsymbol{W}_{xh}$ and there is no notion of similarity between words. In practice, low-dimensional dense representations for words, or *word embeddings*, are often used to address these problems. Specifically, an embedding matrix $\boldsymbol{W}_e \in \mathbb{R}^{d_e \times |V|}$ is looked up for each word x_i to retrieve a representation $\boldsymbol{x}_i \in \mathbb{R}^{d_e}$. As a result, a simple RNN applied to language modeling will generally have $\theta = \{\boldsymbol{W}_{xh}, \boldsymbol{W}_{hh}, \boldsymbol{W}_{hy}, \boldsymbol{W}_e\}$ as its weights as illustrated in Figure 2.2 Draw an RNN with embedding.

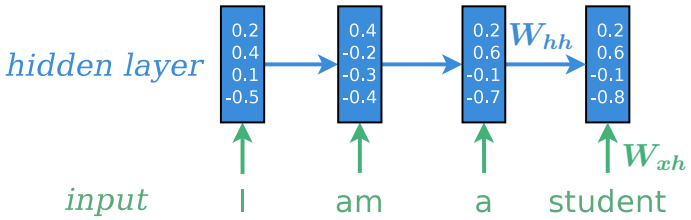

Figure 2.2: **Recurrent language models** – example of a recurrent neural network that processes a sequence of input words "I am a student" to build up hidden representations as input symbols are consumed. The recurrent $\boldsymbol{W_{hh}}$ and feed-forward $\boldsymbol{W_{xh}}$ weights are shared across timesteps.

In language modeling (LM), the task is to specify a probability distribution over sequences of symbols (often, words) so that one can judge if a sequence of words is more likely or "fluent" than another. To accomplish that, an LM decomposes the probability of a word sequence $y = y_1, \ldots, y_m$ as:

$$p(y) = \prod_{i=1}^{m} p(y_i | y_{<i}) \tag{2.6}$$

In the above formula, each of the individual term $p(y_i|y_{<i})$ is the conditional probability of the current word y_i given previous words $y_{<i}$, also referred as the *context* or the *history*. To model these conditional probabilities, traditional n-gram as well as feedforward-based neural language models have to resort to the Markovian assumption to model only a fixed window of context, i.e., $p(y_i|y_{i-n+1}, \ldots, y_{i-1})$. An RNN-based language model naturally lends itself to model the full history as we shall see now.

An RNN-based language model (RNNLM) is a special case of RNNs in which: (a) the input and output are sequences of discrete words, (b) the output sequence ends with a special symbol <eos> that marks the boundary, e.g., $y = \{$ "I", "am", "a", "student", <eos>$\}$, and (c) the input sequence is a shift-by-1 version of the output sequence with <sos> as a starting symbol, e.g., $x = \{$ <sos>, "I", "am", "a", "student"$\}$. We illustrate this in Figure 2.2.

Training Given a training dataset of N discrete output sequences $y^{(1)}, \ldots, y^{(N)}$ with lengths $m_1, \ldots, m_N$ accordingly. The learning objective is to minimize the negative log-likelihood, or the *cross-entropy* loss, of these training examples:

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} -\log p\left(y^{(i)}\right) \tag{2.7}$$

$$= \sum_{i=1}^{N} \sum_{t=1}^{m_i} -\log p\left(y_t^{(i)}|y_{<t}^{(i)}\right) \tag{2.8}$$

RNN learning is often done using mini-batch stochastic gradient descent (SGD) algorithms in which a small set of training examples, a *mini-batch*, is used to compute the gradients and update weights one at a time. Using mini-batches has several advantages: (a) the gradients are more reliable and consistent than the "online" setting which updates per example, (b) less computation is required to update the weights unlike the case of full-batch learning which has to process all examples before updating, and (c) with multiple examples in a mini-batch, one can turn matrix-vector multiplications such as those in Eq. (2.2) and Eq. (2.3) into matrix-matrix multiplications which can be deployed efficiently on GPUs.

The simplest weight update formula with η as a learning rate is given below:

$$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} - \eta \nabla J(\boldsymbol{\theta}) \tag{2.9}$$

Single-timestep Backpropagation To compute the gradients for the loss $J(\boldsymbol{\theta})$, we first need to be able to derive the gradients of the per-timestep loss $l_t = \log \boldsymbol{p}_t(y_t)$ with respect to both the RNN weights $\{\boldsymbol{W}_{xh}, \boldsymbol{W}_{hh}, \boldsymbol{W}_{hy}\}$ and the inputs $\{\boldsymbol{x}_t, \boldsymbol{h}_{t-1}\}$. We denote these gradients as $\{d\boldsymbol{W}_{xh}, d\boldsymbol{W}_{hh}, d\boldsymbol{W}_{hy}, d\boldsymbol{x}_t, d\boldsymbol{h}_{t-1}\}$ respectively and define intermediate gradients $d\boldsymbol{s}_t, d\boldsymbol{h}_t$ similarly. Starting with the loss l_t, we employ backpropagation through structures [18] to derive each gradient one by one in the following order: $l_t \rightarrow \boldsymbol{s}_t \rightarrow \{\boldsymbol{h}_t, \boldsymbol{W}_{hy}\} \rightarrow \{\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{W}_{xh}, \boldsymbol{W}_{hh}\}$. To simplify the math, we will utilize several lemmas and corollaries provided in Appendix A.

First, from Eq. (2.5), we have:

$$d\boldsymbol{s}_t = \frac{\partial l_t}{\partial \boldsymbol{s}_t} = \frac{\partial}{\partial \boldsymbol{s}_t} \left(\boldsymbol{s}_t(y_t) - \log \sum_{y'} e^{\boldsymbol{s}_t(y')} \right) \tag{2.10}$$

Computing per-coordinate gradient $\boldsymbol{s}_t(y)$ gives:

$$\frac{\partial}{\partial \boldsymbol{s}_t(y)} \left(\boldsymbol{s}_t(y_t) - \log \sum_{y'} e^{\boldsymbol{s}_t(y')} \right) = \begin{cases} 1 - \boldsymbol{p}_t(y_t) & y = y_t \\ -\boldsymbol{p}_t(y) & y \neq y_t \end{cases} \tag{2.11}$$

The above gradients can be concisely written in vector form as:

$$d\boldsymbol{s}_t = \mathbf{1}_{y_t} - \boldsymbol{p}_t \tag{2.12}$$

Here, $\boldsymbol{p}_t$ is the probability distribution defined in Eq. (2.4) and has been calculated in the forward pass, so we simply reuse it. $\mathbf{1}_{y_t}$ is a one-hot vector with 1 at position y_t. Applying Corollary 1, noting that $\boldsymbol{s}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t$ in Eq. (2.3), we arrive at:

$$d\boldsymbol{h}_t = \boldsymbol{W}_{hy}^\top \cdot d\boldsymbol{s}_t \tag{2.13}$$

$$d\boldsymbol{W}_{hy} = d\boldsymbol{s}_t \cdot \boldsymbol{h}_t^\top \tag{2.14}$$

At this point, we have derived part of the backpropation procedure which can be applied to any hidden unit type, e.g., the aforementioned vanilla RNN or the LSTM unit that we will describe shortly in the next section.

