Neural Machine Translation and Sequence-to-sequence Models: A Tutorial

Graham Neubig

Language Technologies Institute, Carnegie Mellon University

1   Introduction

This tutorial introduces a new and powerful set of techniques variously called "neural machine translation" or "neural sequence-to-sequence models". These techniques have been used in a number of tasks regarding the handling of human language, and can be a powerful tool in the toolbox of anyone who wants to model sequential data of some sort. The tutorial assumes that the reader knows the basics of math and programming, but does not assume any particular experience with neural networks or natural language processing. It attempts to explain the intuition behind the various methods covered, then delves into them with enough mathematical detail to understand them concretely, and culminates with a suggestion for an implementation exercise, where readers can test that they understood the content in practice.

1.1   Background

Before getting into the details, it might be worth describing each of the terms that appear in the title "Neural Machine Translation and Sequence-to-sequence Models". **Machine translation** is the technology used to translate between human language. Think of the universal translation device showing up in sci-fi movies to allow you to communicate effortlessly with those that speak a different language, or any of the plethora of online translation web sites that you can use to assimilate content that is not in your native language. This ability to remove language barriers, needless to say, has the potential to be very useful, and thus machine translation technology has been researched from shortly after the advent of digital computing.

We call the language input to the machine translation system the **source language**, and call the output language the **target language**. Thus, machine translation can be described as the task of converting a *sequence* of words in the source, and converting into a *sequence* of words in the target. The goal of the machine translation practitioner is to come up with an effective model that allows us to perform this conversion accurately over a broad variety of languages and content.

The second part of the title, **sequence-to-sequence models**, refers to the broader class of models that include *all* models that map one sequence to another. This, of course, includes machine translation, but it also covers a broad spectrum of other methods used to handle other tasks as shown in Figure 1. In fact, if we think of a computer program as something that takes in a sequence of input bits, then outputs a sequence of output bits, we could say that *every single program is a sequence-to-sequence model* expressing some behavior (although of course in many cases this is not the most natural or intuitive way to express things).

Machine translation:
kare wa ringo wo tabeta → he ate an apple
Tagging:
he ate an apple → PRN VBD DET PP
Dialog:
he ate an apple → good, he needs to slim down
Speech Recognition:
↔↔↔↔↔↔↔↔ → he ate an apple
And just about anything...:
1010000111101 → 00011010001101

Figure 1: An example of sequence-to-sequence modeling tasks.

The motivation for using machine translation as a representative of this larger class of sequence-to-sequence models is many-fold:

1. Machine translation is a widely-recognized and useful instance of sequence-to-sequence models, and allows us to use many intuitive examples demonstrating the difficulties encountered when trying to tackle these problems.

2. Machine translation is often one of the main driving tasks behind the development of new models, and thus these models tend to be tailored to MT first, then applied to other tasks.

3. However, there are also cases where MT has learned from other tasks as well, and introducing these tasks helps explain the techniques used in MT as well.

1.2 Structure of this Tutorial

This tutorial first starts out with a general mathematical definition of statistical techniques for machine translation in Section 2. The rest of this tutorial will sequentially describe techniques of increasing complexity, leading up to attentional models, which represent the current state-of-the-art in the field.

First, Sections 3-6 focus on **language models**, which calculate the probability of a target sequence of interest. These models are not capable of performing translation or sequence transduction, but will provide useful preliminaries to understand sequence-to-sequence models.

- Section 3 describes n-**gram language models**, simple models that calculate the probability of words based on their counts in a set of data. It also describes how we evaluate how well these models are doing using measures such as **perplexity**.

- Section 4 describes **log-linear language models**, models that instead calculate the probability of the next word based on features of the context. It describes how we can learn the parameters of the models through **stochastic gradient descent** – calculating derivatives and gradually updating the parameters to increase the likelihood of the observed data.

2

- Section 5 introduces the concept of **neural networks**, which allow us to combine together multiple pieces of information more easily than log-linear models, resulting in increased modeling accuracy. It gives an example of **feed-forward neural language models**, which calculate the probability of the next word based on a few previous words using neural networks.

- Section 6 introduces **recurrent neural networks**, a variety of neural networks that have mechanisms to allow them to remember information over multiple time steps. These lead to **recurrent neural network language models**, which allow for the handling of long-term dependencies that are useful when modeling language or other sequential data.

Finally, Sections 7 and 8 describe actual sequence-to-sequence models capable of performing machine translation or other tasks.

- Section 7 describes **encoder-decoder** models, which use a recurrent neural network to *encode* the target sequence into a vector of numbers, and another network to *decode* this vector of numbers into an output sentence. It also describes **search algorithms** to generate output sequences based on this model.

- Section 8 describes **attention**, a method that allows the model to focus on different parts of the input sentence while generating translations. This allows for a more efficient and intuitive method of representing sentences, and is often more effective than its simpler encoder-decoder counterpart.

2 Statistical MT Preliminaries

First, before talking about any specific models, this chapter describes the overall framework of **statistical machine translation** (SMT) [16] more formally.

First, we define our task of machine translation as translating a source sentence $F = f_1, \ldots, f_J = f_1^{|F|}$ into a target sentence $E = e_1, \ldots, e_I = e_1^{|E|}$. Thus, any type of translation system can be defined as a function

$$\hat{E} = \text{mt}(F), \tag{1}$$

which returns a translation hypothesis $\hat{E}$ given a source sentence F as input.

Statistical machine translation systems are systems that perform translation by creating a probabilistic model for the probability of E given F, $P(E \mid F; \theta)$, and finding the target sentence that maximizes this probability:

$$\hat{E} = \underset{E}{\text{argmax}}\ P(E \mid F; \theta), \tag{2}$$

where θ are the parameters of the model specifying the probability distribution. The parameters θ are learned from data consisting of aligned sentences in the source and target languages, which are called **parallel corpora** in technical terminology. Within this framework, there are three major problems that we need to handle appropriately in order to create a good translation system:

Modeling: First, we need to decide what our model $P(E \mid F; \theta)$ will look like. What parameters will it have, and how will the parameters specify a probability distribution?

Learning: Next, we need a method to learn appropriate values for parameters θ from training data.

Search: Finally, we need to solve the problem of finding the most probable sentence (solving "argmax"). This process of searching for the best hypothesis and is often called **decoding**.[1]

The remainder of the material here will focus on solving these problems.

3 n-gram Language Models

While the final goal of a statistical machine translation system is to create a model of the target sentence E given the source sentence F, $P(E \mid F)$, in this chapter we will take a step back, and attempt to create a **language model** of only the target sentence $P(E)$. Basically, this model allows us to do two things that are of practical use.

Assess naturalness: Given a sentence E, this can tell us, does this look like an actual, natural sentence in the target language? If we can learn a model to tell us this, we can use it to assess the fluency of sentences generated by an automated system to improve its results. It could also be used to evaluate sentences generated by a human for purposes of grammar checking or error correction.

Generate text: Language models can also be used to randomly generate text by sampling a sentence E' from the target distribution: $E' \sim P(E)$.[2] Randomly generating samples from a language model can be interesting in itself – we can see what the model "thinks" is a natural-looking sentences – but it will be more practically useful in the context of the neural translation models described in the following chapters.

In the following sections, we'll cover a few methods used to calculate this probability $P(E)$.

3.1 Word-by-word Computation of Probabilities

As mentioned above, we are interested in calculating the probability of a sentence $E = e_1^T$. Formally, this can be expressed as

$$P(E) = P(|E| = T, e_1^T), \tag{3}$$

the joint probability that the length of the sentence is ($|E| = T$), that the identity of the first word in the sentence is e_1, the identity of the second word in the sentence is e_2, up until the last word in the sentence being e_T. Unfortunately, directly creating a model of this probability distribution is not straightforward,[3] as the length of the sequence T is not determined in advance, and there are a large number of possible combinations of words.[4]

---

[1] This is based on the famous quote from Warren Weaver, likening the process of machine translation to decoding an encoded cipher.

[2] $\sim$ means "is sampled from".

[3] Although it is possible, as shown by **whole-sentence language models** in [88].

[4] *Question: If V is the size of the target vocabulary, how many are there for a sentence of length T?*

P(|E| = 3, e_1="she", e_2="went", e_3="home") =

    P(e_1="she")

    * P(e_2="went" | e_1="she")

    * P(e_3="home" | e_1="she", e_2="went")

    * P(e_4="</s>" | e_1="she", e_2="went", e_3="home")

Figure 2: An example of decomposing language model probabilities word-by-word.

As a way to make things easier, it is common to re-write the probability of the full sentence as the product of single-word probabilities. This takes advantage of the fact that a joint probability – for example $P(e_1, e_2, e_3)$ – can be calculated by multiplying together conditional probabilities for each of its elements. In the example, this means that $P(e_1, e_2, e_3) = P(e_1)P(e_2 \mid e_1)P(e_3 \mid e_1, e_2)$.

Figure 2 shows an example of this incremental calculation of probabilities for the sentence "she went home". Here, in addition to the actual words in the sentence, we have introduced an implicit *sentence end* ("$\langle/s\rangle$") symbol, which we will indicate when we have terminated the sentence. Stepping through the equation in order, this means we first calculate the probability of "she" coming at the beginning of the sentence, then the probability of "went" coming next in a sentence starting with "she", the probability of "home" coming after the sentence prefix "she went", and then finally the sentence end symbol "$\langle/s\rangle$" after "she went home". More generally, we can express this as the following equation:

$$P(E) = \prod_{t=1}^{T+1} P(e_t \mid e_1^{t-1}) \tag{4}$$

where $e_{T+1} = \langle/s\rangle$. So coming back to the sentence end symbol $\langle/s\rangle$, the reason why we introduce this symbol is because it allows us to know when the sentence ends. In other words, by examining the position of the $\langle/s\rangle$ symbol, we can determine the $|E| = T$ term in our original LM joint probability in Equation 3. In this example, when we have $\langle/s\rangle$ as the 4th word in the sentence, we know we're done and our final sentence length is 3.

Once we have the formulation in Equation 4, the problem of language modeling now becomes a problem of calculating the next word given the previous words $P(e_t \mid e_1^{t-1})$. This is much more manageable than calculating the probability for the whole sentence, as we now have a fixed set of items that we are looking to calculate probabilities for. The next couple of sections will show a few ways to do so.

3.2 Count-based n-gram Language Models

The first way to calculate probabilities is simple: prepare a set of training data from which we can count word strings, count up the number of times we have seen a particular string of words, and divide it by the number of times we have seen the context. This simple method,

Figure 3: An example of calculating probabilities using maximum likelihood estimation.

can be expressed by the equation below, with an example shown in Figure 3

$$P_{\mathrm{ML}}(e_t \mid e_1^{t-1}) = \frac{c_{\mathrm{prefix}}(e_1^t)}{c_{\mathrm{prefix}}(e_1^{t-1})}. \tag{5}$$

Here $c_{\mathrm{prefix}}(\cdot)$ is the count of the number of times this particular word string appeared at the beginning of a sentence in the training data. This approach is called **maximum likelihood estimation** (MLE, details later in this chapter), and is both simple and guaranteed to create a model that assigns a high probability to the sentences in training data.

However, let's say we want to use this model to assign a probability to a new sentence that we've never seen before. For example, if we want to calculate the probability of the sentence "i am from utah ." based on the training data in the example. This sentence is extremely similar to the sentences we've seen before, but unfortunately because the string "i am from utah" has not been observed in our training data, $c_{\mathrm{prefix}}(\mathrm{i, am, from, utah}) = 0$, $P(e_4 = \mathrm{utah} \mid e_1 = \mathrm{i}, e_2 = \mathrm{am}, e_3 = \mathrm{from})$ becomes zero, and thus the probability of the whole sentence as calculated by Equation 5 also becomes zero. In fact, this language model will assign a probability of zero to every sentence that it hasn't seen before in the training corpus, which is not very useful, as the model loses ability to tell us whether a new sentence a system generates is natural or not, or generate new outputs.

To solve this problem, we take two measures. First, instead of calculating probabilities from the beginning of the sentence, we set a fixed window of previous words upon which we will base our probability calculations, approximating the true probability. If we limit our context to $n-1$ previous words, this would amount to:

$$P(e_t \mid e_1^{t-1}) \approx P_{\mathrm{ML}}(e_t \mid e_{t-n+1}^{t-1}). \tag{6}$$

Models that make this assumption are called n-**gram models**. Specifically, when models where $n = 1$ are called unigram models, $n = 2$ bigram models, $n = 3$ trigram models, and $n \geq 4$ four-gram, five-gram, etc.

The parameters θ of n-gram models consist of probabilities of the next word given $n-1$ previous words:

$$\theta_{e_{t-n+1}^t} = P(e_t \mid e_{t-n+1}^{t-1}), \tag{7}$$

and in order to train an n-gram model, we have to learn these parameters from data.[5] In the simplest form, these parameters can be calculated using maximum likelihood estimation as follows:

$$\theta_{e_{t-n+1}^t} = P_{\mathrm{ML}}(e_t \mid e_{t-n+1}^{t-1}) = \frac{c(e_{t-n+1}^t)}{c(e_{t-n+1}^{t-1})}, \tag{8}$$

---

[5] *Question: How many parameters does an n-gram model with a particular n have?*

where $c(\cdot)$ is the count of the word string anywhere in the corpus. Sometimes these equations will reference e_{t-n+1} where $t - n + 1 < 0$. In this case, we assume that $e_{t-n+1} = \langle s \rangle$ where $\langle s \rangle$ is a special *sentence start* symbol.

If we go back to our previous example and set $n = 2$, we can see that while the string "i am from utah ." has never appeared in the training corpus, "i am", "am from", "from utah", "utah .", and ". $\langle /s \rangle$" are all somewhere in the training corpus, and thus we can patch together probabilities for them and calculate a non-zero probability for the whole sentence.[6]

However, we still have a problem: what if we encounter a two-word string that has never appeared in the training corpus? In this case, we'll still get a zero probability for that particular two-word string, resulting in our full sentence probability also becoming zero. n-gram models fix this problem by **smoothing** probabilities, combining the maximum likelihood estimates for various values of n. In the simple case of smoothing unigram and bigram probabilities, we can think of a model that combines together the probabilities as follows:

$$P(e_t \mid e_{t-1}) = (1 - \alpha)P_{\mathrm{ML}}(e_t \mid e_{t-1}) + \alpha P_{\mathrm{ML}}(e_t), \qquad (9)$$

where α is a variable specifying how much probability mass we hold out for the unigram distribution. As long as we set $\alpha > 0$, regardless of the context all the words in our vocabulary will be assigned some probability. This method is called **interpolation**, and is one of the standard ways to make probabilistic models more robust to low-frequency phenomena.

If we want to use even more context – $n = 3$, $n = 4$, $n = 5$, or more – we can recursively define our interpolated probabilities as follows:

$$P(e_t \mid e_{t-m+1}^{t-1}) = (1 - \alpha_m)P_{\mathrm{ML}}(e_t \mid e_{t-m+1}^{t-1}) + \alpha_m P(e_t \mid e_{t-m+2}^{t-1}). \qquad (10)$$

The first term on the right side of the equation is the maximum likelihood estimate for the model of order m, and the second term is the interpolated probability for all orders up to $m - 1$.

There are also more sophisticated methods for smoothing, which are beyond the scope of this section, but summarized very nicely in [19].

Context-dependent smoothing coefficients: Instead of having a fixed α, we condition the interpolation coefficient on the context: $\alpha_{e_{t-m+1}^{t-1}}$. This allows the model to give more weight to higher order n-grams when there are a sufficient number of training examples for the parameters to be estimated accurately and fall back to lower-order n-grams when there are fewer training examples. These context-dependent smoothing coefficients can be chosen using heuristics [118] or learned from data [77].

Back-off: In Equation 9, we interpolated together two probability distributions over the full vocabulary V. In the alternative formulation of **back-off**, the lower-order distribution only is used to calculate probabilities for words that were given a probability of zero in the higher-order distribution. Back-off is more expressive but also more complicated than interpolation, and the two have been reported to give similar results [41].

Modified distributions: It is also possible to use a different distribution than P_{ML}. This can be done by subtracting a constant value from the counts before calculating probabilities, a method called **discounting**. It is also possible to modify the counts of

---

[6] *Question: What is this probability?*

lower-order distributions to reflect the fact that they are used mainly as a fall-back for when the higher-order distributions lack sufficient coverage.

Currently, **Modified Kneser-Ney smoothing** (MKN; [19]), is generally considered one of the standard and effective methods for smoothing n-gram language models. MKN uses context-dependent smoothing coefficients, discounting, and modification of lower-order distributions to ensure accurate probability estimates.

3.3 Evaluation of Language Models

Once we have a language model, we will want to test whether it is working properly. The way we test language models is, like many other machine learning models, by preparing three sets of data:

Training data is used to train the parameters θ of the model.

Development data is used to make choices between alternate models, or to tune the **hyper-parameters** of the model. Hyper-parameters in the model above could include the maximum length of n in the n-gram model or the type of smoothing method.

Test data is used to measure our final accuracy and report results.

For language models, we basically want to know whether the model is an accurate model of language, and there are a number of ways we can define this. The most straight-forward way of defining accuracy is the **likelihood** of the model with respect to the development or test data. The likelihood of the parameters θ with respect to this data is equal to the probability that the model assigns to the data. For example, if we have a test dataset $\mathcal{E}_{\text{test}}$, this is:

$$P(\mathcal{E}_{\text{test}}; \theta). \tag{11}$$

We often assume that this data consists of several independent sentences or documents E, giving us

$$P(\mathcal{E}_{\text{test}}; \theta) = \prod_{E \in \mathcal{E}_{\text{test}}} P(E; \theta). \tag{12}$$

Another measure that is commonly used is **log likelihood**

$$\log P(\mathcal{E}_{\text{test}}; \theta) = \sum_{E \in \mathcal{E}_{\text{test}}} \log P(E; \theta). \tag{13}$$

The log likelihood is used for a couple reasons. The first is because the probability of any particular sentence according to the language model can be a very small number, and the product of these small numbers can become a very small number that will cause numerical precision problems on standard computing hardware. The second is because sometimes it is more convenient mathematically to deal in log space. For example, when taking the derivative in gradient-based methods to optimize parameters (used in the next section), it is more convenient to deal with the sum in Equation 13 than the product in Equation 11.

It is also common to divide the log likelihood by the number of words in the corpus

$$\text{length}(\mathcal{E}_{\text{test}}) = \sum_{E \in \mathcal{E}_{\text{test}}} |E|. \tag{14}$$

This makes it easier to compare and contrast results across corpora of different lengths.

The final common measure of language model accuracy is **perplexity**, which is defined as the exponent of the average negative log likelihood per word

$$\mathrm{ppl}(\mathcal{E}_{\mathrm{test}}; \theta) = e^{-(\log P(\mathcal{E}_{\mathrm{test}}; \theta))/\mathrm{length}(\mathcal{E}_{\mathrm{test}})}. \tag{15}$$

An intuitive explanation of the perplexity is "how confused is the model about its decision?" More accurately, it expresses the value "if we randomly picked words from the probability distribution calculated by the language model at each time step, on average how many words would it have to pick to get the correct one?" One reason why it is common to see perplexities in research papers is because the numbers calculated by perplexity are bigger, making the differences in models more easily perceptible by the human eye.[7]

3.4 Handling Unknown Words

Finally, one important point to keep in mind is that some of the words in the test set $\mathcal{E}_{\mathrm{test}}$ will not appear even once in the training set $\mathcal{E}_{\mathrm{train}}$. These words are called **unknown words**, and need to be handeled in some way. Common ways to do this in language models include:

Assume closed vocabulary: Sometimes we can assume that there will be no new words in the test set. For example, if we are calculating a language model over ASCII characters, it is reasonable to assume that all characters have been observed in the training set. Similarly, in some speech recognition systems, it is common to simply assign a probability of zero to words that don't appear in the training data, which means that these words will not be able to be recognized.

