Mathematical Engineering of Deep Learning

Book Draft

Benoit Liquet, Sarat Moka and Yoni Nazarathy

February 28, 2024

Contents

Preface - DRAFT 3

1 Introduction - DRAFT 1
1.1 The Age of Deep Learning . . . 1
1.2 A Taste of Tasks and Architectures . . . 7
1.3 Key Ingredients of Deep Learning . . . 12
1.4 DATA, Data, data! . . . 17
1.5 Deep Learning as a Mathematical Engineering Discipline . . . 20
1.6 Notation and Mathematical Background . . . 23
Notes and References . . . 25

2 Principles of Machine Learning - DRAFT 27
2.1 Key Activities of Machine Learning . . . 27
2.2 Supervised Learning . . . 32
2.3 Linear Models at Our Core . . . 39
2.4 Iterative Optimization Based Learning . . . 48
2.5 Generalization, Regularization, and Validation . . . 52
2.6 A Taste of Unsupervised Learning . . . 62
Notes and References . . . 72

3 Simple Neural Networks - DRAFT 75
3.1 Logistic Regression in Statistics . . . 75
3.2 Logistic Regression as a Shallow Neural Network . . . 82
3.3 Multi-class Problems with Softmax . . . 86
3.4 Beyond Linear Decision Boundaries . . . 95
3.5 Shallow Autoencoders . . . 99
Notes and References . . . 111

4 Optimization Algorithms - DRAFT 113
4.1 Formulation of Optimization . . . 113
4.2 Optimization in the Context of Deep Learning . . . 120
4.3 Adaptive Optimization with ADAM . . . 128
4.4 Automatic Differentiation . . . 135
4.5 Additional Techniques for First-Order Methods . . . 143
4.6 Concepts of Second-Order Methods . . . 152
Notes and References . . . 164

5 Feedforward Deep Networks - DRAFT 167
5.1 The General Fully Connected Architecture . . . 167
5.2 The Expressive Power of Neural Networks . . . 173
5.3 Activation Function Alternatives . . . 180
5.4 The Backpropagation Algorithm . . . 184
5.5 Weight Initialization . . . 192

5.6 Batch Normalization . . . 194
5.7 Mitigating Overfitting with Dropout and Regularization . . . 197
Notes and References . . . 203

6 Convolutional Neural Networks - DRAFT 205
6.1 Overview of Convolutional Neural Networks . . . 205
6.2 The Convolution Operation . . . 209
6.3 Building a Convolutional Layer . . . 216
6.4 Building a Convolutional Neural Network . . . 226
6.5 Inception, ResNets, and Other Landmark Architectures . . . 236
6.6 Beyond Classification . . . 240
Notes and References . . . 247

7 Sequence Models - DRAFT 249
7.1 Overview of Models and Activities for Sequence Data . . . 249
7.2 Basic Recurrent Neural Networks . . . 255
7.3 Generalizations and Modifications to RNNs . . . 265
7.4 Encoders Decoders and the Attention Mechanism . . . 271
7.5 Transformers . . . 279
Notes and References . . . 294

8 Specialized Architectures and Paradigms - DRAFT 297
8.1 Generative Modelling Principles . . . 297
8.2 Diffusion Models . . . 306
8.3 Generative Adversarial Networks . . . 315
8.4 Reinforcement Learning . . . 328
8.5 Graph Neural Networks . . . 338
Notes and References . . . 353

Epilogue - DRAFT 355

A Some Multivariable Calculus - DRAFT 357
A.1 Vectors and Functions in $\mathbb{R}^n$ . . . 357
A.2 Derivatives . . . 359
A.3 The Multivariable Chain Rule . . . 362
A.4 Taylor's Theorem . . . 364

B Cross Entropy and Other Expectations with Logarithms - DRAFT 367
B.1 Divergences and Entropies . . . 367
B.2 Computations for Multivariate Normal Distributions . . . 369

Bibliography 399

Index 401

7 Sequence Models - DRAFT

Many forms of data such as text data in the context of natural language processing appear sequentially. In such a case we require deep learning models that can operate on sequences of arbitrary length, and are well adapted to model temporal relationships in the data. The simple first model of this form is the recurrent neural network (RNN) which can be presented as a variation of the feedforward neural network of Chapter 5. In this chapter we explore such models together with many more advanced variants of these models including long short term memory (LSTM) models, gated recurrent unit (GRU) models, models based on the attention mechanism, and in particular transformer models. An archetypical application is end to end natural language translation and we see how encoder-decoder architecture with sequence models can be used for this purpose. The various forms of models including RNN, LSTM, GRU, or transformers can also be used in such an application among others. These models also form the basis for large language models (LLMs) that have shown to be extremely powerful for general tasks.

In Section 7.1 we consider various forms and application domains of sequence data. As a prime example we consider textual data and ways of encoding textual data as a sequence. In Section 7.2 we introduce and explore basic recurrent neural networks which are naturally suited to deal with sequence data. We present the basic auto-regressive structure of such models and discuss aspects of training. In Section 7.3 we explore generalizations of recurrent neural networks including, stacking and reversing approaches, and importantly long short term memory (LSTM) models, and gated recurrent unit (GRU) models. Prior to the appearance of transformers, LSTMs and GRUs marked the state of the art for sequence modelling. We continue in Section 7.4 where we focus on machine translation applications, and explore how encoder-decoder architectures can be used for end-to-end translation. In the process we introduce the attention mechanism which has become a central pillar of modern sequence models. An encoder-decoder architecture based on attention is also presented. In Section 7.5 we dive into the powerful workhorse of contemporary sequence models, the transformer architecture. Transformers, relying heavily on attention, are presented in detail, culminating with a transformer encoder-decoder architecture.

7.1 Overview of Models and Activities for Sequence Data

Sequence models have been motivated by the analysis of sequential data including text sentences, time-series, and other discrete sequence data such as DNA. These models are especially designed to handle sequential information while convolutional neural networks of Chapter 6 are specialized for processing spatial information. Naturally, most interesting input samples carry some statistical dependence between elements due to the sequential nature of the data. Classical statistical models in time-series such as auto-regressive models are naturally tailored for such data when the sample at each datapoint is a scalar or a low dimensional vector. In contrast, the deep learning models that we cover here allow one to work with high-dimensional samples as appearing in textual data and similar domains.

Forms of Sequence Data

We denote a data sequence via $x = \big(x^{\langle 1 \rangle}, \ldots, x^{\langle T \rangle}\big)$, where the superscripts $\langle t \rangle$ indicate time or position, and capture the order in the sequence. Each $x^{\langle t \rangle}$ is a p-dimensional numerical data point (or vector). The number of elements in the sequence, T, is sometimes fixed, but is also often not fixed and can be essentially unbounded. A classical example is a numerical univariate data sequences ($p = 1$) arising in time-series of economic, natural, or weather data. Similarly, multivariate time-series data ($p > 1$ but typically not huge) also arise in similar settings.

Most of the motivational examples in this chapter are from the context of textual data. In this case, t is typically not the time of the text but rather the index of the word or token[1] within a text sequence. One way to encode text is that each $x^{\langle t \rangle}$ represents a single word using an *embedding vector* in a manner that we discuss below. If for example x is the text associated with the Bible then T is large,[2] whereas if x is the text associated with a movie review as per the IMDB movie dataset (see Figure 1.6 (d) in Chapter 1), then T is on average 231 words. In data formats similar to the latter case, the data involves a collection of data sequences $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$ where each $x^{(i)}$ is an individual movie review and n denotes the total number of movie reviews. While in practice, such data formats often arise, for simplicity, our discussion in this chapter mostly assumes a single (typically long) text sequence x.

To help make the discussion concrete, assume momentarily that we encode input text in the simplest possible manner, where the embedding vector just uses a technique called *one-hot encoding*. With this approach we consider the number of words in the dictionary, vocabulary, or lexicon as $d_{\mathcal{V}}$ (e.g., $d_{\mathcal{V}} \approx 40,000$) and set $p = d_{\mathcal{V}}$. We then associate with each possible word, a unit vector $e_1, \ldots, e_p$ which uniquely identifies the word. At this point, an input data sequence (text) is converted into a sequence of vectors, where $x^{\langle t \rangle} = e_i$ whenever the t-th word in the sequence is the i-th word in lexicographic order in the dictionary. This approach is very simplistic and may appear inefficient. Yet it illustrates that textual data may be easily represented as a numerical input. With more advanced *word embedding* methods discussed below, the dimension of each $x^{\langle t \rangle}$ can be significantly reduced.

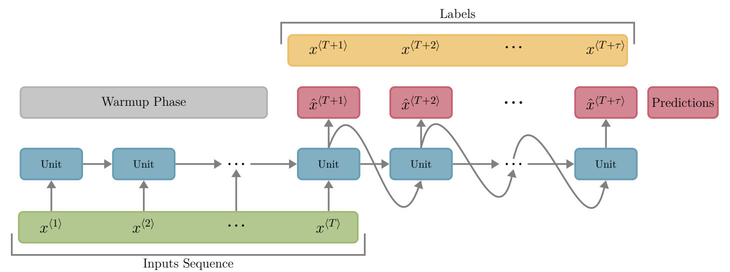
Tasks Involving Sequence Data

There are plenty of tasks and applications involving sequence data. In the context of deep learning such tasks are handled by neural network models. The more classical forms of neural networks for sequence data are generally called *recurrent neural networks* (RNN), while more modern forms are called *transformers*. We focus on the more classical RNN forms in the first sections of this chapter and later visit transformers. The basic forms of RNNs are introduced in Section 7.2. At this point assume that each of these models processes an input sequence x to create some output $\hat{y}$, where the creation of the output is sequential in nature.

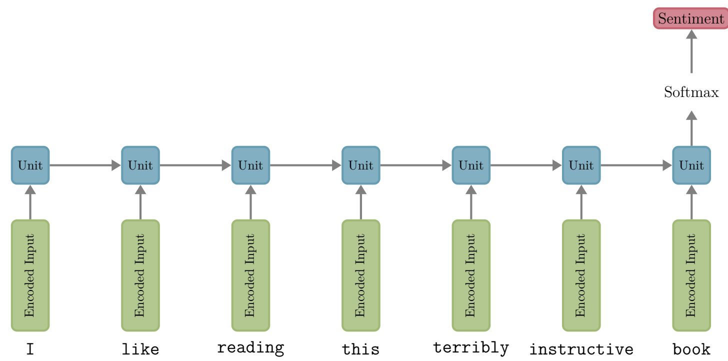
For our discussion, let us focus on text based applications and highlight a few of the tasks and applications in this context. The ideas may then be adapted to domains such as time-

[1] In a complete treatment of textual data analysis or *natural language processing* (NLP) one requires to define and analyze *tokenizers* which break up text into natural "words" or parts of words known as *tokens*. These details are not our focus and we use "word" and "token" synonymously.

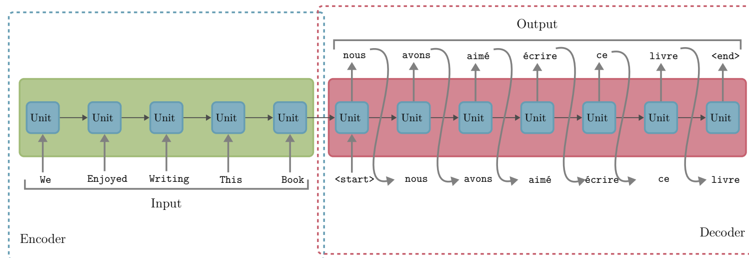
[2] By some counts, there are about half a million sequential words in the old testament of the Bible and more when one considers the new testament and its many variants.



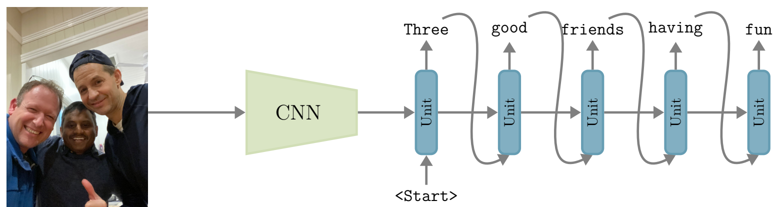
(a)



(b)



(c)



(d)

Figure 7.1: Use of recurrent neural network models (or generalizations) for various sequence data and language tasks. The basic building block, called a unit, is recursively used in the computation. (a) Lookahead prediction of the sequence. (b) Classification of a sequence or sentiment analysis. (c) Machine translation. (d) Image captioning.

series, signal processing, and others. Figure 7.1 illustrates schematically how RNNs (or their generalizations) can be used. The building blocks of these types of models are called *units*, and they are recursively used in the computation of input to output. A basic task presented in (a) is *look-ahead prediction* which in the application context of text, implies predicting the next word (or collection of words) in a sequence. Another type of task presented in (b) is sequence regression or classification which can be used for applications such as *sentiment analysis*. An additional major task illustrated in (c) is *machine translation* where we translate the input sequence from one language to another (e.g., Hebrew to Arabic). Another type of task illustrated in (d) involves decoding an input into a sequence. One such example application is *image captioning* where text is generated to describe the input image. Let us now focus on (a)–(d) in more detail.

Consider Figure 7.1 (a) illustrating lookahead prediction. The simple application in the context of text is to predict the next word (or next few words) based on the sequence of input words. Thus, the output is the sequence of inputs shifted by one and the model attempts to predict the next word at any time t. In the context of time-series this is often referred to as an *auto-regressive* analysis. After a warmup phase, the model predicts the next value in the time series which is also used for predicting the subsequent values until a desired horizon. Hence for an input sequence $x = \big(x^{\langle 1 \rangle}, \ldots, x^{\langle t \rangle}\big)$ we have a future prediction $\hat{y} = \big(\hat{y}^{\langle t+1 \rangle}, \ldots, \hat{y}^{\langle t+\tau \rangle}\big)$ for some *time horizon* τ. Note that the typical use of *large language models* follows this task as well since an input text is given and a response is returned.

Consider now Figure 7.1 (b) illustrating an input sequence processed to produce a single scalar or vector output. An archetypical application in the context of text is *sentiment analysis* where the sentiment or "general vibe" of a sentence is determined. This output may be a vector of probabilities over possible classes, e.g., {`happy`, `sad`, `indifferent`}, and in such a case the output is amenable to classification. Hence $\hat{y}$ is a vector of probabilities and it can also be converted to a categorical output $\widehat{\mathcal{Y}}$ as in (3.34) of Chapter 3.

Moving onto Figure 7.1 (c), consider the application of *machine translation* where the input sentence is from one language and the output sentence is from another language. The architecture of such a model can be composed of two RNNs (or two other types of sequence models) in an *encoder-decoder architecture*. Here the encoder model encodes the input sentence from one language into a *context vector* in a *latent space* and the decoder model decodes from the latent space into a sentence in another language. We describe architectures for such tasks in Section 7.4 and specific transformer models of this form in Section 7.5. Observe that with this task, the input x is a sequence of a certain length while the output $\hat{y}$ is a sequence of a potentially different length. Note that the notion of a latent space was first introduced in a different context of autoencoders in Section 3.5.

Figure 7.1 (d) illustrates the task of *image captioning*. Here for an input image, we wish to output a sentence describing the image. A common way to achieve this is with a convolutional neural network as in Chapter 6 creating a context vector in a latent space. This context vector is then fed into an RNN (or similar sequence model) which acts as a decoder, somewhat similarly to the decoder in the machine translation case. In this application x is an image, and $\hat{y}$ represents an output sentence.

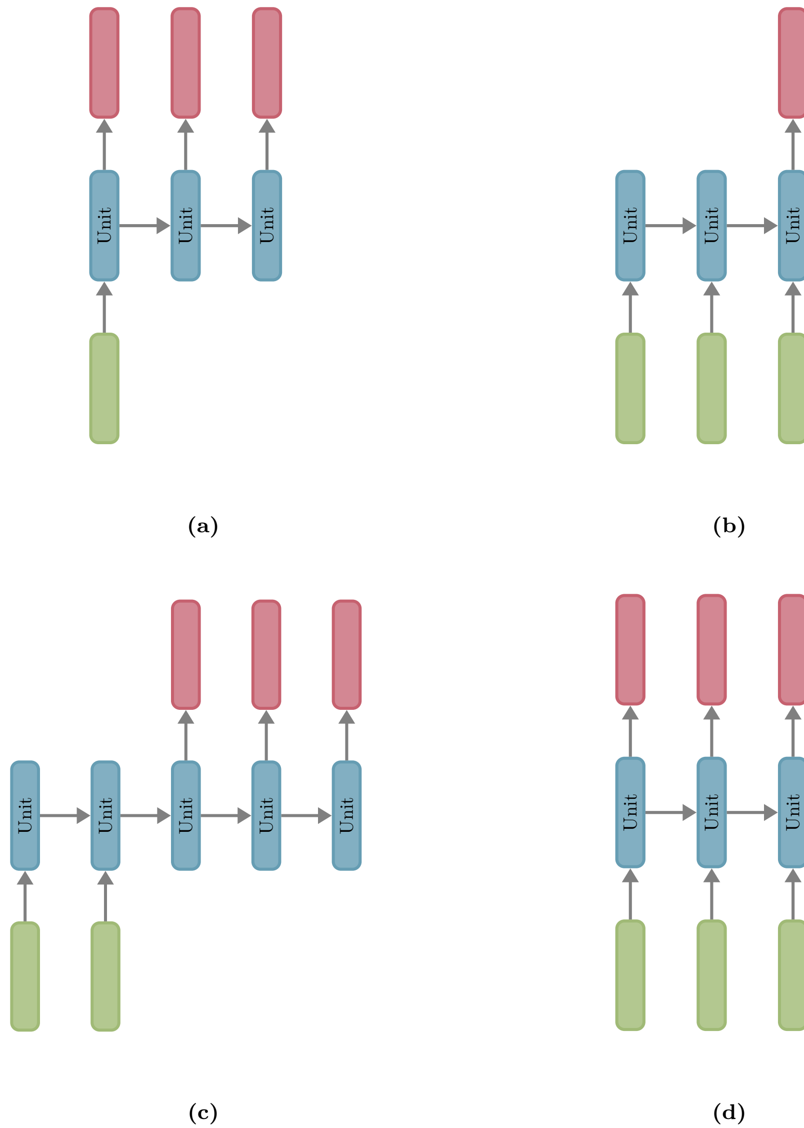


Figure 7.2: Input output paradigms of sequence models. (a) One-to-many. (b) Many-to-one. (c) Many-to-many with partial inputs and outputs. (d) Many-to-many with complete inputs and outputs.

With the tasks and applications highlighted, we see various forms of input x and output $\hat{y}$. Sometimes x and $\hat{y}$ are sequences and at other times they are not. It is often common to describe tasks and models as *one to many*, *many to one*, or *many to many*; see Figure 7.2. In the *one to many* case, x is simply $x^{\langle 1 \rangle}$ while $\hat{y}$ is a sequence, $\hat{y}^{\langle 1 \rangle}, \hat{y}^{\langle 2 \rangle}, \ldots$. In the *many to one* case, $x = \big(x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots\big)$ is a sequence while $\hat{y}$ is a single output (scalar or vector). Finally in the *many to many* case, both x and $\hat{y}$ are sequences. Returning to Figure 7.1, observe that the lookahead prediction task (a) falls in either the *many to one* or the *many to many* case, depending on if the time horizon τ is 1 or greater, respectively. The sentiment

analysis task (b) is a *many to one* case. The machine translation task (c) is a *many to many* case, while the image captioning task (d) is a *one to many* case.

Word Embedding

One-hot encoding which is the simplest way to encode a word results in a very sparse vector of high dimensionality, with the dimension being the size of the lexicon, $d_{\mathcal{V}}$. A popular alternative that has become standard in any application involving text is to use *word embeddings*, where we represent each word (or token) by a vector of real numbers of dimension p, and with p much smaller than $d_{\mathcal{V}}$.

The essence of word embedding techniques is that words from similar contexts have corresponding vectors which are relatively close. Such closeness is often measured via the cosine of the angle[3] between the two vectors in Euclidean space. As an hypothetical example with $p = 4$, the word `king` could be represented by the vector $(0.41, 1.2, 3.4, -1.3)$ and the word `queen` can be represented by a relatively similar vector such as $(0.39, 1.1, 3.5, 1.6)$. Then a completely different word such as `mean` might be represented by a vector such as $(-0.2, -3.2, 1.3, 0.8)$. One can now verify in this example, that the cosine of the angle between `king` and `queen` is about 0.729 while the cosine of the angle between `mean` and the other two words is lower, and is at about -0.04 for `king` and 0.156 for `queen`, respectively.