Vanilla RNN Backpropagation First of all, we can simplify the notation to have $\boldsymbol{T}_{\text{rnn}} = [\boldsymbol{W_{xh}} \boldsymbol{W_{hh}}]$ and $\boldsymbol{z}_t = [\boldsymbol{x}_t; \boldsymbol{h}_{t-1}]$, so the RNN formulation in Eq. (2.2) becomes:

$$\boldsymbol{h_t} = \sigma\left(\boldsymbol{T}_{\text{rnn}} \boldsymbol{z}_t\right) \tag{2.15}$$

Applying Lemma 2, we have:

$$d\boldsymbol{z}_t = \boldsymbol{T}_{\text{rnn}}^{\top} \cdot (\sigma'(\boldsymbol{T}_{\text{rnn}} \boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \tag{2.16}$$

$$d\boldsymbol{T}_{\text{rnn}} = (\sigma'(\boldsymbol{T}_{\text{rnn}} \boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \cdot \boldsymbol{z}_t^{\top} \tag{2.17}$$

This is one of the *tricks* that we use to better utilize GPUs by creating larger matrices and vectors, i.e., $\boldsymbol{T}_{\text{rnn}}$ and $\boldsymbol{z}_t$. From Eq. (2.16) and Eq. (2.17), one can easily extract the following gradients: (a) $d\boldsymbol{x}_t$ – embedding gradients which we use to sparsely update the embedding weights $\boldsymbol{W}_e$, (b) $d\boldsymbol{h}_{t-1}$ – gradients of the previous hidden state, which is needed by the backpropagation-through-time algorithm that we will discuss next, and (c) $d\boldsymbol{W}_{xh}$ as well as $d\boldsymbol{W}_{hh}$ – the RNN input and recurrent connections.[3]

Backpropagation Through Time (BPTT) Having defined a single-timestep backpropagation procedure, we are now ready to go through the BPTT algorithm [55, 68]. Inspired by Sutskever [61], we summarize the BPTT algorithm for RNNs below with the following remarks: (a) Lines 3, 5, 6, 7 accumulate the gradients of RNN weights $\{\boldsymbol{W}_{hy}, \boldsymbol{W}_{xh}, \boldsymbol{W}_{hh}, \boldsymbol{W}_e\}$

[3] One can also separately derive these gradients as follows:

$$d\boldsymbol{x}_t = \boldsymbol{W}_{xh}^{\top} \cdot (\sigma'(\boldsymbol{T}_{\text{rnn}} \boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \tag{2.18}$$

$$d\boldsymbol{h}_{t-1} = \boldsymbol{W}_{hh}^{\top} \cdot (\sigma'(\boldsymbol{T}_{\text{rnn}} \boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \tag{2.19}$$

$$d\boldsymbol{W}_{xh} = (\sigma'(\boldsymbol{T}_{\text{rnn}} \boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \cdot \boldsymbol{x}_t^{\top} \tag{2.20}$$

$$d\boldsymbol{W}_{hh} = (\sigma'(\boldsymbol{T}_{\text{rnn}} \boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \cdot \boldsymbol{h}_{t-1}^{\top} \tag{2.21}$$

over time; (b) In line 7, $d\boldsymbol{x}_t$ refers to gradients of words participating in the current mini-batch which we use to sparsely update $\boldsymbol{W}_e$;[4] and (c) Line 4 accumulates gradients for the current hidden state $\boldsymbol{h}_t$ by considering two paths, a "vertical" one from the current loss at time t and a "recurrent" one from the timestep $t+1$ which was set in Line 8 earlier.

Algorithm 1: BPTT algorithm for "vanilla" RNNs

1 **for** $t = T \rightarrow 1$ **do**
  `// Output backprop`
2 $d\boldsymbol{s}_t \leftarrow \mathbf{1}_{y_t} - \boldsymbol{p}_t$
3 $d\boldsymbol{W}_{hy} \leftarrow d\boldsymbol{W}_{hy} + d\boldsymbol{s}_t \cdot \boldsymbol{h}_t^\top$
4 $d\boldsymbol{h}_t \leftarrow d\boldsymbol{h}_t + \boldsymbol{W}_{hy}^\top \cdot d\boldsymbol{s}_t$
  `// RNN backprop`
5 $d\boldsymbol{W}_{xh} \leftarrow d\boldsymbol{W}_{xh} + (\sigma'(\boldsymbol{T}_{\text{rnn}}\boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \cdot \boldsymbol{x}_t^\top$
6 $d\boldsymbol{W}_{hh} \leftarrow d\boldsymbol{W}_{hh} + (\sigma'(\boldsymbol{T}_{\text{rnn}}\boldsymbol{z}_t) \circ d\boldsymbol{h}_t) \cdot \boldsymbol{h}_{t-1}^\top$
  `// Input backprop`
7 $d\boldsymbol{x}_t \leftarrow \boldsymbol{W}_{xh}^\top \cdot (\sigma'(\boldsymbol{T}_{\text{rnn}}\boldsymbol{z}_t) \circ d\boldsymbol{h}_t)$
8 $d\boldsymbol{h}_{t-1} \leftarrow \boldsymbol{W}_{hh}^\top \cdot (\sigma'(\boldsymbol{T}_{\text{rnn}}\boldsymbol{z}_t) \circ d\boldsymbol{h}_t)$
9 **end**

2.1.2 Better Training RNNs

Even though computing RNN gradients is straightforward once the BPTT algorithm has been plotted out, training is inherently difficult due to the nonlinear iterative nature of RNNs. Among all reasons, the two classic problems of RNNs that often arise when dealing with very long sequences are the *exploding* and *vanishing* gradients as described by Bengio et al. [4]. In short, exploding gradients refers to the phenomenon that the gradients become exponentially large as we backpropagate over time, making learning unstable. Vanishing gradients, on the other hand, is the opposite problem when the gradients go exponentially fast towards zero, turning BPTT into truncated BPTT that is unable to capture long-range dependencies in sequences.

Let us try to explain the aforementioned problems informally and refer readers to more

[4] In multi-layer RNNs, $d\boldsymbol{x}_t$ is used to send gradients down to the below layers.

rigorous and in-depth analyses in [4, 25, 41, 51]. The main cause of these two problems all lies in Line 8 of the BPTT algorithm which can be rewritten as $d\boldsymbol{h}_{t-1} = \boldsymbol{W}_{hh}^{\top} \cdot \text{diag}\left(\sigma'(\boldsymbol{T}_{\text{rnn}}\boldsymbol{z}_t)\right) \cdot d\boldsymbol{h}_t$ (see Lemma 1). We can try to understand the behavior of RNNs over time by assuming for a moment that there is no contribution from intermediate losses, i.e., Line 4 is "ignored". Given such an assumption, a signal backpropagated from the current hidden state over K steps will become $d\boldsymbol{h}_{t-K} = \prod_{i=1}^{K} \left(\boldsymbol{W}_{hh}^{\top} \cdot \text{diag}\left(\sigma'(\boldsymbol{T}_{\text{rnn}}\boldsymbol{z}_{t-i+1})\right)\right) \cdot d\boldsymbol{h}_t$. Assuming that the non-linear function σ is bounded, e.g., sigm and $\tanh$, and behaves "nicely", what we need to deal with now is the multiplication of the recurrent matrix over time. This leads to the fact that the behavior of RNNs is often governed by the characteristics of the recurrent matrix $\boldsymbol{W}_{hh}$ and most analyses examine in terms of the largest eigen value of $\boldsymbol{W}_{hh}$ as well as the norms of these signals. Roughly speaking, if the largest eigen value is large enough, exploding gradients will be likely to happen. On the contrary, if the largest eigen value is below a certain threshold, vanishing gradients will occur as clearly explained by Pascanu et al. [51].

Gradient Clipping In practice, it is generally easy to cope with the exploding gradient problem by applying different forms of gradient clipping. The first approach was proposed by Mikolov [42] through the form of temporal *element-wise* clipping. At each timestep during backpropagation, any elements of $d\boldsymbol{h}$ that are greater than a positive threshold τ or smaller than -τ will be set to τ or -τ respectively. One can also perform gradient *norm* clipping as suggested by Pascanu et al. [51]. The idea is simple: given a final gradient vector $\boldsymbol{g}$ computed per mini-batch, if its norm $||\boldsymbol{g}||$ is greater than a threshold τ, then we will use the following scaled gradient $\frac{\tau}{||\boldsymbol{g}||}\boldsymbol{g}$ instead. The latter approach has been widely used in many systems nowadays and can also be used in conjunction with the former. We take the combined approach in our implementations described later in this thesis.