Interpolate with an unknown words distribution: As mentioned in Equation 10, we can interpolate between distributions of higher and lower order. In the case of unknown words, we can think of this as a distribution of order "0", and define the 1-gram probability as the interpolation between the unigram distribution and unknown word distribution

$$P(e_t) = (1 - \alpha_1)P_{\mathrm{ML}}(e_t) + \alpha_1 P_{\mathrm{unk}}(e_t). \tag{16}$$

Here, P_{unk} needs to be a distribution that assigns a probability to all words V_{all}, not just ones in our vocabulary V derived from the training corpus. This could be done by, for example, training a language model over characters that "spells out" unknown words in the case they don't exist in in our vocabulary. Alternatively, as a simpler approximation that is nonetheless fairer than ignoring unknown words, we can guess the total number of words $|V_{\mathrm{all}}|$ in the language where we are modeling, where $|V_{\mathrm{all}}| > |V|$, and define P_{unk} as a uniform distribution over this vocabulary: $P_{\mathrm{unk}}(e_t) = 1/|V_{\mathrm{all}}|$.

Add an ⟨unk⟩ word: As a final method to handle unknown words we can remove some of the words in $\mathcal{E}_{\mathrm{train}}$ from our vocabulary, and replace them with a special ⟨unk⟩ symbol representing unknown words. One common way to do so is to remove **singletons**, or words that only appear once in the training corpus. By doing this, we explicitly predict in which contexts we will be seeing an unknown word, instead of implicitly predicting it through interpolation like mentioned above. Even if we predict the ⟨unk⟩ symbol, we

---

[7]And, some cynics will say, making it easier for your research papers to get accepted.

will still need to estimate the probability of the actual word, so any time we predict ⟨unk⟩ at position i, we further multiply in the probability of $P_{\text{unk}}(e_t)$.

3.5 Further Reading

To read in more detail about n-gram language models, [41] gives a very nice introduction and comprehensive summary about a number of methods to overcome various shortcomings of vanilla n-grams like the ones mentioned above.

There are also a number of extensions to n-gram models that may be nice for the interested reader.

Large-scale language modeling: Language models are an integral part of many commercial applications, and in these applications it is common to build language models using massive amounts of data harvested from the web for other sources. To handle this data, there is research on efficient data structures [48, 82], distributed parameter servers [14], and lossy compression algorithms [104].

Language model adaptation: In many situations, we want to build a language model for specific speaker or domain. Adaptation techniques make it possible to create large general-purpose models, then adapt these models to more closely match the target use case [6].

Longer-distance language count-based models: As mentioned above, n-gram models limit their context to $n - 1$, but in reality there are dependencies in language that can reach much farther back into the sentence, or even span across whole documents. The recurrent neural network language models that we will introduce in Section 6 are one way to handle this problem, but there are also non-neural approaches such as cache language models [61], topic models [13], and skip-gram models [41].

Syntax-based language models: There are also models that take into account the syntax of the target sentence. For example, it is possible to condition probabilities not on words that occur directly next to each other in the sentence, but those that are "close" syntactically [96].

3.6 Exercise

The exercise that we will be doing in class will be constructing an n-gram LM with linear interpolation between various levels of n-grams. We will write code to:

- Read in and save the training and testing corpora.

- Learn the parameters on the training corpus by counting up the number of times each n-gram has been seen, and calculating maximum likelihood estimates according to Equation 8.

- Calculate the probabilities of the test corpus using linearly interpolation according to Equation 9 or Equation 10.

To handle unknown words, you can use the *uniform distribution* method described in Section 3.4, assuming that there are 10,000,000 words in the English vocabulary. As a sanity check, it may be better to report the number of unknown words, and which portions of the per-word log-likelihood were incurred by the main model, and which portion was incurred by the unknown word probability $\log P_{\text{unk}}$.

In order to do so, you will first need data, and to make it easier to start out you can use some pre-processed data from the German-English translation task of the IWSLT evaluation campaign[8] here: `http://phontron.com/data/iwslt-en-de-preprocessed.tar.gz`.

Potential improvements to the model include reading [19] and implementing a better smoothing method, implementing a better method for handling unknown words, or implementing one of the more advanced methods in Section 3.5.

4   Log-linear Language Models

This chapter will discuss another set of language models: **log-linear language models** [87, 20], which take a very different approach than the count-based n-grams described above.[9]

4.1   Model Formulation

Like n-gram language models, log-linear language models still calculate the probability of a particular word e_t given a particular context e_{t-n+1}^{t-1}. However, their method for doing so is quite different from count-based language models, based on the following procedure.

Calculating features: Log-linear language models revolve around the concept of **features**. In short, features are basically, "something about the context that will be useful in predicting the next word". More formally, we define a feature function $\phi(e_{t-n+1}^{t-1})$ that takes a context as input, and outputs a real-valued **feature vector** $\boldsymbol{x} \in \mathbb{R}^N$ that describe the context using N different features.[10]

For example, from our bi-gram models from the previous chapter, we know that "the identity of the previous word" is something that is useful in predicting the next word. If we want to express the identity of the previous word as a real-valued vector, we can assume that each word in our vocabulary V is associated with a word ID j, where $1 \leq j \leq |V|$. Then, we define our feature function $\phi(e_{t-n+1}^{t})$ to return a feature vector $\boldsymbol{x} = \mathbb{R}^{|V|}$, where if $e_{t-1} = j$, then the jth element is equal to one and the remaining elements in the vector are equal to zero. This type of vector is often called a **one-hot vector**, an example of which is shown in Figure 4(a). For later user, we will also define a function onehot(i) which returns a vector

---

[8] `http://iwslt.org`

[9] It should be noted that the cited papers call these **maximum entropy language models**. This is because models in this chapter can be motivated in two ways: *log-linear models* that calculate un-normalized log-probability scores for each function and normalize them to probabilities, and *maximum-entropy models* that spread their probability mass as evenly as possible given the constraint that they must model the training data. While the maximum-entropy interpretation is quite interesting theoretically and interested readers can reference [11] to learn more, the explanation as log-linear models is simpler conceptually, and thus we will use this description in this chapter.

[10] Alternative formulations that define feature functions that also take the current word as input $\phi(e_{t-n+1}^{t})$ are also possible, but in this book, to simplify the transition into neural language models described in Section 5, we consider features over only the context.
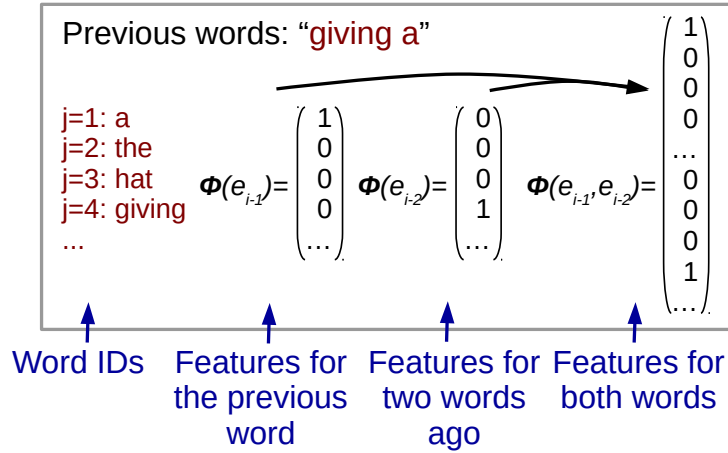
Figure 4: An example of feature values for a particular context.

where only the ith element is one and the rest are zero (assume the length of the vector is the appropriate length given the context).

Of course, we are not limited to only considering one previous word. We could also calculate one-hot vectors for both e_{t-1} and e_{t-2}, then concatenate them together, which would allow us to create a model that considers the values of the two previous words. In fact, there are many other types of feature functions that we can think of (more in Section 4.4), and the ability to flexibly define these features is one of the advantages of log-linear language models over standard n-gram models.

Calculating scores: Once we have our feature vector, we now want to use these features to predict probabilities over our output vocabulary V. In order to do so, we calculate a score vector $\boldsymbol{s} \in \mathbb{R}^{|V|}$ that corresponds to the likelihood of each word: words with higher scores in the vector will also have higher probabilities. We do so using the model parameters θ, which specifically come in two varieties: a **bias vector** $\boldsymbol{b} \in \mathbb{R}^{|V|}$, which tells us how likely each word in the vocabulary is overall, and a **weight matrix** $W = \mathbb{R}^{|V| \times N}$ which describes the relationship between feature values and scores. Thus, the final equation for calculating our scores for a particular context is:

$$\boldsymbol{s} = W\boldsymbol{x} + \boldsymbol{b}. \tag{17}$$

One thing to note here is that in the special case of one-hot vectors or other *sparse* vectors where most of the elements are zero. Because of this we can also think about Equation 17 in a different way that is numerically equivalent, but can make computation more efficient. Specifically, instead of multiplying the large feature vector by the large weight matrix, we can add together the columns of the weight matrix for all *active* (non-zero) features as follows:

$$\boldsymbol{s} = \sum_{\{j:x_j \neq 0\}} W_{\cdot,j} x_j + \boldsymbol{b}, \tag{18}$$

where $W_{\cdot,j}$ is the jth column of W. This allows us to think of calculating scores as "look up the vector for the features active for this instance, and add them together", instead of writing them as matrix math. An example calculation in this paradigm where we have two feature
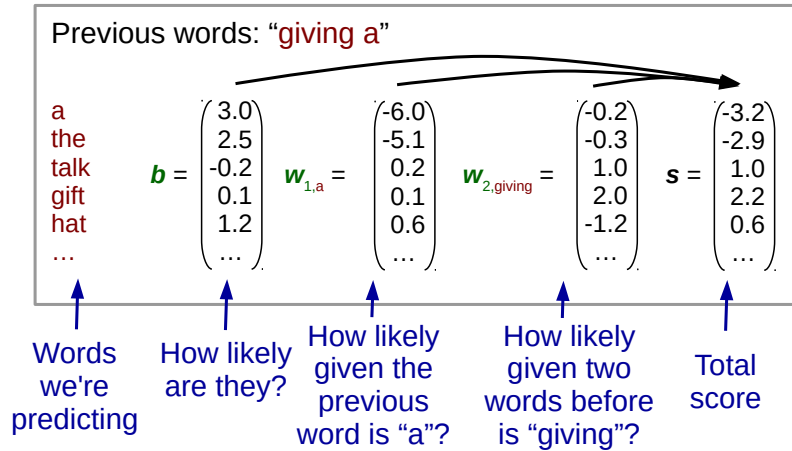
12

Figure 5: An example of the weights for a log linear model in a certain context.

functions (one for the directly preceding word, and one for the word before that) is shown in Figure 5.

Calculating probabilities: It should be noted here that scores $\boldsymbol{s}$ are arbitrary real numbers, not probabilities: they can be negative or greater than one, and there is no restriction that they add to one. Because of this, we run these scores through a function that performs the following transformation:

$$p_j = \frac{\exp(s_j)}{\sum_{\tilde{j}} \exp(s_{\tilde{j}})}. \tag{19}$$

By taking the exponent and dividing by the sum of the values over the entire vocabulary, these scores can be turned into probabilities that are between 0 and 1 and sum to 1.

This function is called the **softmax** function, and often expressed in vector form as follows:

$$\boldsymbol{p} = \text{softmax}(\boldsymbol{s}). \tag{20}$$

Through applying this to the scores calculated in the previous section, we now have a way to go from features to language model probabilities.

4.2   Learning Model Parameters

Now, the only remaining missing link is how to acquire the parameters θ, consisting of the weight matrix W and bias $\boldsymbol{b}$. Basically, the way we do so is by attempting to find parameters that fit the training corpus well.

To do so, we use standard machine learning methods for optimizing parameters. First, we define a **loss function** $\ell(\cdot)$ – a function expressing how poorly we're doing on the training data. In most cases, we assume that this loss is equal to the **negative log likelihood**:

$$\ell(\mathcal{E}_{train}, \boldsymbol{\theta}) = -\log P(\mathcal{E}_{train} \mid \boldsymbol{\theta}) = -\sum_{E \in \mathcal{E}_{train}} \log P(E \mid \boldsymbol{\theta}). \tag{21}$$

We assume we can also define the loss on a per-word level:

$$\ell(e_{t-n+1}^t, \boldsymbol{\theta}) = \log P(e_t \mid e_{t-n+1}^{t-1}). \tag{22}$$

13

Next, we optimize the parameters to reduce this loss. While there are many methods for doing so, in recent years one of the go-to methods is **stochastic gradient descent** (SGD). SGD is an iterative process where we randomly pick a single word e_t (or mini-batch, discussed in Section 5) and take a step to improve the likelihood with respect to e_t. In order to do so, we first calculate the derivative of the loss with respect to each of the features in the full feature set $\boldsymbol{\theta}$:

$$\frac{d\ell(e_{t-n+1}^t, \boldsymbol{\theta})}{d\boldsymbol{\theta}}. \tag{23}$$

We can then use this information to take a step in the direction that will reduce the loss according to the objective function

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{d\ell(e_{t-n+1}^t, \boldsymbol{\theta})}{d\boldsymbol{\theta}}, \tag{24}$$

where η is our **learning rate**, specifying the amount with which we update the parameters every time we perform an update. By doing so, we can find parameters for our model that reduce the loss, or increase the likelihood, on the training data.

This vanilla variety of SGD is quite simple and still a very competitive method for optimization in large-scale systems. However, there are also a few things to consider to ensure that training remains stable:

Adjusting the learning rate: SGD requires also requires us to carefully choose η: if η is too big, training can become unstable and diverge, and if η is too small, training may become incredibly slow or fall into bad local optima. One way to handle this problem is **learning rate decay**: starting with a higher learning rate, then gradually reducing the learning rate near the end of training. Other more sophisticated methods are listed below.

Early stopping: It is common to use a held-out development set, measure our log-likelihood on this set, and save the model that has achieved the best log-likelihood on this held-out set. This is useful in case the model starts to over-fit to the training set, losing its generalization capability, we can re-wind to this saved model. As another method to prevent over-fitting and smooth convergence of training, it is common to measure log likelihood on a held-out development set, and when the log likelihood stops improving or starts getting worse, reduce the learning rate.

Shuffling training order: One of the features of SGD is that it processes training data one at a time. This is nice because it is simple and can be efficient, but it also causes problems if there is some bias in the order in which we see the data. For example, if our data is a corpus of news text where news articles come first, then sports, then entertainment, there is a chance that near the end of training our model will see hundreds or thousands of entertainment examples in a row, resulting in the parameters moving to a space that favors these more recently seen training examples. To prevent this problem, it is common (and highly recommended) to randomly shuffle the order with which the training data is presented to the learning algorithm on every pass through the data.

There are also a number of other update rules that have been proposed to improve gradient descent and make it more stable or efficient. Some representative methods are listed below:

14

SGD with momentum [90]: Instead of taking a single step in the direction of the current gradient, SGD with momentum keeps an exponentially decaying average of past gradients. This reduces the propensity of simple SGD to "jitter" around, making optimization move more smoothly across the parameter space.

AdaGrad [30]: AdaGrad focuses on the fact that some parameters are updated much more frequently than others. For example, in the model above, columns of the weight matrix W corresponding to infrequent context words will only be updated a few times for every pass through the corpus, while the bias b will be updated on every training example. Based on this, AdaGrad dynamically adjusts the training rate η for each parameter individually, with frequently updated (and presumably more stable) parameters such as b getting smaller updates, and infrequently updated parameters such as W getting larger updates.

Adam [60]: Adam is another method that computes learning rates for each parameter. It does so by keeping track of exponentially decaying averages of the mean and variance of past gradients, incorporating ideas similar to both momentum and AdaGrad. Adam is now one of the more popular methods for optimization, as it greatly speeds up convergence on a wide variety of datasets, facilitating fast experimental cycles. However, it is also known to be prone to over-fitting, and thus, if high performance is paramount, it should be used with some caution and compared to more standard SGD methods.

[89] provides a good overview of these various methods with equations and notes a few other concerns when performing stochastic optimization.

4.3 Derivatives for Log-linear Models

Now, the final piece in the puzzle is the calculation of derivatives of the loss function with respect to the parameters. To do so, first we step through the full loss function in one pass as below:

$$\boldsymbol{x} = \boldsymbol{\phi}(e_{t-m+1}^{t-1}) \tag{25}$$

$$\boldsymbol{s} = \sum_{\{j:x_j!=0\}} W_{\cdot,j} x_j + \boldsymbol{b} \tag{26}$$

$$\boldsymbol{p} = \text{softmax}(\boldsymbol{s}) \tag{27}$$

$$\ell = -\log \boldsymbol{p}_{e_t}. \tag{28}$$

And thus, using the chain rule to calculate

$$\frac{d\ell(e_{t-n+1}^t, W, \boldsymbol{b})}{d\boldsymbol{b}} = \frac{d\ell}{d\boldsymbol{p}} \frac{d\boldsymbol{p}}{d\boldsymbol{s}} \frac{d\boldsymbol{s}}{d\boldsymbol{b}} \tag{29}$$

$$\frac{d\ell(e_{t-n+1}^t, W, \boldsymbol{b})}{dW_{\cdot,j}} = \frac{d\ell}{d\boldsymbol{p}} \frac{d\boldsymbol{p}}{d\boldsymbol{s}} \frac{d\boldsymbol{s}}{dW_{\cdot,j}} \tag{30}$$

15

we find that the derivative of the loss function for the bias and each column of the weight matrix is:

$$\frac{d\ell(e_{t-n+1}^t, W, \boldsymbol{b})}{d\boldsymbol{b}} = \boldsymbol{p} - \text{onehot}(e_t) \tag{31}$$

$$\frac{d\ell(e_{t-n+1}^t, W, \boldsymbol{b})}{dW_{\cdot,j}} = x_j(\boldsymbol{p} - \text{onehot}(e_t)) \tag{32}$$

Confirming these equations is left as a (highly recommended) exercise to the reader. Hint: when performing this derivation, it is easier to work with the log probability $\log \boldsymbol{p}$ than working with $\boldsymbol{p}$ directly.

4.4   Other Features for Language Modeling

One reason why log-linear models are nice is because they allow us to flexibly design features that we think might be useful for predicting the next word. For example, these could include:

Context word features: As shown in the example above, we can use the identity of e_{t-1} or the identity of e_{t-2}.

Context class: Context words can be grouped into classes of similar words (using a method such as Brown clustering [15]), and instead of looking up a one-hot vector with a separate entry for every word, we could look up a one-hot vector with an entry for each class [18]. Thus, words from the same class could share statistical strength, allowing models to generalize better.

Context suffix features: Maybe we want a feature that fires every time the previous word ends with "...ing" or other common suffixes. This would allow us to learn more generalized patterns about words that tend to follow progressive verbs, etc.

Bag-of-words features: Instead of just using the past n words, we could use all previous words in the sentence. This would amount to calculating the one-hot vectors for every word in the previous sentence, and then instead of concatenating them simply summing them together. This would lose all information about what word is in what position, but could capture information about what words tend to co-occur within a sentence or document.

It is also possible to combine together multiple features (for example e_{t-1} is a particular word *and* e_{t-2} is another particular word). This is one way to create a more expressive feature set, but also has a downside of greatly increasing the size of the feature space. We discuss these features in more detail in Section 5.1.

4.5   Further Reading

The language model in this section was basically a featurized version of an n-gram language model. There are quite a few other varieties of linear featurized models including:

Whole-sentence language models: These models, instead of predicting words one-by-one, predict the probability over the whole sentence then normalize [88]. This can be conducive to introducing certain features, such as a probability distribution over lengths of sentences, or features such as "whether this sentence contains a verb".

Discriminative language models: In the case that we want to use a language model to determine whether the output of a system is good or not, sometimes it is useful to train directly on this system output, and try to re-rank the outputs to achieve higher accuracy [86]. Even if we don't have real negative examples, it can be possible to "hallucinate" negative examples that are still useful for training [80].