Hence with such an embedding, beyond the value of reducing the dimension of each $x^{\langle t \rangle}$ from $d_{\mathcal{V}}$ (in the order of tens of thousands) to p (in the order of hundreds), we also get the benefit of similarity and context groupings. Having such a contextual representation of words plays a positive role in deep learning models since it allows the models to use context more efficiently.

Simple word embedding techniques map individual words into vectors, while more advanced techniques are context aware and yield a representation of the words based on the context within the rest of the text, enabling models to better deal with homonyms. For example the word `mean` inside the phrase `mean value`, is very different than the same word in side the phrase `mean person`. Hence an advanced word embedding technique will encode each of the occurrences of `mean` differently.

A popular early word embedding technique is *word2vec*. The creation of this embedding relies on a neural network trained on very large corpora, to build the embedding vectors. The basic idea is to train the neural network for a task, and then use an internal layer of the network as the word embedding. Such an approach of a derived feature vector is common throughout deep learning. There are two common variations of the word2vec training algorithm with one approach called the *bag of words* model, seeking to predict a word from its neighboring words, while the other approach, the *skip-gram model*, seeks to predict the words of the context from a central word. In practice, both with word2vec, and with more advanced algorithms,[4] one may choose if to use a fixed pre-trained version of the word embedding, or if to fine tune and adjust the word embedding when used as part of a larger model.

[3] See (A.1) in Appendix A.

[4] See references to other word embedding approaches in the notes and references at the end of the chapter.

7.2 Basic Recurrent Neural Networks

Recurrent neural networks are specifically designed for sequences of data and have the ability to: (i) deal with variable-length sequences, (ii) maintain sequence order, (iii) keep track of long-term dependencies, and (iv) share parameters across an input sequence. In order to achieve all of these goals, the *recurrent neural network* (RNN) model introduces an internal loop which allows information to be passed from one step of the network to the next. The RNN maintains a *hidden state*, also termed *cell state*, which allows the model to keep memory as an input sequence is processed. This state evolves over time, as the input sequence is fed into the model. See Figure 7.3 for a schematic illustration of both a *recursive graph* representation and an *unfolded graph* representation of the model. In the figure we schematically see how an RNN transforms an input sequence to an output sequence, with the blue nodes representing units of the model; details follow.

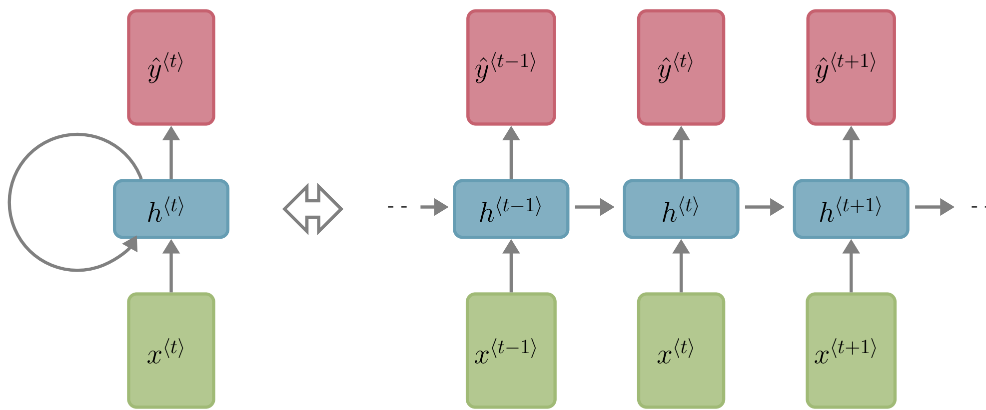


Figure 7.3: A recursive neural network RNN can be represented (left) via a recursive graph and (right) via an unfolded representation of that graph. An input sequence $x^{\langle 1 \rangle}, x^{\langle 2 \rangle} \ldots$ is transformed to an output sequence $\hat{y}^{\langle 1 \rangle}, \hat{y}^{\langle 2 \rangle} \ldots$, where in each step t, the unit, with cell state represented via $h^{\langle t \rangle}$, performs the computation.

Recurrent neural networks apply a *recurrence relation* where at every time step the next input is combined with the previous state to update the new state and a new output. This internal loop is the key difference between traditional feedforward neural networks of Chapter 5 and RNNs. In the traditional models, the flow of information from input to output via hidden layers is only in the forward direction, whereas in RNNs, the input plays a role as information flows. More specifically, in the traditional models, there is no cyclic connection between neurons; contrast Figure 5.1 of Chapter 5 with Figure 7.3. Moreover, the traditional feedforward neural networks work with fixed length input and fixed length output while the RNN input sequence and output sequence are each allowed to be of variable (essentially unbounded) length.

The neurons inside RNNs implement the cell state and are typically denoted via $h^{\langle t \rangle}$ for the state at time t. Mathematically, the *state evolution* can be represented via the recurrence relation,

$$\underbrace{h^{\langle t \rangle}}_{\text{current state}} = f_{\theta_{hx}, \theta_{hh}}\big(\underbrace{h^{\langle t-1 \rangle}}_{\text{old state}}, \underbrace{x^{\langle t \rangle}}_{\text{input vector}} \big),$$

acting on the sequence of input data $x = \big(x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots\big)$, to create a sequence of cell states $h^{\langle 1 \rangle}, h^{\langle 2 \rangle}, \ldots$, where the initial state $h^{\langle 0 \rangle}$ is typically taken as a zero vector. It is important to note that at every time step t the same function $f_{\theta_{hx},\theta_{hh}}(\cdot)$ is used with the same fixed (over time) sets of parameters θ_{hx} and θ_{hh}.

The output sequence $\hat{y} = \big(\hat{y}^{\langle 1 \rangle}, \hat{y}^{\langle 2 \rangle}, \ldots\big)$ is defined at each time step by an another function $g_{\theta_{yh}}(\cdot)$ with,

$$\hat{y}^{\langle t \rangle} = g_{\theta_{yh}}(h^{\langle t \rangle}),$$

where again, the parameters θ_{yh} are fixed over time.

The recursive loop enables us to express the cell state at time t, $h^{\langle t \rangle}$, in terms of the t first inputs, namely, omitting the parameter subscripts from $f_{\theta_{hx},\theta_{hh}}(\cdot)$, we have,

$$h^{\langle t \rangle} = f(\ldots \overbrace{f(\underbrace{f(\overbrace{\underbrace{f(h^{\langle 0 \rangle}, x^{\langle 1 \rangle})}_{h^{\langle 1 \rangle}}, x^{\langle 2 \rangle})}^{h^{\langle 2 \rangle}}, x^{\langle 3 \rangle})}_{h^{\langle 3 \rangle}} \ldots}^{h^{\langle t-1 \rangle}}, x^{\langle t \rangle}). \tag{7.1}$$

Thus, since at time t, the output $\hat{y}^{\langle t \rangle}$ is a function of $h^{\langle t \rangle}$, we can also express the output as a function of the inputs up until time step t. Namely,

$$\hat{y}^{\langle t \rangle} = g_{\theta}^{(t)}(x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, x^{\langle 3 \rangle}, \ldots, x^{\langle t \rangle}),$$

where the function $g_{\theta}^{(t)}(\cdot)$ is specific to time t and captures the unrolling of the state as in (7.1) or Figure 7.3 (b). This highlights the ability of RNN to deal with variable length input and output sequences. Here $\theta = (\theta_{hx}, \theta_{hh}, \theta_{yh})$ is the collection of all learnable parameters of the RNN.

The functions $f_{\theta_{hx},\theta_{hh}}(\cdot)$ and $g_{\theta_{yh}}(\cdot)$ are concretely defined via affine transformations and non-linear activations similarly to other common neural network models. Specifically,

$$\begin{cases} h^{\langle t \rangle} = S_h(W_{hh} h^{\langle t-1 \rangle} + W_{hx} x^{\langle t \rangle} + b_h) \\ \hat{y}^{\langle t \rangle} = S_y(W_{yh} h^{\langle t \rangle} + b_y). \end{cases} \tag{7.2}$$

The parameters $\theta = (\theta_{hx}, \theta_{hh}, \theta_{yh})$ are captured via weight matrices and bias vectors.[5] Further, $S_h(\cdot)$ and $S_y(\cdot)$ are vector activation functions typically composed of element-wise scalar activations $\sigma(\cdot)$, similarly to Chapter 5. We denote the dimension of $x^{\langle t \rangle}$ as p, the dimension of $y^{\langle t \rangle}$ as q, and the dimension of the cell state of $h^{\langle t \rangle}$ as m. Hence $W_{hx} \in \mathbb{R}^{m \times p}$, $W_{hh} \in \mathbb{R}^{m \times m}$, $W_{yh} \in \mathbb{R}^{q \times m}$, $b_h \in \mathbb{R}^m$, and $b_y \in \mathbb{R}^q$.

One variant is to feed the output of the previous time step, $\hat{y}^{\langle t-1 \rangle}$, into the input so that the input at every time is not $x^{\langle t \rangle}$ but rather some merging of $x^{\langle t \rangle}$ and $\hat{y}^{\langle t-1 \rangle}$. We can denote this variant via,

$$\begin{cases} h^{\langle t \rangle} = S_h(W_{hh} h^{\langle t-1 \rangle} + W_{hx}(x^{\langle t \rangle} + \mathcal{T}\big(\hat{y}^{\langle t-1 \rangle}\big)) + b_h) \\ \hat{y}^{\langle t \rangle} = S_y(W_{yh} h^{\langle t \rangle} + b_y), \end{cases} \tag{7.3}$$

[5] The actual mapping of the weight and bias vectors to each of $(\theta_{hx}, \theta_{hh}, \theta_{yh})$ is not important. Specifically, b_h can be viewed as either part of θ_{hx} or θ_{hh}.

where $\mathcal{T}(\,\cdot\,)$ is an abstraction[6] of some transformation which results in a vector of dimension p (like $x^{\langle t \rangle}$).

Note that the forms in (7.2) and (7.3) are suitable for many-to-many mappings, as in Figure 7.2 (d), since each input $x^{\langle t \rangle}$ has an associated $h^{\langle t \rangle}$ and $\hat{y}^{\langle t \rangle}$. If we use these recursions for one-to-many tasks, then we only use a first initial $x^{\langle 1 \rangle}$ and then continue the recursion with 0 values for $x^{\langle t \rangle}$ on $t = 2, 3, \ldots$ until some stopping criterion is met. A typical criterion is to have a special `<stop>` token appear within $\hat{y}^{\langle t \rangle}$. A similar adaptation can be done for many-to-one tasks, where we simply ignore all $\hat{y}^{\langle t \rangle}$ for $t < T$.

One often refers to the mechanism of computation described in (7.2) as a *gate*, a *simple gate*, an *RNN cell*, or an *RNN unit*. Figure 7.4 depicts this computation where $\oplus$ and $\otimes$ are the usual vector/matrix addition and multiplication operations respectively. In the sequel we see that the gate structure as in Figure 7.4 can be modified to more complicated forms such as LSTM and GRU gates appearing in Figure 7.8. In terms of applications, the simple structure of RNN gates has already proven useful for many basic tasks such as for example dealing with short sentences for next word prediction as well as for sentiment analysis.

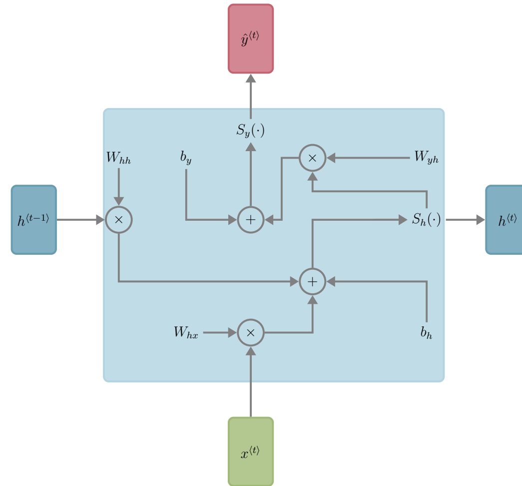


Figure 7.4: An RNN unit, also known as a gate, operating on input $x^{\langle t \rangle}$, and previous cell state $h^{\langle t-1 \rangle}$. The output vector of the unit is $\hat{y}^{\langle t \rangle}$. The unit also determines the cell state $h^{\langle t \rangle}$.

A Simple Concrete Toy Example

To illustrate the application of RNNs let us consider a simple concrete toy example of lookahead text prediction. For simplicity we resort to one-hot encoding (in contrast to more advanced word embedding methods). In our toy example assume a lexicon with $d_{\mathcal{V}} = 8$ words, appearing here in lowercase alphabetical order as,

[6] In practice, this variant is often useful in decoders, described in sections 7.4 and 7.5, where $x^{\langle t \rangle}$ is often set to 0 except for an initial `<start>` token, and the transformation $\mathcal{T}(\cdot)$ typically transforms an output embedding into the desired token (e.g., via argmax) and then transforms the token back into an input word embedding.

$$\Big[\texttt{deep},\ \ \texttt{engineering},\ \ \texttt{learning},\ \ \texttt{machine},\ \ \texttt{mathematical},\ \ \texttt{of},\ \ \texttt{statistics},\ \ \texttt{the}\Big],$$

where each word is represented by a unit vector in $\mathbb{R}^8$. For example, the text,

$$\texttt{the mathematical engineering of deep learning},$$

is represented via a sequence $x = \big(x^{\langle 1\rangle}, x^{\langle 2\rangle}, x^{\langle 3\rangle}, x^{\langle 4\rangle}, x^{\langle 5\rangle}, x^{\langle 6\rangle}\big)$. Here for example $x^{\langle 1\rangle} = (0, 0, 0, 0, 0, 0, 0, 1)$ because the first word in the sequence, `the`, is the 8-th word in the lexicon, and similarly $x^{\langle 2\rangle} = (0, 0, 0, 0, 1, 0, 0, 0)$ because the second word in the sequence, `mathematical`, is the 5-th word in the lexicon, etc. Observe that here with one-hot encoding, $p = d_{\mathcal{V}}$.

For a lookahead prediction application we set the network output to be of dimension $q = 8$ since each output is of the size of the lexicon. Here when the network is fed a partial input $x^{\langle 1\rangle}, \ldots, x^{\langle t\rangle}$, the output at time (step) t, denoted via $\hat{y}^{\langle t\rangle}$ should ideally be equal to or be close to the one-hot encoded target $y^{\langle t\rangle} := x^{\langle t+1\rangle}$ (the next word). Similarly to the classification examples arising in multinomial regression in Section 3.3, our RNN will output $\hat{y}^{\langle t\rangle}$ vectors that are probability vectors over the lexicon. In this case, using maximum a posteriori probability decisions as in (3.34) of Chapter 3, the coordinate of $\hat{y}^{\langle t\rangle}$ with the highest probability can be taken as a prediction $\widehat{\mathcal{Y}}^{\langle t\rangle}$ which is an index into the lexicon, determining the predicted word.

Now following the RNN evolution equations defined in (7.2), we present concrete dimensions for this illustrative example. A design choice is the size of the hidden state m, which in this case we arbitrarily take as $m = 20$. Hence, the weight matrix W_{hh} dealing with state evolution is 20×20, the weight matrix W_{hx} is 20×8 dimensional as it converts the inputs to the state, and the bias vector b_h is a 20 dimensional vector. The vector activation function $S_h(\cdot)$ is composed of scalar activations, which can be of any of the standard types (e.g., sigmoid); see Section 5.3. Further, the matrix translating state to output, W_{yh} is 8×20 since $q = 8$, and finally b_y is an 8 dimensional vector. Importantly, in this case, we take the output activation function $S_y(\cdot)$ as a softmax function since it converts the affine transformation of the state into a probability vector. Hence in summary, such a toy network would have $20 \times 20 \ + \ 20 \times 8 \ + \ 20 \ + \ 8 \times 20 \ + \ 8 = 748$ parameters to learn.

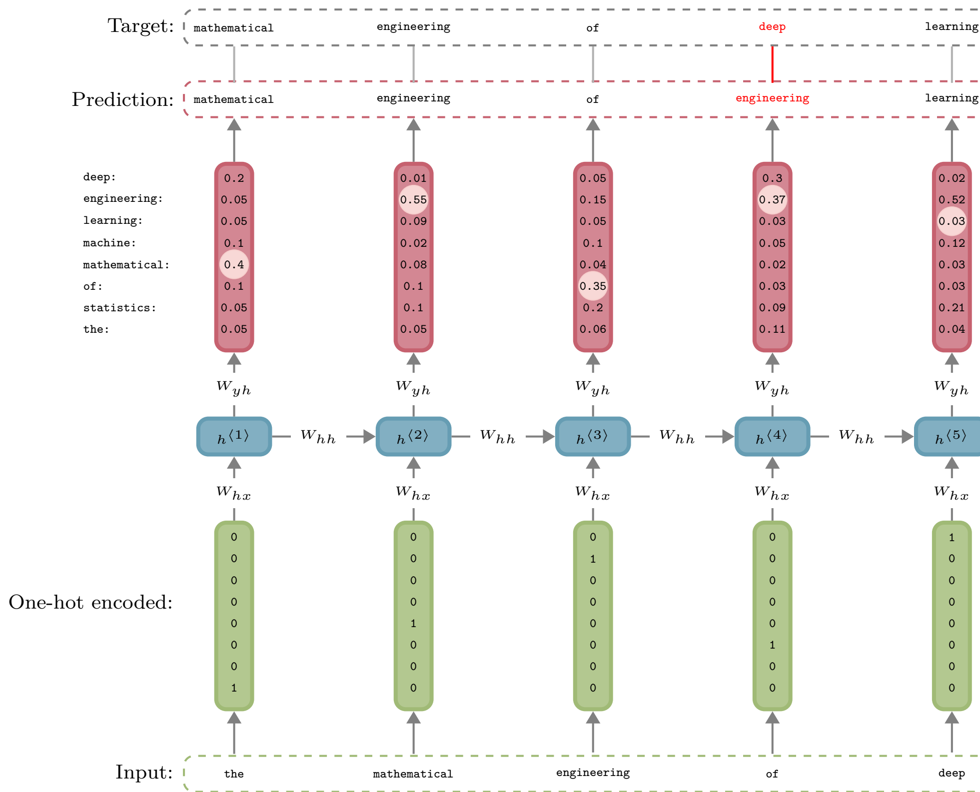


Figure 7.5: A schematic of RNN cells unrolled for language modeling. In this illustration, the input sentence `the mathematical engineering of deep learning`, yields lookahead prediction `mathematical engineering of` *engineering* `learning`, with a single error.

Figure 7.5 presents a schematic illustration of the unrolling of this toy network. When the model is used in training, the shifted sequence $y = \big(x^{\langle 2 \rangle}, x^{\langle 3 \rangle}, \ldots\big)$ serves as the sequence of target labels for comparison; these are one-hot encoded vectors. Then for given weight and bias parameters, we predict $\hat{y} = \big(\hat{y}^{\langle 1 \rangle}, \hat{y}^{\langle 2 \rangle}, \ldots\big)$ as the predicted labels, where each $\hat{y}^{\langle t \rangle}$ is a vector of probabilities over the lexicon for output word at step t. We then use categorical cross entropy (see equation (3.30) in Chapter 3) to compute the loss. We train using gradient based learning similarly to all other deep learning models, yet for evaluation of the gradient, we use a variant of backpropagation called *backpropagation through time* which is described below.

In production, the way that the model is used, is by selecting the word at time t, with the highest probability in $\hat{y}^{\langle t \rangle}$. We denote the index of this selection via $\widehat{\mathcal{Y}}^{\langle t \rangle}$. In the illustration of Figure 7.5, most of the words are properly predicted with the exception of the fourth word at $t = 4$ which is predicted as `deep` while the target is `engineering`.