Long Short-Term Memory The vanishing gradient problem, on the other hand, is more challenging to tackle. There have been many proposed approaches to alleviate the problem such as skip connections [34, 66], hierarchical architectures [14], leaky integrators [27],

second-order methods [41], and regularization [51], to name a few; also, see [6] for a comparison of some of these techniques. Among all, Long Short-term Memory (LSTM), invented by Hochreiter and Schmidhuber [25], appears to be one of the most widely adopted solutions to the vanishing gradient problem. Graves and colleagues deserve credit for popularizing LSTM through a series of work [19, 20, 21]. The key idea of LSTM is to augment RNNs with linear *memory* units that allow the gradient to flow smoothly through time. In addition, there are gating units that control how much an RNN wants to reuse memory (*forget* gates), receive input signal (*input* gates), and extract information (*output* gates) at each timestep. There are many implementation instances of LSTM, differing in terms of whether and which biases are used, how gates are built, etc; however, it turns out that these different choices do not matter much for most cases [22, 28]. As such, in this section and through out this thesis, we will stick to the formulation described in [69].

Instead of jumping directly into the detailed formulation, let us provide intuitions on how to gradually build up an LSTM architecture. First, we can construct a simple memory unit as follows:

$$\boldsymbol{c}_t = \boldsymbol{c}_{t-1} + \sigma\left(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}\right)) \tag{2.22}$$

$$\boldsymbol{h}_t = \boldsymbol{c}_t \tag{2.23}$$

This architecture can be viewed as a form of "leaky" integration mentioned in [6, 61] since it is equivalent to $\boldsymbol{h}_t = \boldsymbol{h}_{t-1} + \sigma(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1})$. Training this network over long sequences is easy since among the exponentially many backpropagation paths, there is exactly one path that goes through all the memory units $\boldsymbol{c}_i$ ($i = \overline{1, T}$) and is guaranteed to not vanish since $d\boldsymbol{c}_t = d\boldsymbol{c}_{t-1}$ along that path.

Such architecture, however, does not account for the fact that certain inputs, e.g., function words or punctuations, are, sometimes, not relevant to the task at hand and should be downweighted. Occasionally, we might also want to reset the memory, e.g., at the beginning of each sentence in a paragraph. To add more flexibility and power to this architecture,

the LSTM adds forget, input, and output gates as follows:

$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \sigma\left(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}\right) \tag{2.24}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \sigma\left(\boldsymbol{c}_t\right)) \tag{2.25}$$

We note that, in Eq. (2.25), the memory cell $\boldsymbol{c}_t$ is passed through a nonlinear function σ before the output gate $\boldsymbol{o}_t$ is used to extract relevant information in the hope for better information retrieval. As an evidence, Greff et al. [22] have shown that such a output nonlinearity is critical to the performance of an LSTM. Moving on, to ensure that the gates are adaptive, we build them from the information given by the current input $\boldsymbol{x}_t$ and the previous hidden state $\boldsymbol{h}_{t-1}$. We also want the gates to be in $[0, 1]$, so sigm will be used. All of these desiderata lead to the below LSTM formulation described in [69] in which σ is chosen to be $\tanh$:

$$\begin{pmatrix} \boldsymbol{i}_t \\ \boldsymbol{f}_t \\ \boldsymbol{o}_t \\ \hat{\boldsymbol{h}}_{\boldsymbol{t}} \end{pmatrix} = \begin{pmatrix} \mathrm{sigm} \\ \mathrm{sigm} \\ \mathrm{sigm} \\ \tanh \end{pmatrix} \begin{bmatrix} \boldsymbol{W}_{xi}\boldsymbol{W}_{hi} \\ \boldsymbol{W}_{xf}\boldsymbol{W}_{hf} \\ \boldsymbol{W}_{xo}\boldsymbol{W}_{ho} \\ \boldsymbol{W}_{xh}\boldsymbol{W}_{hh} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} \tag{2.26}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \hat{\boldsymbol{h}}_{\boldsymbol{t}} \tag{2.27}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t) \tag{2.28}$$

Following the same spirit as Eq. (2.15), we can be GPU-efficient with Eq. (2.26) since the 8 different submatrices is grouped into a single big matrix, which we call $\boldsymbol{T}_{\text{lstm}}$. Let $\boldsymbol{z}_t = [\boldsymbol{x}_t; \boldsymbol{h}_{t-1}]$, what we do is first multiply $\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t$ and then apply different non-linear functions to corresponding parts of the output. For the ease of deriving backpropagation equations later, we can rewrite Eq. (2.26) as:

$$\boldsymbol{u}_t = g(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \tag{2.29}$$

$$= g(\boldsymbol{T}_{\text{x}}\boldsymbol{x}_t + \boldsymbol{T}_{\text{h}}\boldsymbol{h}_{t-1}) \tag{2.30}$$

Here, g is a non-linear function applied element-wise and we define g loosely in the sense that it uses $\tanh$ only for the vector part corresponding to $\hat{\boldsymbol{h}}_{\boldsymbol{t}}$ and sigm for the rest.

LSTM Training In the LSTM training pipeline, there are many components that are exactly the same or very similar to RNN training. We will now highlight some key differences. First of all, LSTM extends the recurrence function to have not just the hidden states but also the memory cells as both inputs and outputs. The definition is as below:

$$(\boldsymbol{h}_t, \boldsymbol{c}_t) = f\left(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{c}_{t-1}\right) \tag{2.31}$$

In our case, the abstract function f is implemented by Eq. 2.26-2.28. Once $\boldsymbol{h}_t$ is computed, the prediction process is the same as that of RNNs which is given by Eq. 2.3-2.5. The training objective in Eq. (2.8) remains unchanged as well.

LSTM Backpropagation Since the prediction procedure is the same, LSTM backpropagation pipeline mimics that of RNNs up to Eq. (2.13) and Eq. (2.14), which computes $d\boldsymbol{h}_t$ and $d\boldsymbol{W}_{hy}$ respectively.

Given $d\boldsymbol{h}_t$, we now work backwark to derive other gradients. First, starting from Eq. (2.28) and by applying Lemma 3, we have:

$$d\boldsymbol{o}_t = \tanh(\boldsymbol{c}_t) \circ d\boldsymbol{h}_t \tag{2.32}$$

$$d\boldsymbol{c}_t = \tanh'(\boldsymbol{c_t}) \circ \boldsymbol{o_t} \circ d\boldsymbol{h}_t \tag{2.33}$$

Before backpropagating Eq. (2.27), once must *remember* to update $d\boldsymbol{c}_t$ with the gradient sent back from $\boldsymbol{c}_{t+1}$, which is accomplished by Lines 6 and 10 of Algorithm 2. Given the updated $d\boldsymbol{c}_t$, we apply Corollary 2 to derive:

$$d\boldsymbol{f}_t = \boldsymbol{c}_{t-1} \circ d\boldsymbol{c}_t \tag{2.34}$$

$$d\boldsymbol{c}_{t-1} = \boldsymbol{f}_t \circ d\boldsymbol{c}_t \tag{2.35}$$

$$d\boldsymbol{i}_t = \hat{\boldsymbol{h}}_{\boldsymbol{t}} \circ d\boldsymbol{c}_t \tag{2.36}$$

$$d\hat{\boldsymbol{h}}_{\boldsymbol{t}} = \boldsymbol{i}_t \circ d\boldsymbol{c}_t \tag{2.37}$$