4.6 Exercise

In the exercise for this chapter, we will construct a log-linear language model and evaluate its performance. I highly suggest that you try to use the `NumPy` library to hold and perform calculations over feature vectors, as this will make things much easier. If you have never used `NumPy` before, you can take a look at this tutorial to get started: `https://docs.scipy.org/doc/numpy-dev/user/quickstart.html`.

Writing the program will entail:

- Writing a function to read in the training and test corpora, and converting the words into numerical IDs.

- Writing the feature function $\phi(e_{t-n+1}^{t-1})$, which takes in a string and returns which features are active (for example, as a baseline these can be features with the identity of the previous two words).

- Writing code to calculate the loss function.

- Writing code to calculate gradients and perform stochastic gradient descent updates.

- Writing (or re-using from the previous exercise) code to evaluate the language models.

Similarly to the n-gram language models, we will measure the per-word log likelihood and perplexity on our text corpus, and compare it to n-gram language models. Handling unknown words will similarly require that you use the uniform distribution with 10,000,000 words in the English vocabulary.

Potential improvements to the model include designing better feature functions, adjusting the learning rate and measuring the results, and researching and implementing other types of optimizers such as AdaGrad or Adam.

5 Neural Networks and Feed-forward Language Models

In this chapter, we describe language models based on **neural networks**, a way to learn more sophisticated functions to improve the accuracy of our probability estimates with less feature engineering.

5.1 Potential and Problems with Combination Features

Before moving into the technical detail of neural networks, first let's take a look at a motivating example in Figure 6. From the example, we can see e_{t-2} = "farmers" is compatible with e_t = "hay" (in the context "farmers grow hay"), and e_{t-1} = "eat" is also compatible (in the context "cows eat hay"). If we are using a log-linear model with one set of features dependent

farmers eat	steak → **high** hay → **low**	cows eat	steak → **low** hay → **high**
farmers grow	steak → **low** hay → **high**	cows grow	steak → **low** hay → **low**

Figure 6: An example of the effect that combining multiple words can have on the probability of the next word.

on e_{t-1}, and another set of features dependent on e_{t-2}, neither set of features can rule out the unnatural phrase "farmers eat hay."

One way we can fix this problem is by creating another set of features where we learn one vector for each pair of words e_{t-2}, e_{t-1}. If this is the case, our vector for the context $e_{t-2} =$ "farmers", $e_{t-1} =$ "eat" could assign a low score to "hay", resolving this problem. However, adding these combination features has one major disadvantage: it greatly expands the parameters: instead of $O(|V|^2)$ parameters for each pair e_{i-1}, e_i, we need $O(|V|^3)$ parameters for each triplet e_{i-2}, e_{i-1}, e_i. These numbers greatly increase the amount of memory used by the model, and if there are not enough training examples, the parameters may not be learned properly.

Because of both the importance of and difficulty in learning using these combination features, a number of methods have been proposed to handle these features, such as **kernelized support vector machines** [28] and **neural networks** [91, 39]. Specifically in this section, we will cover neural networks, which are both flexible and relatively easy to train on large data, desiderata for sequence-to-sequence models.

5.2 A Brief Overview of Neural Networks

To understand neural networks in more detail, let's take a very simple example of a function that we cannot learn with a simple linear classifier like the ones we used in the last chapter: a function that takes an input $\boldsymbol{x} \in \{-1, 1\}^2$ and outputs $y = 1$ if both x_1 and x_2 are equal and $y = -1$ otherwise. This function is shown in Figure 7.
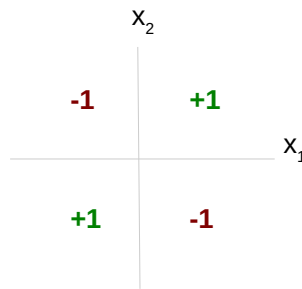

Figure 7: A function that cannot be solved by a linear transformation.

A first attempt at solving this function might define a linear model (like the log-linear models from the previous chapter) that solves this problem using the following form:
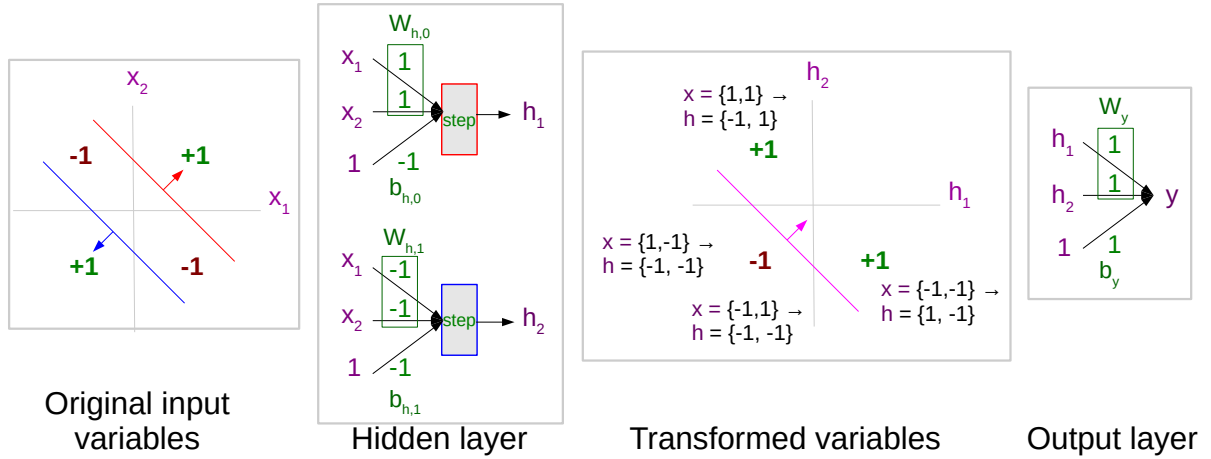
$$y = W\boldsymbol{x} + b. \tag{33}$$

18

Figure 8: A simple neural network that represents the nonlinear function of Figure 7.

However, this class of functions is not powerful enough to represent the function at hand.[11] Thus, we turn to a slightly more complicated class of functions taking the following form:

$$\boldsymbol{h} = \text{step}(W_{xh}\boldsymbol{x} + \boldsymbol{b}_h)$$
$$y = \boldsymbol{w}_{hy}\boldsymbol{h} + b_y. \tag{34}$$

Computation is split into two stages: calculation of the **hidden layer**, which takes in input $\boldsymbol{x}$ and outputs a vector of hidden variables $\boldsymbol{h}$, and calculation of the **output layer**, which takes in $\boldsymbol{h}$ and calculates the final result y. Both layers consist of an **affine transform**[12] using weights W and biases $\boldsymbol{b}$, followed by a step($\cdot$) function, which calculates the following:

$$\text{step}(x) = \begin{cases} 1 & \text{if } x > 0, \\ -1 & \text{otherwise.} \end{cases} \tag{35}$$

This function is one example of a class of neural networks called **multi-layer perceptrons** (MLPs). In general, MLPs consist one or more hidden layers that consist of an affine transform followed by a non-linear function (such as the step function used here), culminating in an output layer that calculates some variety of output.

Figure 8 demonstrates why this type of network does a better job of representing the non-linear function of Figure 7. In short, we can see that the first hidden layer *transforms* the input $\boldsymbol{x}$ into a hidden vector $\boldsymbol{h}$ in a different space that is more conducive for modeling our final function. Specifically in this case, we can see that $\boldsymbol{h}$ is now in a space where we can define a linear function (using $\boldsymbol{w}_y$ and b_y) that correctly calculates the desired output y.

As mentioned above, MLPs are one specific variety of neural network. More generally, neural networks can be thought of as a chain of functions (such as the affine transforms and step functions used above, but also including many, many others) that takes some input and calculates some desired output. The power of neural networks lies in the fact that chaining together a variety of simpler functions makes it possible to represent more complicated functions

---

[11] *Question: Prove this by trying to solve the system of equations.*
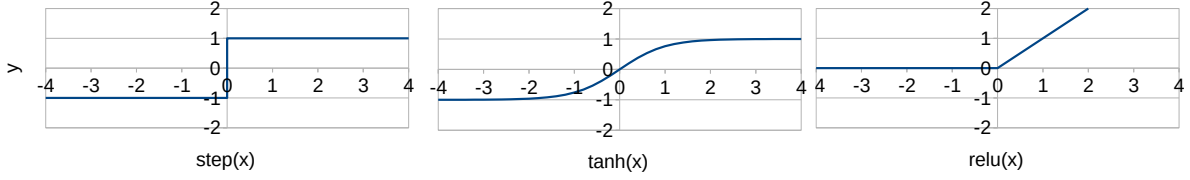[12] A fancy name for a multiplication followed by an addition.

Figure 9: Types of non-linearities.

in an easily trainable, parameter-efficient way. In fact, the simple single-layer MLP described above is a **universal function approximator** [51], which means that it can approximate any function to arbitrary accuracy if its hidden vector $\boldsymbol{h}$ is large enough.

We will see more about training in Section 5.3 and give some more examples of how these can be more parameter efficient in the discussion of neural network language models in Section 5.5.

5.3 Training Neural Networks

Now that we have a model in Equation 34, we would like to train its parameters W_{mh}, $\boldsymbol{b}_h$, $\boldsymbol{w}_{hy}$, and b_y. To do so, remembering our gradient-based training methods from the last chapter, we need to define the loss function $\ell(\cdot)$, calculate the derivative of the loss with respect to the parameters, then take a step in the direction that will reduce the loss. For our loss function, let's use the **squared-error loss**, a commonly used loss function for regression problems which measures the difference between the calculated value y and correct value y^* as follows

$$\ell(y^*, y) = (y^* - y)^2. \tag{36}$$

Next, we need to calculate derivatives. Here, we run into one problem: the step($\cdot$) function is not very derivative friendly, with its derivative being:

$$\frac{d\text{step}(x)}{dx} = \begin{cases} \text{undefined} & \text{if } x = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{37}$$

Because of this, it is more common to use other non-linear functions, such as the hyperbolic tangent (tanh) function. The tanh function, as shown in Figure 9, looks very much like a softened version of the step function that has a continuous gradient everywhere, making it more conducive to training with gradient-based methods. There are a number of other alternatives as well, the most popular of which being the rectified linear unit (RelU)

$$\text{RelU}(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{38}$$

shown in the left of Figure 9. In short, RelUs solve the problem that the tanh function gets "saturated" and has very small gradients when the absolute value of input x is very large (x is a large negative or positive number). Empirical results have often shown it to be an effective alternative to tanh, including for the language modeling task described in this chapter [110].
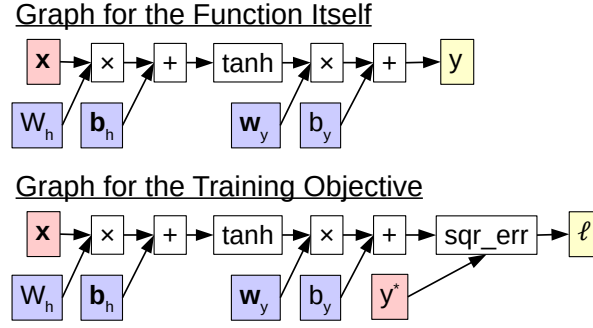
20

Figure 10: Computation graphs for the function itself, and the loss function.

So let's say we swap in a tanh non-linearity instead of the step function to our network, we can now proceed to calculate derivatives like we did in Section 4.3. First, we perform the full calculation of the loss function:

$$
\begin{aligned}
\boldsymbol{h}' &= W_{xh}\boldsymbol{x} + \boldsymbol{b}_h \\
\boldsymbol{h} &= \tanh(\boldsymbol{h}') \\
y &= \boldsymbol{w}_{hy}\boldsymbol{h} + b_y \\
\ell &= (y^* - y)^2.
\end{aligned}
\tag{39}
$$

Then, again using the chain rule, we calculate the derivatives of each set of parameters:

$$
\begin{aligned}
\frac{d\ell}{db_y} &= \frac{d\ell}{dy}\frac{dy}{db_y} \\
\frac{d\ell}{d\boldsymbol{w}_{hy}} &= \frac{d\ell}{dy}\frac{dy}{d\boldsymbol{w}_{hy}} \\
\frac{d\ell}{d\boldsymbol{b}_h} &= \frac{d\ell}{dy}\frac{dy}{d\boldsymbol{h}}\frac{d\boldsymbol{h}}{d\boldsymbol{h}'}\frac{d\boldsymbol{h}'}{d\boldsymbol{b}_h} \\
\frac{d\ell}{dW_{xh}} &= \frac{d\ell}{dy}\frac{dy}{d\boldsymbol{h}}\frac{d\boldsymbol{h}}{d\boldsymbol{h}'}\frac{d\boldsymbol{h}'}{dW_{xh}}.
\end{aligned}
\tag{40}
$$

We could go through all of the derivations above by hand and precisely calculate the gradients of all parameters in the model. Interested readers are free to do so, but even for a simple model like the one above, it is quite a lot of work and error prone. For more complicated models, like the ones introduced in the following chapters, this is even more the case.

Fortunately, when we actually implement neural networks on a computer, there is a very useful tool that saves us a large portion of this pain: **automatic differentiation** (autodiff) [116, 44]. To understand automatic differentiation, it is useful to think of our computation in Equation 39 as a data structure called a **computation graph**, two examples of which are shown in Figure 10. In these graphs, each node represents either an input to the network or the result of one computational operation, such as a multiplication, addition, tanh, or squared error. The first graph in the figure calculates the function of interest itself and would be used when we want to make predictions using our model, and the second graph calculates the loss function and would be used in training.

21

Automatic differentiation is a two-step dynamic programming algorithm that operates over the second graph and performs:

- **Forward calculation**, which traverses the nodes in the graph in topological order, calculating the actual result of the computation as in Equation 39.

- **Back propagation**, which traverses the nodes in reverse topological order, calculating the gradients as in Equation 40.

The nice thing about this formulation is that while the overall function calculated by the graph can be relatively complicated, as long as it can be created by combining multiple simple nodes for which we are able to calculate the function $f(x)$ and derivative $f'(x)$, we are able to use automatic differentiation to calculate its derivatives using this dynamic program without doing the derivation by hand.

Thus, to implement a general purpose training algorithm for neural networks, it is necessary to implement these two dynamic programs, as well as the atomic forward function and backward derivative computations for each type of node that we would need to use. While this is not trivial in itself, there are now a plethora of toolkits that either perform general-purpose auto-differentiation [7, 50], or auto-differentiation specifically tailored for machine learning and neural networks [1, 12, 26, 105, 78]. These implement the data structures, nodes, back-propogation, and parameter optimization algorithms needed to train neural networks in an efficient and reliable way, allowing practitioners to get started with designing their models. In the following sections, we will take this approach, taking a look at how to create our models of interest in a toolkit called `DyNet`,[13] which has a programming interface that makes it relatively easy to implement the sequence-to-sequence models covered here.[14]

5.4   An Example Implementation

Figure 11 shows an example of implementing the above neural network in `DyNet`, which we'll step through line-by-line. Lines 1-2 import the necessary libraries. Lines 4-5 specify parameters of the models: the size of the hidden vector $\boldsymbol{h}$ and the number of epochs (passes through the data) for which we'll perform training. Line 7 initializes a `DyNet` model, which will store all the parameters we are attempting to learn. Lines 8-11 initialize parameters W_{xh}, $\boldsymbol{b}_h$, $\boldsymbol{w}_{hy}$, and b_y to be the appropriate size so that dimensions in the equations for Equation 39 match. Line 12 initializes a "trainer", which will update the parameters in the model according to an update strategy (here we use simple stochastic gradient descent, but trainers for AdaGrad, Adam, and other strategies also exist). Line 14 creates the training data for the function in Figure 7.

Lines 16-25 define a function that takes input $\boldsymbol{x}$ and creates a computation graph to calculate Equation 39. First, line 17 creates a new computation graph to hold the computation for this particular training example. Lines 18-21 take the parameters (stored in the model) and adds them to the computation graph as `DyNet` variables for this particular training example. Line 22 takes a Python list representing the current input and puts it into the computation graph as a `DyNet` variable. Line 23 calculates the hidden vector $\boldsymbol{h}$, Line 24 calculates the value y, and Line 25 returns it.

---

[13]http://github.com/clab/dynet
[14]It is also developed by the author of these materials, so the decision might have been a wee bit biased.

22

```python
import dynet as dy
import random
# Parameters of the model and training
HIDDEN_SIZE = 20
NUM_EPOCHS = 20
# Define the model and SGD optimizer
model = dy.Model()
W_xh_p = model.add_parameters((HIDDEN_SIZE, 2))
b_h_p = model.add_parameters(HIDDEN_SIZE)
W_hy_p = model.add_parameters((1, HIDDEN_SIZE))
b_y_p = model.add_parameters(1)
trainer = dy.SimpleSGDTrainer(model)
# Define the training data, consisting of (x,y) tuples
data = [([1,1],1), ([-1,1],-1), ([1,-1],-1), ([-1,-1],1)]
# Define the function we would like to calculate
def calc_function(x):
  dy.renew_cg()
  w_xh = dy.parameter(w_xh_p)
  b_h = dy.parameter(b_h_p)
  W_hy = dy.parameter(W_hy_p)
  b_y = dy.parameter(b_y_p)
  x_val = dy.inputVector(x)
  h_val = dy.tanh(w_xh * x_val + b_h)
  y_val = W_hy * h_val + b_y
  return y_val
# Perform training
for epoch in range(NUM_EPOCHS):
  epoch_loss = 0
  random.shuffle(data)
  for x, ystar in data:
    y = calc_function(x)
    loss = dy.squared_distance(y, dy.scalarInput(ystar))
    epoch_loss += loss.value()
    loss.backward()
    trainer.update()
  print("Epoch %d: loss=%f" % (epoch, epoch_loss))
# Print results of prediction
for x, ystar in data:
  y = calc_function(x)
  print("%r -> %f" % (x, y.value()))
```

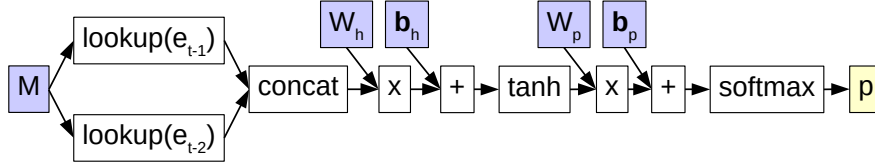Figure 11: An example of training a neural network for a multi-layer perceptron using the toolkit DyNet.

23

Figure 12: A computation graph for a tri-gram feed-forward neural language model.

Lines 27-36 perform training for NUM_EPOCHS passes over the data (one pass through the training data is usually called an "epoch"). Line 28 creates a variable to keep track of the loss for this epoch for later reporting. Line 29 shuffles the data, as recommended in Section 4.2. Lines 30-35 perform stochastic gradient descent, looping over each of the training examples. Line 31 creates a computation for the function itself, and Line 32 adds computation for the loss function. Line 33 runs the forward calculation to calculate the loss and adds it to the loss for this epoch. Line 34 runs back propagation, and Line 35 updates the model parameters. At the end of the epoch, we print the loss for the epoch in Line 36 to make sure that the loss is going down and our model is converging.

Finally, at the end of training in Lines 38-40, we print the output results. In an actual scenario, this would be done on a separate set of test data.

5.5 Neural-network Language Models

Now that we have the basics down, it is time to apply neural networks to language modeling [76, 9]. A feed-forward neural network language model is very much like the log-linear language model that we mentioned in the previous section, simply with the addition of one or more non-linear layers before the output.

First, let's recall the tri-gram log-linear language model. In this case, assume we have two sets of features expressing the identity of e_{t-1} (represented as $W^{(1)}$) and e_{t-2} (as $W^{(2)}$), the equation for the log-linear model looks like this:

$$\boldsymbol{s} = W^{(1)}_{\cdot,e_{t-1}} + W^{(2)}_{\cdot,e_{t-2}} + \boldsymbol{b}$$
$$\boldsymbol{p} = \text{softmax}(\boldsymbol{s}), \tag{41}$$

where we add the appropriate columns from the weight matricies to the bias to get the score, then take the softmax to turn it into a probability.