Training an RNN with Backpropagation Through Time

In general when training an RNN, the loss function is accumulated over all time steps t. In particular, during the execution of gradient descent or some generalization of gradient

descent, such as ADAM described in Chapter 4, we compute the loss and its derivatives with respect to weight and bias parameters, i.e., θ. A general expression for the loss is

$$C(\theta) = \frac{1}{T}\sum_{t=1}^{T} C^{\langle t \rangle}(\theta). \tag{7.4}$$

Here $C^{\langle t \rangle}(\theta)$ denotes the individual loss associated with time t. For example, continuing with the text language model from above, we may set,

$$C^{\langle t \rangle}(\theta) = -\sum_{k=1}^{d_{\mathcal{V}}} y_k^{\langle t \rangle} \log \hat{y}_k^{\langle t \rangle}, \tag{7.5}$$

similarly to the categorical cross entropy in (3.30) of Chapter 3. Here, keep in mind that $y^{\langle t \rangle}$ is one-hot encoded of dimension $q = d_{\mathcal{V}}$, and hence the summation in (7.5) has a single non-zero summand at the index k for which $y_k^{\langle t \rangle} = 1$. Further, note that even if word embeddings are used for the input $x^{\langle t \rangle}$, then the output $\hat{y}^{\langle t \rangle}$ still represents a probability vector over the lexicon of size $d_{\mathcal{V}}$ so that it is comparable to the target output $y^{\langle t \rangle}$.

Gradient computation of $C(\theta)$ with respect to the various weight matrices and bias vectors in θ is somewhat similar to the backpropagation algorithm described in Section 5.4; see also Section 4.4 for automatic differentiation basics. However, a key difference lies in the fact that the same weight and bias parameters are used for all time t; see the unfolded representation of the RNN in Figure 7.3. This difference as well as the fact that inputs to RNN are of arbitrary size T, imposes some hardships on gradient computation. The basic algorithm is called *backpropagation through time*.

One way to view the algorithm is to momentarily return to the feedforward networks of Chapter 5 and assume that the weight matrices and bias vectors of layers are all constrained to be the same with a single set of parameters, W and b, for all layers. Further, momentarily assume that the network depth L, is fixed at the sequence input length T. This form of a feedforward network is essentially an unfolded RNN if we consider every recursive step of the RNN as a layer in the feedforward network, and if we ignore inputs to the RNN beyond the first input $x^{\langle 1 \rangle}$, and impose loss on the RNN only for the last output.

One can also modify feedforward networks to have additional external inputs at each of the hidden layers. In our feedforward analogy, assume now that an external input to the ℓ-th layer is $x^{\langle \ell \rangle}$, where the layer ℓ and the time t play the same role. Further, one may impose loss functions on feedforward networks that not only take the neurons at the last layer as arguments, but rather use all layers, similarly to (7.4). If we also employ such a loss function on the feedforward network analogy then we see that we can treat the unfolded recurrent neural network as a feedforward network, where for simplicity we treat the transformation from $h^{\langle t \rangle}$ to $y^{\langle t \rangle}$ in (7.2) as the identity transformation. With this, let us return to the details of the backpropagation algorithm in Section 5.4 and see how it can be adapted for recurrent neural networks.

At first a forward pass is carried out to populate the neuron values. In feedforward networks these were denoted $a^{[\ell]}$ whereas in the unfolded recurrent neural network they are denoted via $h^{\langle t \rangle}$. Then a backward pass is used to compute the adjoint elements, denoted $\zeta^{\langle t \rangle}$, similarly

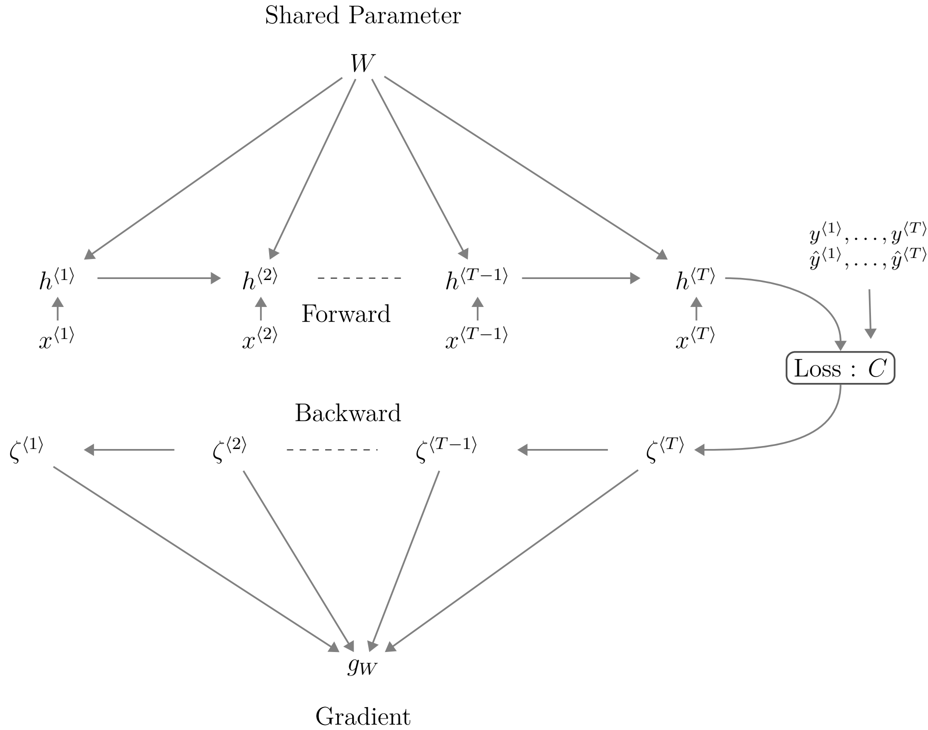


Figure 7.6: The variables and flow of information in the backpropagation through time algorithm. The shared parameter W influences the recursive forward pass computation of all cell states $h^{\langle 1\rangle},\dots,h^{\langle T\rangle}$. Once the backwards pass computation is carried out for all adjoints $\zeta^{\langle T\rangle},\dots,\zeta^{\langle 1\rangle}$, they are all used to compute the gradient of the loss, g_W.

to the adjoints defined in (5.24) of Chapter 5. In the RNN context these are,

$$\zeta^{\langle t\rangle} = \frac{\partial C(\theta)}{\partial h^{\langle t\rangle}}.$$

The essence of backpropagation is computing $\zeta^{\langle t-1\rangle}$ based on $\zeta^{\langle t\rangle}$. This computation follows similar lines to (5.26) of Chapter 5, adapted here to be,

$$\zeta^{\langle t\rangle} = \begin{cases} \frac{1}{T}\sum_{\tau=1}^{T} \dot{C}^{\langle \tau\rangle}(h^{\langle \tau\rangle}), & t = T, \\ \frac{\partial h^{\langle t+1\rangle}}{\partial h^{\langle t\rangle}}\,\zeta^{\langle t+1\rangle}, & t = T-1,\dots,1, \end{cases} \tag{7.6}$$

where $\dot{C}^{\langle \tau\rangle}(h^{\langle \tau\rangle})$ is the derivative of (7.5) with respect to the prediction $\hat{y}^{\langle \tau\rangle}$ for which we are given computable expressions.

Once the backpropagation through time recursion (7.6) is carried out, we use the computed adjoint sequence $\zeta^{\langle T\rangle},\dots,\zeta^{\langle 1\rangle}$ to evaluate the gradient of the loss with respect to components of θ. For simplicity let us focus only on W_{hh} as appearing in (7.2), denoted here as W for

brevity. Specifically, we are interested in evaluating the $m \times m$ derivative matrix,

$$g_W = \frac{\partial C}{\partial W} = \frac{1}{T} \sum_{t=1}^{T} \frac{\partial C^{\langle t \rangle}\left(h^{\langle t \rangle}, y^{\langle t \rangle}; W\right)}{\partial W}, \tag{7.7}$$

similarly to the notation in the feedforward case as in (5.22) of Chapter 5. A noticeable difference between (7.7) and (5.22) is that due to the loss function structure in (7.4), g_W is a direct function of the cell state at all times t (all internal layers of the unfolded graph). However, a more important difference is due to the fact that all time steps (unfolded layers) share the same parameter W, and thus the computational graph connecting W and the loss dictates that all adjoints affect g_W. See Figure 7.6 and contrast it with Figure 5.7 of Chapter 5.

While the feedforward case in Chapter 5 with individual parameters per layer has an easy translation of an adjoint into a gradient, as in the right hand side of (5.25), here the translation of adjoints to a gradient is more complicated and more computationally costly. Specifically, using the multivariate chain rule, we can be informally[7] represent the gradient as,

$$g_W = \frac{1}{T} \sum_{t=1}^{T} \sum_{\tau=1}^{t} \frac{\partial h^{\langle \tau \rangle}}{\partial W} \underbrace{\frac{\partial C}{\partial h^{\langle \tau \rangle}}}_{\zeta^{\langle \tau \rangle}}. \tag{7.8}$$

To understand the internal summation in (7.8), recall that the output $\hat{y}^{\langle t \rangle}$, used in the individual loss $C^{\langle t \rangle}$, depends on all cell states $h^{\langle 1 \rangle}, \ldots, h^{\langle t \rangle}$, where each cell state is parameterized by a common W. Hence the computational graph for this loss component, needs to be taken into account when applying the chain rule. This is also illustrated in the top part of Figure 7.6.

Note that formally the expression $\frac{\partial h^{\langle \tau \rangle}}{\partial W}$ in (7.8) is a derivative of a vector with respect to a matrix, and we do not handle such objects in this book. An alternative is to represent each scalar component of $h^{\langle t \rangle}$ separately. Using (7.2) and assuming the vector activation function $S_h(\cdot)$ is composed of scalar activation functions $\sigma(\cdot)$, we have,

$$h_j^{\langle \tau \rangle} = \sigma([W_{hh} h^{\langle \tau - 1 \rangle} + W_{hx} x^{\langle \tau \rangle} + b_h]_j).$$

Now using (A.15) from Appendix A, we have that the derivative of the scalar $h_j^{\langle \tau \rangle}$ with respect to the weight matrix W_{hh} (abbreviated as W) is given by the matrix,

$$\frac{\partial h_j^{\langle \tau \rangle}}{\partial W} = \dot{\sigma}(W_{hh} h^{\langle \tau - 1 \rangle} + W_{hx} x^{\langle \tau \rangle} + b_h)\, e_j \left(h^{\langle \tau - 1 \rangle}\right)^\top, \tag{7.9}$$

where e_j is the m-dimensional unit vector with 1 at the j-th coordinate, and $\dot{\sigma}(\cdot)$ is the derivative of the scalar activation function; see also Section 5.3.

Continuing with the approach of treating individual neurons $h_j^{\langle \tau \rangle}$, let us now present a more precise version of (7.8). For this consider the fact that in computing each individual loss, $C^{\langle \tau \rangle}$, we rely on the neurons with the cell states $h_j^{\langle 1 \rangle}, \ldots, h_j^{\langle \tau \rangle}$, for all $j = 1, \ldots, m$. In turn, each of these neurons is influenced by W (shorthand for W_{hh}), as in (7.9). Now (also

[7] The representation in (7.8) is informal because the vector-matrix derivative $\frac{\partial h^{\langle \tau \rangle}}{\partial W}$ is not a matrix.

summing up over all individual losses for $t = 1, \ldots, T$), we use the multivariate chain rule to arrive at,

$$g_W = \frac{1}{T} \sum_{t=1}^{T} \sum_{\tau=1}^{t} \sum_{j=1}^{m} \frac{\partial h_j^{\langle \tau \rangle}}{\partial W} \zeta_j^{\langle \tau \rangle}, \tag{7.10}$$

which is fully computable using the backpropagated adjoints from (7.6) and (7.9).

To summarize backpropagation through time, we first carry out a forward pass to populate $h^{\langle 1 \rangle}, \ldots, h^{\langle T \rangle}$ using (7.2) or the (7.3) variant. We then carry out a backward pass to populate the adjoints $\zeta^{\langle T \rangle}, \ldots, \zeta^{\langle 1 \rangle}$ using (7.6). We then compute the gradient g_W via (7.10). This summary is for our simplified case focusing only on $W = W_{hh}$ and ignoring the fact that $y^{\langle t \rangle}$ is generally not $h^{\langle t \rangle}$, but rather constructed via the second equation in (7.2). Hence in our simplified presentation we focused on the essence and ignored less complicated details for the complete set of θ parameters.

Let us also consider the Jacobian $\frac{\partial h^{\langle t+1 \rangle}}{\partial h^{\langle t \rangle}}$ appearing in (7.6). Again, assuming that the vector activation function $S_h(\cdot)$ of (7.2) is composed of element wise scalar activation functions $\sigma(\cdot)$, this Jacobian can be represented as,

$$\frac{\partial h^{\langle t+1 \rangle}}{\partial h^{\langle t \rangle}} = W_{hh}^{\top} \operatorname{diag}\Big(\dot{\sigma}(W_{hh} h^{\langle t \rangle} + W_{hx} x^{\langle t \rangle} + b_h)\Big), \tag{7.11}$$

where the derivative of the activation function is denoted via $\dot{\sigma}(\cdot)$ and is applied element wise to the components of its input.

Computational Challenges

We discussed vanishing and exploding gradient phenomena in Section 5.4, where in equations (5.34) and (5.35) we saw how both the forward pass and the backwards pass involve actions of repeated matrix multiplication. More specifically, the backpropagation based equation (5.35) is based on a simplification of a feedforward neural network that ignores the effect of activation functions, ignores the bias, and assumes that each layer of the network has the same weight matrix. In such a case, it is evident that for deep networks (L large), vanishing or exploding gradient phenomena are likely to occur.

In recurrent neural networks, such phenomena are even more problematic than typical deep feedforward networks because the input size T (paralleling the depth of the feedforward network L), can be large. Unrolling (7.6) we get for $t = 1, \ldots, T-1$,

$$\zeta^{\langle t \rangle} = \frac{\partial h^{\langle t+1 \rangle}}{\partial h^{\langle t \rangle}} \frac{\partial h^{\langle t+2 \rangle}}{\partial h^{\langle t+1 \rangle}} \quad \cdots \quad \frac{\partial h^{\langle T-1 \rangle}}{\partial h^{\langle T-2 \rangle}} \frac{\partial h^{\langle T \rangle}}{\partial h^{\langle T-1 \rangle}} \zeta^{\langle T \rangle}.$$

Now using (7.11) and for simplicity ignoring the action of the activation function (treating it as an identity function), ignoring the input $x^{\langle t \rangle}$, and ignoring the bias term, we obtain,

$$\zeta^{\langle t \rangle} = \left(W_{hh}^{\top}\right)^{T-t} \zeta^{\langle T \rangle}. \tag{7.12}$$

This representation is similar to (5.35) of Chapter 5, and is even more realistic since in recurrent neural networks, the weight matrices of all unrolled layers are the same, whereas in the Chapter 5 analysis of feedforward networks fixing the weight matrix was a simplification.

Hence, in recurrent neural networks trained on inputs with large sizes T, it is very likely that during backpropagation, the adjoint values $\zeta^{\langle t \rangle}$ vanish or explode. This follows from the matrix power in (7.12), since in most situations, the maximal eigenvalue of W_{hh} is likely to not be at or near unity (see also discussion on the effect of eigenvalues on vanishing and exploding phenomena in Section 5.4).

Considering (7.12) and assuming $W_{hh}^{\top}$ has a maximal eigenvalue less than unity in absolute value, then if T is large, for small t, $\zeta^{\langle t \rangle} \approx 0$. One way to express this is to consider some $T_0 < T$ such for example if $T = 300$, set $T_0 = 250$, and then for $t < T_0$ assume $\zeta^{\langle t \rangle} = 0$. In this case the gradient computation (7.10) can be roughly represented as,

$$g_W \approx \frac{1}{T} \sum_{t=T_0}^{T} \sum_{\tau=T_0}^{t} \sum_{j=1}^{m} \frac{\partial h_j^{\langle \tau \rangle}}{\partial W} \zeta_j^{\langle \tau \rangle}. \tag{7.13}$$

Now considering the influence of the input via (7.13) and (7.9), we see that the gradient is only updated based on "near effects", and not based on "long-term effects" since inputs to the sequence $x^{\langle t \rangle}$ for $t < T_0$ do not play a role. For example in language modelling, the contribution of faraway words to predicting the next word at time-step diminishes when the gradient vanishes early on. As an example consider the text

```
Slava grew up in Ukraine before he moved around the world, first to the
United States, and then to Australia. He loves teaching languages and is an
avid teacher of his own mother tongue _.'
```

In this case, completion of the end of the text, marked via _, requires information from the start of the text. Models presented in the sections below, were also designed to overcome such difficulties.

Further, with recurrent neural networks, computation of the loss and of the gradients across an entire corpus is generally infeasible or too expensive. In practice, a batch of sentences is used to compute the loss to limit the sequence size T. Note also that in cases where $W_{hh}^{\top}$ has eigenvalues greater than unity in absolute value an exploding gradient phenomena is likely to occur. For this, *gradient clipping* may be employed as described at the end of Section 5.4. Another technique is to use *truncated backpropagation through time* (TBPTT) which limits the number of time steps the signal can backpropagate in each forward pass.

Other Aspects of Training

Some practices of training recurrent neural networks are very similar to training feedforward or convolutional networks. For example, one uses similar weight initialization techniques to those introduced in Section 5.5 in the context of feedforward networks. However, there are some differences as well. An important aspect to keep in mind is that unlike the supervised setting that prevailed with the models of Chapter 5 and Chapter 6, with recurrent models we are often able to train with *self-supervision*. Specifically, as already discussed in the example of Figure 7.5 we may use a shifted sequence $y = \big(x^{\langle 2 \rangle}, x^{\langle 3 \rangle}, \dots\big)$ as the desired output for the loss, and simply train the model for one step lookahead prediction.

Note however, that not all training is of the self-supervised form. In some cases, often arising in machine translation applications described in the sequel, we are naturally presented with an input sequence $x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \dots$ which may result from word embedding of one natural language

(e.g., English) and a corresponding output sequence $y^{\langle 1\rangle}, y^{\langle 2\rangle}, \ldots$, of one-hot encoded vectors, associated with another natural language (e.g., Arabic). Hence recurrent neural networks can be trained in a supervised setting as well.

In both the self-supervised and supervised settings, in cases where we use the formulation (7.3), where the output $\hat{y}^{\langle t-1\rangle}$ is fed into the input, we sometimes use a training technique called *teacher forcing*. The idea of teacher forcing is to use the actual (correct) one-hot encoded label $y^{\langle t-1\rangle}$ in place of the model generated (predicted) probability vector $\hat{y}^{\langle t-1\rangle}$ during training. That is, the recursion (7.3) has now inputs that are based on the actual labels instead of the predictions. Note that in this case, $\mathcal{T}(\cdot)$ in (7.3), can be viewed as also converting the probability vector into a word embedding, if needed. This technique accelerates training by removing the errors in the labels. We revisit the teacher forcing technique both at the end of Section 7.4 in the context of encoder-decoder models, and at the end of Section 7.5 in the context of transformers where it is extremely powerful due to parallelization.

7.3 Generalizations and Modifications to RNNs

The basic recurrent networks of Section 7.2, while powerful, still suffer from some drawbacks in terms of training, vanishing and exploding gradient, and expressivity. In this section we highlight a few generalizations and modifications to RNNs that enable more powerful models for sequence data. An underlying concept is the connection of gates in various creative ways that enable more expressive and robust models. The notion of a gate was already illustrated in Figure 7.4. In this section we see how such gates can be connected in diverse ways, as well as how the internals of the gate can be extended to yield more powerful models.

Stacking and Reversing Gates

Basic extensions to recurrent neural networks are possible by interconnecting gates in more complicated forms than just a forward direction of data flow. In particular, common approaches are to either stack the gates to form deeper networks, reverse the gates, or combine the two approaches. See Figure 7.7 for a schematic representation of such interconnections of gates.