Let $d\boldsymbol{u}_t = [d\boldsymbol{i}_t; d\boldsymbol{f}_t; d\boldsymbol{o}_t; d\hat{\boldsymbol{h}}_{\boldsymbol{t}}]$ (vertical concatenation), we are now ready to backpropagate through Eq. (2.30). In a similar manner as RNNs, Eq. 2.18-2.21, we arrive at:

$$d\boldsymbol{x}_t = \boldsymbol{T}_{\text{x}}^\top \cdot (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t) \tag{2.38}$$

$$d\boldsymbol{h}_{t-1} = \boldsymbol{T}_{\text{h}}^\top \cdot (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t) \tag{2.39}$$

$$d\boldsymbol{T}_{\text{x}} = (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t) \cdot \boldsymbol{x}_t^\top \tag{2.40}$$

$$d\boldsymbol{T}_{\text{h}} = (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t) \cdot \boldsymbol{h}_{t-1}^\top \tag{2.41}$$

All of these gradients can now be put together in the below BPTT algorithm for LSTM:

Algorithm 2: BPTT algorithm for LSTM

for $t = T \rightarrow 1$ **do**

`// Output backprop`

$d\boldsymbol{s}_t \leftarrow \mathbf{1}_{y_t} - \boldsymbol{p}_t$

$d\boldsymbol{W}_{hy} \leftarrow d\boldsymbol{W}_{hy} + d\boldsymbol{s}_t \cdot \boldsymbol{h}_t^\top$

$d\boldsymbol{h}_t \leftarrow d\boldsymbol{h}_t + \boldsymbol{W}_{hy}^\top \cdot d\boldsymbol{s}_t$

`// LSTM backprop`

$d\boldsymbol{o}_t \leftarrow \tanh(\boldsymbol{c}_t) \circ d\boldsymbol{h}_t$

$d\boldsymbol{c}_t \leftarrow d\boldsymbol{c}_t + \tanh'(\boldsymbol{c}_t) \circ \boldsymbol{o}_t \circ d\boldsymbol{h}_t$; `// Already included` $d\boldsymbol{c}_{t+1}$

$d\boldsymbol{f}_t \leftarrow \boldsymbol{c}_{t-1} \circ d\boldsymbol{c}_t$

$d\boldsymbol{i}_t \leftarrow \hat{\boldsymbol{h}}_{\boldsymbol{t}} \circ d\boldsymbol{c}_t$

$d\hat{\boldsymbol{h}}_{\boldsymbol{t}} \leftarrow \boldsymbol{i}_t \circ d\boldsymbol{c}_t$

$d\boldsymbol{c}_{t-1} \leftarrow \boldsymbol{f}_t \circ d\boldsymbol{c}_t$; `// Compute` $d\boldsymbol{c}_{t-1}$

$d\boldsymbol{u}_t = [d\boldsymbol{i}_t; d\boldsymbol{f}_t; d\boldsymbol{o}_t; d\hat{\boldsymbol{h}}_{\boldsymbol{t}}]$

$d\boldsymbol{T}_{\text{x}} \leftarrow (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t) \cdot \boldsymbol{x}_t^\top$

$d\boldsymbol{T}_{\text{h}} \leftarrow (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t) \cdot \boldsymbol{h}_{t-1}^\top$

`// Input backprop`

$d\boldsymbol{x}_t \leftarrow \boldsymbol{T}_{\text{x}}^\top \cdot (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t)$

$d\boldsymbol{h}_{t-1} \leftarrow \boldsymbol{T}_{\text{h}}^\top \cdot (g'(\boldsymbol{T}_{\text{lstm}}\boldsymbol{z}_t) \circ d\boldsymbol{u}_t)$

end

2.2 Neural Machine Translation

Having introduced recurrent language models, one can simply think of neural machine translation (NMT) as a recurrent language model that conditions on the source sentence. More formally, NMT aims to directly model the conditional probability $p(y|x)$ of translating a source sentence, $x_1, \ldots, x_n$, to a target sentence, $y_1, \ldots, y_m$. It accomplishes this goal through an *encoder-decoder* or *sequence-to-sequence* framework [11, 29, 63]. The *encoder* computes a representation $\boldsymbol{s}$ for each source sentence. Based on that source representation, the *decoder* generates a translation, one target word at a time, and hence, decomposes the log conditional probability as:

$$\log p(y|x) = \sum_{t=1}^{m} \log p\left(y_t|y_{<t}, \boldsymbol{s}\right) \tag{2.42}$$

NMT models vary in terms of the exact architectures to use. A natural choice for sequential data is the recurrent neural network (RNN), used by most of the recent NMT work and for both the encoder and decoder. RNN models, however, differ in terms of: (a) *directionality* – unidirectional or bidirectional; (b) *depth* – single or deep multi-layer; and (c) *type* – often either a vanilla, an LSTM [25], or a gated recurrent unit (GRU) [11]. In general, for the encoder, almost any architecture can be used since we have fully observed the source sentence. For example, Kalchbrenner and Blunsom [29] used a convolutional neural network for encoding the source. Choices on the decoder side are more limited since we need to be able to generate a translation. At the time of this thesis, the most popular choice is a unidirectional RNN, which simplifies the beam-search decoding algorithm by producing translations from left to right.

In this thesis, all our NMT models are deep multi-layer RNNs which are unidirectional and have LSTM as the recurrent unit. We show an example of such model in Figure 2.3 though it should be easy to extend to other RNN architectures. In this example, we train our model to translate a source sentence "I am a student" into a target one "Je suis étudiant". At a high level, our NMT models consist of two recurrent language models as described in Section (2.1.1): the *encoder* RNN simply consumes the input source words without making any prediction; the *decoder*, on the other hand, processes the target sentence while predicting the next words.

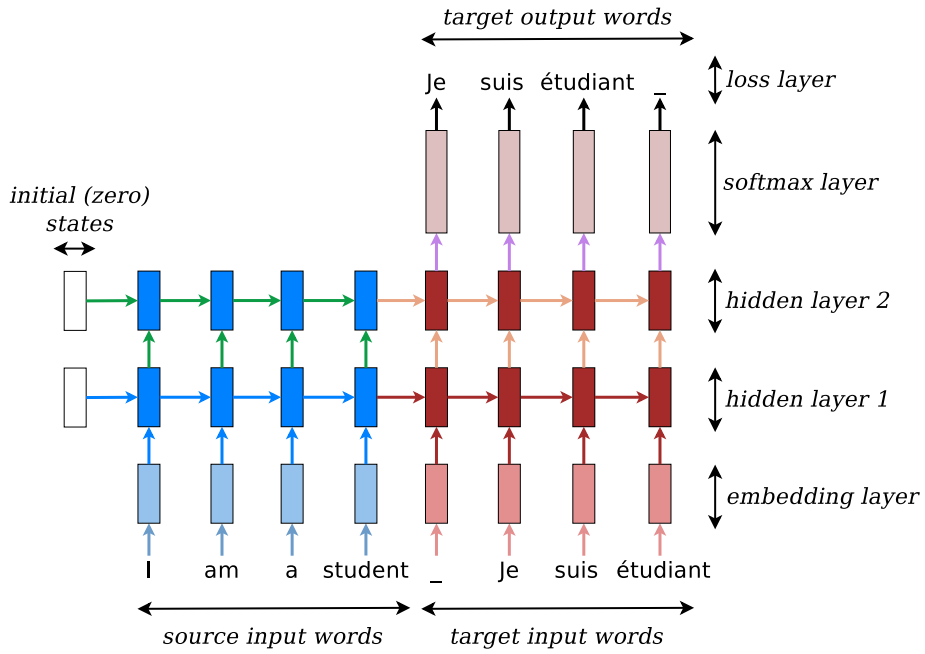

Figure 2.3: **Neural machine translation** – example of a deep recurrent architecture proposed by Sutskever et al. [63] for translating a source sentence "I am a student" into a target sentence "Je suis étudiant". Here, "_" marks the end of a sentence.