Compared to this, a tri-gram neural network model with a single layer is structured as shown in Figure 12 and described in equations below:

$$\boldsymbol{m} = \text{concat}(M_{\cdot,e_{t-2}}, M_{\cdot,e_{t-1}})$$
$$\boldsymbol{h} = \tanh(W_{mh}\boldsymbol{m} + \boldsymbol{b}_h)$$
$$\boldsymbol{s} = W_{hs}\boldsymbol{h} + \boldsymbol{b}_s$$
$$\boldsymbol{p} = \text{softmax}(\boldsymbol{s}) \tag{42}$$

In the first line, we obtain a vector $\boldsymbol{m}$ representing the context e_{i-n+1}^{i-1} (in the particular case above, we are handling a tri-gram model so $n = 3$). Here, M is a matrix with $|V|$ columns, and L_m rows, where each column corresponds to an L_m-length vector representing a single

24

```
1  # Define the lookup parameters at model definition time
2  # VOCAB_SIZE is the number of words in the vocabulary
3  # EMBEDDINGS_SIZE is the length of the word embedding vector
4  M_p = model.add_lookup_parameters((VOCAB_SIZE, EMBEDDING_SIZE))
5  # Load the parameters into the computation graph
6  M = dy.lookup(M_p)
7  # And look up the vector for word i
8  m = M[i]
```

Figure 13: Code for looking things up in DyNet.

word in the vocabulary. This vector is called a **word embedding** or a **word representation**, which is a vector of real numbers corresponding to particular words in the vocabulary.[15] The interesting thing about expressing words as vectors of real numbers is that each element of the vector could reflect a different aspect of the word. For example, there may be an element in the vector determining whether a particular word under consideration could be a noun, or another element in the vector expressing whether the word is an animal, or another element that expresses whether the word is countable or not.[16] Figure 13 shows an example of how to define parameters that allow you to look up a vector in DyNet.

The vector $\boldsymbol{m}$ then results from the concatenation of the word vectors for all of the words in the context, so $|\boldsymbol{m}| = L_m * (n - 1)$. Once we have this $\boldsymbol{m}$, we run the vectors through a hidden layer to obtain vector $\boldsymbol{h}$. By doing so, the model can learn combination features that reflect information regarding multiple words in the context. This allows the model to be expressive enough to represent the more difficult cases in Figure 6. For example, given a context is "cows eat", and some elements of the vector $M_{\cdot,\text{cows}}$ identify the word as a "large farm animal" (e.g. "cow", "horse", "goat"), while some elements of $M_{\cdot,\text{eat}}$ corresponds to "eat" and all of its relatives ("consume", "chew", "ingest"), then we could potentially learn a unit in the hidden layer $\boldsymbol{h}$ that is active when we are in a context that represents "things farm animals eat".

Next, we calculate the score vector for each word: $\boldsymbol{s} \in \mathbb{R}^{|V|}$. This is done by performing an affine transform of the hidden vector $\boldsymbol{h}$ with a weight matrix $W_{hs} \in \mathbb{R}^{|V| \times |\boldsymbol{h}|}$ and adding a bias vector $\boldsymbol{b}_s \in \mathbb{R}^{|V|}$. Finally, we get a probability estimate $\boldsymbol{p}$ by running the calculated scores through a softmax function, like we did in the log-linear language models. For training, if we know e_t we can also calculate the loss function as follows, similarly to the log-linear model:

$$\ell = -\log(p_{e_t}). \tag{43}$$

DyNet has a convenience function that, given a score vector $\boldsymbol{s}$, will calculate the negative log likelihood loss:

---

[15]For the purposes of the model in this chapter, these vectors can basically be viewed as one set of tunable parameters in the neural language model, but there has also been a large amount of interest in learning these vectors for use in other tasks. Some methods are outlined in Section 5.6.

[16]In reality, it is rare that single elements in the vector have such an intuitive meaning unless we impose some sort of constraint, such as sparsity constraints [75].

```
1 loss = dy.pickneglogsoftmax(s, e_t)
```

The reasons why the neural network formulation is nice becomes apparent when we compare this to n-gram language models in Section 3:

Better generalization of contexts: n-gram language models treat each word as its own discrete entity. By using input embeddings M, it is possible to group together similar words so they behave similarly in the prediction of the next word. In order to do the same thing, n-gram models would have to explicitly learn word classes and using these classes effectively is not a trivial problem [15].

More generalizable combination of words into contexts: In an n-gram language model, we would have to remember parameters for all combinations of $\{\text{cow}, \text{horse}, \text{goat}\} \times \{\text{consume}, \text{chew}, \text{ingest}\}$ to represent the context "things farm animals eat". This would be quadratic in the number of words in the class, and thus learning these parameters is difficult in the face of limited training data. Neural networks handle this problem by learning nodes in the hidden layer that can represent this quadratic combination in a feature-efficient way.

Ability to skip previous words: n-gram models generally fall back sequentially from longer contexts (e.g. "the two previous words e_{t-2}^{t-1}") to shorter contexts (e.g. "the previous words e_{t-1}"), but this doesn't allow them to "skip" a word and only reference for example, "the word two words ago e_{t-2}". Log-linear models and neural networks can handle this skipping naturally.

5.6   Further Reading

In addition to the methods described above, there are a number of extensions to neural-network language models that are worth discussing.

Softmax approximations: One problem with the training of log-linear or neural network language models is that at every training example, they have to calculate the large score vector $\boldsymbol{s}$, then run a softmax over it to get probabilities. As the vocabulary size $|V|$ grows larger, this can become quite time-consuming. As a result, there are a number of ways to reduce training time. One example are methods that sample a subset of the vocabulary $V' \in V$ where $|V'| \ll V$, then calculate the scores and approximate the loss over this smaller subset. Examples of these include methods that simply try to get the true word e_t to have a higher score (by some margin) than others in the subsampled set [27] and more probabilistically motivated methods, such as **importance sampling** [10] or **noise-contrastive estimation** (NCE; [74]). Interestingly, for other objective functions such as linear regression and special variety of softmax called the **spherical softmax**, it is possible to calculate the objective function in ways that do not scale linearly with the vocabulary size [111].

Other softmax structures: Another interesting trick to improve training speed is to create a softmax that is structured so that its loss functions can be computed efficiently. One

way to do so is the class-based softmax [40], which assigns each word e_t to a class c_t, then divides computation into two steps: predicting the probability of class c_t given the context, then predicting the probability of the word e_t given the class and the current context $P(e_t \mid c_t, e_{t-n+1}^{t-1})P(c_t \mid e_{t-n+1}^{t-1})$. The advantage of this method is that we only need to calculate scores for the correct class c_t out of $|C|$ classes, then the correct word e_t out of the vocabulary for class c_t, which is size $|V_{c_t}|$. Thus, our computational complexity becomes $O(|C| + |V_{c_t}|)$ instead of $O(|V|)$.[17] The hierarchical softmax [73] takes this a step further by predicting words along a binary-branching tree, which results in a computational complexity of $O(\log_2|V|)$.

Other models to learn word representations: As mentioned in Section 5.5, we learn word embeddings M as a by-product of training our language models. One very nice feature of word representations is that language models can be trained purely on raw text, but the resulting representations can capture semantic or syntactic features of the words, and thus can be used to effectively improve down-stream tasks that don't have a lot of annotated data, such as part-of-speech tagging or parsing [107].[18] Because of their usefulness, there have been an extremely large number of approaches proposed to learn different varieties of word embeddings,[19] from early work based on distributional similarity and dimensionality reduction [93, 108] to more recent models based on predictive models similar to language models [107, 71], with the general current thinking being that predictive models are the more effective and flexible of the two [5].The most well-known methods are the continuous-bag-of-words and skip-gram models implemented in the software `word2vec`,[20] which define simple objectives for predicting words using the immediately surrounding context or vice-versa. `word2vec` uses a sampling-based approach and parallelization to easily scale up to large datasets, which is perhaps the primary reason for its popularity. One thing to note is that these methods are not language models in themselves, as they do not calculate a probability of the sentence $P(E)$, but many of the parameter estimation techniques can be shared.

5.7 Exercise

In the exercise for this chapter, we will use `DyNet` to construct a feed-forward language model and evaluate its performance.

Writing the program will entail:

- Writing a function to read in the data and (turn it into numerical IDs).

- Writing a function to calculate the loss function by looking up word embeddings, then running them through a multi-layer perceptron, then predicting the result.

- Writing code to perform training using this function.

---

[17] *Question: What is the ideal class size to achieve the best computational efficiency?*

[18] Manning (2015) called word embeddings the "Sriracha sauce of NLP", because you can add them to anything to make it better http://nlp.stanford.edu/~manning/talks/ NAACL2015-VSM-Compositional-Deep-Learning.pdf

[19] So many that Daumé III (2016) called word embeddings the "Sriracha sauce of NLP: it sounds like a good idea, you add too much, and now you're crying" https://twitter.com/haldaume3/status/ 706173575477080065

[20] https://code.google.com/archive/p/word2vec/

- Writing evaluation code that measures the perplexity on a held-out data set.

Language modeling accuracy should be measured in the same way as previous exercises and compared with the previous models.

Potential improvements to the model include tuning the various parameters of the model. How big should $\boldsymbol{h}$ be? Should we add additional hidden layers? What optimizer with what learning rate should we use? What happens if we implement one of the more efficient versions of the softmax explained in Section 5.6?

6 Recurrent Neural Network Language Models

The neural-network models presented in the previous chapter were essentially more powerful and generalizable versions of n-gram models. In this section, we talk about language models based on recurrent neural networks (RNNs), which have the additional ability to capture long-distance dependencies in language.

6.1 Long Distance Dependencies in Language

He doesn't have very much confidence in himself
She doesn't have very much confidence in herself

Figure 14: An example of long-distance dependencies in language.

Before speaking about RNNs in general, it's a good idea to think about the various reasons a model with a limited history would not be sufficient to properly model all phenomena in language.

One example of a long-range grammatical constraint is shown in Figure 14. In this example, there is a strong constraint that the starting "he" or "her" and the final "himself" or "herself" must match in gender. Similarly, based on the subject of the sentence, the conjugation of the verb will change. These sorts of dependencies exist regardless of the number of intervening words, and models with a finite history e_{i-n+1}^{i-1}, like the one mentioned in the previous chapter, will never be able to appropriately capture this. These dependencies are frequent in English but even more prevalent in languages such as Russian, which has a large number of forms for each word, which must match in case and gender with other words in the sentence.[21]

Another example where long-term dependencies exist is in **selectional preferences** [85]. In a nutshell, selectional preferences are basically common sense knowledge of "what will do what to what". For example, "I ate salad with a fork" is perfectly sensible with "a fork" being a tool, and "I ate salad with my friend" also makes sense, with "my friend" being a companion. On the other hand, "I ate salad with a backpack" doesn't make much sense because a backpack is neither a tool for eating nor a companion. These selectional preference violations lead to nonsensical sentences and can also span across an arbitrary length due to the fact that subjects, verbs, and objects can be separated by a great distance.

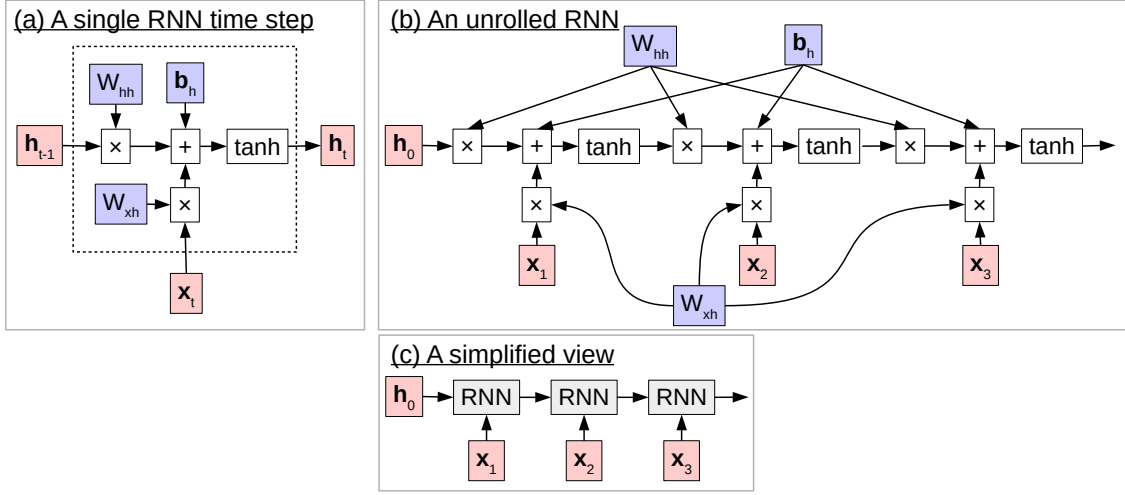[21]See `https://en.wikipedia.org/wiki/Russian_grammar` for an overview.

Figure 15: Examples of computation graphs for neural networks. (a) shows a single time step. (b) is the unrolled network. (c) is a simplified version of the unrolled network, where gray boxes indicate a function that is parameterized (in this case by W_{xh}, W_{hh}, and $\boldsymbol{b}_h$).

Finally, there are also dependencies regarding the **topic** or **register** of the sentence or document. For example, it would be strange if a document that was discussing a technical subject suddenly started going on about sports – a violation of topic consistency. It would also be unnatural for a scientific paper to suddenly use informal or profane language – a lack of consistency in register.

These and other examples describe why we need to model long-distance dependencies to create workable applications.

6.2   Recurrent Neural Networks

Recurrent neural networks (RNNs; [33]) are a variety of neural network that makes it possible to model these long-distance dependencies. The idea is simply to add a connection that references the previous hidden state $\boldsymbol{h}_{t-1}$ when calculating hidden state $\boldsymbol{h}$, written in equations as:

$$\boldsymbol{h}_t = \begin{cases} \tanh(W_{xh}\boldsymbol{x}_t + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h) & t \geq 1, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{44}$$

As we can see, for time steps $t \geq 1$, the only difference from the hidden layer in a standard neural network is the addition of the connection $W_{hh}\boldsymbol{h}_{t-1}$ from the hidden state at time step $t-1$ connecting to that at time step t. As this is a recursive equation that uses $\boldsymbol{h}_{t-1}$ from the previous time step. This single time step of a recurrent neural network is shown visually in the computation graph shown in Figure 15(a).

When performing this visual display of RNNs, it is also common to "unroll" the neural network in time as shown in Figure 15(b), which makes it possible to explicitly see the information flow between multiple time steps. From unrolling the network, we can see that we are still dealing with a standard computation graph in the same form as our feed-forward networks, on which we can still do forward computation and backward propagation, making

it possible to learn our parameters. It also makes clear that the recurrent network has to start somewhere with an initial hidden state $\boldsymbol{h}_0$. This initial state is often set to be a vector full of zeros, treated as a parameter $\boldsymbol{h}_{\text{init}}$ and learned, or initialized according to some other information (more on this in Section 7).

Finally, for simplicity, it is common to abbreviate the whole recurrent neural network step with a single block "RNN" as shown in Figure 15. In this example, the boxes corresponding to RNN function applications are gray, to show that they are internally parameterized with W_{xh}, W_{hh}, and $\boldsymbol{b}_h$. We will use this convention in the future to represent parameterized functions.

RNNs make it possible to model long distance dependencies because they have the ability to pass information between timesteps. For example, if some of the nodes in $\boldsymbol{h}_{t-1}$ encode the information that "the subject of the sentence is male", it is possible to pass on this information to $\boldsymbol{h}_t$, which can in turn pass it on to $\boldsymbol{h}_{t+1}$ and on to the end of the sentence. This ability to pass information across an arbitrary number of consecutive time steps is the strength of recurrent neural networks, and allows them to handle the long-distance dependencies described in Section 6.1.

Once we have the basics of RNNs, applying them to language modeling is (largely) straight-forward [72]. We simply take the feed-forward language model of Equation 42 and enhance it with a recurrent connection as follows:

$$
\begin{aligned}
\boldsymbol{m}_t &= M_{.,e_{t-1}} \\
\boldsymbol{h}_t &= \begin{cases} \tanh(W_{mh}\boldsymbol{m}_t + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h) & t \geq 1, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \\
\boldsymbol{p}_t &= \text{softmax}(W_{hs}\boldsymbol{h}_t + b_s).
\end{aligned}
\tag{45}
$$

One thing that should be noted is that compared to the feed-forward language model, we are only feeding in the previous word instead of the two previous words. The reason for this is because (if things go well) we can expect that information about e_{t-2} and all previous words are already included in $\boldsymbol{h}_{t-1}$, making it unnecessary to feed in this information directly.

Also, for simplicity of notation, it is common to abbreviate the equation for $\boldsymbol{h}_t$ with a function $\text{RNN}(\cdot)$, following the simplified view of drawing RNNs in Figure 15(c):

$$
\begin{aligned}
\boldsymbol{m}_t &= M_{.,e_{t-1}} \\
\boldsymbol{h}_t &= \text{RNN}(\boldsymbol{m}_t, \boldsymbol{h}_{t-1}) \\
\boldsymbol{p}_t &= \text{softmax}(W_{hs}\boldsymbol{h}_t + b_s).
\end{aligned}
\tag{46}
$$

6.3 The Vanishing Gradient and Long Short-term Memory

However, while the RNNs in the previous section are conceptually simple, they also have problems: the **vanishing gradient** problem and the closely related cousin, the **exploding gradient** problem.

A conceptual example of the vanishing gradient problem is shown in Figure 16. In this example, we have a recurrent neural network that makes a prediction after several times steps, a model that could be used to classify documents or perform any kind of prediction over a sequence of text. After it makes its prediction, it gets a loss that it is expected to back-propagate over all time steps in the neural network. However, at each time step, when we run

$$\frac{dl}{d_{h_0}} = \text{tiny} \qquad \frac{dl}{d_{h_1}} = \text{small} \qquad \frac{dl}{d_{h_2}} = \text{med.} \qquad \frac{dl}{d_{h_3}} = \text{large}$$
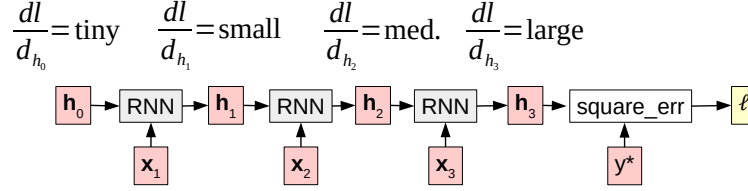
Figure 16: An example of the vanishing gradient problem.

the back propagation algorithm, the gradient gets smaller and smaller, and by the time we get back to the beginning of the sentence, we have a gradient so small that it effectively has no ability to have a significant effect on the parameters that need to be updated. The reason why this effect happens is because unless $\frac{d\boldsymbol{h}_{t-1}}{d\boldsymbol{h}_t}$ is exactly one, it will tend to either diminish or amplify the gradient $\frac{d\ell}{d\boldsymbol{h}_t}$, and when this diminishment or amplification is done repeatedly, it will have an exponential effect on the gradient of the loss.[22]

One method to solve this problem, in the case of diminishing gradients, is the use of a neural network architecture that is specifically designed to ensure that the derivative of the recurrent function is exactly one. A neural network architecture designed for this very purpose, which has enjoyed quite a bit of success and popularity in a wide variety of sequential processing tasks, is the **long short-term memory** (LSTM; [49]) neural network architecture. The most fundamental idea behind the LSTM is that in addition to the standard hidden state $\boldsymbol{h}$ used by most neural networks, it also has a **memory cell** $\boldsymbol{c}$, for which the gradient $\frac{d\boldsymbol{c}_t}{d\boldsymbol{c}_{t-1}}$ is exactly one. Because this gradient is exactly one, information stored in the memory cell does not suffer from vanishing gradients, and thus LSTMs can capture long-distance dependencies more effectively than standard recurrent neural networks.