Let us first consider reversing of gates as in Figure 7.7 (a) to create a *bidirectional recurrent neural network*. For such a modification we extended the RNN evolution equation (7.2) to,

$$\begin{cases} h_f^{\langle t\rangle} = S_h(W_{hh}^f h_f^{\langle t-1\rangle} + W_{hx}^f x^{\langle t\rangle} + b_h^f) \\ h_r^{\langle t\rangle} = S_h(W_{hh}^r h_r^{\langle t+1\rangle} + W_{hx}^r x^{\langle t\rangle} + b_h^r) \\ \hat{y}^{\langle t\rangle} = S_y(W_{yh}^f h_f^{\langle t\rangle} + W_{yh}^r h_r^{\langle t\rangle} + b_y), \end{cases} \tag{7.14}$$

where now for every time t there are two cell states, $h_f^{\langle t\rangle}$ and $h_r^{\langle t\rangle}$, representing the forward direction and reverse direction respectively. Observe in (7.14) that $h_f^{\langle t\rangle}$ evolves based on the input $x^{\langle t\rangle}$ and $h_f^{\langle t-1\rangle}$, while $h_r^{\langle t\rangle}$ evolves based on the input $x^{\langle t\rangle}$ and $h_f^{\langle t+1\rangle}$. Naturally with such an extension there are more trained parameters, superscripted via f and r respectively in (7.14).

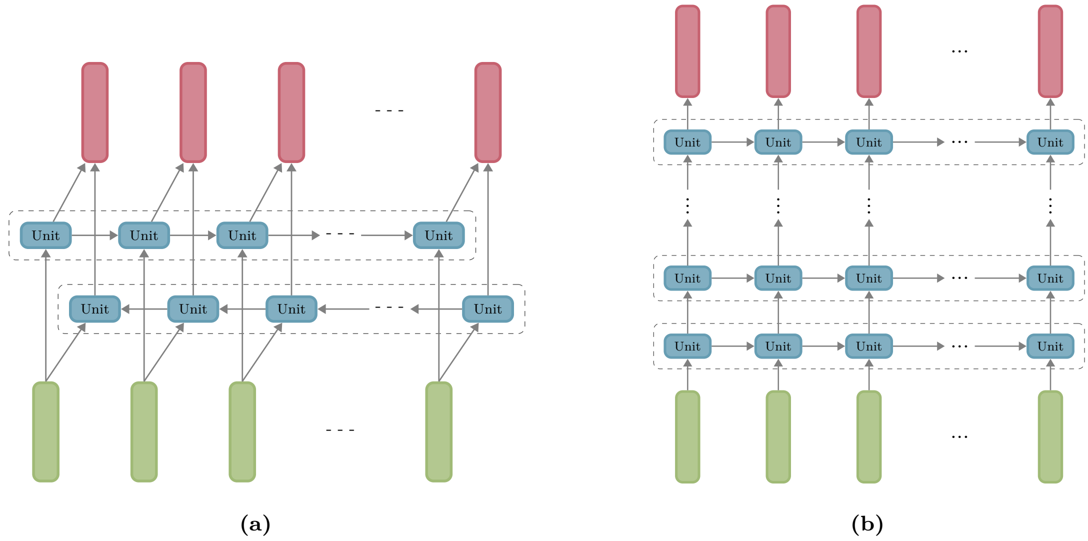


Figure 7.7: Alternative configurations and extensions of recurrent neural networks. (a) Stacked RNN. (b) Bidirectional RNN.

As is evident from (7.14), the forward sequence of cell states $h_f^{\langle 1\rangle}, h_f^{\langle 2\rangle}, \ldots, h_f^{\langle T\rangle}$ and the reverse sequence of cell states $h_r^{\langle T\rangle}, h_r^{\langle t-1\rangle}, \ldots, h_r^{\langle 1\rangle}$, evolve without interaction. Once computed, these sequences are then combined to obtain the output sequence. Such bidirectional data flow enables the model to be more versatile, especially for cases where the entire input sequence is available. This is the setup in applications such as handwritten text recognition, machine translation, speech recognition, and part-of-speech tagging, among others.

Let us now consider stacking of gates as in Figure 7.7 (b) to create a deeper model, also sometimes known as a *stacked recurrent neural network*. With this paradigm we extend the evolution equations (7.2) to,

$$\begin{cases} h_{[1]}^{\langle t\rangle} = S_h^{[1]}(W_{hh}^{[1]} h_{[1]}^{\langle t-1\rangle} + W_{hx}^{[1]} x^{\langle t\rangle} + b_h^{[1]}) \\ h_{[2]}^{\langle t\rangle} = S_h^{[2]}(W_{hh}^{[2]} h_{[2]}^{\langle t-1\rangle} + W_{hx}^{[2]} h_{[1]}^{\langle t\rangle} + b_h^{[2]}) \\ \quad \vdots \\ h_{[L]}^{\langle t\rangle} = S_h^{[L]}(W_{hh}^{[L]} h_{[L]}^{\langle t-1\rangle} + W_{hx}^{[L]} h_{[L-1]}^{\langle t\rangle} + b_h^{[L]}) \\ \hat{y}^{\langle t\rangle} = S_y(W_{yh} h_{[L]}^{\langle t\rangle} + b_y), \end{cases} \tag{7.15}$$

where we now use notation such as $[1], \ldots, [L]$ to signify the depth of individual components and L is the number of stacked layers, similarly to the notation of Chapter 5. Observe that the cell state at time t and depth ℓ, denoted via $h_{[\ell]}^{\langle t\rangle}$ is computed based on the cell state at depth $\ell - 1$ and the same time t using the matrix $W_{hx}^{[\ell]}$ (where the notation x here in the subscript implies the previous level). It is also computed using the cell state at the same depth, ℓ, and the previous time, $t - 1$ using the matrix $W_{hh}^{[\ell]}$.

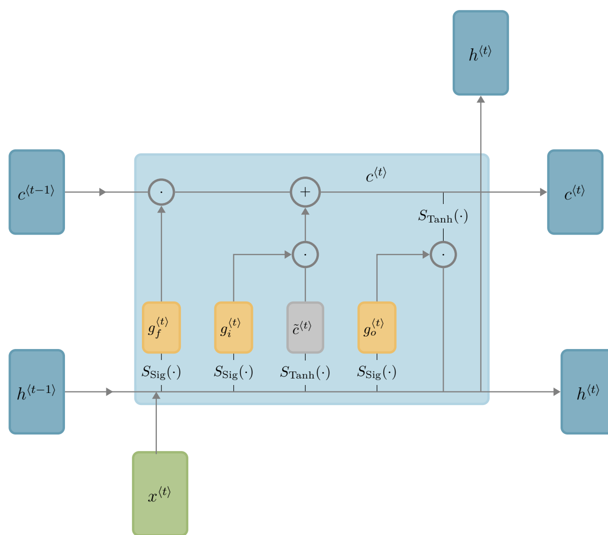
Such stacked RNN models are clearly more expressive and thus they generally outperform single-layer recurrent neural networks when trained with enough data. However, they are harder to train as the number of parameters clearly grows proportionally to the number of layers. We also mention that combinations of stacking and reversing are also possible.

Long Short Term Memory Models

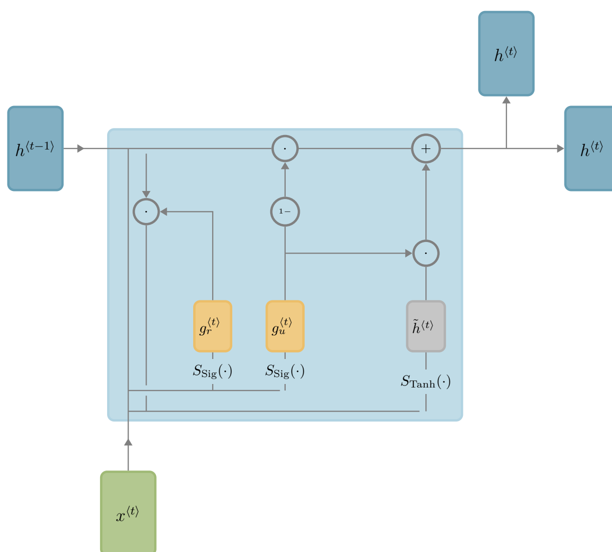
Long short term memory (LSTM) models are generalizations of basic recurrent neural networks that are designed to preserve information over many time steps. To understand the idea behind LSTM, it is constructive to think in terms of logical operations that are approximated via multiplication of vectors. In particular, as we see below, different components of LSTM interplay in a way that can heuristically be described as computation of a logical circuit. More specifically, some of the neurons inside LSTM can be called *internal gates* and are represented as values in the range $[0, 1]$ and these are then multiplied by other neurons with arbitrary real values. In particular, when a vector of neurons in such an internal gate, say g, has elements in the range $[0, 1]$, and another vector of neurons, say c has general real values, then the element wise multiplication $g \odot c$ can be viewed as a restriction which approximately zeros out (forgets) entries of c when the corresponding entry of g is near 0. We informally say that the entry of the internal gate is "open" when it is approximately at 1 and similarly "closed" when it is approximately 0. Using internal gates for this type of "approximate logical masking" is common in these models as well as the gated recurrent units described in the sequel.

A key concept in LSTM is to extend the hidden units of RNNs by separating the information flow between units into two groups where one group is called the *cell state* and denoted $c^{\langle t \rangle}$, while the other group is called the *hidden state* and denoted $h^{\langle t \rangle}$. The model is designed so that long term dependancies are generally retained through $c^{\langle t \rangle}$ while short term dependencies are carried by $h^{\langle t \rangle}$. The interaction between these groups of neurons is enabled via additional groups of neurons, namely the internal gates, which are generally vectors with entries in the range $[0, 1]$, denoted via $g_f^{\langle t \rangle}$, $g_i^{\langle t \rangle}$, and $g_o^{\langle t \rangle}$. An additional internal group of neurons, denoted via $\tilde{c}^{\langle t \rangle}$, is sometimes called the *internal cell state*.

For the basic recurrent neural network models of Section 7.2, we used m for the number of neurons and this is also the dimension of information flow between successive units. However, for LSTM, only some of the neurons are used for information flow between units, namely $c^{\langle t \rangle}$ and $h^{\langle t \rangle}$. In terms of dimension, we retain m as the number of neurons, and assume that $m = 6\,\tilde{m}$ where the dimensions of all vectors $c^{\langle t \rangle}$, $h^{\langle t \rangle}$, $\tilde{c}^{\langle t \rangle}$, $g_f^{\langle t \rangle}$, $g_i^{\langle t \rangle}$, and $g_o^{\langle t \rangle}$ is $\tilde{m}$.



(a)



(b)

Figure 7.8: Representation of the LSTM and the GRU units. Internal gates are represented in yellow and internal states are in gray. The output $\hat{y}^{\langle t \rangle}$ is not presented. (a) In LSTM there are three internal gates and the internal state is called the internal cell state. (b) In GRU there are two internal gates and the internal state is called the internal hidden state.

A basic LSTM unit is illustrated in Figure 7.8 (a) which summarizes the evolution associated with this unit. Like the simpler RNN counterpart in Figure 7.4, and equations (7.2), the

evolution equations of LSTM describe how the pair $(c^{\langle t \rangle}, h^{\langle t \rangle})$ evolves as a function of the previous pair $(c^{\langle t-1 \rangle}, h^{\langle t-1 \rangle})$ and the input $x^{\langle t \rangle}$. Further, the output $\hat{y}^{\langle t \rangle}$ evolves based on $(c^{\langle t \rangle}, h^{\langle t \rangle})$, directly via $h^{\langle t \rangle}$ and indirectly based on $c^{\langle t \rangle}$. Unlike the RNN (7.2), the LSTM evolution is more complex since it also involves the internal gates and neurons.

The LSTM evolution equations are,

$$\begin{cases} c^{\langle t \rangle} = g_f \odot c^{\langle t-1 \rangle} \ + \ g_i \odot \underbrace{S_{\text{Tanh}}\Big(W_{\tilde{c}h}\, h^{\langle t-1 \rangle} + W_{\tilde{c}x}\, x^{\langle t \rangle} + b_{\tilde{c}}\Big)}_{\tilde{c}^{\langle t \rangle}} & \text{(cell state)} \\ h^{\langle t \rangle} = g_o \odot S_{\text{Tanh}}(c^{\langle t \rangle}) & \text{(hidden state)} \\ \hat{y}^{\langle t \rangle} = S_y(W_{yh} h^{\langle t \rangle} + b_y), \end{cases} \tag{7.16}$$

where for clarity we omit the time superscripts from the internal gates and denote them via g_f, g_i, and g_o. Importantly, at every time t these internal gates are computed as,

$$\begin{cases} g_f = S_{\text{Sig}}\big(W_{fh} h^{\langle t-1 \rangle} + W_{fx} x^{\langle t \rangle} + b_f\big) & \text{(forget gate)} \\ g_i = S_{\text{Sig}}\big(W_{ih} h^{\langle t-1 \rangle} + W_{ix} x^{\langle t \rangle} + b_i\big) & \text{(input gate)} \\ g_o = S_{\text{Sig}}\big(W_{oh} h^{\langle t-1 \rangle)} + W_{ox} x^{\langle t \rangle} + b_o\big). & \text{(output gate)} \end{cases} \tag{7.17}$$

Note that to restrict the value of internal gates to the range $[0, 1]$, sigmoid activation functions are typically used and we denote the associated vector activation function via $S_{\text{Sig}}(\cdot)$. The hidden state, the cell state, and the internal cell state information is not generally restricted to $[0, 1]$ and a typical activation function is tanh where we denote the associated vector activation function via $S_{\text{Tanh}}(\cdot)$.

As evident from (7.16) and (7.17), for an LSTM with input of dimension p and output of dimension q, the trained LSTM parameters include the following. First there are four $\tilde{m} \times \tilde{m}$ weight matrices $W_{\tilde{c}h}$, W_{fh}, W_{ih}, and W_{oh}. Further there are the four $\tilde{m} \times p$ weight matrices $W_{\tilde{c}x}$, W_{fx}, W_{ix}, and W_{ox}. In addition there is the $q \times \tilde{m}$ weight matrix W_{yh}, as well as the five associated bias vectors.

The specific structure of an LSTM unit interconnects the internal gates in a way that enables using both long term and short term memory, captured in $c^{\langle t \rangle}$ and $h^{\langle t \rangle}$ respectively. Specifically, the internal gates help select which information is “forgotten”, “used as input”, or “used as output”. At each time step t the entries in the internal gate vectors $g_f^{\langle t \rangle}$, $g_i^{\langle t \rangle}$, and $g_o^{\langle t \rangle}$ can be “open”, “closed”, or somewhere in-between where entries that are near 1 are considered open and entries that are near 0 are considered closed. The forget gate $g_f^{\langle t \rangle}$ is multiplied element wise with the previous cell state $c^{\langle t-1 \rangle}$ to “forget” information from the previous cell state or not, depending on being closed or open respectively. Similarly, the input gate, $g_i^{\langle t \rangle}$ controls what parts of the new cell content are written to the cell and this is applied to the internal cell state $\tilde{c}^{\langle t \rangle}$ which models the “selected information” based on the current input and the previous short term memory. Finally the output gate, $g_o^{\langle t \rangle}$, controls what parts of the cell are written to the hidden state $h^{\langle t \rangle}$ which is then used both for output $\hat{y}^{\langle t \rangle}$ and the short term memory passed onto the next unit.

It is interesting to consider the magnitudes of the LSTM elements, specifically in the first equation of (7.16). At time t, the previous cell state $c^{\langle t-1 \rangle}$ may have entries with general

values (not limited to $[-1,1]$). These values may then be "forgotten" if multiplied by $g_f^{\langle t\rangle}$ in cases where it is approximately at 0. Further, new long term memory is accumulated when $g_i^{\langle t\rangle}$ is approximately at 1. Observe that since the tanh activation function's range is $[-1,1]$, the accumulation of this new memory is limited at every time step. Specifically, based on the internal cell state, $\tilde{c}^{\langle t\rangle}$, the memory may increase or decrease by at most 1 per time step.

The interconnection of LSTM units follows the same principles as the interconnection of recurrent neural network units outlined above. Specifically, one may view an unrolled representation of LSTM in the same manner as an unrolled representation of basic recurrent neural networks, presented in Figure 7.3. The difference is that both $c^{\langle t\rangle}$ and $h^{\langle t\rangle}$ are passed between time t and time $t+1$, and not just $h^{\langle t\rangle}$ as in basic RNN. With this, the same extensions that one may consider for basic RNNs can be applied to LSTM. Specifically, LSTM can be reversed as in Figure 7.7 (a) or stacked into a deeper architecture as in Figure 7.7 (b). In reversing LSTM, the reverse direction LSTM passes $(c^{\langle t+1\rangle}, h^{\langle t+1\rangle})$ into the unit computing $(c^{\langle t\rangle}, h^{\langle t\rangle})$. In stacking LSTMs, the hidden state $h_{[\ell]}^{\langle t\rangle}$ of layer ℓ is passed as an input (similar to $x^{\langle t\rangle}$) for the unit above at layer $\ell+1$ but not the cell state. Note that in stacked LSTM we may view $c_{[1]}^{\langle t\rangle},\ldots,c_{[L]}^{\langle t\rangle}$ as a representation of long term memory across all layers at step t. This long term memory is passed to the next step, $t+1$.

Gated Recurrent Unit Models

An alternative to the LSTM architecture is the *gated recurrent unit* (GRU) architecture, with a unit illustrated in Figure 7.8 (b). While LSTMs make an explicit separation of neurons to be long term or short term, with GRUs we return to a somewhat simpler architecture with only one key set of neurons $h^{\langle t\rangle}$, again called the *hidden state*. Gated recurrent units store both long-term dependencies and short-term memory in the single hidden state. Like LSTMs, gated recurrent units use internal gates with values in the range $[0,1]$, this time called the *reset gate* $g_r^{\langle t\rangle}$ and the *update gate* $g_u^{\langle t\rangle}$. Similarly to LSTMs that maintain an internal cell state, GRUs maintain an *internal hidden state* $\tilde{h}^{\langle t\rangle}$. Setting $\tilde{m}$ as the number of neurons in each of these groups, the total number of neurons in a GRU is $m=4\tilde{m}$. Hence with 4 components instead of 6 components, gated recurrent units provide a simpler architecture in comparison to LSTM as there are only two internal gates (in comparison to three) and a single group of states passed between time units (in comparison to two).

The basic evolution equation for gated recurrent units is,

$$\begin{cases} h^{\langle t\rangle} = (1-g_u)\odot h^{\langle t-1\rangle} \; + \; g_u \odot \underbrace{S_{\text{Tanh}}\Big(W_{\tilde{h}h}\,(g_r\odot h^{\langle t-1\rangle}) + W_{\tilde{h}x}\,x^{\langle t\rangle} + b_{\tilde{h}}\Big)}_{\tilde{h}^{\langle t\rangle}} \\ \hat{y}^{\langle t\rangle} = S_y(W_{yh}h^{\langle t\rangle}+b_y), \end{cases} \tag{7.18}$$

where for clarity we omit the time superscripts from the internal gates and denote them via g_r and g_u. At every time t, these internal gates are computed as,

$$\begin{cases} g_r = S_{\text{Sig}}\big(W_{rh}h^{\langle t-1\rangle} + W_{rx}x^{\langle t\rangle} + b_r\big) & \text{(reset gate)} \\ g_u = S_{\text{Sig}}\big(W_{uh}h^{\langle t-1\rangle} + W_{ux}x^{\langle t\rangle} + b_u\big). & \text{(update gate)} \end{cases} \tag{7.19}$$

A key attribute of the first equation of (7.18) is that new entries of the cell state $h^{\langle t\rangle}$ are computed as a convex combination of the entries of the previous cell state $h^{\langle t-1\rangle}$ and the

hidden cell state $\tilde{h}^{\langle t \rangle}$. This convex combination is determined by the entries of $g_u^{\langle t \rangle}$ where an entry near 1 implies "update" of the cell state based on the internal cell state, and an entry near 0 implies retaining the previous value (not updating).

As evident from (7.18) and (7.19), for a GRU with input of dimension p and output of dimension q, the trained parameters include the following. First there are three $\tilde{m} \times \tilde{m}$ weight matrices $W_{\tilde{h}h}$, W_{rh}, and W_{uh}. Further there are the three $\tilde{m} \times p$ weight matrices $W_{\tilde{h}x}$, W_{rx}, and W_{ux}. In addition there is the $q \times \tilde{m}$ weight matrix W_{yh}, as well as the four associated bias vectors. Again as evident, the number of parameter groups is smaller than that of LSTM.