Highlight that the parameters of the RNN are all changed when moving from the blue part to the red part. It is 2 RNNs glued together.

In more detail, at the bottom layer, the encoder and decoder RNNs receive as *input* the following: first, the source sentence, then a boundary marker "_" which indicates the transition from the encoding to the decoding mode, and the target sentence. Given these discrete words, the model looks up the source and target embeddings to retrieve the corresponding word representations. For this *embedding layer* to work, a vocabulary is chosen for each language, and often the top V frequent words are selected. These embedding weights, one set per language, are learned during training. While one can choose to initialize embedding weights with pretrained word representations, such as word2vec [46] and Glove [53], we found, in this thesis, that these embeddings can be initialized randomly and learned from scratch given large training datasets.

Once retrieved, the word embeddings are then fed as input into the main network, which consists of two multi-layer RNNs 'stuck together' — an encoder for the source language and a decoder for the target language. The encoder RNN uses zero vectors as its starting states. The decoder, on the other hand, needs to have access to the source information, so

one simple way to achieve that is to initialize it with the last hidden state of the encoder.[5] In Figure 2.3, we pass the hidden state at the source word "student" to the decoder side. The *feed-forward* (vertical) weights connect the hidden unit from the layer below to the upper one; whereas, the *recurrent* (horizontal) weights transfer the history knowlege from the previous timestep to the next one. Often, we use different weights across the encoder and decoder as well as across different layers; in the current example, we have 4 different LSTM weight sets $\boldsymbol{T}_{\text{lstm}}$, detailed in Eq. (2.29), over $\{\text{encoder, decoder}\} \times \{1^{\text{st}}, 2^{\text{nd}} \text{ layer}\}$. Finally, for each target word, the hidden state at the top layer is transformed by the *softmax* weights into a probability distribution over the target vocabulary of size V according to Eq. (2.3) and Eq. (2.4).

Training Training neural machine translation is similar to training a recurrent language model that we have discussed in Section (2.1) except that we need to handle the conditioning part on source sentences. The training objective for NMT is formulated as:

$$J = \sum_{(x,y)\in\mathbb{D}} -\log p(y|x) \tag{2.43}$$

Here, $\mathbb{D}$ refers to our parallel training corpus of source and target sentence pairs (x, y). Given the aforementioned NMT architecture, computing the NMT loss for (x, y) during the *forward* pass is almost the same as how we compute the regular RNN loss on just y. The only difference is that we have to first compute representations for the source sentence x to initialize the decoder RNN instead of just starting from zero states. For the *backpropagation* phase, computing gradients for the decoder is the same as what we have described in Algorithm 2 for regular RNNs. The last hidden-state gradient from the decoder is passed back to the encoder. We then continue backpropating through the encoder in a similar fashion as that of the decoder but without any prediction losses.

More concretely, we present in Algorithm 3 details in the forward pass of an NMT model which uses a deep multi-layer LSTM architecture. Since the encoder and decoder share many operations in common, we combine both the source sentence x (length m_x) and the target sentence y (length m_y) together to form an input sequence s as shown in Line 1,

[5] This is not the only way to initialize the decoder, e.g., Cho et al. [11] connect the last encoder state to every timestep in the decoder.

~~which also includes~~ the end-of-sentence marker "_". We first start with the encoder weights and initial states set to zero (Line 2-3). The algorithm switches to the decoder mode at time $m_x + 1$ (Line 5). The same LSTM codebase (Line 8-11) is used for both the encoder and decoder in which embeddings are first looked up for the input s_t; after that, hidden states as well as LSTM cell memories are built from the bottom layer to the top one (the L^{th} layer). In Line 10, `LSTM` refers to the entire formulation in Eq 2.26-2.28, which one can easily replace with other hidden units such as RNN and GRU. Lastly, on the decoder side, the top hidden state is used to predict the next symbol s_{t+1} (Line 13); then, a loss value l_t and a probability distribution $\boldsymbol{p}_t$ computed according to Eq 2.3-2.4 are returned.

Algorithm 3: NMT training algorithm – *forward* pass.

1 $s \leftarrow [x, \_, y, \_]$; `// Length of` s `is` $m_x + 1 + m_y + 1$
2 $\boldsymbol{W}_e, \boldsymbol{T}_{\text{lstm}}^{(1..L)} \leftarrow \boldsymbol{W}_e^{\text{encoder}}, \boldsymbol{T}_{\text{lstm}}^{\text{encoder}}$; `// Encoder weights`
3 $\boldsymbol{h}_0^{(1..L)}, \boldsymbol{c}_0^{(1..L)} \leftarrow \boldsymbol{0}$; `// Zero init`
4 **for** $t = 1 \rightarrow (m_x + 1 + m_y)$ **do**
  `// Decoder transition`
5 **if** $t == (m_x + 1)$ **then**
6 $\boldsymbol{W}_e, \boldsymbol{T}_{\text{lstm}}^{(1..L)} \leftarrow \boldsymbol{W}_e^{\text{decoder}}, \boldsymbol{T}_{\text{lstm}}^{\text{decoder}}$;
7 **end**
  `// Multi-layer LSTM`
8 $\boldsymbol{h}_t^{(0)} \leftarrow \texttt{Emb\_LookUp}(s_t, \boldsymbol{W}_e)$;
9 **for** $l = 1 \rightarrow L$ **do**
10 $\boldsymbol{h}_t^{(l)}, \boldsymbol{c}_t^{(l)} \leftarrow \texttt{LSTM}\left(\boldsymbol{h}_{t-1}^{(l)}, \boldsymbol{c}_{t-1}^{(l)}, \boldsymbol{h}_t^{(l-1)}, \boldsymbol{T}_{\text{lstm}}^{(l)}\right)$; `// LSTM hidden unit`
11 **end**
  `// Target-side prediction`
12 **if** $t \geq (m_x + 1)$ **then**
13 $l_t, \boldsymbol{p}_t \leftarrow \texttt{Predict}(s_{t+1}, \boldsymbol{h}_t^{(L)}, \boldsymbol{W}_{hy})$;
14 **end**
15 **end**

Next, we describe details of the backpropagation step in Algorithm 4. A quick glance through the algorithm reveals many similarities compared to the forward pass algorithm except that we have reversed the procedue. First, we start with the decoder weights and initialize all gradients to zero (Line 1-2). At time m_x, we switch to the encoder mode while

Algorithm 4: NMT training algorithm – *backpropagation* pass.