So how do LSTMs do this? To understand this, let's take a look at the LSTM architecture in detail, as shown in Figure 17 and the following equations:

$$\boldsymbol{u}_t = \tanh(W_{xu}\boldsymbol{x}_t + W_{hu}h_{t-1} + \boldsymbol{b}_u) \tag{47}$$

$$\boldsymbol{i}_t = \sigma(W_{xi}\boldsymbol{x}_t + W_{hi}h_{t-1} + \boldsymbol{b}_i) \tag{48}$$

$$\boldsymbol{o}_t = \sigma(W_{xo}\boldsymbol{x}_t + W_{ho}h_{t-1} + \boldsymbol{b}_o) \tag{49}$$

$$\boldsymbol{c}_t = \boldsymbol{i}_t \odot \boldsymbol{u}_t + \boldsymbol{c}_{t-1} \tag{50}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t). \tag{51}$$

Taking the equations one at a time: Equation 47 is the update, which is basically the same as the RNN update in Equation 44; it takes in the input and hidden state, performs an affine transform and runs it through the tanh non-linearity.

Equation 48 and Equation 49 are the **input gate** and **output gate** of the LSTM respectively. The function of "gates", as indicated by their name, is to either allow information to pass through or block it from passing. Both of these gates perform an affine transform

---

[22]This is particularly detrimental in the case where we receive a loss only once at the end of the sentence, like the example above. One real-life example of such a scenario is document classification, and because of this, RNNs have been less successful in this task than other methods such as convolutional neural networks, which do not suffer from the vanishing gradient problem [59, 63]. It has been shown that pre-training an RNN as a language model before attempting to perform classification can help alleviate this problem to some extent [29].
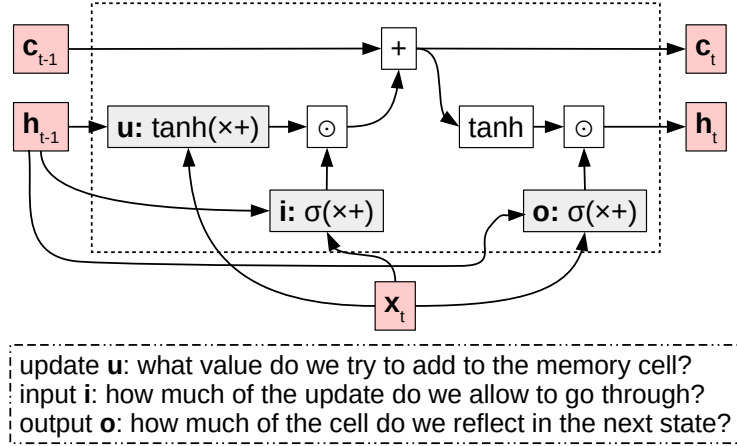
Figure 17: A single time step of long short-term memory (LSTM). The information flow between the $\boldsymbol{h}$ and cell $\boldsymbol{c}$ is modulated using parameterized input and output gates.

followed by the **sigmoid function**, also called the **logistic function**[23]

$$\sigma(x) = \frac{1}{1 + \exp(-x)}, \tag{52}$$

which squashes the input between 0 (which $\sigma(x)$ will approach as x becomes more negative) and 1 (which $\sigma(x)$ will approach as x becomes more positive). The output of the sigmoid is then used to perform a componentwise multiplication

$$\boldsymbol{z} = \boldsymbol{x} \odot \boldsymbol{y}$$
$$z_i = x_i * y_i$$

with the output of another function. This results in the "gating" effect: if the result of the sigmoid is close to one for a particular vector position, it will have little effect on the input (the gate is "open"), and if the result of the sigmoid is close to zero, it will block the input, setting the resulting value to zero (the gate is "closed").

Equation 50 is the most important equation in the LSTM, as it is the equation that implements the intuition that $\frac{d\boldsymbol{c}_t}{d\boldsymbol{c}_{t-1}}$ must be equal to one, which allows us to conquer the vanishing gradient problem. This equation sets $\boldsymbol{c}_t$ to be equal to the update $\boldsymbol{u}_t$ modulated by the input gate $\boldsymbol{i}_t$ plus the cell value for the previous time step $\boldsymbol{c}_{t-1}$. Because we are directly adding $\boldsymbol{c}_{t-1}$ to $\boldsymbol{c}_t$, if we consider only this part of Equation 50, we can easily confirm that the gradient will indeed be one.[24]

Finally, Equation 51 calculates the next hidden state of the LSTM. This is calculated by using a tanh function to scale the cell value between -1 and 1, then modulating the output

---

[23] To be more accurate, the sigmoid function is actually any mathematical function having an s-shaped curve, so the tanh function is also a type of sigmoid function. The logistic function is also a slightly broader class of functions $f(x) = \frac{L}{1 + \exp(-k(x - x_0))}$. However, in the machine learning literature, the "sigmoid" is usually used to refer to the particular variety in Equation 52.

[24] In actuality $\boldsymbol{i}_t \odot \boldsymbol{u}_t$ is also affected by $\boldsymbol{c}_{t-1}$, and thus $\frac{d\boldsymbol{c}_t}{d\boldsymbol{c}_{t-1}}$ is not exactly one, but the effect is relatively indirect. Especially for vector elements with $\boldsymbol{i}_t$ close to zero, the effect will be minimal.

using the output gate value $\boldsymbol{o}_t$. This will be the value actually used in any downstream calculation, such as the calculation of language model probabilities.

$$\boldsymbol{p}_t = \mathrm{softmax}(W_{hs}\boldsymbol{h}_t + b_s). \tag{53}$$

6.4 Other RNN Variants

Because of the importance of recurrent neural networks in a number of applications, many variants of these networks exist. One modification to the standard LSTM that is used widely (in fact so widely that most people who refer to "LSTM" are now referring to this variant) is the addition of a **forget gate** [38]. The equations for the LSTM with a forget gate are shown below:

$$\begin{aligned}
\boldsymbol{u}_t &= \tanh(W_{xu}\boldsymbol{x}_t + W_{hu}h_{t-1} + \boldsymbol{b}_u) \\
\boldsymbol{i}_t &= \sigma(W_{xi}\boldsymbol{x}_t + W_{hi}h_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{f}_t &= \sigma(W_{xf}\boldsymbol{x}_t + W_{hf}h_{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{o}_t &= \sigma(W_{xo}\boldsymbol{x}_t + W_{ho}h_{t-1} + \boldsymbol{b}_o) \\
\boldsymbol{c}_t &= \boldsymbol{i}_t \odot \boldsymbol{u}_t + \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t).
\end{aligned} \tag{54} \tag{55}$$

Compared to the standard LSTM, there are two changes. First, in Equation 54, we add an additional gate, the forget gate. Second, in Equation 55, we use the gate to modulate the passing of the previous cell $\boldsymbol{c}_{t-1}$ to the current cell $\boldsymbol{c}_t$. This forget gate is useful in that it allows the cell to easily clear its memory when justified: for example, let's say that the model has remembered that it has seen a particular word strongly correlated with another word, such as "he" and "himself" or "she" and "herself" in the example above. In this case, we would probably like the model to remember "he" until it is used to predict "himself", then forget that information, as it is no longer relevant. Forget gates have the advantage of allowing this sort of fine-grained information flow control, but they also come with the risk that if $\boldsymbol{f}_t$ is set to zero all the time, the model will forget everything and lose its ability to handle long-distance dependencies. Thus, at the beginning of neural network training, it is common to initialize the bias $\boldsymbol{b}_f$ of the forget gate to be a somewhat large value (e.g. 1), which will make the neural net start training without using the forget gate, and only gradually start forgetting content after the net has been trained to some extent.

While the LSTM provides an effective solution to the vanishing gradient problem, it is also rather complicated (as many readers have undoubtedly been feeling). One simpler RNN variant that has nonetheless proven effective is the **gated recurrent unit** (GRU; [24]), expressed in the following equations:

$$\boldsymbol{r}_t = \sigma(W_{xr}\boldsymbol{x}_t + W_{hr}h_{t-1} + \boldsymbol{b}_r) \tag{56}$$

$$\boldsymbol{z}_t = \sigma(W_{xz}\boldsymbol{x}_t + W_{hz}h_{t-1} + \boldsymbol{b}_z) \tag{57}$$

$$\tilde{\boldsymbol{h}}_t = \tanh(W_{xh}\boldsymbol{x}_t + W_{hh}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{58}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t)\boldsymbol{h}_{t-1} + \boldsymbol{z}_t\tilde{\boldsymbol{h}}_t. \tag{59}$$

The most characteristic element of the GRU is Equation 59, which interpolates between a candidate for the updated hidden state $\tilde{\boldsymbol{h}}_t$ and the previous state $\tilde{\boldsymbol{h}}_{t-1}$. This interpolation is

modulated by an **update gate z_t** (Equation 57), where if the update gate is close to one, the GRU will use the new candidate hidden value, and if the update is close to zero, it will use the previous value. The candidate hidden state is calculated by Equation 58, which is similar to a standard RNN update but includes an additional modulation of the hidden state input by a **reset gate r_t** calculated in Equation 56. Compared to the LSTM, the GRU has slightly fewer parameters (it performs one less parameterized affine transform) and also does not have a separate concept of a "cell". Thus, GRUs have been used by some to conserve memory or computation time.
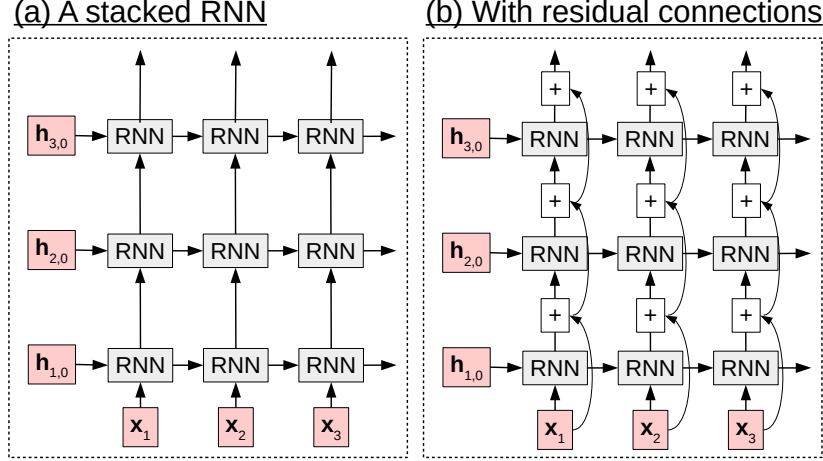

Figure 18: An example of (a) stacked RNNs and (b) stacked RNNs with residual connections.

One other important modification we can do to RNNs, LSTMs, GRUs, or really any other neural network layer is simple but powerful: stack multiple layers on top of each other (**stacked RNNs** Figure 18(a)). For example, in a 3-layer stacked RNN, the calculation at time step t would look as follows:

$$\begin{aligned}
\boldsymbol{h}_{1,t} &= \mathrm{RNN}_1(\boldsymbol{x}_t, \boldsymbol{h}_{1,t-1}) \\
\boldsymbol{h}_{2,t} &= \mathrm{RNN}_2(\boldsymbol{h}_{1,t}, \boldsymbol{h}_{2,t-1}) \\
\boldsymbol{h}_{3,t} &= \mathrm{RNN}_3(\boldsymbol{h}_{2,t}, \boldsymbol{h}_{3,t-1}),
\end{aligned}$$

where $\boldsymbol{h}_{n,t}$ is the hidden state for the nth layer at time step t, and $\mathrm{RNN}(\cdot)$ is an abbreviation for the RNN equation in Equation 44. Similarly, we could substitute this function for $\mathrm{LSTM}(\cdot)$, $\mathrm{GRU}(\cdot)$, or any other recurrence step. The reason why stacking multiple layers on top of each other is useful is for the same reason that non-linearities proved useful in the standard neural networks introduced in Section 5: they are able to progressively extract more abstract features of the current words or sentences. For example, [98] find evidence that in a two-layer stacked LSTM, the first layer tends to learn granular features of words such as part of speech tags, while the second layer learns more abstract features of the sentence such as voice or tense.

While stacking RNNs has potential benefits, it also has the disadvantage that it suffers from the vanishing gradient problem in the vertical direction, just as the standard RNN did in the horizontal direction. That is to say, the gradient will be back-propagated from the layer

close to the output (RNN_3) to the layer close to the input (RNN_1), and the gradient may vanish in the process, causing the earlier layers of the network to be under-trained. A simple solution to this problem, analogous to what the LSTM does for vanishing gradients over time, is **residual networks** (Figure 18(b)) [47]. The idea behind these networks is simply to add the output of the previous layer directly to the result of the next layer as follows:

$$\boldsymbol{h}_{1,t} = \text{RNN}_1(\boldsymbol{x}_t, \boldsymbol{h}_{1,t-1}) + \boldsymbol{x}_t$$
$$\boldsymbol{h}_{2,t} = \text{RNN}_2(\boldsymbol{h}_{1,t}, \boldsymbol{h}_{2,t-1}) + \boldsymbol{h}_{1,t}$$
$$\boldsymbol{h}_{3,t} = \text{RNN}_3(\boldsymbol{h}_{2,t}, \boldsymbol{h}_{3,t-1}) + \boldsymbol{h}_{2,t}.$$

As a result, like the LSTM, there is no vanishing of gradients due to passing through the $\text{RNN}(\cdot)$ function, and even very deep networks can be learned effectively.

6.5 Online, Batch, and Minibatch Training

As the observant reader may have noticed, the previous sections have gradually introduced more and more complicated models; we started with a simple linear model, added a hidden layer, added recurrence, added LSTM, and added more layers of LSTMs. While these more expressive models have the ability to model with higher accuracy, they also come with a cost: largely expanded parameter space (causing more potential for overfitting) and more complicated operations (causing much greater potential computational cost). This section describes an effective technique to improve the stability and computational efficiency of training these more complicated networks, **minibatching**.

Up until this point, we have used the stochastic gradient descent learning algorithm introduced in Section 4.2 that performs updates according to the following iterative process. This type of learning, which performs updates a single example at a time is called **online learning**.

Algorithm 1 A fully online training algorithm

1: **procedure** ONLINE
2:　　**for** several epochs of training **do**
3:　　　　**for** each training example in the data **do**
4:　　　　　　Calculate gradients of the loss
5:　　　　　　Update the parameters according to this gradient
6:　　　　**end for**
7:　　**end for**
8: **end procedure**

In contrast, we can also think of a **batch learning** algorithm, which treats the entire data set as a single unit, calculates the gradients for this unit, then only performs update after making a full pass through the data.

These two update strategies have trade-offs.

- Online training algorithms usually find a relatively good solution more quickly, as they don't need to make a full pass through the data before performing an update.

- However, at the end of training, batch learning algorithms can be more stable, as they are not overly influenced by the most recently seen training examples.

35

Algorithm 2 A batch learning algorithm

1: **procedure** BATCH
2:     **for** several epochs of training **do**
3:         **for** each training example in the data **do**
4:             Calculate and accumulate gradients of the loss
5:         **end for**
6:         Update the parameters according to the accumulated gradient
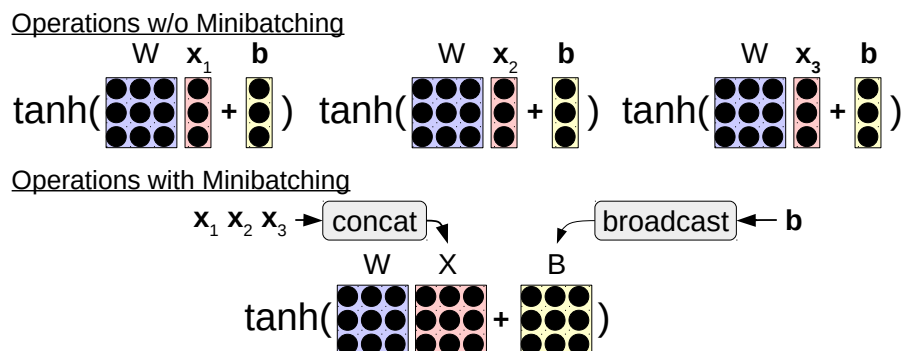7:   **end for**
8: **end procedure**

Figure 19: An example of combining multiple operations together when minibatching.

- Batch training algorithms are also more prone to falling into local optima; the randomness in online training algorithms often allows them to bounce out of local optima and find a better global solution.

Minibatching is a happy medium between these two strategies. Basically, minibatched training is similar to online training, but instead of processing a single training example at a time, we calculate the gradient for n training examples at a time. In the extreme case of $n = 1$, this is equivalent to standard online training, and in the other extreme where n equals the size of the corpus, this is equivalent to fully batched training. In the case of training language models, it is common to choose minibatches of $n = 1$ to $n = 128$ sentences to process at a single time. As we increase the number of training examples, each parameter update becomes more informative and stable, but the amount of time to perform one update increases, so it is common to choose an n that allows for a good balance between the two.

One other major advantage of minibatching is that by using a few tricks, it is actually possible to make the simultaneous processing of n training examples significantly faster than processing n different examples separately. Specifically, by taking multiple training examples and grouping similar operations together to be processed simultaneously, we can realize large gains in computational efficiency due to the fact that modern hardware (particularly GPUs, but also CPUs) have very efficient vector processing instructions that can be exploited with appropriately structured inputs. As shown in Figure 19, common examples of this in neural networks include grouping together matrix-vector multiplies from multiple examples into a single matrix-matrix multiply or performing an element-wise operation (such as tanh) over
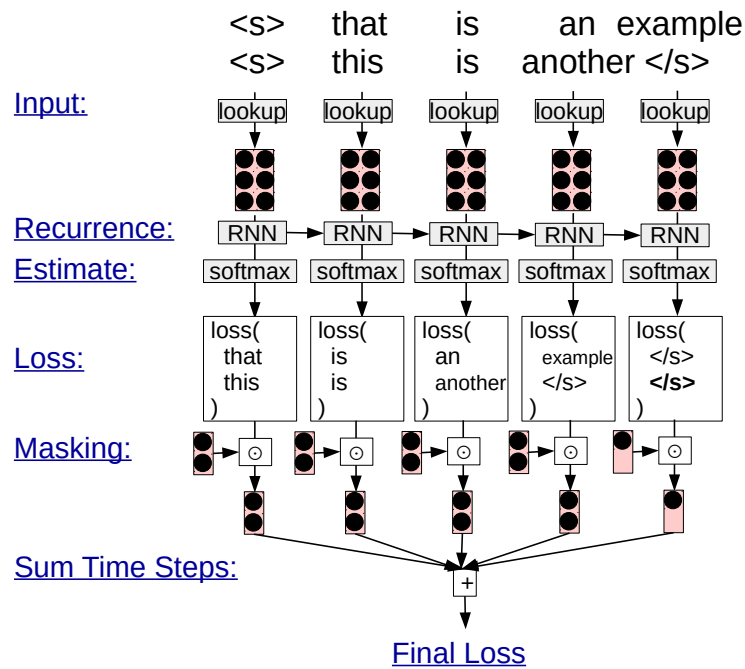
Figure 20: An example of minibatching in an RNN language model.

multiple vectors at the same time as opposed to processing single vectors individually. Luckily, in DyNet, the library we are using, this is relatively easy to do, as much of the machinery for each elementary operation is handled automatically. We'll give an example of the changes that we need to make when implementing an RNN language model below.

The basic idea in the batched RNN language model (Figure 20) is that instead of processing a single sentence, we process multiple sentences at the same time. So, instead of looking up a single word embedding, we look up multiple word embeddings (in DyNet, this is done by replacing the `lookup` function with the `lookup_batch` function, where we pass in an array of word IDs instead of a single word ID). We then run these batched word embeddings through the RNN and softmax as normal, resulting in two separate probability distributions over words in the first and second sentences. We then calculate the loss for each word (again in DyNet, replacing the `pickneglogsoftmax` function with the `pickneglogsoftmax_batch` function and pass word IDs). We then sum together the losses and use this as the loss for our entire sentence.