To gain some intuition about the GRU architecture we may observe that the update gate g_u plays a role similar to both the forget gate, g_f, and input gate, g_i in LSTM. Specifically compare the first equation in (7.18) with the first equation in (7.16). The simplification offered by GRU is to use a convex combination $(1 - g_u, g_u)$ instead of a general linear combination (g_f, g_i) as in LSTM. In both architectures this operation controls what parts of long term memory information are updated versus preserved. One may also observe that GRU's internal hidden state $\tilde{h}^{\langle t \rangle}$ is updated via a slightly more complex mechanism than LSTM's internal cell state $\tilde{c}^{\langle t \rangle}$. The innovation in GRUs is to use the reset gate, g_r. Practice has shown that with such an architecture, GRUs are able to maintain both long term and short term memory inside the hidden state sequence, $h^{\langle 1 \rangle}, h^{\langle 2 \rangle}, \ldots$.

Note that the interconnection of GRUs can follow the exact same lines as other recurrent neural network architectures. Again, bi-directional connections as well as stacked configurations are possible; see Figure 7.7.

7.4 Encoders Decoders and the Attention Mechanism

One of the great application successes of sequence models is in the domain of *machine translation* tasks, namely the translation of one human language to another. For this, a general paradigm involving an *encoder* neural network and a *decoder* neural network is common. Other applications of encoders and decoders include, *image to text* models and *text to image* models. Yet, the main motivation we consider here is machine translation, since this application was the main driver in the development of *encoder-decoder architectures* within sequence models.

An important machine learning concept that has advances machine translation and other tasks, is the *attention mechanism*. This idea is incorporated in *transformer models* that currently drive state of the art *large language models*. Transformers are the topic of the next section and in this current section, we first introduce general ideas of encoder-decoder architectures with the motivation of machine translation. We then formally define the attention mechanism. Finally, we see an encoder-decoder architecture that incorporates the attention mechanism at the interface of the encoder and the decoder.

Encoder-Decoder Architectures for Machine Translation

Recall from Section 7.1 that in general, when considering natural language, the input text is converted into a sequence of word embeddings denoted $x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots$. With such a sequence, at some point, an embedding of a word or token such as `<stop>` appears and marks the end

of the text. In a machine translation application, our goal is to convert this input sequence to an output sequence $\hat{y}^{\langle 1 \rangle}, \hat{y}^{\langle 2 \rangle}, \ldots$, also containing a `<stop>` token representation at its end. Clearly the input is in one natural language, e.g., French, and the output is an another natural language, e.g., Telugu.

Machine translation handled via an encoder-decoder architecture, uses a setup similar to Figure 7.9 (a). First, an encoder model which is a recurrent neural network, or a variant such as LSTM, or GRU, is used to convert the input sequence into the latent space by creating a *context vector* also known as the *code*, denoted via $z^\star$. Ideally this code encompasses the meaning and style of the input text. Then, a second sequence model, known as the decoder, takes the code $z^\star$ as input and converts it to the output sentence. Clearly the encoder model in this setup is configured as a many to one model, while the decoder model is configured as a one to many model. In the decoder, the output at each time fed into the input for the next time as in (7.3). Note that the code $z^\star$ is a vector of fixed dimension. Further the dimension p of each $x^{\langle t \rangle}$, the dimension q of each $\hat{y}^{\langle t \rangle}$ typically differ and each typically has their own encoding. The input and output sequences are of arbitrary length, where the length of the input sequence and the length of the output sequence may differ.

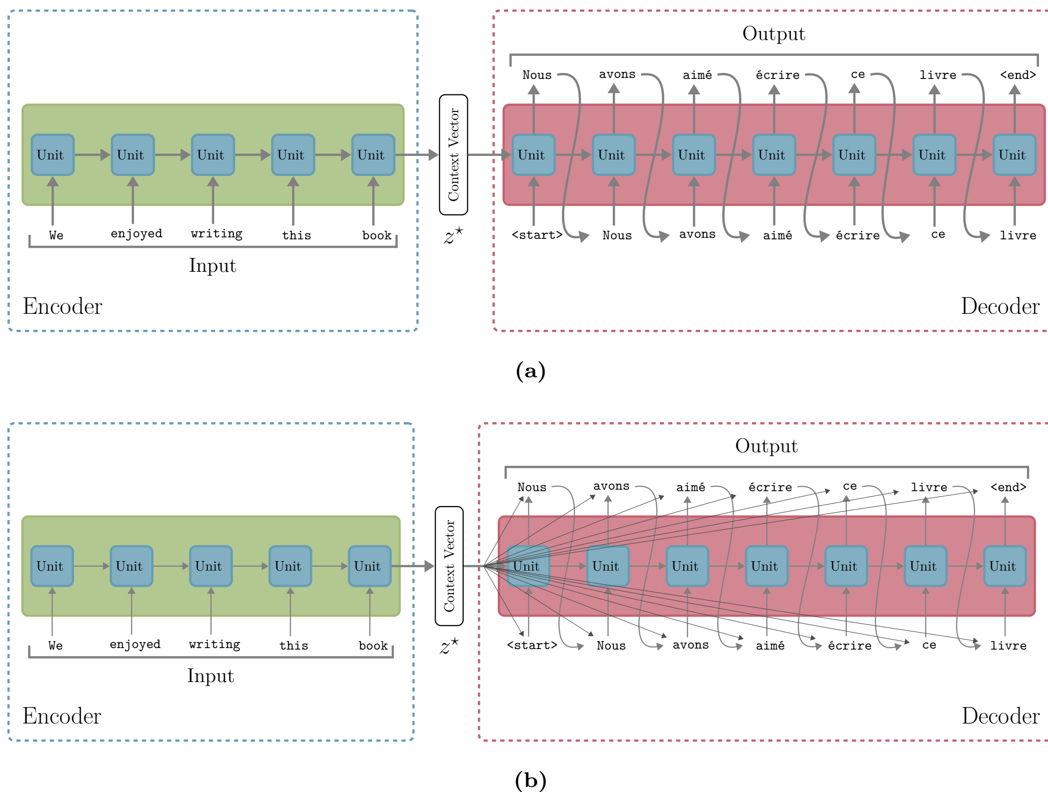


Figure 7.9: Unrolling of basic encoder-decoder architectures for machine translation. (a) A basic architecture where the encoder output context vector $z^\star$ is computed and fed as the initial state to the decoder. (b) A more advanced architecture where $z^\star$ is also presented at the input and output at each time step of the decoder.

This basic type of encoder-decoder architecture, as in Figure 7.9 (a), has already proven quite useful for early attempts of machine translation using deep sequence models. The choice between basic RNN, LSTM, GRU, or stacked combinations of one of these types of units is a modelling choice that one can make. No matter what type of unit is used, a key weakness is that the impact of the code $z^\star$ on the output $\hat{y}^{\langle 1 \rangle}, \hat{y}^{\langle 2 \rangle}, \ldots,$ decreases as t grows within the predicted output. Nevertheless, this architecture is a starting point for more advanced architectures.

A natural improvement is to make the context vector accessible for all steps in the decoder. With such a setup, at each time the decoder is fed the concatenation of the previous output and the code vector $z^\star$. A second improvement is to also present the code vector $z^\star$ for the computation of the output,[8] where at this point the output computation is based on a concatenation of the cell state, $h^{\langle t \rangle}$, and the code $z^\star$. This architecture is depicted in Figure 7.9 (b).

In the context of machine translation it is often useful to modify the encoder-decoder pair such that the encoder accepts the *text in reverse order*. As an example, assume that the input text is,

```
I am going to read another chapter.
```

Then with the text in reverse order paradigm, this input is fed to the encoder as,

```
chapter another read to going am I.
```

The training process then uses reverse order inputs as above, yet outputs are expected in the normal order. Clearly when the model is used operationally, the input text is also reversed.

The benefit of this approach is in keeping inputs and their respective outputs closer on average. For example, assume that we are translating from English to French where the output should be (the non-reversed French text),

```
je vais lire un autre chapitre.
```

Note that with this approach the (reversed) input phrase `going am I` is near the output phrase `je vais` (which means "I am going"), and similarly with other pairs. Whereas if the text was not reversed, then generally (assuming inputs and outputs of the same length) the distance between an input and the respective output is in the order of the length of the text.

Even when employing techniques such as reversal of the text, a key drawback of all of the above encoder-decoder architectures is that performance degrades rapidly as the length of the input sentence increases. This is because the encoded vector needs to capture the entire input text, and in doing so, it might skip many important details. The attention mechanism that we describe below, and its application in machine translation architectures, overcome many of these difficulties.

[8] Having the code available to the output is in a sense a residual connection similar to the ResNets discussed in Section 6.5. It is particularly useful if a stacked architecture is used in the decoder.

The Attention Mechanism

As we saw above, one of the key considerations in encoder-decoder architectures has to do with the way in which the encoder output enters as input to the decoder. Towards this, we now introduce a general paradigm called the *attention mechanism*. One may view the attention mechanism as a method for "annotating" elements of the input or intermediate sequences, which require more focus, or attention, than others. We first define attention mathematically, and later we see how it can interplay within an encoder-decoder architecture. The attention mechanism defined here is also central to transformer models which encompass Section 7.5 below, and are used for most contemporary large language models.

In general, an *attention mechanism* can be viewed as a transformation of a sequence of vectors to a sequence of vectors. The input sequence has T_{in} vectors and the output sequence has T_{out} vectors. We assume the vectors are m dimensional and denote the input sequence via $v^{\langle 1\rangle}, v^{\langle 2\rangle}, \ldots, v^{\langle T_{\text{in}}\rangle}$ and the output sequence via $u^{\langle 1\rangle}, u^{\langle 2\rangle}, \ldots, u^{\langle T_{\text{out}}\rangle}$.

As an aid to the attention mechanism, we also have two sequences of vectors, which we call the *proxy vectors*. These are denoted via $z_q^{\langle 1\rangle}, \ldots, z_q^{\langle T_{\text{out}}\rangle}$, and $z_k^{\langle 1\rangle}, \ldots, z_k^{\langle T_{\text{in}}\rangle}$, where the dimension of each vector in the first sequence is m_q, and similarly m_k for the second sequence. The notation using subscripts q and k stems from *query* and *key* respectively. These terms, query and key, are more common in the application of transformer models in the next section.

One of the components of the attention mechanism is a *score function*, also known as an *alignment function*, $s : \mathbb{R}^{m_q} \times \mathbb{R}^{m_k} \to \mathbb{R}$ which when applied to a pair of proxy vectors, z_q and z_k and denoted via $s(z_q, z_k)$, measures the similarity between the two proxy vectors. A typical simple score function, suitable when $m_q = m_k$, is the inner product. Yet other possibilities, also potentially with learned parameters, can be employed, and in some instances this component is known as an *alignment model*. It is typical to have normalization as part of the score function with a factor such as $\sqrt{\max(m_q, m_k)}$. This normalization maintains score values at a reasonable range for numerical stability.

At the heart of the attention mechanism, for any time $t = 1, \ldots, T_{\text{out}}$, we apply the score function on a fixed $z_q^{\langle t\rangle}$ against all $z_k^{\langle 1\rangle}, \ldots, z_k^{\langle T_{\text{in}}\rangle}$ and then using softmax we obtain a vector of *attention weights*, also known as *alignment scores*. This vector, denoted $\alpha^{\langle t\rangle}$, is of length T_{in}, and is computed via

$$\alpha^{\langle t\rangle} = S_{\text{softmax}}\left(\begin{bmatrix} s(z_q^{\langle t\rangle}, z_k^{\langle 1\rangle}) \\ s(z_q^{\langle t\rangle}, z_k^{\langle 2\rangle}) \\ \vdots \\ s(z_q^{\langle t\rangle}, z_k^{\langle T_{\text{in}}\rangle}) \end{bmatrix}\right), \qquad \text{for} \qquad t = 1, \ldots, T_{\text{out}}. \tag{7.20}$$

Here the vector to vector softmax function, $S_{\text{softmax}}(\cdot)$ is as defined in (3.25). The attention weights associated with time t, can also be written as $\alpha^{\langle t\rangle} = \big(\alpha_1^{\langle t\rangle}, \ldots, \alpha_{T_{\text{in}}}^{\langle t\rangle}\big)$. Thus in general, for any $t \in \{1, \ldots, T_{\text{out}}\}$, the attention weight vector $\alpha^{\langle t\rangle}$ captures similarity between the proxy vector $z_q^{\langle t\rangle}$ and all of the proxy vectors $z_k^{\langle 1\rangle}, \ldots, z_k^{\langle T_{\text{in}}\rangle}$.

With attention weights available, the attention mechanism operates on the input sequence $v^{\langle 1\rangle}, \ldots, v^{\langle T_{\text{in}}\rangle}$. The mechanism produces an output sequence $u^{\langle 1\rangle}, \ldots, u^{\langle T_{\text{out}}\rangle}$ where each

$u^{\langle t \rangle}$ is computed via the linear combination,

$$\begin{cases} u^{\langle t \rangle} = \sum_{\tau=1}^{T_{\text{in}}} \alpha_\tau^{\langle t \rangle} v^{\langle \tau \rangle} & \text{(Non-causal attention)} \\ \qquad\qquad \text{or} \\ u^{\langle t \rangle} = \sum_{\tau=1}^{t} \alpha_\tau^{\langle t \rangle} v^{\langle \tau \rangle}. & \text{(Causal attention)} \end{cases} \tag{7.21}$$

Note that in the causal form the output at time t, $u^{\langle t \rangle}$, only depends on the inputs up to time t, while in the non-causal form $u^{\langle t \rangle}$ depends on inputs at all times $t \in \{1, \ldots, T_{\text{in}}\}$. Also note that the causal form is only possible when $T_{\text{out}} \le T_{\text{in}}$ (this is the case in the next section where in particular we use $T_{\text{out}} = T_{\text{in}}$).

As we see below, this general mechanism is applied in various sequence model architectures where in each case, the proxy vector sequences $z_q^{\langle 1 \rangle}, \ldots, z_q^{\langle T_{\text{out}} \rangle}$, and $z_k^{\langle 1 \rangle}, \ldots, z_k^{\langle T_{\text{in}} \rangle}$, and the score function can be defined differently. Note that the attention mechanism can also be employed for *graph neural networks* (GNN); see Section 8.5.

Encoder-Decoder with an Attention Mechanism

As a first application of the attention mechanism let us see an encoder-decoder framework. This architecture is described in Figure 7.10 where an attention mechanism is used to tie the encoder output with the decoder. A key attribute of the attention mechanism is to provide more importance to some of the input words, in comparison to others, during machine translation. There are two main ideas in this architecture. The first idea is to use a bi-directional encoder, and importantly the second idea is to incorporate an attention mechanism.

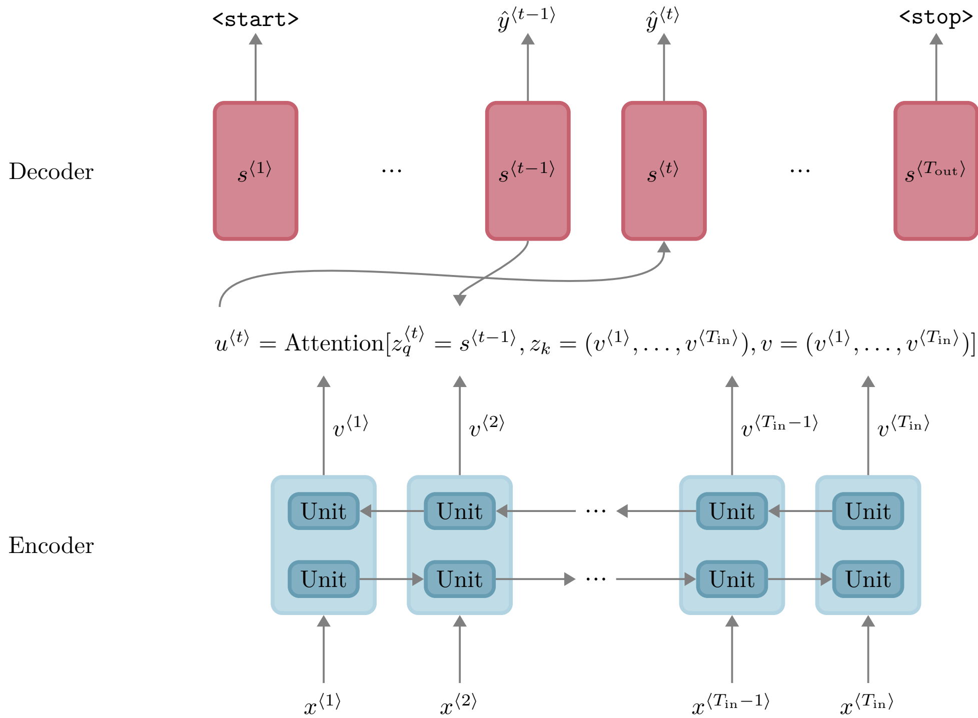


Figure 7.10: An encoder-decoder model with a bi-directional encoder and tying of the encoder and the decoder via an attention mechanism. The output sequence of the encoder is used as input to the attention mechanism. In the attention mechanism the previous decoder state is used as a query and the encoder outputs are used as keys.

The encoder is constructed via a bidirectional recurrent neural network, similar to Figure 7.7 (a) and the first two recursions in (7.14). Specifically for an input $x^{\langle 1\rangle}, \ldots, x^{\langle T_{\text{in}}\rangle}$, we obtain the sequences of encoder hidden states $h_f^{\langle 1\rangle}, \ldots, h_f^{\langle T_{\text{in}}\rangle}$ and $h_r^{\langle 1\rangle}, \ldots, h_r^{\langle T_{\text{in}}\rangle}$, resulting from the forward directional and reverse directional recursions, respectively. Note that LSTM or GRU alternatives can be used in place of these forward and reverse recursions as well.

Now the output of the encoder is taken as a concatenation of the forward and reverse direction. Specifically, we denote this encoder output via $v^{\langle 1\rangle}, \ldots, v^{\langle T_{\text{in}}\rangle}$ where $v^{\langle t\rangle}$ is a concatenation of $h_f^{\langle t\rangle}$ and $h_r^{\langle t\rangle}$. Note that the encoder output is a sequence of vectors of length T_{in}, namely the same length of the input sequence to the whole architecture.

The decoder is a variant of a recurrent neural network with hidden state $s^{\langle t\rangle}$, following the recursion,

$$\begin{cases} s^{\langle t\rangle} = f_{\text{decoder}}\big(s^{\langle t-1\rangle}, \hat{y}^{\langle t-1\rangle}, u^{\langle t\rangle}\big) \\ \hat{y}^{\langle t\rangle} = f_{\text{decoder-out}}\big(s^{\langle t\rangle}\big), \end{cases} \tag{7.22}$$

where $u^{\langle t\rangle}$ marks the input to the decoder and is computed via an attention mechanism as we describe below. The other two inputs are $s^{\langle t-1\rangle}$, the hidden decoder state at the previous time step, and $\hat{y}^{\langle t-1\rangle}$, the output of the decoder at the previous time step.

To see how the attention mechanism is used for computing $u^{\langle t\rangle}$, observe our notation where the encoder output is $v^{\langle t\rangle}$ and the decoder input at time t is $u^{\langle t\rangle}$. This notation agrees with the attention mechanism described above and in this architecture, an attention mechanism tying the encoder and the decoder, converts $v^{\langle t\rangle}$ to $u^{\langle t\rangle}$. Specifically, non-causal attention as in (7.21) is used. For the attention computation, the proxy vectors determining the attention weights via (7.20) are set as $z_q^{\langle t\rangle} = s^{\langle t-1\rangle}$ and $z_k^{\langle t\rangle} = v^{\langle t\rangle}$.

Observe that when the next decoder output token, $\hat{y}^{\langle t\rangle}$ is created using (7.22), it is based on the decoder state $s^{\langle t\rangle}$ which is based on the previous decoder state, on the previous decoder output, and importantly, on the attention output $u^{\langle t\rangle}$. This attention output is a linear combination of all of the previous decoder outputs, weighted by the attention weights, $\alpha^{\langle t\rangle}$ of time t.