$\boldsymbol{W}_e, \boldsymbol{T}_{\text{lstm}}^{(1..L)} \leftarrow \boldsymbol{W}_e^{\text{decoder}}, \boldsymbol{T}_{\text{lstm}}^{\text{decoder}}$; // Decoder weights

$d\boldsymbol{h}^{(1..L)}, d\boldsymbol{c}^{(1..L)}, d\boldsymbol{T}_{\text{lstm}}^{(1..L)}, d\boldsymbol{W}_e, d\boldsymbol{W}_{hy} \leftarrow \mathbf{0}$; // Zero init

for $t = (m_x + 1 + m_y) \rightarrow 1$ **do**

  // Encoder transition

  if $t == m_x$ **then**

    $\boldsymbol{W}_e, \boldsymbol{T}_{\text{lstm}}^{(1..L)} \leftarrow \boldsymbol{W}_e^{\text{encoder}}, \boldsymbol{T}_{\text{lstm}}^{\text{encoder}}$;

    $d\boldsymbol{W}_e^{\text{decoder}}, d\boldsymbol{T}_{\text{lstm}}^{\text{decoder}} \leftarrow d\boldsymbol{W}_e, d\boldsymbol{T}_{\text{lstm}}^{(1..L)}$; // Save decoder gradients

    $d\boldsymbol{T}_{\text{lstm}}^{(1..L)}, d\boldsymbol{W}_e \leftarrow \mathbf{0}$;

  end

  // Target-side prediction

  if $t \geq (m_x + 1)$ **then**

    $d\boldsymbol{h}, d\boldsymbol{W} \leftarrow \texttt{Predict\_grad}(s_{t+1}, \boldsymbol{p}_t, \boldsymbol{h}_t^{(L)}, \boldsymbol{W}_{hy})$;

    $d\boldsymbol{h}^{(L)} \leftarrow d\boldsymbol{h}^{(L)} + d\boldsymbol{h}$;

    $d\boldsymbol{W}_{hy} \leftarrow d\boldsymbol{W}_{hy} + d\boldsymbol{W}$;

  end

  // Multi-layer LSTM

  for $l = L \rightarrow 1$ **do**

    $d\boldsymbol{h}^{(l)}, d\boldsymbol{c}^{(l)}, d\boldsymbol{x}, d\boldsymbol{T} \leftarrow \texttt{LSTM\_grad}\left(d\boldsymbol{h}^{(l)}, d\boldsymbol{c}^{(l)}, \boldsymbol{h}_{t-1}^{(l)}, \boldsymbol{c}_{t-1}^{(l)}, \boldsymbol{h}_t^{(l-1)}, \boldsymbol{T}_{\text{lstm}}^{(l)}\right)$;

    $d\boldsymbol{h}^{(l-1)} \leftarrow d\boldsymbol{h}^{(l-1)} + d\boldsymbol{x}$;

    $d\boldsymbol{T}_{\text{lstm}}^{(l)} \leftarrow d\boldsymbol{T}_{\text{lstm}}^{(l)} + d\boldsymbol{T}$;

  end

  $d\boldsymbol{W}_e \leftarrow \texttt{Emb\_grad\_update}(s_t, d\boldsymbol{h}^{(0)}, d\boldsymbol{W}_e)$;

end

$d\boldsymbol{W}_e^{\text{encoder}}, d\boldsymbol{T}_{\text{lstm}}^{\text{encoder}} \leftarrow d\boldsymbol{W}_e, d\boldsymbol{T}_{\text{lstm}}^{(1..L)}$; // Save encoder gradients

I'm not sure this is all correct. Where does $dh^{(l)}$ feed into $dh^{(l-1)}$? Or $dc_t^{(l)}$ into $dc_{(t-1)}^{(l)}$? I could understand Fig. 2. This seems less clear.

saving the currently accumulated LSTM and embedding gradients for the decoder (Line 5-7). Thanks to the backpropagation procedure presented earlier for LSTM, we can simplify the core NMT gradient computation (Line 9-19) by making the following two referents: (a) `Predict_grad` (Line 2-4 of Algorithm 2) which computes gradients for the target-side losses with respect to the hidden states at the top layer and the softmax weights $\boldsymbol{W}_{hy}$; and (b) `LSTM_grad` (Line 5-15 of Algorithm 2) which computes gradients for inputs to LSTM and the LSTM weights per layer $\boldsymbol{T}^{(l)}_{\text{lstm}}$. It is important to note that in Lines 11 and 16 of Algorithm 4, we add the gradients (flowed vertically from either the loss or the upper LSTM layer) to the gradient of the below layer (which already contains the gradient backpropagated horizontally) instead of overriding it. Lastly, in Line 19, we perform sparse updates on the corresponding embedding matrix for participating words only.

2.2.1 Testing

Having trained an NMT model, we, of course, need to be able to use it to translate, or decode, unseen source sentences! This section explains a few different ways to accomplish this goal and how to decode with an ensemble of models.

The simplest strategy to translate a source sentence is to perform *greedy decoding* which we illustrate in Figure 2.4. The idea is simple: (a) we first encode the source sentence, "I am a student" in our example, similar to the training process; (b) the decoding process is started as soon as an end-of-sentence marker "_" for the source sentence is fed as an input; and (c) for each timestep on the decoder side, we pick the most likely word (a greedy choice), e.g., "moi" has the highest translation probability in the first decoding step, then use it as an input to the next timestep, and continue until the end-of-sentence marker "_" is produced as an output symbol. Step (c) is what makes testing different from training: unlike training in which correct target words in y are always fed as an input, testing, on the other hand, uses words predicted by the model.

More concretely, we adopt the NMT forward algorithm to arrive at the greedy decoding strategy in Algorithm 5. We present the greedy algorithm in a slightly more abstract way by reusing elements of the NMT forward pass in Algorithm 3. First, we run through the encoder in Line 1 to obtain a representation $\boldsymbol{h}_0, \boldsymbol{c}_0$ for the source sentence x (length m_x).

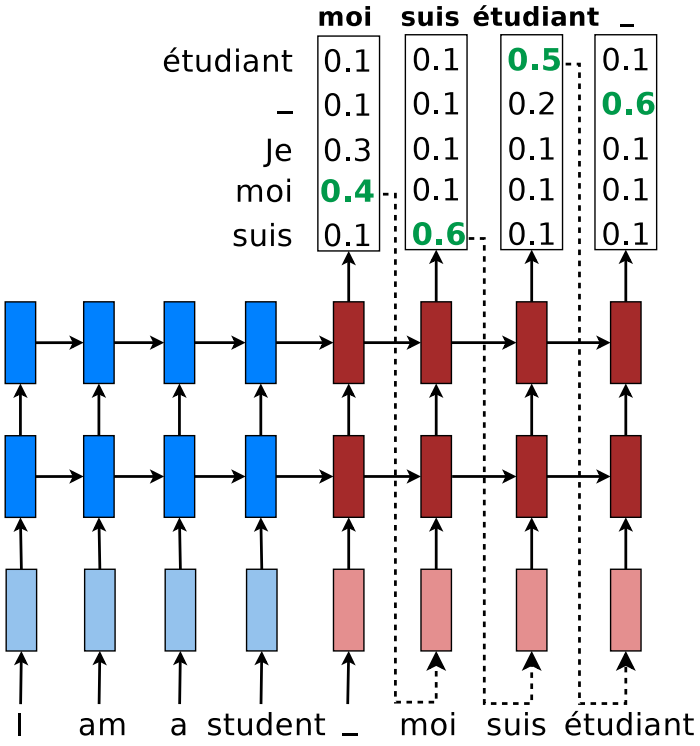

Figure 2.4: **Greedy Decoding** – example of how a trained NMT model produces a translation for a source sentence "I am a student" using greedy search.

We then use the end-of-sentence marker "_" as an input to start the decoding process and restrict the final translation to have a maximum length of $\alpha * m_x$.[6] At each timestep on the decoder side, we call `MultiLayerLSTM`, which refers to Line 8-11 in Algorithm 3, to build up representations over L stacking LSTM layers. The hidden state at the top layer is used to compute the predictive distribution $\boldsymbol{p}_t$ from which we make a greedy choice to produce the index of the translation word at that timestep (Line 7). The process ends when we have produced the marker "_" as a translation word or when the translation length exceeds the length threshold.

For NMT, it turns out that such a simple strategy of greedy decoding can produce very good translations [63]. However, to achieve better result, a more popular strategy is to use *beam-search* decoding algorithm which has been the core of phrase-based statistical machine translation for years [31]. However, unlike phrase-based SMT, NMT has a much

[6] We often set α to 1.5.

Algorithm 5: NMT *greedy* decoding algorithm.