One sticking point, however, is that we may need to create batches with sentences of different sizes, also shown in the figure. In this case, it is common to perform **sentence padding** and **masking** to make sure that sentences of different lengths are treated properly. Padding works by simply adding the "end-of-sentence" symbol to the shorter sentences until they are of the same length as the longest sentence in the batch. Masking works by multiplying all loss functions calculated over these padded symbols by zero, ensuring that the losses for sentence end symbols don't get counted twice for the shorter sentences.

By taking these two measures, it becomes possible to process sentences of different lengths, but there is still a problem: if we perform lots of padding on sentences of vastly different

37

lengths, we'll end up wasting a lot of computation on these padded symbols. To fix this problem, it is also common to sort the sentences in the corpus by length before creating mini-batches to ensure that sentences in the same mini-batch are approximately the same size.

6.6   Further Reading

Because of the prevalence of RNNs in a number of tasks both on natural language and other data, there is significant interest in extensions to them. The following lists just a few other research topics that people are handling:

What can recurrent neural networks learn?: RNNs are surprisingly powerful tools for language, and thus many people have been interested in what exactly is going on inside them. [57] demonstrate ways to visualize the internal states of LSTM networks, and find that some nodes are in charge of keeping track of length of sentences, whether a parenthesis has been opened, and other salietn features of sentences. [65] show ways to analyze and visualize which parts of the input are contributing to particular decisions made by an RNN-based model, by back-propagating information through the network.

Other RNN architectures: There are also quite a few other recurrent network architectures. [42] perform an interesting study where they ablate various parts of the LSTM and attempt to find the best architecture for particular tasks. [123] take it a step further, explicitly training the model to find the best neural network architecture.

6.7   Exercise

In the exercise for this chapter, we will construct a recurrent neural network language model using LSTMs.

Writing the program will entail:

- Writing a function such as `lstm_step` or `gru_step` that takes the input of the previous time step and updates it according to the appropriate equations. For reference, in DyNet, the componentwise multiply and sigmoid functions are `dy.cmult` and `dy.logistic` respectively.

- Adding this function to the previous neural network language model and measuring the effect on the held-out set.

- Ideally, implement mini-batch training by using the functionality implemented in DyNet, `lookup_batch` and `pickneglogsoftmax_batch`.

Language modeling accuracy should be measured in the same way as previous exercises and compared with the previous models.

Potential improvements to the model include: Measuring the speed/stability improvements achieved by mini-batching. Comparing the differences between recurrent architectures such as RNN, GRU, or LSTM.

7   Neural Encoder-Decoder Models

From Section 3 to Section 6, we focused on the language modeling problem of calculating the probability $P(E)$ of a sequence E. In this section, we return to the statistical machine translation problem (mentioned in Section 2) of modeling the probability $P(E \mid F)$ of the output E given the input F.

7.1   Encoder-decoder Models

The first model that we will cover is called an **encoder-decoder** model [22, 36, 53, 101]. The basic idea of the model is relatively simple: we have an RNN language model, but before starting calculation of the probabilities of E, we first calculate the initial state of the language model using another RNN over the source sentence F. The name "encoder-decoder" comes from the idea that the first neural network running over F "encodes" its information as a vector of real-valued numbers (the hidden state), then the second neural network used to predict E "decodes" this information into the target sentence.
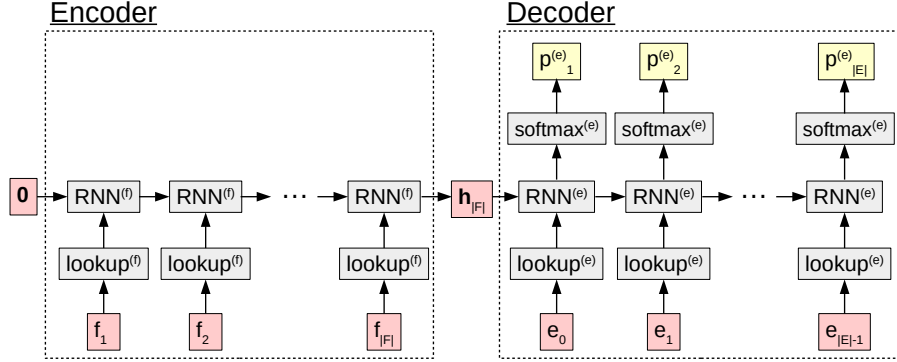

Figure 21: A computation graph of the encoder-decoder model.

If the encoder is expressed as $\text{RNN}^{(f)}(\cdot)$, the decoder is expressed as $\text{RNN}^{(e)}(\cdot)$, and we have a softmax that takes $\text{RNN}^{(e)}$'s hidden state at time step t and turns it into a probability, then our model is expressed as follows (also shown in Figure 21):

$$\boldsymbol{m}_t^{(f)} = M_{\cdot, f_t}^{(f)}$$

$$\boldsymbol{h}_t^{(f)} = \begin{cases} \text{RNN}^{(f)}(\boldsymbol{m}_t^{(f)}, \boldsymbol{h}_{t-1}^{(f)}) & t \geq 1, \\ \boldsymbol{0} & \text{otherwise.} \end{cases}$$

$$\boldsymbol{m}_t^{(e)} = M_{\cdot, e_{t-1}}^{(e)}$$

$$\boldsymbol{h}_t^{(e)} = \begin{cases} \text{RNN}^{(e)}(\boldsymbol{m}_t^{(e)}, \boldsymbol{h}_{t-1}^{(e)}) & t \geq 1, \\ \boldsymbol{h}_{|F|}^{(f)} & \text{otherwise.} \end{cases}$$

$$\boldsymbol{p}_t^{(e)} = \text{softmax}(W_{hs}\boldsymbol{h}_t^{(e)} + b_s) \tag{60}$$

In the first two lines, we look up the embedding $\boldsymbol{m}_t^{(f)}$ and calculate the encoder hidden state $\boldsymbol{h}_t^{(f)}$ for the tth word in the source sequence F. We start with an empty vector $\boldsymbol{h}_0^{(f)} = \boldsymbol{0}$, and

by $\boldsymbol{h}_{|F|}^{(f)}$, the encoder has seen all the words in the source sentence. Thus, this hidden state should theoretically be able to encode all of the information in the source sentence.

In the decoder phase, we predict the probability of word e_t at each time step. First, we similarly look up $\boldsymbol{m}_t^{(e)}$, but this time use the previous word e_{t-1}, as we must condition the probability of e_t on the previous word, not on itself. Then, we run the decoder to calculate $\boldsymbol{h}_t^{(e)}$. This is very similar to the encoder step, with the important difference that $\boldsymbol{h}_0^{(e)}$ is set to the final state of the encoder $\boldsymbol{h}_{|F|}^{(f)}$, allowing us to condition on F. Finally, we calculate the probability $\boldsymbol{p}_t^{(e)}$ by using a softmax on the hidden state $\boldsymbol{h}_t^{(e)}$.

While this model is quite simple (only 5 lines of equations), it gives us a straightforward and powerful way to model $P(E \mid F)$. In fact, [101] have shown that a model that follows this basic pattern is able to perform translation with similar accuracy to heavily engineered systems specialized to the machine translation task (although it requires a few tricks over the simple encoder-decoder that we'll discuss in later sections: beam search (Section 7.2), a different encoder (Section 7.3), and ensembling (Section 7.4)).

7.2 Generating Output

At this point, we have only mentioned how to create a probability model $P(E \mid F)$ and haven't yet covered how to actually generate translations from it, which we will now cover in the next section. In general, when we generate output we can do so according to several criteria:

Random Sampling: Randomly select an output E from the probability distribution $P(E \mid F)$. This is usually denoted $\hat{E} \sim P(E \mid F)$.

1-best Search: Find the E that maximizes $P(E \mid F)$, denoted $\hat{E} = \underset{E}{\operatorname{argmax}} P(E \mid F)$.

n-best Search: Find the n outputs with the highest probabilities according to $P(E \mid F)$.

Which of these methods we will choose will depend on our application, so we will discuss some use cases along with the algorithms themselves.

7.2.1 Random Sampling

First, **random sampling** is useful in cases where we may want to get a variety of outputs for a particular input. One example of a situation where this is useful would be in a sequence-to-sequence model for a dialog system, where we would prefer the system to not always give the same response to a particular user input to prevent monotony. Luckily, in models like the encoder-decoder above, it is simple to exactly generate samples from the distribution $P(E \mid F)$ using a method called **ancestral sampling**. Ancestral sampling works by sampling variable values one at a time, gradually conditioning on more context, so at time step t, we will sample a word from the distribution $P(e_t \mid \hat{e}_1^{t-1})$. In the encoder-decoder model, this means we simply have to calculate $\boldsymbol{p}_t$ according to the previously sampled inputs, leading to the simple generation algorithm in Algorithm 3.

One thing to note is that sometimes we also want to know the probability of the sentence that we sampled. For example, given a sentence $\hat{E}$ generated by the model, we might want to know how certain the model is in its prediction. During the sampling process, we can calculate $P(\hat{E} \mid F) = \prod_t^{|\hat{E}|} P(\hat{e}_t \mid F, \hat{E}_1^{t-1})$ incrementally by stepping along and multiplying together

40

the probabilities of each sampled word. However, as we remember from the discussion of probability vs. log probability in Section 3.3, using probabilities as-is can result in very small numbers that cause numerical precision problems on computers. Thus, when calculating the full-sentence probability it is more common to instead add together log probabilities for each word, which avoids this problem.

Algorithm 3 Generating random samples from a neural encoder-decoder

1: **procedure** SAMPLE
2:   **for** t from 1 to $|F|$ **do**
3:     Calculate $\boldsymbol{m}_t^{(f)}$ and $\boldsymbol{h}_t^{(f)}$
4:   **end for**
5:   Set $\hat{e}_0 = $ "$\langle \mathrm{s} \rangle$" and $t \leftarrow 0$
6:   **while** $\hat{e}_t \neq$ "$\langle /\mathrm{s} \rangle$" **do**
7:     $t \leftarrow t + 1$
8:     Calculate $\boldsymbol{m}_t^{(e)}$, $\boldsymbol{h}_t^{(e)}$, and $\boldsymbol{p}_t^{(e)}$ from $\hat{e}_{t-1}$
9:     Sample $\hat{e}_t$ according to $\boldsymbol{p}_t^{(e)}$
10:   **end while**
11: **end procedure**

7.2.2 Greedy 1-best Search

Next, let's consider the problem of generating a 1-best result. This variety of generation is useful in machine translation, and most other applications where we simply want to output the translation that the model thought was best. The simplest way of doing so is **greedy search**, in which we simply calculate $\boldsymbol{p}_t$ at every time step, select the word that gives us the highest probability, and use it as the next word in our sequence. In other words, this algorithm is exactly the same as Algorithm 3, with the exception that on Line 9, instead of sampling $\hat{e}_t$ randomly according to $\boldsymbol{p}_t^{(e)}$, we instead choose the max: $\hat{e}_t = \operatorname*{argmax}_i p_{t,i}^{(e)}$.

Interestingly, while ancestral sampling exactly samples outputs from the distribution according to $P(E \mid F)$, greedy search is not guaranteed to find the translation with the highest probability. An example of a case in which this is true can be found in the graph in Figure 22, which is an example of search graph with a vocabulary of $\{\mathrm{a}, \mathrm{b}, \langle /\mathrm{s} \rangle\}$.[25] As an exercise, I encourage readers to find the true 1-best (or n-best) sentence according to the probability $P(E \mid F)$ and the probability of the sentence found according to greedy search and confirm that these are different.

7.2.3 Beam Search

One way to solve this problem is through the use of **beam search**. Beam search is similar to greedy search, but instead of considering only the one best hypothesis, we consider b best hypotheses at each time step, where b is the "width" of the beam. An example of beam search where $b = 2$ is shown in Figure 23 (note that we are using log probabilities here because they

---

[25]In reality, we will never have a probability of exactly $P(e_t = \langle /\mathrm{s} \rangle \mid F, e_1^{t-1}) = 1.0$, but for illustrative purposes, we show this here.
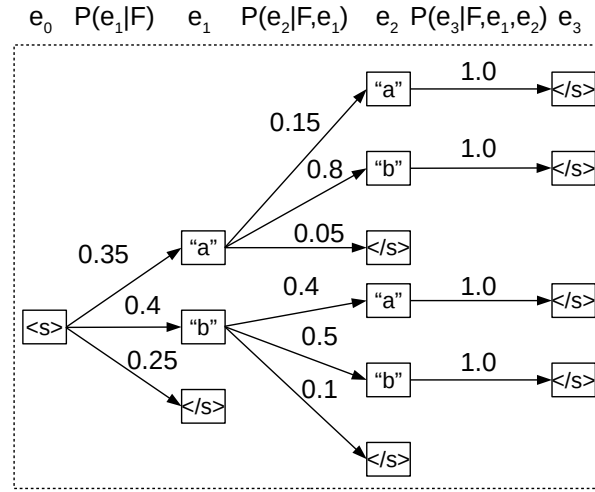
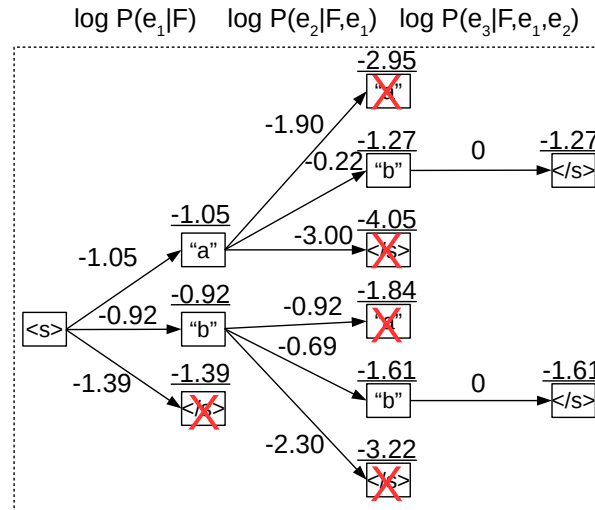Figure 22: A search graph where greedy search fails.

Figure 23: An example of beam search with $b = 2$. Numbers next to arrows are log probabilities for a single word $\log P(e_t \mid F, e_1^{t-1})$, while numbers above nodes are log probabilities for the entire hypothesis up until this point.

are more conducive to comparing hypotheses over the entire sentence, as mentioned before). In the first time step, we expand hypotheses for e_1 corresponding to all of the three words in the vocabulary, then keep the top two ("b" and "a") and delete the remaining one ("$\langle /s \rangle$"). In the second time step, we expand hypotheses for e_2 corresponding to the continuation of the first hypotheses for all words in the vocabulary, temporarily creating $b * |V|$ active hypotheses. These active hypotheses are also pruned down to the b active hypotheses ("a b" and "b b"). This process of calculating scores for $b * |V|$ continuations of active hypotheses, then pruning back down to the top b, is continued until the end of the sentence.

One thing to be careful about when generating sentences using models, such as neural machine translation, where $P(E \mid F) = \prod_t^{|E|} P(e_t \mid F, e_1^{t-1})$ is that they tend to prefer shorter sentences. This is because every time we add another word, we multiply in another probability, reducing the probability of the whole sentence. As we increase the beam size, the search algorithm gets better at finding these short sentences, and as a result, beam search with a larger beam size often has a significant **length bias** towards these shorter sentences.

There have been several attempts to fix this length bias problem. For example, it is possible to put a prior probability on the length of the sentence given the length of the previous sentence $P(|E| \mid |F|)$, and multiply this with the standard sentence probability $P(E \mid F)$ at decoding time [34]:

$$\hat{E} = \underset{E}{\operatorname{argmax}} \ \log P(|E| \mid |F|) + \log P(E \mid F). \tag{61}$$

This prior probability can be estimated from data, and [34] simply estimate this using a multinomial distribution learned on the training data:

$$P(|E| \mid |F|) = \frac{c(|E|, |F|)}{c(|F|)}. \tag{62}$$

A more heuristic but still widely used approach normalizes the log probability by the length of the target sentence, effectively searching for the sentence that has the highest average log probability per word [21]:

$$\hat{E} = \underset{E}{\operatorname{argmax}} \ \log P(E \mid F)/|E|. \tag{63}$$

7.3   Other Ways of Encoding Sequences

In Section 7.1, we described a model that works by encoding sequences linearly, one word at a time from left to right. However, this may not be the most natural or effective way to turn the sentence F into a vector $\boldsymbol{h}$. In this section, we'll discuss a number of different ways to perform encoding that have been reported to be effective in the literature.

7.3.1   Reverse and Bidirectional Encoders

First, [101] have proposed a **reverse encoder**. In this method, we simply run a standard linear encoder over F, but instead of doing so from left to right, we do so from right to left.

$$\overleftarrow{\boldsymbol{h}}_t^{(f)} = \begin{cases} \overleftarrow{\mathrm{RNN}}^{(f)}(\boldsymbol{m}_t^{(f)}, \overleftarrow{\boldsymbol{h}}_{t+1}^{(f)}) & t \leq |F|, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{64}$$

(a) Dependency Distances in Forward Encoder

$|F|$ $|F|$ $|F|$

```
                              e₁   e₂        e|E|
  0 → enc → enc → ···· → enc → dec → dec → ···· → enc
      ↑     ↑           ↑     ↑     ↑           ↑
      f₁    f₂          f|F|  e₀    e₁          e|E|-1
```

(b) Dependency Distances in Reverse Encoder

$|F|+|E|-1$ 3   1

```
                              e₁   e₂        e|E|
  0 → enc → enc → ···· → enc → dec → dec → ···· → enc
      ↑     ↑           ↑     ↑     ↑           ↑
      f|F|  f|F|-1      f₁    e₀    e₁          e|E|-1
```
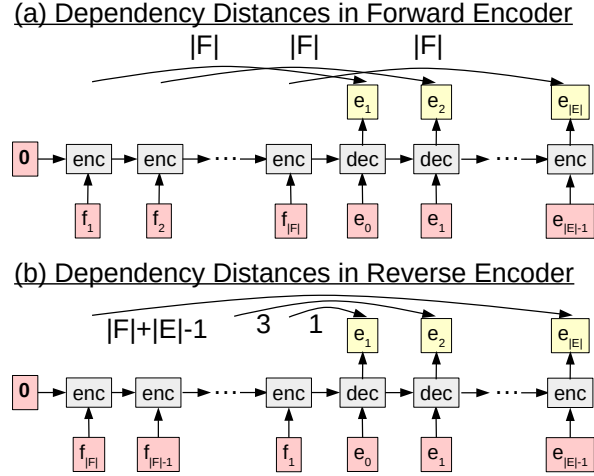
Figure 24: The distances between words with the same index in the forward and reverse decoders.

The motivation behind this method is that for pairs of languages with similar ordering (such as English-French, which the authors experimented on), the words at the beginning of F will generally correspond to words at the beginning of E. Assuming the extreme case that words with identical indices correspond to each-other (e.g. f_1 corresponds to e_1, f_2 to e_2, etc.), the distance between corresponding words in the linear encoding and decoding will be $|F|$, as shown in Figure 24(a). Remembering the vanishing gradient problem from Section 6.3, this means that the RNN has to propagate the information across $|F|$ time steps before making a prediction, a difficult feat. At the beginning of training, even RNN variants such as LSTMs have trouble, as they have to essentially "guess" what part of the information encoded in their hidden state is being used without any prior bias.

Reversing the encoder helps solve this problem by reducing the length of dependencies for a subset of the words in the sentence, specifically the ones at the beginning of the sentences. As shown in Figure 24(b), the length of the dependency for f_1 and e_1 is 1, and subsequent pairs of f_t and e_t have a distance of $2t-1$. During learning, the model can "latch on" to these short-distance dependencies and use them as a way to bootstrap the model training, after which it becomes possible to gradually learn the longer dependencies for the words at the end of the sentence. In [101], this proved critical to learn effective models in the encoder-decoder framework.