The strength of this attention based architecture is that any input token can receive attention during the construction of the output sequence, even if the construction is at a location in the sequence far away from the input token. The application of a bi-directional architecture for the encoder enables the model to capture earlier and later information which help to disambiguate the input embedded word. The application of an attention mechanism introduces a form of a *dynamic context vector* between the encoder and decoder, in place of the static context vector $z^\star$ used in the simpler architectures above. Namely, architectures as depicted in Figure 7.9 have a fixed $z^\star$ which does not change during the operation of the decoder. The attention based approach replaces this fixed $z^\star$ with the sequence $u^{\langle 1\rangle}, u^{\langle 2\rangle}, \ldots$, which itself depends on the decoder state (through the proxy $z_q^{\langle t\rangle} = s^{\langle t-1\rangle}$).

An Illustration of Attention Weights

Let us see parts of an illustrative toy example of English to French translation using the encoder-decoder with attention architecture as in Figure 7.10. Assume the following:

Input: `we love deep learning <stop>`

Output: `nous aimons l' apprentissage en profondeur <stop>`

Observe that in this case, we explicitly use the `<stop>` token, and assume that with our tokenization and word embedding setup, each word or the `<stop>` token is a single vector. Also note that `l'` is considered a word. In this case we have $T_{\text{in}} = 5$ and as resulting from the model, $T_{\text{out}} = 7$.

When the input is processed via the architecture of Figure 7.10, first the encoder creates the sequences $h_f^{\langle 1\rangle}, \ldots, h_f^{\langle T_{\text{in}}\rangle}$ and $h_r^{\langle 1\rangle}, \ldots, h_r^{\langle T_{\text{in}}\rangle}$. Then, a sequence where each element is a concatenation of $h_f^{\langle t\rangle}$ and $h_r^{\langle t\rangle}$ is fed into the attention mechanism. The attention and the decoder operation run together, where with each additional time step of the output, (7.22) is applied, and the attention computation of (7.21) takes place. Importantly, with each t, the attention weights $\alpha^{\langle t\rangle}$ are computed via (7.20) where the proxy vector $z_q^{\langle t\rangle}$ is taken as the previous decoder state, and the proxy sequence $z_k^{\langle 1\rangle}, \ldots, z_k^{\langle T_{\text{in}}\rangle}$ is taken as the concatenated output of the encoder, summarizing all of the input text.

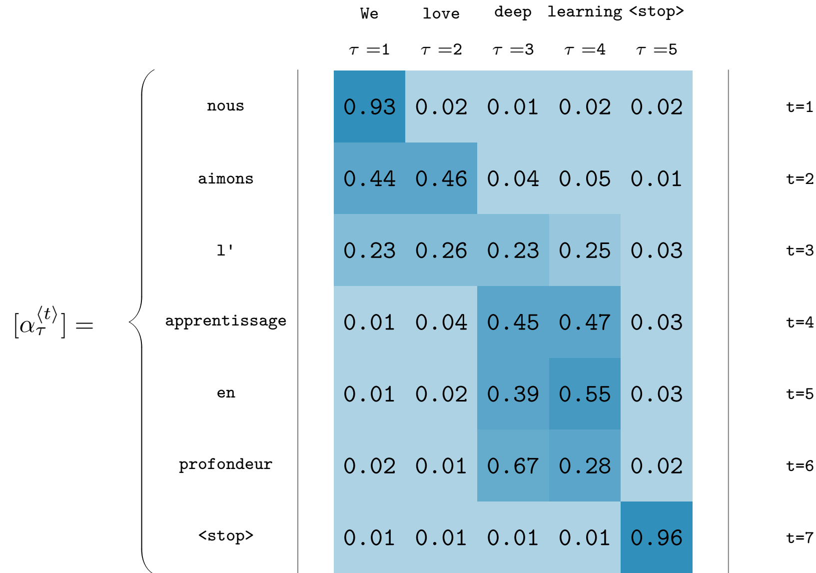


Figure 7.11: An attention matrix where each row is the attention vector $\alpha^{\langle t \rangle}$. When used in an encoder-decoder, we may consider the attention in a row associated with time t as based on the previous output. So for example for time $t = 4$, the available output from the decoder so far is an encoding of `nous aimons l'` and the attention vector $\alpha^{\langle 4 \rangle}$ dictates which encoded English words to focus on (in this case `deep` at $\alpha_3^{\langle 4 \rangle}$ and `learning` at $\alpha_4^{\langle 4 \rangle}$).

Figure 7.11 illustrates possible values of the attention weights, as they are computed via the machine translation process. The input is in English and the output is in French. Specifically, each row of this matrix is an hypothetical vector of attention weights, $\alpha^{\langle t \rangle}$, computed at time t of the decoder; namely $t = 1, \ldots, T_{\text{out}}$. Observe that each row sum is 1 and entries with higher probabilities are emphasized. This sequence of attention weight vectors shows how attention weights can adjust according to context. For example at time $t = 4$ in the decoder, an encoding of the partial sequence `nous aimons l'` is already available via $z_q = s^{\langle 3 \rangle}$. For creation of the next word, `apprentissage` (which directly means "learning" in English), most of the attention is put on the encoder hidden states associated both with `deep` and with `learning`.

Variants of the Score Function

If we use the inner product score function, then we are constrained that the dimension of the decoder hidden state, $s^{\langle t \rangle}$ be twice the dimension of each of the directional encoder hidden states (this is the dimension of $h_f^{\langle t \rangle}$ and $h_r^{\langle t \rangle}$). However, as already stated, other score functions are also possible. In each alternative, the score function operates on $z_q \in \mathbb{R}^{m_q}$ and $z_k \in \mathbb{R}^{m_k}$. These are common alternatives,

$$s(z_q, z_k) = \begin{cases} z_q^\top W_s \, z_k, & \text{(General)} \\ w_s^\top \tanh(\widetilde{W}_s [z_q, z_k]), & \text{(Concatenation)} \\ w_s^\top \tanh(W_{sa} \, z_q + W_{sb} \, z_k). & \text{(Additive)} \end{cases}$$

Each of these score function alternatives has parameters that are learned during training. In the first case the parameter matrix is $W_s \in \mathbb{R}^{m_q \times m_k}$. In the second case, the parameter vector w_s is $\tilde{m}$ dimensional, and the parameter matrix is $\widetilde{W}_s \in \mathbb{R}^{\tilde{m} \times (m_q + m_k)}$. Note that in this case $[z_q, z_k]$ denotes a concatenation of the two vectors. In third case, $W_{sa} \in \mathbb{R}^{\tilde{m} \times m_q}$ and $W_{sb} \in \mathbb{R}^{\tilde{m} \times m_k}$ and again w_s is a $\tilde{m}$ dimensional vector. In each of these cases tanh is applied element wise.

Training Encoder-Decoder Models

Continuing with the application of machine translation we now consider various approaches for training encoder-decoder models. Assume we are training models as in the architectures of Figure 7.9 (a) and (b) as well as Figure 7.10. Our discussion here is also relevant for training transformer encoder-decoder models presented in the next section (Figure 7.16).

As input data we have n sequences in the source language (e.g., English), where each sequence, denoted via $x^{(i)}$ is already encoded into vectors $x^{i,\langle 1 \rangle}, \ldots, x^{i,\langle T_x^i \rangle}$, using some word embedding technique (e.g., word2vec or some more advanced variant). Similarly we have n sequences in the target language (e.g., French), where in this case, each sequence $y^{(i)}$ is one-hot encoded into vectors $y^{i,\langle 1 \rangle}, \ldots, y^{i,\langle T_y^i \rangle}$, according to the lexicon of the target language. Clearly we assume that for any i, the pair of sequences $x^{(i)}$ and $y^{(i)}$ have the same semantic meaning.

For a given set of parameters, when feeding an input sequence $x^{(i)}$ into the encoder-decoder architecture, we are presented with an output sequence $\hat{y}^{(i)}$ where each element $\hat{y}^{i,\langle t \rangle}$ is a probability vector in the lexicon of the target language. We then use cross-entropy loss,[9] comparing $\hat{y}^{(i)}$ to $y^{(i)}$, summing over all elements of the sequence, similarly to (7.4) and (7.5); see also the discussion around Figure 7.5 in Section 7.2. One may also use mini-batches over multiple sequences, where for each mini-batch backpropagation (or backpropagation through time) is applied, and then a variant of gradient descent is used, similarly to any other deep learning model.

Teacher forcing, as discussed at the end of Section 7.2 is very commonly used when training encoder-decoder models. For example in the encoder-decoder with an atttention architecture of Figure 7.10, during training we replace $\hat{y}^{\langle t-1 \rangle}$ by $y^{\langle t-1 \rangle}$ in (7.22), and similarly for the other encoder-decoder models.

Note also, that once an encoder-decoder model is trained, we may sometimes fine tune either the encoder, or decoder, by *freezing* the layers of one of the components while training the other component. Also, it is common to freeze both the encoder and decoder, and only fine tune an output layer on top of the decoder.

7.5 Transformers

We now introduce a family of models called *transformers*. The transformer architecture was originally introduced to handle machine translation and has since found applications in many other paradigms including large language models, but also non sequence data

[9] A minor technical issue is that often the lengths of $\hat{y}^{(i)}$ and $y^{(i)}$ may differ. In such a case, the shorter sequence is padded with `<empty>` tokens and no loss is incurred at time t if both the predicted and training sequences have an `<empty>` token at time t.

domains such as images. The approach we present here continues to focus on the machine translation application, yet the reader should keep in mind that transformers have much wider applicability.

Transformers mark a paradigm shift in dealing with sequence data, as the architecture is no longer of a recurrent nature, but rather works using parallel computation. That is, while recurrent neural networks, LSTMs, GRUs, and other variants discussed in earlier sections may seem natural for sequence data, with transformers we return to fixed length input-output schemes. Similarly to the feedforward networks of Chapter 5 or the convolutional networks of Chapter 6, transformers operate on inputs of a fixed length, and yield outputs of a fixed length. Nevertheless, note that transformer decoders are used in an auto-regressive manner with a variable number of iterations.

In the context of sequence data, when using transformers, sequences are converted to have fixed length by padding with representations of `<empty>` tokens at the end of the sequence, when needed. Similarly if the input or output exceed the dimensions, mechanisms external to the transformer are used to raise an error, break up the computation, truncate the input, or carry out similar workarounds.

As a simple illustration, return to the English to French translation example from the previous section and as a toy example, assume that the transformer input and output dimensions are both 9. In this case, we can expect the padded input and output to be,

```
 Input:  we    love     deep  learning       <stop>  <empty>     <empty>  <empty>  <empty>
Output:  nous  aimons   l'    apprentissage  en      profondeur  <stop>   <empty>  <empty>
```

While the abandonment of variable length inputs and outputs may seem like a step back from recurrent neural networks, transformers have shown great benefits in performance. In addition to yielding state of the performance on many language benchmarks, these architectures enable parallel computation which is not possible with recurrent neural networks.

The transformer architecture that we introduce here inherits the encoder-decoder pattern used in the previous section, and is well suited for machine translation. Note however that for other tasks, one may sometimes only use the encoder part of the transformer, the decoder part, or slight variants of the architecture that we present here. The key mechanism used in transformers is attention, with various forms of the attention mechanism interconnected in a novel way.

In our overview of transformers, we first describe the notion of *self attention*. We then describe *multi-head self attention*, often called *multi-head attention* in short. We then describe *positional embeddings* and then move onto introducing the *transformer block* which is the basic building block of the transformer architecture both for the encoder and the decoder. We close the section with an outline of the transformer encoder-decoder architecture followed by a discussion of how transformers are used in production and training.

Self Attention

We have already seen the general attention mechanism in equation (7.21) which transforms a sequence $v^{\langle 1\rangle},\ldots,v^{\langle T_{\text{in}}\rangle}$ to an output sequence $u^{\langle 1\rangle},\ldots,u^{\langle T_{\text{out}}\rangle}$, using linear combinations with attention weights $\alpha_{\tau}^{\langle t\rangle}$. The attention weights are defined in equation (7.20) and are

based on the score function applied to the proxy vectors, denoted $z_q^{\langle 1 \rangle}, \ldots, z_q^{\langle T_{\text{out}} \rangle}$, and $z_k^{\langle 1 \rangle}, \ldots, z_k^{\langle T_{\text{in}} \rangle}$.

In the context of the transformer architecture, we refer to the proxy vectors $z_q^{\langle t \rangle}$ as *queries*, we refer to the proxy vectors $z_k^{\langle t \rangle}$ as *keys*, and we refer to the input vectors $v^{\langle t \rangle}$ as *values*. This terminology is rooted in information retrieval systems and captures the fact that when we compute an attention vector $\alpha^{\langle t \rangle}$, we are "searching" via (7.20) for a query represented via $z_q^{\langle t \rangle}$ against all keys $z_k^{\langle 1 \rangle}, \ldots, z_k^{\langle T_{\text{in}} \rangle}$. Then the attention weights are used to combine the "search results" via (7.21).

The mechanism of *self attention*, illustrated in Figure 7.12, is a form of attention where we convert an input sequence $x^{\langle 1 \rangle}, \ldots, x^{\langle T_{\text{in}} \rangle}$ to an output $u^{\langle 1 \rangle}, \ldots, u^{\langle T_{\text{out}} \rangle}$, with $T = T_{\text{in}} = T_{\text{out}}$. In the simplest form (ignore blue in Figure 7.12) we set the queries, the keys, and the values directly as elements of the input. Namely,

$$z_q^{\langle t \rangle} = x^{\langle t \rangle}, \qquad z_k^{\langle t \rangle} = x^{\langle t \rangle}, \qquad v^{\langle t \rangle} = x^{\langle t \rangle}, \qquad \text{(Simple self attention)}$$

and this implies that in the causal form (ignore red in Figure 7.12), with score function $s(\cdot, \cdot)$, the self attention mechanism yields output for any time $t \in \{1, \ldots, T\}$, via,

$$u^{\langle t \rangle} = \sum_{\tau \le t} \alpha_\tau^{\langle t \rangle} x^{\langle \tau \rangle}, \qquad \text{with} \qquad \alpha_\tau^{\langle t \rangle} = \frac{e^{s(x^{\langle t \rangle}, x^{\langle \tau \rangle})}}{\sum_{t'=1}^{t} e^{s(x^{\langle t \rangle}, x^{\langle t' \rangle})}}. \tag{7.23}$$

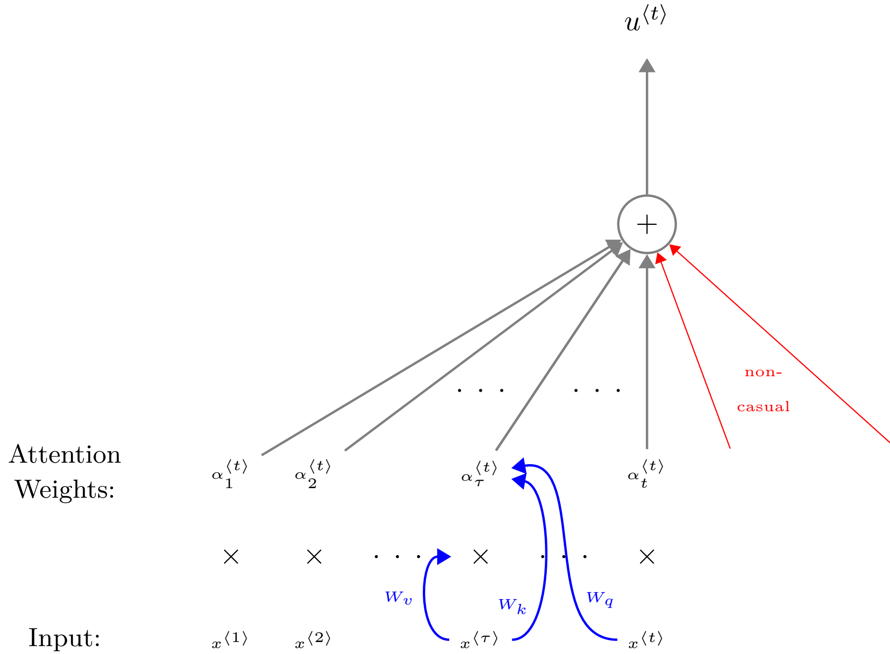


Figure 7.12: The flow of information in self attention. Ignoring the blue and the red, this is causal simple self attention where the output at time t, $u^{\langle t \rangle}$ is a linear combination of $x^{\langle 1 \rangle}, \ldots, x^{\langle t \rangle}$ with attention weights $\alpha_1^{\langle t \rangle}, \ldots, \alpha_t^{\langle t \rangle}$. Considering the blue, this is more versatile self attention where each attention weight $\alpha_\tau^{\langle t \rangle}$ is computed using weighting matrices W_k and W_q of the input, and where the linear combination is of weighted inputs with W_v. Considering also the red, it is non-causal.

A more versatile form of self attention, this time involving learned parameters, has queries, keys, and values that are not directly taken as the input, but are rather linear transformations of the input. Namely,

$$z_q^{\langle t \rangle} = W_q x^{\langle t \rangle}, \qquad z_k^{\langle t \rangle} = W_k x^{\langle t \rangle}, \qquad v^{\langle t \rangle} = W_v x^{\langle t \rangle}, \qquad \text{(More versatile self attention)}$$

where the learnable parameter matrices W_q, W_k, and W_v are each $p \times p$ dimensional,[10] with p the dimension of each $x^{\langle t \rangle}$. Hence in this case, the attention mechanism has output for any time $t \in \{1, \ldots, T\}$, via,

$$u^{\langle t \rangle} = \sum_{\tau \leq t} \alpha_\tau^{\langle t \rangle} W_v \, x^{\langle \tau \rangle}, \qquad \text{with} \qquad \alpha_\tau^{\langle t \rangle} = \frac{e^{s(W_q \, x^{\langle t \rangle}, W_k \, x^{\langle \tau \rangle})}}{\sum_{t'=1}^{t} e^{s(W_q \, x^{\langle t \rangle}, W_k \, x^{\langle t' \rangle})}}. \tag{7.24}$$

Observe that that in (7.23) and (7.24) we use the causal form from (7.21). An alternative non-causal form is also applicable (consider also the red part of Figure 7.12). In such a case, the summations are not on $\tau \leq t$ but are rather on $\tau \in \{1, \ldots, T\}$, similarly to the non-causal form appearing in (7.21).

[10] Note that here we use the same dimension for the input, the output, and the proxy vectors. More generally one may set different dimensions for these entities, as we do in the case of multi-head attention below.

Multi-Head Self Attention

A generalization of self attention is to use multiple self attention mechanisms in parallel and then combine the outputs of these mechanisms. With this parallelism, we can treat each individual attention mechanism as searching for a different set of features in the input, and then have information content of the output as a combination of the derived features.

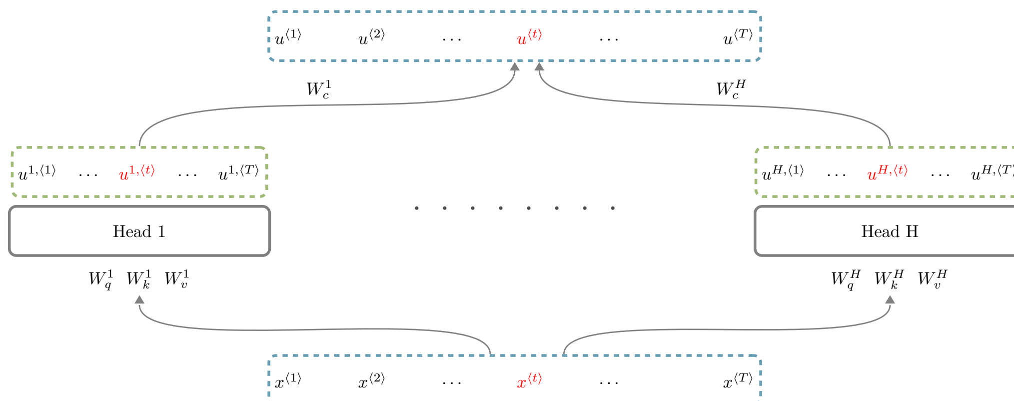


Figure 7.13: Multi-head self attention is the parallel application of H attention heads, where head h has parameter matrices W_q^h, W_k^h, and W_v^h. Each head h operates on the full input, $x^{\langle 1\rangle},\ldots,x^{\langle T\rangle}$ and results in output for the head, $u^{h,\langle 1\rangle},\ldots,u^{h,\langle T\rangle}$. When determining the output of the whole multi-head self attention mechanism, each output at time t, denoted $u^{\langle t\rangle}$ combines $u^{1,\langle t\rangle},\ldots,u^{H,\langle t\rangle}$ weighted by $W_c^1,\ldots,W_c^H$.