1 $\boldsymbol{h}_0, \boldsymbol{c}_0 \leftarrow \text{Encoder}(x, \boldsymbol{W}_e^{\text{encoder}}, \boldsymbol{T}_{\text{lstm}}^{\text{encoder}})$;
2 $t \leftarrow 1$;
3 $y_1 \leftarrow \_$;
4 **while** $t \leq \alpha * m_x$ **do** // Length factor $\alpha \geq 1$
5 $\quad \boldsymbol{h}_t, \boldsymbol{c}_t \leftarrow \texttt{MultiLayerLSTM}\left(\boldsymbol{h}_{t-1}, \boldsymbol{c}_{t-1}, y_t, \boldsymbol{W}_e^{\text{decoder}}, \boldsymbol{T}_{\text{lstm}}^{\text{decoder}}\right)$;
6 $\quad \boldsymbol{p}_t \leftarrow \texttt{Softmax}(\boldsymbol{h}_t^{(L)}, \boldsymbol{W}_{hy})$;
7 $\quad y_{t+1} \leftarrow \text{argmax}_i\, \boldsymbol{p}_t(i)$; // ***Greedy*** choice
8 $\quad$ **if** $y_{t+1} == \text{Index}(\_)$ **then** // Ending condition
9 $\quad\quad$ break;
10 $\quad$ **end**
11 $\quad t \leftarrow t + 1$
12 **end**
13 **return** $y_{2..t}$

PBSMT generates one phrase at a time from left to right, so NMT isn't really simpler in this respect.

simpler beam-search decoding algorithm since it generates translations word-by-word from left to right. One can modify the greedy decoding algorithm as follows to build a beam-search decoder: (a) at each timestep on the decoder side, we keep track of the top B (the beam size) best translations together with their corresponding hidden states; (b) in Line 7 of Algorithm 5, instead of applying argmax, we select the top B most likely words; and (c) given B previous best translation $\times B$ best words, we select a new set of B best translations for the current timestep based on the combined scores (previous translation scores + current word translation scores). Extra care needs to be taken to make sure that in step (c) we select correct hidden states for the new set of B best translations. Sutskever et al. [63] observed that for NMT, a minimal beam size of 2 already provides a significant boost in translation quality. A beam of size 10 is often used, which is significant smaller that what phrase-based SMT tends to use (> 1000). Maybe omit or broaden range. It's often only 100-200.

Lastly, to achieve the very best result, one simple strategy which has been widely adopted for deep neural networks is to use an ensemble of models. For NMT decoding, using multiple models is pretty straightforward. The idea is that each model produces a distribution at each timestep in the decoder (Line 6 of Algorithm 5). These different distributions are then averaged to produce a new ensemble distribution which we can use for both greedy and beam-search decoders as if we decode from a single model.

The "simplicity" comes only by not exploring different places to pay attention and maintaining coverage set... ideas re-emerging in NMT!

Need some short paragraph concluding the chapter.

Chapter 3

Copy Mechanisms

Chapter 4

Attention Mechanisms

Chapter 5

Hybrid Models

Chapter 6

NMT Future

Chapter 7

Conclusion

Appendix A

Miscellaneous

Lemma 1. *Let $\boldsymbol{u}$, $\boldsymbol{v}$ be any vectors and $\circ$ be element-wise vector multiplication, we have:*

$$diag(\boldsymbol{u}) \cdot \boldsymbol{v} = \boldsymbol{u} \circ \boldsymbol{v} \tag{A.1}$$

Lemma 2. *Let l be a loss value that in which we already knew how to compute its gradient $d\boldsymbol{v}$ with respect to a vector $\boldsymbol{v}$. Given that $\boldsymbol{v} = f(\boldsymbol{W}\boldsymbol{h})$, the gradients $d\boldsymbol{h}, d\boldsymbol{W}$ of the loss l with respect to the vector $\boldsymbol{h}$ and the matrix $\boldsymbol{W}$ can be derived as follows:*

$$d\boldsymbol{h} = \boldsymbol{W}^\top \cdot (f'(\boldsymbol{W}\boldsymbol{h}) \circ d\boldsymbol{v}) \tag{A.2}$$

$$d\boldsymbol{W} = (f'(\boldsymbol{W}\boldsymbol{h}) \circ d\boldsymbol{v}) \cdot \boldsymbol{h}^\top \tag{A.3}$$

Proof. Let $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{h}$, we have the following derivations:

$$\begin{aligned}
d\boldsymbol{h} &= \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{h}} \cdot \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{z}} \cdot d\boldsymbol{v} && \text{[Vector calculus chain rules]} \\
&= \frac{\partial \boldsymbol{W}\boldsymbol{h}}{\partial \boldsymbol{h}} \cdot \frac{\partial f(\boldsymbol{z})}{\partial \boldsymbol{z}} \cdot d\boldsymbol{v} \\
&= \boldsymbol{W}^\top \cdot \text{diag}\left(f'(\boldsymbol{z})\right) \cdot d\boldsymbol{v} \\
&= \boldsymbol{W}^\top \cdot (f'(\boldsymbol{W}\boldsymbol{h}) \circ d\boldsymbol{v}) && \text{[Lemma 1]}
\end{aligned}$$

Let $\boldsymbol{w}_i^\top$ be the i$^{\text{th}}$ row vector of matrix $\boldsymbol{W}$ and v_i, z_i be the i$^{\text{th}}$ elements of vectors $\boldsymbol{v}, \boldsymbol{z}$.

Also denoting $d\boldsymbol{w}_i, dv_i$ to be the gradients of l with respect to $\boldsymbol{w}_i, v_i$, we have:

$$\begin{aligned}
d\boldsymbol{w}_i &= \frac{\partial z_i}{\partial \boldsymbol{w}_i} \cdot \frac{\partial v_i}{\partial z_i} \cdot dv_i && \text{[Vector calculus chain rules]} \\
&= \frac{\partial \boldsymbol{w}_i^\top \boldsymbol{h}}{\partial \boldsymbol{w}_i} \cdot f'(z_i) \cdot dv_i \\
&= \boldsymbol{h} \cdot f'(z_i) \cdot dv_i \\
d\boldsymbol{w}_i^\top &= (f'(z_i) \cdot dv_i) \cdot \boldsymbol{h}^\top && \text{[Tranposing]} \\
d\boldsymbol{W} &= (f'(\boldsymbol{W}\boldsymbol{h}) \circ d\boldsymbol{v}) \cdot \boldsymbol{h}^\top && \text{[Concatenating row derivatives]}
\end{aligned}$$

□

Corollary 1. *As a special case of Lemma 2, when f is an identity function, i.e., $\boldsymbol{v} = \boldsymbol{W}\boldsymbol{h}$, we have:*

$$d\boldsymbol{h} = \boldsymbol{W}^\top \cdot d\boldsymbol{v} \tag{A.4}$$

$$d\boldsymbol{W} = d\boldsymbol{v} \cdot \boldsymbol{h}^\top \tag{A.5}$$

Lemma 3. *Let $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{s}$ be any vectors such that $\boldsymbol{s} = \boldsymbol{u} \circ f(\boldsymbol{v})$. Also, let $d\boldsymbol{u}, d\boldsymbol{v}, d\boldsymbol{s}$ be the gradients of a loss l with respect to the corresponding vectors. We have:*

$$d\boldsymbol{u} = f(\boldsymbol{v}) \circ d\boldsymbol{s} \tag{A.6}$$

$$d\boldsymbol{v} = f'(\boldsymbol{v}) \circ \boldsymbol{u} \circ d\boldsymbol{s} \tag{A.7}$$

Corollary 2. *As a special case of Lemma 3 when f is an identity function, i.e., $\boldsymbol{s} = \boldsymbol{u} \circ \boldsymbol{v}$. We have:*

$$d\boldsymbol{u} = \boldsymbol{v} \circ d\boldsymbol{s} \tag{A.8}$$

$$d\boldsymbol{v} = \boldsymbol{u} \circ d\boldsymbol{s} \tag{A.9}$$

Bibliography

[1] Michael Auli, Michel Galley, Chris Quirk, and Geoffrey Zweig. Joint language and translation modeling with recurrent neural networks. In *ACL*, 2013.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[3] Yoshua Bengio and Jean-Sébastien Senécal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Trans. Neural Networks*, 19(4):713–722, 2008.