However, this approach of reversing the encoder relies on the strong assumption that the order of words in the input and output sequences are very similar, or at least that the words at the beginning of sentences are the same. This is true for languages like English and French, which share the same "subject-verb-object (SVO)" word ordering, but may not be true for more typologically distinct languages. One type of encoder that is slightly more robust to these differences is the **bi-directional encoder** [4]. In this method, we use two different
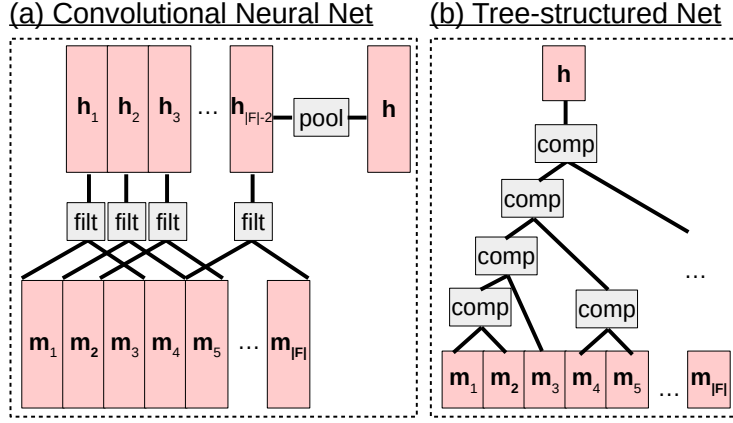
44

Figure 25: Examples of convolutional and tree-structured networks.

encoders: one traveling forward and one traveling backward over the input sentence

$$
\overrightarrow{\boldsymbol{h}}_t^{(f)} = \begin{cases} \overrightarrow{\mathrm{RNN}}^{(f)}(\boldsymbol{m}_t^{(f)}, \overrightarrow{\boldsymbol{h}}_{t+-}^{(f)}) & t \geq 1, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{65}
$$

$$
\overleftarrow{\boldsymbol{h}}_t^{(f)} = \begin{cases} \overleftarrow{\mathrm{RNN}}^{(f)}(\boldsymbol{m}_t^{(f)}, \overleftarrow{\boldsymbol{h}}_{t+1}^{(f)}) & t \leq |F|, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{66}
$$

which are then combined into the initial vector $\boldsymbol{h}_0^{(e)}$ for the decoder RNN. This combination can be done by simply concatenating the two final vectors $\overrightarrow{\boldsymbol{h}}_{|F|}$ and $\overleftarrow{\boldsymbol{h}}_1$. However, this also requires that the size of the vectors for the decoder RNN be exactly equal to the combined size of the two encoder RNNs. As a more flexible alternative, we can add an additional parameterized hidden layer between the encoder and decoder states, which allows us to convert the bidirectional encoder states into an appropriately-sized state for the decoder:

$$
\boldsymbol{h}_0^{(e)} = \tanh(W_{\overrightarrow{f}e}\overrightarrow{\boldsymbol{h}}_{|F|} + W_{\overleftarrow{f}e}\overleftarrow{\boldsymbol{h}}_1 + \boldsymbol{b}_e). \tag{67}
$$

7.3.2 Convolutional Neural Networks

In addition, there are also methods for decoding that move beyond a simple linear view of the input sentence. For example, **convolutional neural networks** (CNNs; [37, 114, 62]), Figure 25(a)) are a variety of neural net that combines together information from spatially or temporally local segments. They are most widely applied to image processing but have also been used for speech processing, as well as the processing of textual sequences. While there are many varieties of CNN-based models of text (e.g. [55, 63, 54]), here we will show an example from [59]. This model has n **filters** with a width w that are passed incrementally over w-word segments of the input. Specifically, given an embedding matrix M of width $|F|$, we generate a hidden layer matrix H of width $|F| - w + 1$, where each column of the matrix is equal to

$$
\boldsymbol{h}_t = W\mathrm{concat}(\boldsymbol{m}_t, \boldsymbol{m}_{t+1}, \ldots, \boldsymbol{m}_{t+w-1}) \tag{68}
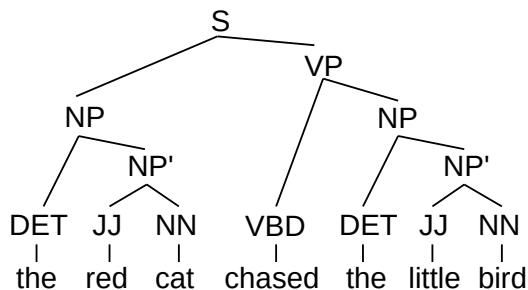$$

45

Figure 26: An example of a syntax tree for a sentence showing the sentence structure and phrase types (DET="determiner", JJ="adjective", NN="noun", VBD="past tense verb", NP="noun phrase", NP'="part of a noun phrase", VP="verb phrase", S="sentence").

where $W \in \mathbb{R}^{n \times w|\boldsymbol{m}|}$ is a matrix where the ith row represents the parameters of filter i that will be multiplied by the embeddings of w consecutive words. If $w = 3$, we can interpret this as $\boldsymbol{h}_1$ extracting a vector of features for f_1^3, $\boldsymbol{h}_2$ as extracting a vector of features for f_2^4, etc. until the end of the sentence.

Finally, we perform a **pooling** operation that converts this matrix H (which varies in width according to the sentence length) into a single vector $\boldsymbol{h}$ (which is fixed-size and can thus be used in down-stream processing). Examples of pooling operations include average, max, and k-max [55].

Compared to RNNs and their variants, CNNs have several advantages and disadvantages:

- On the positive side, CNNs provide a relatively simple way to detect features of short word sequences in sentence text and accumulate them across the entire sentence.

- Also on the positive side, CNNs do not suffer as heavily from the vanishing gradient problem, as they do not need to propagate gradients across multiple time steps.

- On the negative side, CNNs are not quite as expressive and are a less natural way of expressing complicated patterns that move beyond their filter width.

In general, CNNs have been found to be quite effective for text classification, where it is more important to pick out the most indicative features of the text and there is less of an emphasis on getting an overall view of the content [59]. There have also been some positive results reported using specific varieties of CNNs for sequence-to-sequence modeling [54].

7.3.3 Tree-structured Networks

Finally, one other popular form of encoder that is widely used in a number of tasks are **tree-structured networks** ([83, 100], Figure 25(b)). The basic idea behind these networks is that the way to combine the information from each particular word is guided by some sort of structure, usually the syntactic structure of the sentence, an example of which is shown in Figure 26. The reason why this is intuitively useful is because each syntactic phrase usually also corresponds to a coherent semantic unit. Thus, performing the calculation and manipulation of vectors over these coherent units will be more appropriate compared to using random substrings of words, like those used by CNNs.

For example, let's say we have the phrase "the red cat chased the little bird" as shown in the figure. In this case, following a syntactic tree would ensure that we calculate vectors for coherent units that correspond to a grammatical phrase such as "chased" and "the little bird", and combine these phrases together one by one to obtain the meaning of larger coherent phrase such as "chased the little bird". By doing so, we can take advantage of the fact that language is **compositional**, with the meaning of a more complex phrase resulting from regular combinations and transformation of smaller constituent phrases [102]. By taking this linguistically motivated and intuitive view of the sentence, we hope will help the neural networks learn more generalizable functions from limited training data.

Perhaps the most simple type of tree-structured network is the **recursive neural network** proposed by [100]. This network has very strong parallels to standard RNNs, but instead of calculating the hidden state $\boldsymbol{h}_t$ at time t from the previous hidden state $\boldsymbol{h}_{t-1}$ as follows:

$$\boldsymbol{h}_t = \tanh(W_{xh}\boldsymbol{x}_t + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h), \tag{69}$$

we instead calculate the hidden state of the parent node $\boldsymbol{h}_p$ from the hidden states of the left and right children, $\boldsymbol{h}_l$ and $\boldsymbol{h}_r$ respectively:

$$\boldsymbol{h}_p = \tanh(W_{xp}\boldsymbol{x}_t + W_{lp}\boldsymbol{h}_l + W_{rp}\boldsymbol{h}_r + \boldsymbol{b}_p). \tag{70}$$

Thus, the representation for each node in the tree can be calculated in a bottom-up fashion.

Like standard RNNs, these recursive networks suffer from the vanishing gradient problem. To fix this problem there is an adaptation of LSTMs to tree-structured networks, fittingly called **tree LSTMs** [103], which fixes this vanishing gradient problem. There are also a wide variety of other kinds of tree-structured composition functions that interested readers can explore [99, 31, 32]. Also of interest is the study by [66], which examines the various tasks in which tree structures are necessary or unnecessary for NLP.

7.4 Ensembling Multiple Models

One other method that is widely used in encoder-decoders, or other models of translation is **ensembling**: the combination of the prediction of multiple independently trained models to improve the overall prediction results. The intuition behind ensembling is that different models will make different mistakes, and that on average it is more common for models to agree when the answer is correct than when it is mistaken. Thus, if we combine multiple models together, it becomes possible to smooth over these mistakes, finding the correct answer more often.

The first step in ensembling encoder-decoder models is to independently train N different models $P_1(\cdot), P_2(\cdot), \ldots, P_N(\cdot)$, for example, by randomly initializing the weights of the neural network differently before training. Next, during search, at each time step we calculate the probability of the next word as the average probability of the N models:

$$P(e_t \mid F, e_1^{t-1}) = \frac{1}{N} \sum_{i=1}^{N} P_i(e_t \mid F, e_1^{t-1}). \tag{71}$$

This probability is used in searching for our hypotheses.

7.5 Exercise

In the exercise for this chapter, we will create an encoder-decoder translation model and make it possible to generate translations.

Writing the program will entail:

- Extend your RNN language model code to first read in a source sentence to calculate the initial hidden state.

- On the training set, write code to calculate the loss function and perform training.

- On the development set, generate translations using greedy search.

- Evaluate your generated translations by comparing them to the reference translations to see if they look good or not. Translations can also be evaluated by automatic means, such as BLEU score [81]. A reference implementation of a BLEU evaluation script can be found here: `https://github.com/moses-smt/mosesdecoder/blob/master/scripts/generic/multi-bleu.perl`.

Potential improvements to the model include: Implementing beam search and comparing the results with greedy search. Implementing an alternative encoder. Implementing ensembling.

8 Attentional Neural MT

In the past chapter, we described a simple model for neural machine translation, which uses an encoder to encode sentences as a fixed-length vector. However, in some ways, this view is overly simplified, and by the introduction of a powerful mechanism called **attention**, we can overcome these difficulties. This section describes the problems with the encoder-decoder architecture and what attention does to fix these problems.

8.1 Problems of Representation in Encoder-Decoders

Theoretically, a sufficiently large and well-trained encoder-decoder model should be able to perform machine translation perfectly. As mentioned in Section 5.2, neural networks are universal function approximators, meaning that they can express any function that we wish to model, including a function that accurately predicts our predictive probability for the next word $P(e_t \mid F, e_1^{t-1})$. However, in practice, it is necessary to learn these functions from limited data, and when we do so, it is important to have a proper **inductive bias** – an appropriate model structure that allows the network to learn to model accurately with a reasonable amount of data.

There are two things that are worrying about the standard encoder-decoder architecture. The first was described in the previous section: there are long-distance dependencies between words that need to be translated into each other. In the previous section, this was alleviated to some extent by reversing the direction of the encoder to bootstrap training, but still, a large number of long-distance dependencies remain, and it is hard to guarantee that we will learn to handle these properly.

The second, and perhaps more, worrying aspect of the encoder-decoder is that it attempts to store information sentences of any arbitrary length in a hidden vector of fixed size. In other words, even if our machine translation system is expected to translate sentences of lengths from 1 word to 100 words, it will still use the same intermediate representation to store all of the information about the input sentence. If our network is too small, it will not be able to encode all of the information in the longer sentences that we will be expected to translate. On the other hand, even if we make the network large enough to handle the largest sentences in our inputs, when processing shorter sentences, this may be overkill, using needlessly large amounts of memory and computation time. In addition, because these networks will have large numbers of parameters, it will be more difficult to learn them in the face of limited data without encountering problems such as overfitting.

The remainder of this section discusses a more natural way to solve the translation problem with neural networks: attention.

8.2   Attention

The basic idea of attention is that instead of attempting to learn a single vector representation for each sentence, we instead keep around vectors for every word in the input sentence, and reference these vectors at each decoding step. Because the number of vectors available to reference is equivalent to the number of words in the input sentence, long sentences will have many vectors and short sentences will have few vectors. As a result, we can express input sentences in a much more efficient way, avoiding the problems of inefficient representations for encoder-decoders mentioned in the previous section.

First we create a set of vectors that we will be using as this variably-lengthed representation. To do so, we calculate a vector for every word in the source sentence by running an RNN in both directions:

$$\overrightarrow{\boldsymbol{h}}_j^{(f)} = \text{RNN}(\text{embed}(f_j), \overrightarrow{\boldsymbol{h}}_{j-1}^{(f)})$$
$$\overleftarrow{\boldsymbol{h}}_j^{(f)} = \text{RNN}(\text{embed}(f_j), \overleftarrow{\boldsymbol{h}}_{j+1}^{(f)}).$$

Then we concatenate the two vectors $\overrightarrow{\boldsymbol{h}}_j^{(f)}$ and $\overleftarrow{\boldsymbol{h}}_j^{(f)}$ into a bidirectional representation $\boldsymbol{h}_j^{(f)}$

$$\boldsymbol{h}_j^{(f)} = [\overleftarrow{\boldsymbol{h}}_j^{(f)}; \overrightarrow{\boldsymbol{h}}_j^{(f)}].$$

We can further concatenate these vectors into a matrix:

$$H^{(f)} = \text{concat\_col}(\boldsymbol{h}_1^{(f)}, \ldots, \boldsymbol{h}_{|F|}^{(f)}).$$

This will give us a matrix where every column corresponds to one word in the input sentence.

However, we are now faced with a difficulty. We have a matrix $H^{(f)}$ with a variable number of columns depending on the length of the source sentence, but would like to use this to compute, for example, the probabilities over the output vocabulary, which we only know how to do (directly) for the case where we have a vector of input. The key insight of attention is that we calculate a vector $\boldsymbol{\alpha}_t$ that can be used to combine together the columns of H into a vector $\boldsymbol{c}_t$

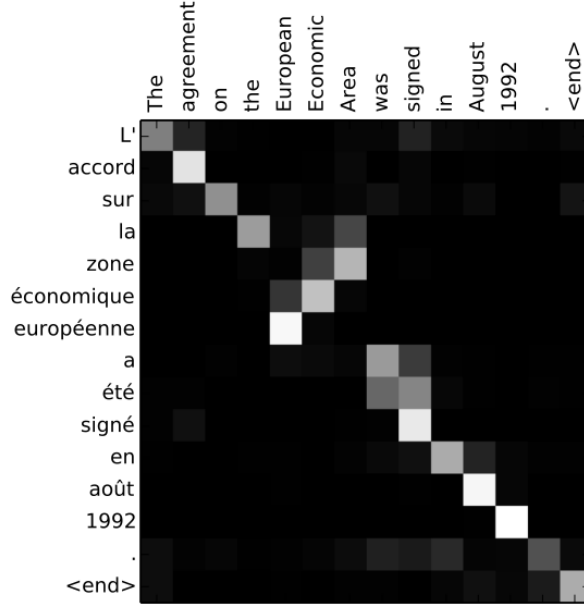$$\boldsymbol{c}_t = H^{(f)} \boldsymbol{\alpha}_t. \tag{72}$$

Figure 27: An example of attention from [4]. English is the source, French is the target, and a higher attention weight when generating a particular target word is indicated by a lighter color in the matrix.

$\boldsymbol{\alpha}_t$ is called the **attention vector**, and is generally assumed to have elements that are between zero and one and add to one.

The basic idea behind the attention vector is that it is telling us how much we are "focusing" on a particular source word at a particular time step. The larger the value in $\boldsymbol{\alpha}_t$, the more impact a word will have when predicting the next word in the output sentence. An example of how this attention plays out in an actual translation example is shown in Figure 27, and as we can see the values in the alignment vectors generally align with our intuition.

8.3 Calculating Attention Scores

The next question then becomes, from where do we get this $\boldsymbol{\alpha}_t$? The answer to this lies in the *decoder* RNN, which we use to track our state while we are generating output. As before, the decoder's hidden state $\boldsymbol{h}_t^{(e)}$ is a fixed-length continuous vector representing the previous target words e_1^{t-1}, initialized as $\boldsymbol{h}_0^{(e)} = \boldsymbol{h}_{|F|+1}^{(f)}$. This is used to calculate a context vector $\boldsymbol{c}_t$ that is used to summarize the source attentional context used in choosing target word e_t, and initialized as $\boldsymbol{c}_0 = \boldsymbol{0}$.

First, we update the hidden state to $\boldsymbol{h}_t^{(e)}$ based on the word representation and context vectors from the previous target time step

$$\boldsymbol{h}_t^{(e)} = \text{enc}([\text{embed}(e_{t-1}); \boldsymbol{c}_{t-1}], \boldsymbol{h}_{t-1}^{(e)}). \tag{73}$$

Based on this $\boldsymbol{h}_t^{(e)}$, we calculate an **attention score** $\boldsymbol{a}_t$, with each element equal to

$$a_{t,j} = \text{attn\_score}(\boldsymbol{h}_j^{(f)}, \boldsymbol{h}_t^{(e)}). \tag{74}$$

50
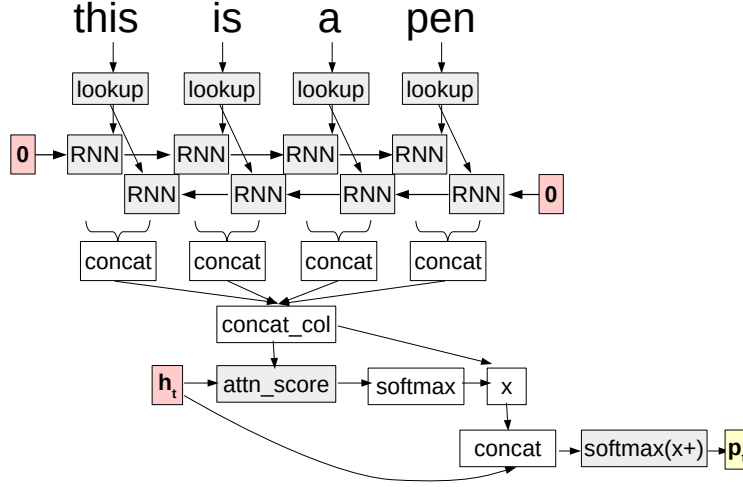
Figure 28: A computation graph for attention.

attn_score($\cdot$) can be an arbitrary function that takes two vectors as input and outputs a score about how much we should focus on this particular input word encoding $\boldsymbol{h}_j^{(f)}$ at the time step $\boldsymbol{h}_t^{(e)}$. We describe some examples at a later point in Section 8.4.

We then normalize this into the actual attention vector itself by taking a softmax over the scores:

$$\boldsymbol{\alpha}_t = \mathrm{softmax}(\boldsymbol{a}_t). \tag{75}$$

This attention vector is then used to weight the encoded representation $H^{(f)}$ to create a context vector $\boldsymbol{c}_t$ for the current time step, as mentioned in Equation 72.

We now have a context vector $\boldsymbol{c}_t$ and hidden state $\boldsymbol{h}_t^{(e)}$ for time step t, which we can pass on down to downstream tasks. For example, we can concatenate both of these together when calculating the softmax distribution over the next words:

$$\boldsymbol{p}_t^{(e)} = \mathrm{softmax}(W_{hs}[\boldsymbol{h}_t^{(e)}; \boldsymbol{c}_t] + b_s). \tag{76}$$

It is worth noting that this means that the encoding of each source word $\boldsymbol{h}_j^{(f)}$ is considered much more directly in the calculation of output probabilities. In contrast to the encoder-decoder, where the encoder-decoder will only be able to access information about the first encoded word in the source by passing it over $|F|$ time steps, here the source encoding is accessed (in a weighted manner) through the context vector Equation 72.