Figure 7.13 illustrates multi-head self attention. Specifically, assume we have H self attention mechanisms, where mechanism $h \in \{1,\ldots,H\}$ has its own set of parameter matrices W_q^h, W_k^h, and W_v^h. Here $W_q^h \in \mathbb{R}^{m\times p}$, $W_k^h \in \mathbb{R}^{m\times p}$, and $W_v^h \in \mathbb{R}^{m_v\times p}$, where m is the dimension of the query and the key ($m = m_q = m_k$), p is the dimension of the input $x^{\langle t\rangle}$ as previously, and m_v is the dimension of the value (and the output of the individual attention head).

At first, each attention head h operates independently similarly to (7.24), yielding an output sequence $u^{h,\langle 1\rangle},\ldots,u^{h,\langle T\rangle}$ of m_v dimensional vectors. This operation is via,

$$u^{h,\langle t\rangle} = \sum_{\tau=1}^{T} \alpha_\tau^{h,\langle t\rangle}\, W_v^h\, x^{\langle\tau\rangle}, \qquad \text{with} \qquad \alpha_\tau^{h,\langle t\rangle} = \frac{e^{s(W_q^h\, x^{\langle t\rangle}, W_k^h\, x^{\langle\tau\rangle})}}{\sum_{t'=1}^{t} e^{s(W_q^h\, x^{\langle t\rangle}, W_k^h\, x^{\langle t'\rangle})}}. \tag{7.25}$$

Note that in (7.25) we use a non-causal form of attention, in contrast to the causal form appearing in (7.24). Below we comment on how one may practically convert such a non-causal form to a causal form via a mechanism called *masked self attention*.

Now for any time index t, we have H vectors $u^{1,\langle t\rangle},\ldots,u^{H,\langle t\rangle}$ which we use to produce the output of the multi-head attention for the specific time index t. For this we apply an additional linear transformation to each vector, converting it from dimension m_v back to dimension p. We then sum up over $h = 1,\ldots,H$. Thus, the output of the multi-head self

attention mechanism is

$$u^{\langle t \rangle} = \sum_{h=1}^{H} W_c^h u^{h,\langle t \rangle}, \tag{7.26}$$

where for each h, the matrix W_c^h is $p \times m_v$ dimensional. Each W_c^h captures the transformation from the single attention output $u^{h,\langle t \rangle}$ of dimension m_v to dimension p. The combination of these H matrices can also capture weightings between the attention heads.

Multi-head self attention plays a central role in transformer blocks, both in the encoder and the decoder. In certain cases such as the encoder, we use the non-causal form, as in (7.25). However, in other cases, such as the decoder, we use a causal form. Practically we may enforce a causal form via masked self attention, also known as *masking* for short. With this approach, for a computation of attention weights at time t, we set entries of the (key) proxy vectors of times after time t to negative infinity. That is,

$$z_k^{\langle t+1 \rangle} = -\infty, \quad z_k^{\langle t+2 \rangle} = -\infty, \quad \ldots \, , z_k^{\langle T \rangle} = -\infty. \qquad \text{(Masking)} \tag{7.27}$$

Then with this masking, through the softmax computation, the $-\infty$ values yield zeros for each of $\alpha_{t+1}^{\langle t \rangle}, \alpha_{t+2}^{\langle t \rangle}, \ldots, \alpha_T^{\langle t \rangle}$, and thus, when computing the output at time t, no attention is given to future times.

Implementing causality via masking is especially important when one considers a matrix representation of the multi-head self attention mechanism. Specifically, one may treat the input as a matrix X of dimension $p \times T$ and then represent all of the attention operations as matrix on matrix operations. While we omit the details of such a representation, the reader should keep in mind that unlike recurrent neural networks where time t implies a step in the computation, for transformers time t is simply a dimension of the input matrix X and operations can be parallelized based on the equations above.

Positional Embeddings

Sequence models have a natural time index, t, where $x^{\langle t \rangle}$ followed by $x^{\langle t+1 \rangle}$, embodies some relationship between the two vectors in the sequence. As a simple example consider some input text, `deep learning`, and the reverse text, `learning deep`. These two short sequences have different semantic meaning, since the order matters. However, as is evident from the multi-head self attention mechanism (7.25), the order in the input sequence $x^{\langle 1 \rangle}, \ldots, x^{\langle T \rangle}$ is not captured by the mechanism at all. This stands in stark contrast to previous sequence models such as recurrent neural networks and their generalizations, where the recurrent nature of the model makes use of sequence order.

Hence, for using non-causal multi-head self attention effectively, we require a mechanism for encoding the order of the sequence in the input data. Such mechanisms are generally called *positional embeddings*. A basic and primitive form of positional embedding is to extend each input vector $x^{\langle t \rangle}$ with an additional one-hot encoded vector that captures its position in the sequence. For example for a sequence of length $T = 4$, we extend $x^{\langle 1 \rangle}$ with $e_1 = (1, 0, 0, 0)$, extend $x^{\langle 2 \rangle}$ with $e_2 = (0, 1, 0, 0)$, and so forth. Then the input sequence is no longer taken as having vectors of length p, but rather as having vectors of length $\tilde{p} = p + T$.

To further illustrate the point using the toy *one-hot encoding positional embedding* example with $p = 2$ and $T = 4$, assume the first vector is $x^{\langle 1 \rangle} = (0.2, -1.3)$ and assume that as a

matter of coincidence the last vector $x^{\langle 4 \rangle}$ has the same values. Then after applying such positional embedding, the vectors are transformed to

$$\tilde{x}^{\langle 1 \rangle} = (0.2, -1.3, \underbrace{1, 0, 0, 0}_{e_1}) \qquad \text{and} \qquad \tilde{x}^{\langle 4 \rangle} = (0.2, -1.3, \underbrace{0, 0, 0, 1}_{e_4}),$$

and then when these positionally embedded vectors are processed by non-causal multi-head self attention, the model can distinguish between $\tilde{x}^{\langle 1 \rangle}$ and $\tilde{x}^{\langle 4 \rangle}$. Even in the (more common) case where different vectors will not have repeated values, the order in the sequence is still encoded and this enhances performance.

However, this type of encoding is clearly wasteful and inefficient, yielding an excessively large $\tilde{p}$. A more advanced form is to encode the vectors and the embeddings jointly. For example, one might use a transformation such as

$$\tilde{x}^{\langle t \rangle} = W_1 S\left(W_2 x^{\langle t \rangle} + W_3 e_t + b\right) \in \mathbb{R}^{\tilde{p}}, \tag{7.28}$$

where $W_1 \in \mathbb{R}^{\tilde{p} \times \tilde{p}}$, $W_2 \in \mathbb{R}^{\tilde{p} \times p}$, and $W_3 \in \mathbb{R}^{\tilde{p} \times T}$ are learnable weight matrices, $b \in \mathbb{R}^{\tilde{p}}$ is a learnable bias vector, and $S(\cdot)$ is some vector activation function such as for example ReLU applied element wise. In this case, we may just use $\tilde{p} = p$, yet larger $\tilde{p}$ are also possible.

With this type of encoding, after training the parameters, positional embeddings are ideally encoded within the word vectors. This is similar to how vector representations encode words from a dictionary into a lower dimensional space when using word embeddings.

With the introduction of transformers, a different type of positional encoding, motivated by *Fourier analysis*, was popularized. With this approach we set some $\tilde{p} > p$ and denote $p_e = \tilde{p} - p$. That is, p_e is the increase of dimension from the original encoding in $\mathbb{R}^p$ to the new encoding in $\mathbb{R}^{\tilde{p}}$. Assume also that p_e is even. Unlike the trained positional encoding in (7.28), here we just use sines and cosines without any trainable parameters.

Specifically for $i \in \{0, \ldots, \frac{p_e}{2} - 1\}$, and $t \in \{1, \ldots, T\}$ we set,

$$r_{2i}^{\langle t \rangle} = \sin\left(\frac{t}{M^{2i/p_e}}\right), \quad r_{2i+1}^{\langle t \rangle} = \cos\left(\frac{t}{M^{2i/p_e}}\right), \tag{7.29}$$

where a common value for M is $10,000$. The vector with positional embedding is then,

$$\tilde{x}^{\langle t \rangle} = (x_1^{\langle t \rangle}, \ldots, x_p^{\langle t \rangle}, \underbrace{\overbrace{r_0^{\langle t \rangle}}^{\sin}, \overbrace{r_1^{\langle t \rangle}}^{\cos}, \overbrace{r_2^{\langle t \rangle}}^{\sin}, \ \ldots \ , \overbrace{r_{p_e-1}^{\langle t \rangle}}^{\cos}}_{\text{positional embedding}}).$$

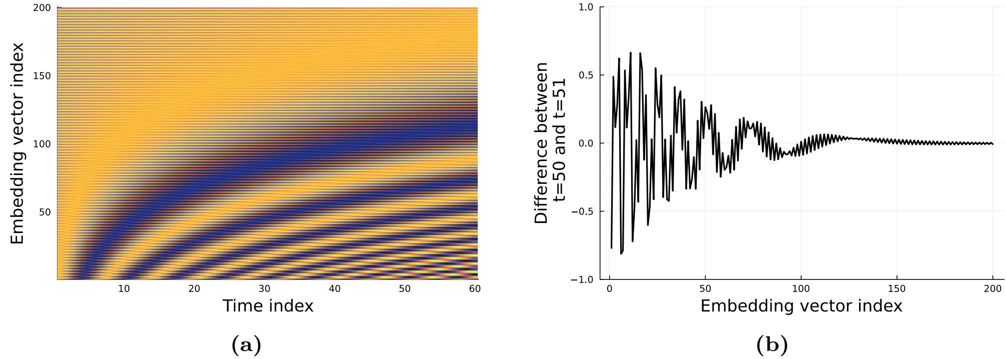


Figure 7.14: The sine and cosine based positional embedding as in (7.29). (a) A heat map of $r_j^{\langle t \rangle}$ with $p_e = 768$ and $T = 4000$ plotted for t only over the first 60 time indexes for j only over the first 200 positions in the embedding vector. Positive values are yellow and negative values are blue. (b) A comparison of the positional embeddings at the time index $t = 50$ and $t = 51$ via a plot of the difference. As is evident, there is a significant difference marked by around the first hundred vector positions.

One may then also reduce the dimension back from $\tilde{p}$ to a lower dimension in similar vein to (7.28). Specifically one common simplistic approach which has empirically worked well is,

$$\tilde{x}^{\langle t \rangle} = (x_1^{\langle t \rangle} + r_0^{\langle t \rangle}, x^{\langle 2 \rangle} + r_1^{\langle t \rangle}, \quad \ldots \quad , x_p^{\langle t \rangle} + r_{p-1}^{\langle t \rangle}) \in \mathbb{R}^p, \tag{7.30}$$

where here $p_e = p$.

The benefit of using positional embedding such as (7.30) first arises from the fact that sines and cosines are bounded within $[-1, 1]$ and is further aided by the idea of a Fourier representation of the position. In particular, consider Figure 7.14 where (a) illustrates embedding values from (7.29). By adding a vertical slice (for fixed t) of the embedding values (7.29) to the original $x^{\langle t \rangle}$ vector, we enable the model to distinguish the time value from typical other values. This is particularly evident by considering (b) where we compare two neighbouring time steps by plotting their difference.

The Transformer Block

The basic building block of the transformer architecture is a unit called a *transformer block* which is used multiple times within a transformer, interconnected in series, both in the encoder and the decoder. There are several variations of transformer blocks and here we focus on the basic block used in the encoder. Later when we describe the decoder architecture we highlight differences.

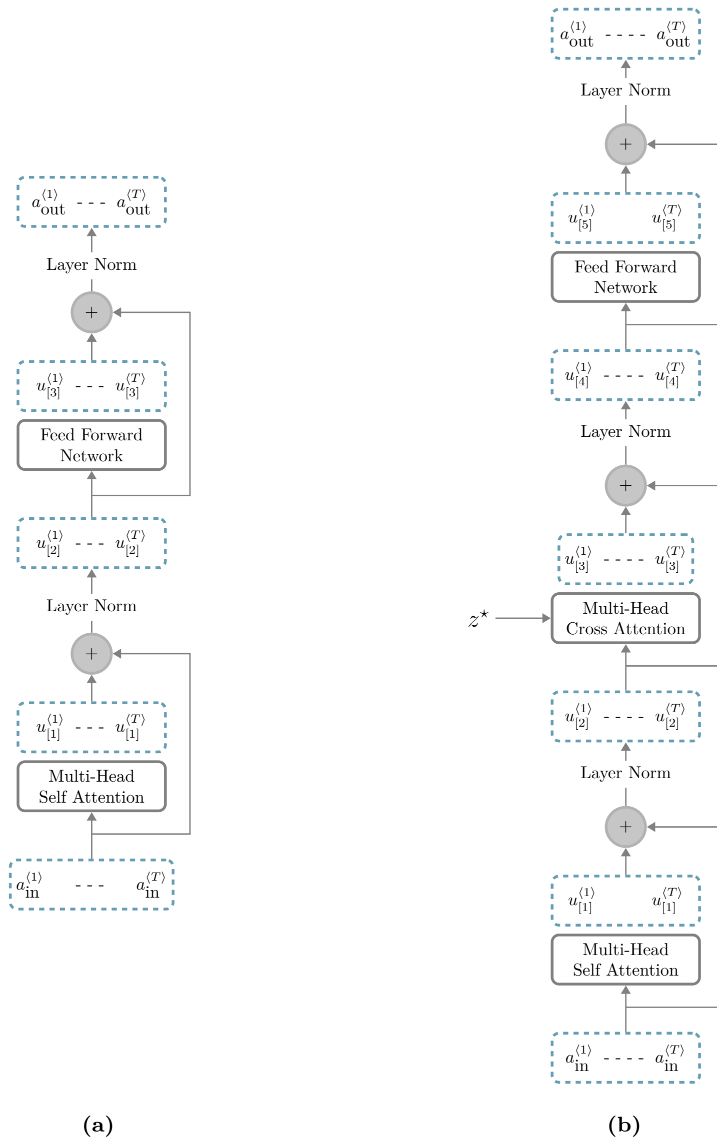


Figure 7.15: Architecture of transformer blocks. (a) A single transformer block. The input a_{in} passes through a multi-head self attention layer, followed by a feedforward layer. Layer normalization and residual connections are applied in these steps as well. (b) A transformer decoder block. In addition to the components of a transformer block, it also has a multi-head cross attention layer that is fed the context vector $z^\star$.

We denote the input of a transformer block as a_{in} and the output as a_{out} where both a_{in} and a_{out} are $p \times T$ matrices. Thus in general, we can view the block as a function $f_\theta : \mathbb{R}^{p\times T} \to \mathbb{R}^{p\times T}$, where θ represents trained parameters of the block and $a_{\text{out}} = f_\theta(a_{\text{in}})$. For example, the encoder of the transformer architecture has a first transformer block that operates on input,

$$a_{\text{in}} = \big[\tilde{x}^{\langle 1\rangle}, \ldots, \tilde{x}^{\langle T\rangle}\big],$$

where each column is a positional encoding vector as in (7.30). Then the output of this block, a_{out}, is fed into a second transformer block in the encoder, and so forth. Common architectures have an encoder composed of a sequence of half a dozen or more transformer blocks. Thus for example the input of the second transformer block has a_{in} set as the a_{out} of the first block, etc.

A transformer block has several internal layers with the two main layers being a multi-head self attention layer, and downstream to it, a feedforward layer. Each of these also utilizes a residual connection and a normalization layer. A schematic of a typical transformer block is in Figure 7.15 (a). The main idea of the transformer block is to enhance multi-head self attention with further connections using the feedforward layer. The residual connections enhance training performance, similar to ResNets discussed in Section 6.5. Normalization, in the form of *layer normalization*, stabilizes training and production performance by ensuring that values remain in a sensible dynamic range. We now present the details of a block.

Let us denote the columns of the input a_{in} as $a_{\text{in}}^{\langle 1\rangle},\ldots,a_{\text{in}}^{\langle T\rangle}$. The first step of the block with multi-head self attention is to employ (7.25) for every head $h=1,\ldots,H$, and then (7.26) where in these equations $x^{\langle t\rangle}$ is replaced by $a_{\text{in}}^{\langle t\rangle}$. The output of this multi-head self attention layer is then denoted $u_{[1]}^{\langle 1\rangle},\ldots,u_{[1]}^{\langle T\rangle}$, where we use the subscript [1] to indicate it is an output of the first layer.

We then apply residual connections and layer normalization to each of $u_{[1]}^{\langle 1\rangle},\ldots,u_{[1]}^{\langle T\rangle}$ yielding $u_{[2]}^{\langle 1\rangle},\ldots,u_{[2]}^{\langle T\rangle}$. This step can be summarized as.

$$u_{[2]}^{\langle t\rangle} = \text{LayerNorm}\,\big(\underbrace{a_{\text{in}}^{\langle t\rangle} + u_{[1]}^{\langle t\rangle}}_{\text{Residual connection}}\,;\gamma,\beta\big). \tag{7.31}$$

Here, the LayerNorm$(\cdot)$ operator is defined for $z\in\mathbb{R}^p$ with parameters $\gamma,\beta\in\mathbb{R}^p$, via,

$$\text{LayerNorm}(z\,;\gamma,\beta) = \gamma\odot\frac{(z-\mu_z)}{\sqrt{\sigma_z^2+\varepsilon}}+\beta,$$

where

$$\mu_z = \frac{1}{p}\sum_{i=1}^{p} z_i, \qquad \text{and} \qquad \sigma_z = \sqrt{\frac{1}{p}\sum_{i=1}^{p}(z_i-\mu_z)^2},$$

$\varepsilon>0$ is a small fixed quantity that ensures that we do not divide by zero, and the addition, division, and square root operations are all element-wise operations.

Layer normalization is somewhat similar to batch normalization, outlined in Section 5.6, and group normalization outlined in Section 6.4. A major difference between layer normalization and batch normalization is that for batch normalization we obtain the mean and standard deviation per feature over a mini batch, whereas with layer normalization we use a single sample, yet compute statistics over all features (of a single feature vector). In both cases, the normalization forces feature values to remain at normalized values, further aided by the learnable parameter vectors γ and β.

The next step in the transformer block is the application of a fully connected neural network on each $u_{[2]}^{\langle t\rangle}$ to yield $u_{[3]}^{\langle t\rangle}$. Note that the same learnable network parameters are used for each

$t \in \{1, \ldots, T\}$. Commonly this network has a single hidden layer with non-linear activation, followed by a layer with linear (identity) activation,[11] yet there are other possibilities as well. Sticking with the commonly used architecture we have,

$$u_{[3]}^{\langle t \rangle} = W^{[2]} S(W^{[1]} u_{[2]}^{\langle t \rangle} + b^{[1]}) + b^{[2]},$$

where $S(\cdot)$ is commonly an element wise application of ReLU. Here we denote the dimension of the inner layer as N_1 and the learnable parameters are $b^{[1]} \in \mathbb{R}^{N_1}$, $b^{[2]} \in \mathbb{R}^p$, $W^{[1]} \in \mathbb{R}^{N_1 \times p}$, and $W^{[2]} \in \mathbb{R}^{p \times N_1}$.

Finally, to yield the output of the transformer block, we apply residual connections and layer normalization in the same manner as (7.31). Specifically we use,

$$a_{\text{out}}^{\langle t \rangle} = \text{LayerNorm}\left(u_{[2]}^{\langle t \rangle} + u_{[3]}^{\langle t \rangle}; \tilde{\gamma}, \tilde{\beta}\right),$$

where here $\tilde{\gamma}, \tilde{\beta} \in \mathbb{R}^p$ are trainable parameters for this layer normalization.

Note that when considered as a variation of a neural network, one may view a transformer block as "wide and shallow". Even though such a single transformer block is not deep, the residual connections provide direct access to the previous levels of abstraction and enable the levels above to infer more fine grained features without having to remember or store previous ones.