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2): 157–166, 1994.

[5] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *JMLR*, 3:1137–1155, 2003.

[6] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *ICASSP*, 2013.

[7] Christopher M. Bishop. Mixture density networks. Technical report, Aston University, 1994.

[8] Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 06 1993.

[9] Daniel Cer, Michel Galley, Daniel Jurafsky, and Christopher D. Manning. Phrasal: A statistical machine translation toolkit for exploring new model features. In *ACL, Demonstration Session*, 2010.

[10] David Chiang. Hierarchical phrase-based translation. *Computational Linguistics*, 33 (2):201–228, 2007.

[11] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

[12] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *ACL*, 2014.

[13] Chris Dyer, Jonathan Weese, Hendra Setiawan, Adam Lopez, Ferhan Ture, Vladimir Eidelman, Juri Ganitkevitch, Phil Blunsom, and Philip Resnik. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *ACL, Demonstration Session*, 2010.

[14] Salah El Hihi and Yoshua Bengio. Hierarchical recurrent neural networks for long-term dependencies. In *NIPS*, 1996.

[15] Jeffrey L. Elman. Finding structure in time. In *Cognitive Science*, 1990.

[16] Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. IRSTLM: an open source toolkit for handling large scale language models. In *Interspeech*, 2008.

[17] Mikel L. Forcada and Ramón Neco. Recursive hetero-associative memories for translation. *Biological and Artificial Computation: From Neuroscience to Technology*, pages 453–462, 1997.

[18] C. Goller and A. Kchler. Learning task-dependent distributed representations by backpropagation through structure. *IEEE Transactions on Neural Networks*, 1:347–352, 1996.

[19] A. Graves. Generating sequences with recurrent neural networks. In *Arxiv preprint arXiv:1308.0850*, 2013.

[20] Alex Graves and Juergen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *NIPS*. 2009.

[21] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6): 602–610, 2005.

[22] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[23] Kenneth Heafield. KenLM: faster and smaller language model queries. In *WMT*, 2011.

[24] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. Scalable modified Kneser-Ney language model estimation. In *ACL*, 2013.

[25] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[26] W. John Hutchins. Machine translation: A concise history, 2007.

[27] Herbert Jaeger, Mantas Lukoševičius, Dan Popovici, and Udo Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Networks*, 20(3):335–352, 2007.

[28] Rafal Józefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *ICML*, 2015.

[29] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *EMNLP*, 2013.

[30] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`, 2015. Accessed: 2016-07-05.

[31] Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *NAACL*, 2003.

[32] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *ACL, Demonstration Session*, 2007.

[33] Percy Liang, Alexandre Bouchard-Côté, Dan Klein, and Ben Taskar. An end-to-end discriminative approach to machine translation. In *ACL*, 2006.

[34] Tsungnan Lin, Bill G. Horne, Peter Tiňo, and C. Lee Giles. Learning long-term dependencies in narx recurrent neural networks. *IEEE Transactions on Neural Networks*, 7(6):1329–1338, 1996.

[35] Minh-Thang Luong and Christopher D. Manning. Stanford neural machine translation systems for spoken language domain. In *IWSLT*, 2015.

[36] Minh-Thang Luong and Christopher D. Manning. Achieving open vocabulary neural machine translation with hybrid word-character models. In *ACL*, 2016.

[37] Minh-Thang Luong, Michael Kayser, and Christopher D. Manning. Deep neural language models for machine translation. In *CoNLL*, 2015.

[38] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *EMNLP*, 2015.

[39] Minh-Thang Luong, Ilya Sutskever, Quoc V. Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. In *ACL*, 2015.

[40] Minh-Thang Luong, Quoc V. Le, Ilya Sutskever, Oriol Vinyals, and Lukasz Kaiser. Multi-task sequence to sequence learning. In *ICLR*, 2016.

[41] James Martens and Ilya Sutskever. Learning recurrent neural networks with Hessian-free optimization. In *ICML*, 2011.

[42] Tomáš Mikolov. *Statistical Language Models Based on Neural Networks*. PhD thesis, Brno University of Technology, 2012.

[43] Tomáš Mikolov and Geoffrey Zweig. Context dependent recurrent neural network language model. In *SLT*, 2012.

[44] Tomáš Mikolov, Martin Karafit, Lukas Burget, Jan Cernock, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, 2010.

[45] Tomáš Mikolov, Stefan Kombrink, Lukas Burget, Jan Cernock, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *ICASSP*, 2011.

[46] Tomáš Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[47] Andriy Mnih and Geoffrey Hinton. A scalable hierarchical distributed language model. In *NIPS*, 2009.

[48] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. In *ICML*, 2012.

[49] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *AISTATS*, 2005.

[50] Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, 2003.

[51] Razvan Pascanu, Tomáš Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *ICML*, 2013.

[52] Adam Pauls and Dan Klein. Faster and smaller n-gram language models. In *ACL*, 2011.

[53] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[54] Ronald Rosenfeld. Two decades of statistical language modeling: Where do we go from here? In *IEEE*, volume 88, pages 1270–1278, 2000.

[55] David E. Rumelhart and James L. McClelland. On learning the past tenses of English verbs. In J. L. McClelland, D. E. Rumelhart, and PDP Research Group, editors, *Parallel Distributed Processing. Volume 2: Psychological and Biological Models*, pages 216–271. MIT Press, 1986.

[56] Holger Schwenk. Continuous space language models. *Computer Speech and Languages*, 21(3):492–518, 2007.

[57] Holger Schwenk. Continuous space translation models for phrase-based statistical machine translation. In *COLING*, 2012.

[58] Abigail See, Minh-Thang Luong, and Christopher D. Manning. Compression of neural machine translation models via pruning. In *CoNLL*, 2016.

[59] Le Hai Son, Alexandre Allauzen, and Franois Yvon. Continuous space translation models with neural networks. In *NAACL-HLT*, 2012.

[60] Andreas Stolcke. SRILM – an extensible language modeling toolkit. In *ICSLP*, 2002.

[61] Ilya Sutskever. *Training Recurrent Neural Networks*. PhD thesis, University of Toronto, 2012.

[62] Ilya Sutskever, James Martens, and Geoffrey Hinton. Generating text with recurrent neural networks. In *ICML*, 2011.

[63] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[64] Yee Whye Teh. A hierarchical Bayesian language model based on Pitman-Yor processes. In *ACL*, 2006.

[65] Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. Decoding with large-scale neural language models improves translation. In *EMNLP*, 2013.

[66] Alexander Waibel, Toshiyuki Hanazawa, Geofrey Hinton, Kiyohiro Shikano, and Kevin J. Lang. Readings in speech recognition. chapter Phoneme Recognition Using Time-delay Neural Networks, pages 393–404. 1990. ISBN 1-55860-124-4.

[67] Warren Weaver. Translation. In William N. Locke and A. Donald Boothe, editors, *Machine Translation of Languages*, pages 15–23. MIT Press, Cambridge, MA, 1949. Reprinted from a memorandum written by Weaver in 1949.

[68] Paul J. Werbos. Back propagation through time: What it does and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.

[69] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.