This whole, rather involved, process is shown in Figure 28.

8.4 Ways of Calculating Attention Scores

As mentioned in Equation 74, the final missing piece to the puzzle is how to calculate the attention score $a_{t,j}$.

[68] test three different attention functions, all of which have their own merits:

Dot product: This is the simplest of the functions, as it simply calculates the similarity between $\boldsymbol{h}_t^{(e)}$ and $\boldsymbol{h}_j^{(f)}$ as measured by the dot product:

$$\text{attn\_score}(\boldsymbol{h}_j^{(f)}, \boldsymbol{h}_t^{(e)}) := \boldsymbol{h}_j^{(f)\intercal}\boldsymbol{h}_t^{(e)}. \tag{77}$$

This model has the advantage that it adds no additional parameters to the model. However, it also has the intuitive disadvantage that it forces the input and output encodings to be in the same space (because similar $\boldsymbol{h}_t^{(e)}$ and $\boldsymbol{h}_j^{(f)}$ must be close in space in order for their dot product to be high). It should also be noted that the dot product can be calculated efficiently for every word in the source sentence by instead defining the attention score over the concatenated matrix $H^{(f)}$ as follows:

$$\text{attn\_score}(H^{(f)}, \boldsymbol{h}_t^{(e)}) := H_j^{(f)\intercal}\boldsymbol{h}_t^{(e)}. \tag{78}$$

Combining the many attention operations into one can be useful for efficient impementation, especially on GPUs. The following attention functions can also be calculated like this similarly.[26]

Bilinear functions: One slight modification to the dot product that is more expressive is the **bilinear function**. This function helps relax the restriction that the source and target embeddings must be in the same space by performing a linear transform parameterized by W_a before taking the dot product:

$$\text{attn\_score}(\boldsymbol{h}_j^{(f)}, \boldsymbol{h}_t^{(e)}) := \boldsymbol{h}_j^{(f)\intercal}W_a\boldsymbol{h}_t^{(e)}. \tag{79}$$

This has the advantage that if W_a is not a square matrix, it is possible for the two vectors to be of different sizes, so it is possible for the encoder and decoder to have different dimensions. However, it does introduce quite a few parameters to the model ($|\boldsymbol{h}^{(f)}| \times |\boldsymbol{h}^{(e)}|$), which may be difficult to train properly.

Multi-layer perceptrons: Finally, it is also possible to calculate the attention score using a multi-layer perceptron, which was the method employed by [4] in their original implementation of attention:

$$\text{attn\_score}(\boldsymbol{h}_t^{(e)}, \boldsymbol{h}_j^{(f)}) := \boldsymbol{w}_{a2}^{\intercal}\tanh(W_{a1}[\boldsymbol{h}_t^{(e)}; \boldsymbol{h}_j^{(f)}]), \tag{80}$$

where W_{a1} and $\boldsymbol{w}_{a2}$ are the weight matrix and vector of the first and second layers of the MLP respectively. This is more flexible than the dot product method, usually has fewer parameters than the bilinear method, and generally provides good results.

In addition to these methods above, which are essentially the defacto-standard, there are a few more sophisticated methods for calculating attention as well. For example, it is possible to use recurrent neural networks [120], tree-structured networks based on document structure [121], convolutional neural networks [2], or structured models [58] to calculate attention.

---

[26] *Question: What do the equations look like for the combined versions of the following functions?*

8.5 Copying and Unknown Word Replacement

One pleasant side-effect of attention is that it not only increases translation accuracy, but also makes it easier to tell which words are translated into which words in the output. One obvious consequence of this is that we can draw intuitive graphs such as the one shown in Figure 27, which aid error analysis.

Another advantage is that it also becomes possible to handle unknown words in a more elegant way, performing **unknown word replacement** [67]. The idea of this method is simple, every time our decoder chooses the unknown word token ⟨unk⟩ in the output, we look up the source word with the highest attention weight at this time step, and output that word instead of the unknown token ⟨unk⟩. If we do so, at least the user can see which words have been left untranslated, which is better than seeing them disappear altogether or be replaced by a placeholder.

It is also common to use alignment models such as those described in [16] to obtain a translation dictionary, then use this to aid unknown word replacement even further. Specifically, instead of copying the word as-is into the output, if the chosen source word is f, we output the word with the highest translation probability $P_{\text{dict}}(e \mid f)$. This allows words that are included in the dictionary to be mapped into their most-frequent counterpart in the target language.

8.6 Intuitive Priors on Attention

Because of the importance of attention in modern NMT systems, there have also been a number of proposals to improve accuracy of estimating the attention itself through the introduction of intuitively motivated prior probabilities. [25] propose several methods to incorporate biases into the training of the model to ensure that the attention weights match our belief of what alignments between languages look like.

These take several forms, and are heavily inspired by the alignment models used in more traditional SMT systems such as those proposed by [16]. These models can be briefly summarized as:

Position Bias: If two languages have similar word order, then it is more likely that alignments should fall along the diagonal. This is demonstrated strongly in Figure 27. It is possible to encourage this behavior by adding a prior probability over attention that makes it easier for things near the diagonal to be aligned.

Markov Condition: In most languages, we can assume that most of the time if two words in the target are contiguous, the aligned words in the source will also be contiguous. For example, in Figure 27, this is true for all contiguous pairs of English words except "the, European" and "Area, was". To take advantage of this property, it is possible to impose a prior that discourages large jumps and encourages local steps in attention. A model that is similar in motivation, but different in implementation, is the **local attention** model [68], which selects which part of the source sentence to focus on using the neural network itself.

Fertility: We can assume that some words will be translated into a certain number words in the other langauge. For example, the English word "cats" will be translated into two words "les chats" in French. Priors on fertility takes advantage of this fact by giving

the model a penalty when particular words are not attended too much, or attended to too much. In fact one of the major problems with poorly trained neural MT systems is that they repeat the same word over and over, or drop words, a violation of this fertility constraint. Because of this, several other methods have been proposed to incorporate coverage in the model itself [106, 69], or as a constraint during the decoding process [119].

Bilingual Symmetry: Finally, we expect that words that are aligned when performing translation from F to E should also be aligned when performing translation from E to F. This can be enforced by training two models in parallel, and enforcing constraints that the alignment matrices look similar in both directions.

[25] experiment extensively with these approaches, and find that the bilingual symmetry constraint is particularly effective among the various methods.

8.7   Further Reading

This section outlines some further directions for reading more about improvements to attention:

Hard Attention: As shown in Equation 75, standard attention uses a soft combination of various contents. There are also methods for hard attention that make a hard binary decision about whether to focus on a particular context, with motivations ranging from learning explainable models [64], to processing text incrementally [122, 45].

Supervised Training of Attention: In addition, sometimes we have hand-annotated data showing us true alignments for a particular language pair. It is possible to train attentional models using this data by defining a loss function that penalizes the model when it does not predict these alignments correctly [70].

Other Ways of Memorizing Input: Finally, there are other ways of accessing relevant information other than attention. [115] propose a method using **memory networks**, which have a separate set of memory that can be written to or read from as the processing continues.

8.8   Exercise

In the exercise for this chapter, we will create code to train and generate translations with an attentional neural MT model.

Writing the program will entail extending your encoder-decoder code to add attention. You can then generate translations and compare them to others.

- Extend your encoder-decoder code to add attention.

- On the training set, write code to calculate the loss function and perform training.

- On the development set, generate translations using greedy search.

- Evaluate these translations, either manually or automatically.

It is also highly recommended, but not necessary, that you attempt to implement unknown word replacement.

Potential improvements to the model include implementing any of the improvements to attention mentioned in Section 8.6 or Section 8.7.

9 Conclusion

This tutorial has covered the basics of neural machine translation and sequence-to-sequence models. It gradually stepped through models of increasing sophistication, starting with n-gram language models, and culminating in attention, which now represents the state-of-the-art in many sequence-to-sequence modeling tasks.

It should be noted that this is a very active reserach field, and there are a number of advanced research topics that are beyond the scope of this tutorial, but may be of interest to readers who have mastered the basics and would like to learn more.

Handling large vocabularies: One difficulty of neural MT models is that they perform badly when using large vocabularies; it is hard to learn how to properly translate rare words with limited data, and computation becomes a burden. One method to handle this is to break words into smaller units such as characters [23] or subwords [94]. It is also possible to incorporate translation dictionaries with broad coverage to handle low-frequency phenomena [3].

Optimizing translation performance: While the models presented in this tutorial are trained to maximize the likelihood of the target sentence given the source $P(E \mid F)$, in reality what we actually care about is the accuracy of the generated sentences. There have been a number of works proposed to resolve this disconnect by directly considering the accuracy of the generated results when training our models. These include methods that sample translation results from the current model and move towards parameters that result in good translations [84, 97], methods that optimize parameters towards partially mistaken hypotheses to try to improve robustness to mistakes in generation [8, 79], or methods that try to prevent mistakes that may occur during the search process [117].

Multi-lingual learning: Up until now we assumed that we were training a model between two languages F and E. However, in reality there are many languages in the world, and some work has shown that we can benefit by using data from all these languages to learn models together [35, 52, 46]. It is also possible to perform transfer across languages, training a model first on one language pair, then fine-tuning it to others [124].

Other applications: Similar sequence-to-sequence models have been used for a wide variety of tasks, from dialog systems [112, 95] to text summarization [92], speech recognition [17], speech synthesis [109], image captioning [56, 113], image generation [43], and more.

This is just a small sampling of topics from this exciting and rapidly expanding field, and hopefully this tutorial gave readers the tools to strike out on their own and apply these models to their applications of interest.

Acknowledgements

References

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. *arXiv preprint arXiv:1602.03001*, 2016.

[3] Philip Arthur, Graham Neubig, and Satoshi Nakamura. Incorporating discrete translation lexicons into neural machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[5] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 238–247, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[6] Jerome R Bellegarda. Statistical language model adaptation: review and perspectives. *Speech communication*, 42(1):93–108, 2004.

[7] Claus Bendtsen and Ole Stauning. Fadbad, a flexible c++ package for automatic differentiation. *Department of Mathematical Modelling, Technical University of Denmark*, 1996.

[8] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1171–1179, 2015.

[9] Yoshua Bengio, Holger Schwenk, Jean-Sébastien Senécal, Fréderic Morin, and Jean-Luc Gauvain. Neural probabilistic language models. In *Innovations in Machine Learning*, volume 194, pages 137–186. 2006.

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

[11] Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22, 1996.

[12] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[13] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3, 2003.

[14] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, 2007.

[15] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, 1992.

[16] 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*, 19:263–312, 1993.

[17] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 4960–4964. IEEE, 2016.

[18] Stanley Chen. Shrinking exponential language models. In *Proceedings of the Human Language Technologies: The 2009 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 468–476, 2009.

[19] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 310–318, 1996.

[20] Stanley F. Chen and Roni Rosenfeld. A survey of smoothing techniques for me models. *Speech and Audio Processing, IEEE Transactions on*, 8(1):37–50, Jan 2000.

[21] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of the Workshop on Syntax and Structure in Statistical Translation*, pages 103–111, 2014.

[22] Lonnie Chrisman. Learning recursive distributed representations for holistic computation. *Connection Science*, 3(4):345–366, 1991.

[23] Junyoung Chung, Kyunghyun Cho, and Yoshua Bengio. A character-level decoder without explicit segmentation for neural machine translation. In *Proceedings of the*

54th Annual Meeting of the Association for Computational Linguistics (ACL), pages 1693–1703, 2016.

[24] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[25] Trevor Cohn, Cong Duy Vu Hoang, Ekaterina Vymolova, Kaisheng Yao, Chris Dyer, and Gholamreza Haffari. Incorporating structural alignment biases into an attentional neural translation model. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 876–885, 2016.

[26] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[27] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.

[28] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[29] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3079–3087, 2015.

[30] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[31] Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 334–343, 2015.

[32] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 199–209, 2016.

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

[34] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. Tree-to-sequence attentional neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 823–833, 2016.

[35] Orhan Firat, Kyunghyun Cho, and Yoshua Bengio. Multi-way, multilingual neural machine translation with a shared attention mechanism. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 866–875, 2016.

[36] Mikel L Forcada and Ramón P Ñeco. Recursive hetero-associative memories for translation. In *International Work-Conference on Artificial Neural Networks*, pages 453–462. Springer, 1997.

[37] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

[38] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[39] Yoav Goldberg. A primer on neural network models for natural language processing. *arXiv preprint arXiv:1510.00726*, 2015.

[40] Joshua Goodman. Classes for fast maximum entropy training. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 561–564. IEEE, 2001.

[41] Joshua T. Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403–434, 2001.

[42] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015.

[43] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[44] Andreas Griewank. Automatic differentiation of algorithms: theory, implementation, and application. In *proceedings of the first SIAM Workshop on Automatic Differentiation*, 1991.

[45] Jiatao Gu, Graham Neubig, Kyunghyun Cho, and Victor OK Li. Learning to translate in real-time with neural machine translation. 2017.

[46] Thanh-Le Ha, Jan Niehues, and Alexander Waibel. Toward multilingual neural machine translation with universal encoder and decoder. *arXiv preprint arXiv:1611.04798*, 2016.

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[48] Kenneth Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the 6th Workshop on Statistical Machine Translation (WMT)*, pages 187–197, 2011.

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

[50] Robin J Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):26, 2014.

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

[52] Melvin Johnson, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, et al. Google's multilingual neural machine translation system: Enabling zero-shot translation. *arXiv preprint arXiv:1611.04558*, 2016.

[53] Nal Kalchbrenner and Phil Blunsom. Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1700–1709, 2013.

[54] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

[55] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 655–665, 2014.

[56] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. pages 3128–3137, 2015.

[57] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[58] Y. Kim, C. Denton, L. Hoang, and A. M. Rush. Structured Attention Networks. *ArXiv e-prints*, February 2017.

[59] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, 2014.

[60] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[61] Roland Kuhn and Renato De Mori. A cache-based natural language model for speech recognition. *IEEE transactions on pattern analysis and machine intelligence*, 12(6):570–583, 1990.

[62] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[63] Tao Lei, Regina Barzilay, and Tommi Jaakkola. Molding cnns for text: non-linear, non-consecutive convolutions. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1565–1575, 2015.

[64] Tao Lei, Regina Barzilay, and Tommi Jaakkola. Rationalizing neural predictions. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 107–117, 2016.

[65] Jiwei Li, Xinlei Chen, Eduard Hovy, and Dan Jurafsky. Visualizing and understanding neural models in nlp. *arXiv preprint arXiv:1506.01066*, 2015.

[66] Jiwei Li, Thang Luong, Dan Jurafsky, and Eduard Hovy. When are tree structures necessary for deep learning of representations? In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2304–2314, 2015.

[67] Minh-Thang Luong, Ilya Sutskever, Quoc Le, Oriol Vinyals, and Wojciech Zaremba. Addressing the rare word problem in neural machine translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 11–19, 2015.

[68] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1412–1421, 2015.

[69] Haitao Mi, Baskaran Sankaran, Zhiguo Wang, and Abe Ittycheriah. Coverage embedding models for neural machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 955–960, 2016.

[70] Haitao Mi, Zhiguo Wang, and Abe Ittycheriah. Supervised attentions for neural machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2283–2288, 2016.

[71] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[72] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 1045–1048, 2010.

[73] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3111–3119, 2013.

[74] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

[75] Brian Murphy, Partha Talukdar, and Tom Mitchell. Learning effective and interpretable semantic models using non-negative sparse embedding. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 1933–1950, 2012.

[76] Masami Nakamura, Katsuteru Maruyama, Takeshi Kawabata, and Kiyohiro Shikano. Neural network approach to word category prediction for English texts. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, 1990.

[77] Graham Neubig and Chris Dyer. Generalizing and hybridizing count-based and neural language models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[78] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[79] Mohammad Norouzi, Samy Bengio, Navdeep Jaitly, Mike Schuster, Yonghui Wu, Dale Schuurmans, et al. Reward augmented maximum likelihood for neural structured prediction. In *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1723–1731, 2016.

[80] Daisuke Okanohara and Jun'ichi Tsujii. A discriminative language model with pseudo-negative samples. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 73–80, 2007.

[81] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318, 2002.

[82] Adam Pauls and Dan Klein. Faster and smaller n-gram language models. pages 258–267, 2011.

[83] Jordan B Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1):77–105, 1990.

[84] MarcAurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[85] Philip Resnik. Selectional preference and sense disambiguation. In *Proceedings of the ACL SIGLEX Workshop on Tagging Text with Lexical Semantics: Why, What, and How*, pages 52–57. Washington, DC, 1997.

[86] Brian Roark, Murat Saraclar, Michael Collins, and Mark Johnson. Discriminative language modeling with conditional random fields and the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 47–54, 2004.

[87] Ronald Rosenfeld. A maximum entropy approach to adaptive statistical language modelling. *Computer Speech and Language*, 10(3):187 – 228, 1996.

[88] Ronald Rosenfeld, Stanley F Chen, and Xiaojin Zhu. Whole-sentence exponential language models: a vehicle for linguistic-statistical integration. *Computer Speech & Language*, 15(1):55–73, 2001.

[89] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[90] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, 1986.

[91] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[92] Alexander M. Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 379–389, 2015.

[93] Hinrich Sch utze. Word space. 5:895–902, 1993.

[94] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1715–1725, 2016.

[95] Lifeng Shang, Zhengdong Lu, and Hang Li. Neural responding machine for short-text conversation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1577–1586, 2015.

[96] Libin Shen, Jinxi Xu, and Ralph Weischedel. A new string-to-dependency machine translation algorithm with a target dependency language model. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 577–585, 2008.

[97] Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. Minimum risk training for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1683–1692, 2016.

[98] Xing Shi, Inkit Padhi, and Kevin Knight. Does string-based neural mt learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1526–1534, 2016.

[99] Richard Socher, John Bauer, Christopher D. Manning, and Ng Andrew Y. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 455–465, 2013.

[100] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 129–136, 2011.

[101] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014.

[102] Zoltán Gendler Szabó. Compositionality. *Stanford encyclopedia of philosophy*, 2010.

[103] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2015.

[104] David Talbot and Thorsten Brants. Randomized language models via perfect hash functions. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 505–513, 2008.

[105] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[106] Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. Modeling coverage for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 76–85, 2016.

[107] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 384–394. Association for Computational Linguistics, 2010.

[108] Peter D Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37:141–188, 2010.

[109] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499*, 2016.

[110] Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. Decoding with large-scale neural language models improves translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1387–1392, 2013.

[111] Pascal Vincent, Alexandre de Brébisson, and Xavier Bouthillier. Efficient exact gradient update for training deep networks with very large sparse targets. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1108–1116, 2015.

[112] Oriol Vinyals and Quoc Le. A neural conversational model. *arXiv preprint arXiv:1506.05869*, 2015.

[113] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. pages 3156–3164, 2015.

[114] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. 37(3):328–339, 1989.

[115] Mingxuan Wang, Zhengdong Lu, Hang Li, and Qun Liu. Memory-enhanced decoder for neural machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 278–286, 2016.

[116] R.E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[117] Sam Wiseman and Alexander M. Rush. Sequence-to-sequence learning as beam-search optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1296–1306, 2016.

[118] I.H. Witten and T.C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Information Theory, IEEE Transactions on*, 37(4):1085–1094, 1991.

[119] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[120] Zichao Yang, Zhiting Hu, Yuntian Deng, Chris Dyer, and Alex Smola. Neural machine translation with recurrent attention modeling. *arXiv preprint arXiv:1607.05108*, 2016.

[121] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1480–1489, San Diego, California, June 2016. Association for Computational Linguistics.

[122] Lei Yu, Jan Buys, and Phil Blunsom. Online segment to segment neural transduction. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1307–1316, 2016.

[123] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[124] Barret Zoph, Deniz Yuret, Jonathan May, and Kevin Knight. Transfer learning for low-resource neural machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1568–1575, 2016.