Let us summarize the learned parameters of a single transformer block with dimensions p for the vector length, T for the sequence length, H for the number of self attention heads, m_v for the dimension inside each self attention block, and m for the dimension of the query and key inside each self attention block. In this case, the total number of parameters is,

$$\underbrace{4p}_{\gamma,\beta,\tilde{\gamma},\tilde{\beta}} + \underbrace{2N_1 p}_{W^{[1]},W^{[2]}} + \underbrace{N_1 + p}_{b^{[1]},b^{[2]}} + H \times \Big(\underbrace{2mp}_{W_k^h \text{ and } W_q^h} + \underbrace{2m_v p}_{W_v^h \text{ and } W_c^h} \Big). \tag{7.32}$$

As a quantitative example, agreeing with the first transformer architecture introduced in 2017,[12] let us consider a case with $p = 512$, $N_1 = 2048$, $m = m_v = 64$, and $H = 8$. In this case there are just over 3 million learnable parameters for such a transformer block. Specifically (7.32) evaluates to $3,150,336$. As we see now, multiple transformer blocks are typically connected, yielding architectures with many millions of parameters.

Putting the Bits Together Into an Encoder-Decoder Framework

Now that we have seen the design of the transformer block, we are ready to interconnect such blocks in an encoder, and also interconnect variations of this block in a decoder. We now describe a basic *transformer encoder-decoder architecture* using such blocks and interconnections. It is useful to recall the more classical encoder-decoder architectures as appearing in (a) and (b) of Figure 7.9. A transformer architecture is somewhat similar to the architecture in (b), since the output of the encoder is fed into each of the decoder steps.

[11] Incidentally, this two-layer architecture with an activation function only in the single hidden layer, is the same network used in Theorem 5.1 of Chapter 5.

[12] See the "Attention is all you need" paper [410], as well as other notes and references at the end of the chapter.

However, unlike the architectures in Figure 7.9, transformers do not process one word at a time. A schematic of a transformer encoder-decoder architecture is in Figure 7.16.

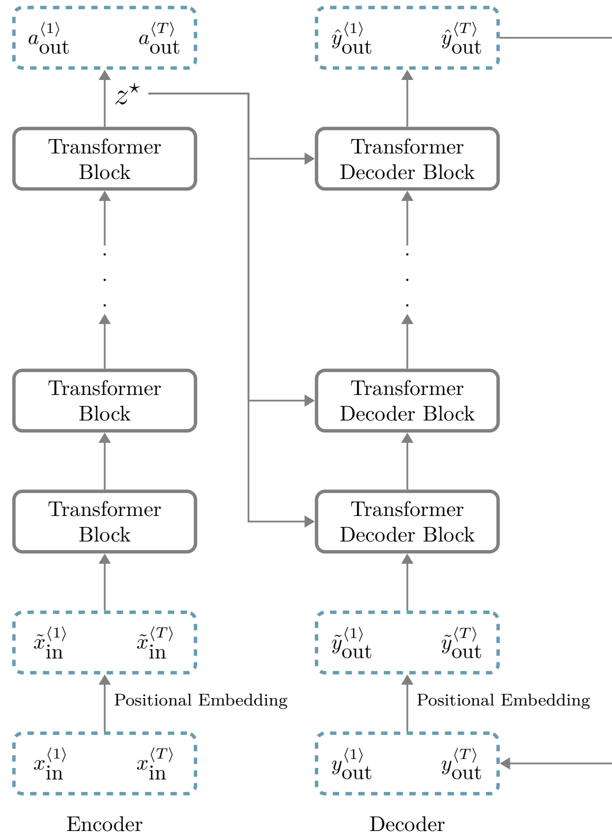


Figure 7.16: An encoder-decoder transformer architecture. The encoder is composed of multiple transformer blocks, and the decoder is composed of multiple transformer decoder blocks. Each block in the decoder is fed the code $z^\star$. The loop from the output of the decoder going back into the input of the decoder illustrates the auto-regressive application of transformer decoders.

Transformer encoders simply stack the transformer blocks in series, where each block has exactly the specifications described above as in Figure 7.15 (a). The first block is fed with the positional encoded input, and the output of that block goes into the second block, and so forth. The output of the encoder is $a^{\langle 1 \rangle}_{\text{out}}, \ldots, a^{\langle T \rangle}_{\text{out}}$ resulting from the last transformer block. We also denote this output via $z^\star$ as it describes a code, and thus for each position t we denote the encoder output via $z^{\star\langle t \rangle}$.

Transformer decoders use a variation of the transformer block which we call the *transformer decoder block*, illustrated in Figure 7.15 (b). This block architecture differs from the transformer block specified above in two ways. First, the multi-head self attention layer is causal. This is implemented via masking as in (7.27). Such causality prevents attendance of future positions as suitable for auto-regressive prediction. Second, an additional layer, called a *cross attention layer*, is used between the causal multi-head self attention and the

feedforward layer. The new cross attention layer is handled with layer normalization and residual connections, similar to the other two layers.

In addition to the flow of information within the transformer decoder block, the cross attention layer is fed with the encoder output $z^\star$. This is similar to the encoder-decoder architecture in Figure 7.9 (b), and it allows the transformer decoder block to directly incorporate the encoder's code. Transformer decoders are constructed by stacking several transformer decoder blocks in sequence, similarly to the stacking in the encoder, where each block gets the same $z^\star$. Similar to the encoder, the first block operates on positional embedded inputs. Further discussion of what these inputs are, is in the following subsection. On top of the final transformer decoder block (not illustrated in Figure 7.16), we add an additional layer transforming each output vector to a token. This is similar to the outputs of other encoder-decoder architectures. Often it is simply a linear layer with a softmax (i.e., a multinomial regression as in Section 3.3).

The cross attention layer inside each transformer decoder block is in fact a *multi-head cross attention* layer and follows equations similar to (7.25) and (7.26), with the difference being that the key and value inputs are the decoder output $z^\star$. Specifically, if we denote the input to the multi-head cross attention layer from earlier layers as $\tilde{u}^{\langle 1\rangle}, \ldots, \tilde{u}^{\langle T\rangle}$, then the self attention equation (7.25) is now modified to have cross attention (between $z^\star$ and the input to the layer $\tilde{u}$). This attention computation for head h is then,

$$u^{h,\langle t\rangle} = \sum_{\tau=1}^{T} \alpha_\tau^{h,\langle t\rangle}\, W_v^h\, z^{\star\langle \tau\rangle}, \qquad \text{with} \qquad \alpha_\tau^{h,\langle t\rangle} = \frac{e^{s(W_q^h\, \tilde{u}^{\langle t\rangle}, W_k^h\, z^{\star\langle \tau\rangle})}}{\sum_{t'=1}^{t} e^{s(W_q^h\, \tilde{u}^{\langle t\rangle}, W_k^h\, z^{\star\langle t'\rangle})}}, \tag{7.33}$$

followed by a combination of the heads using (7.26).

Observe that this pattern of cross attention is similar to the earlier encoder-decoder with an attention mechanism architecture presented in Figure 7.10. In the earlier architecture, the proxy vector $z_q^{\langle t\rangle}$ used the previous decoder state, somewhat similarly $W_q^h\, \tilde{u}^{\langle t\rangle}$ in (7.33). Further, in the earlier architecture, the proxy vector $z_k^{\langle t\rangle}$ is the same vector used as input to the attention mechanism. This agrees with using the decoder output, $z^\star$ as key and value inputs in (7.33).

Now after highlighting the differences between a transformer block as in Figure 7.15 (a), and the transformer decoder block in Figure 7.15 (b), we can briefly summarize the layers and steps of the transformer decoder block. The matrix of inputs to each block, with each input vector denoted $a_{\text{in}}^{\langle t\rangle}$, is first processed with causal multi-head self attention to yield $u_{[1]}^{\langle 1\rangle}, \ldots, u_{[1]}^{\langle T\rangle}$. Now exactly as in (7.31), layer normalization and residual connections yield $u_{[2]}^{\langle 1\rangle}, \ldots, u_{[2]}^{\langle T\rangle}$. Then this sequence of vectors (or matrix) is processed via the multi-head cross attention layer using (7.33) and (7.26), where $\tilde{u}^{\langle t\rangle}$ is $u_{[2]}^{\langle t\rangle}$, and the encoder output $z^\star$ is put to use. The result is $u_{[3]}^{\langle 1\rangle}, \ldots, u_{[3]}^{\langle T\rangle}$. Then layer normalization and residual connections are applied again yielding $u_{[4]}^{\langle 1\rangle}, \ldots, u_{[4]}^{\langle T\rangle}$. This sequence is now fed into the feedforward layer, to yield $u_{[5]}^{\langle 1\rangle}, \ldots, u_{[5]}^{\langle T\rangle}$. Finally, each $u_{[5]}^{\langle t\rangle}$ is again applied with layer normalization and residual connections to yield the output $a_{\text{out}}^{\langle t\rangle}$. There are 6 steps here, in comparison to the 4 steps used in the transformer block of the encoder.[13]

[13] We count layer normalization and residual connection as a single step.

The parameters of each transformer decoder block include those of the transformer block from Figure 7.15, as well as parameters resulting from the multi-head cross attention layer and its normalization. Specifically, for the decoder, the parameter count in (7.32) needs to be augmented with an additional $H \times (2mp + 2m_v p)$ term for the multi-head cross attention as well as $2p$ for the additional normalization parameters. As an example, using the same dimensions as above, we now have over 4 million parameters, or $4,199,936$ exactly, for a transformer decoder block.

If we now consider an encoder-decoder transformer architecture with 6 transformer blocks in the encoder and 6 transformer decoder blocks in the decoder, then the number of parameters in the encoder is about 19 million, and the number of parameters in the decoder is about 25 million. As mentioned above, we also require an additional layer on top of the decoder, transforming each output vector to a token (for example to a natural language word or part of it). This is simply a linear layer with a softmax (i.e., a multinomial regression). This type of layer is needed at the end of any pipeline that generates text. If we assume that the number of word tokens[14] is $d_{\mathcal{V}} \approx 37,000$, then the number of parameters of the final multinomial regression is $d_{\mathcal{V}} \times p + p$, which is about 19 million parameters in our case. Hence putting the pieces together we have about 63 million parameters in the whole model.[15]

The encoder-decoder transformer model is the cornerstone of *large language models*, and indeed by 2023, models with up to half a trillion parameters are already in use. Such parameter count is about 100 times more parameters than the size of the transformer discussed above.

Using the Encoder-Decoder in Production and Training

In our description of the encoder-decoder architecture above, except for indicating that positional embedding is applied, we did not specify the decoder inputs. The way that decoder inputs are used depends on the task at hand, and the form of the inputs varies between production and training. We now present the details.

First in production (inference or test-time), note that we use the decoder in an auto-regressive manner as in Figure 7.17. Specifically, the code from the encoder $z^\star$ is presented to the decoder and we iterate executions of the decoder until a `<stop>` token (or word) is realized. In the first iteration we set the input sequence to the decoder to only have a `<start>` token embedding, and then with every iteration we present the output sequence up to the previous iteration as input to the decoder. That is, at iteration t, the decoder computation can be represented as

$$\hat{y}^{\langle t \rangle} = f_{\text{decoder}}\big(z^\star, (\tilde{\hat{y}}^{\langle 1 \rangle}, \ldots, \tilde{\hat{y}}^{\langle t-1 \rangle})\big), \tag{7.34}$$

where $z^\star$ is the code vector from the encoder, and $\tilde{\hat{y}}^{\langle t \rangle}$ is an embedded and positionally embedded vector, resulting from the decoder output token $\widehat{\mathcal{Y}}^{\langle t \rangle}$. The transformation from the decoder output $\hat{y}^{\langle t \rangle}$ to the decoder output token $\widehat{\mathcal{Y}}^{\langle t \rangle}$, can naively be done via an argmax as in (3.34) in Chapter 3, or by sampling tokens according to the probability output $\hat{y}^{\langle t \rangle}$. More advanced multi-step techniques such as *beam search* can also be employed, where several consecutive tokens are considered together; we omit the details. In summary, as we see in (7.34), the transformer decoder output at time t is a function of its previous outputs while the first input $\tilde{\hat{y}}^{\langle 1 \rangle}$ is an embedded and positionally embedded representation of `<start>`.

[14]This is a common tokenization dimension and was used in the original transformers paper.

[15]The first introduced transformer architecture, [410], is estimated to have used about 65 million parameters. Such relatively small discrepancies are due to implementation.

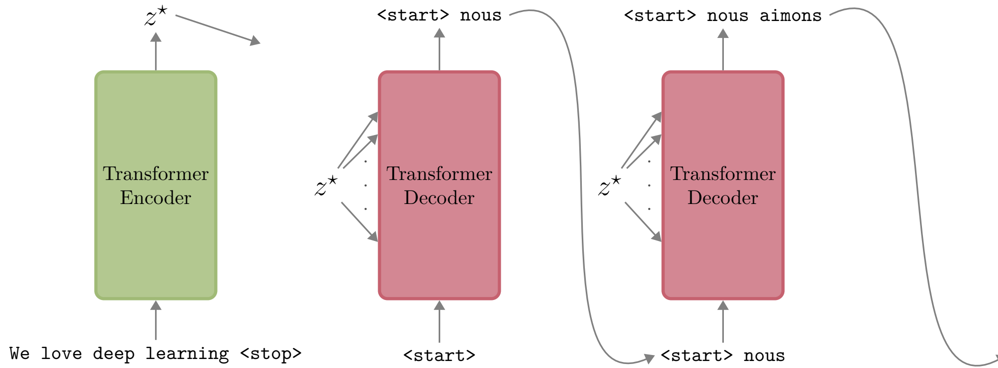


Figure 7.17: Auto-regressive application of a transformer decoder block for machine translation. The transformer encoder runs once on the whole input English sentence, creating the code $z^\star$. This code is then used with every iteration of the transformer decoder block, where with each iteration an additional word (or token) is created, and the previous output is fed as input in an auto-regressive manner. Generation stops (not illustrated in figure) when a `<stop>` token appears at the output.

Now considering training, a natural naive approach is to use (7.34) directly in each forward pass and backward pass iteration, when computing gradients. Namely, to use backward propagation through time. However, (7.34) naturally lends itself to use teacher forcing, similarly to the use of teacher forcing when training other encoder-decoder models, as discussed at the end of Section 7.4. With this approach, (7.34) is converted to

$$\hat{y}^{\langle t \rangle} = f_{\text{decoder}}\big(z^\star, (\tilde{y}^{\langle 1 \rangle}, \dots, \tilde{y}^{\langle t-1 \rangle})\big),$$

where now the training data one-hot encoded labels $y^{\langle 1 \rangle}, \dots, y^{\langle t \rangle}$, in their embedded and positionally embedded form, $\tilde{y}^{\langle 1 \rangle}, \dots, \tilde{y}^{\langle t \rangle}$, are used as input to the transformer decoder instead of the predictions. This teacher forcing technique accelerates training by removing error during early phases of the process. Further, an important difference in teacher forcing of transformers vs. teacher forcing of recurrent encoder-decoder frameworks, is that with transformers we can exploit parallelization. Specifically, for the forward pass, we may compute each of these in parallel:

$$\begin{cases} \hat{y}^{\langle 2 \rangle} = f_{\text{decoder}}\big(z^\star, (\tilde{y}^{\langle 1 \rangle})\big) \\ \hat{y}^{\langle 3 \rangle} = f_{\text{decoder}}\big(z^\star, (\tilde{y}^{\langle 1 \rangle}, \tilde{y}^{\langle 2 \rangle})\big) \\ \hat{y}^{\langle 4 \rangle} = f_{\text{decoder}}\big(z^\star, (\tilde{y}^{\langle 1 \rangle}, \tilde{y}^{\langle 2 \rangle}, \tilde{y}^{\langle 3 \rangle})\big) \\ \quad\vdots \\ \hat{y}^{\langle T \rangle} = f_{\text{decoder}}\big(z^\star, (\tilde{y}^{\langle 1 \rangle}, \tilde{y}^{\langle 2 \rangle}, \dots, \tilde{y}^{\langle T-1 \rangle})\big). \end{cases}$$

Note that while transformers were introduced for machine translation and are currently the power house of large language models for generative text modeling, adaptations of transformers have also been successful for other non-language tasks, including image tasks. In fact, transformer models compete with convolutional models, and in certain cases outperform convolutional models on images, especially in the presence of huge training datasets.

Notes and References

A useful applied introductory text about *time-series* sequence data analysis is [198] and a more theoretical book is [64]. Yet, while these are texts about sequence models, the traditional statistical and forecasting time-series focus is on cases where each $x^{\langle t \rangle}$ is a scalar or a low dimensional vector. Neural network models, the topic of this chapter, are very different, and for an early review of recurrent neural networks and generalizations see chapter 10 of [142] and the many references there-in, where key references are also listed below.

As the most common application of sequence models is textual data, let us mention early texts on *natural language processing (NLP)*. General early approaches to NLP are summarized in [280] and [216] where the topic is tackled via rule-based approaches based on the statistics of grammar. A much more modern summary of applications is [228] and a review of applications of neural networks for NLP is in [138], yet this field is quickly advancing at the time of publishing of this current book. See also chapter 7 of the book [4] for a comprehensive discussion of RNNs as well as their *long short term memory (LSTM)* and *gated recurrent units* (GRU) generalizations.

Recurrent neural networks (RNN) are useful for broad applications such as DNA sequencing, see for example [375], image captioning as in [188], time series prediction as in [22], sentiment analysis as in [274], speech recognition as in [146], and many other applications. Possibly one of the first constructions of recurrent neural networks (RNN) in their modern form appeared in [117] and is sometimes referred to as an *Elman network*. Yet this was not the inception of ideas for recurrent neural networks and earlier ideas appeared in several influential works over the previous decades. See [367] for an historical account with notable earlier publications including [13] in 1972, and [185], and [357] in the 1980's.

The introduction of *bidirectional RNN* is in [369]. The introduction of long short term memory (LSTM) models in the late 1990's was in [184]. Gated recurrent units (GRUs) are much more recent concepts and were introduced in [80] and [85] after the big spark of interest in deep learning occurred. An empirical comparison of these various approaches is in [214]. A more contemporary review of LSTM is in [438]. These days, for advanced NLP tasks LSTMs and GRUs are generally outperformed by *transformer models*, yet in non-NLP applications we expect to see LSTMs remain a useful tool for many years to come. Some recent example application papers include [220], [295], [351], and [446], among many others.

Moving onto textual data, the idea of *word embeddings* is now standard in the world of NLP. The key principle originates with the *word2vec* work in [288]. Word embedding was further developed with *GloVe* in [327]. These days when considering dictionaries, lexicons, tokenizations, and word embeddings, one may often use dedicated libraries such as for example those supplied (and contributed to) with *HuggingFace*.[16] An applied book in this domain is [403] and since the field is moving quickly, many others are to appear as well.

The modern neural encoder-decoder approach was pioneered by [218] and then in the context of machine translation, influential works are [81] and [392]. The idea of using attention in recent times, first for *handwriting recognition*, was proposed in [145] and then the work in [20] extended the idea, and applied it to machine translation as we illustrate in our Figure 7.10. A massive advance was with the 2017 paper, "Attention is all you need", [410], which introduced the transformer architecture, the backbone of almost all of today's leading large language models. Ideas of *layer normalization* are from [19]. Further details of transformers can be found in [331], and a survey of variants of transformers as well as non-NLP applications can be found in [262].

At the time of publishing of this book the hottest topics in the world of deep learning are large language models and their multi-modal counterparts. A recent comprehensive survey is in [449], and other surveys are [73] and [155]. We should note that as this particular field is moving very rapidly at the time of publication of the book, there will surely be significant advances in the years coming. *Multimodal models* are being developed and deployed as well, and these models have images as input and output in addition to text; see [428] for a survey. Indeed beyond the initial task of machine translation, transformers have also been applied to images with incredible success. A first landmark paper on this avenue is [107]. See also the survey papers [227] and [269].

[16] https://huggingface.co.

While this list is certainly non-exhaustive, we also mention some of the key large language model architectures that emerged following the “Attention is all you need” paper [410]. Some of these are *BERT* [103], *Roberta* [268], *XLNET* [433], *GPT-2*[340] *GPT-3* [66]. Other notable LLMs are *GLaM* [110], *Gopher* [341], *Chinchilla* [341], *Megatron-Turing NLG* [382], and *LaMDa* [400]. The topic of training and using these models is beyond our scope.