Chapter 3

Machine Translation

3.1 Problem

Machine translation (MT) is one of the very oldest problems in computer science. Many different approaches to MT have been tried, and here we will look at the statistical approach, which learns translation models from *parallel text*.

Parallel text is a corpus of text that expresses the same meaning in two (or more) different languages. Usually, we assume that a parallel text has already been *sentence-aligned*, that is, it consists of *sentence pairs*, each of which expresses the same meaning in two languages. In the original work on statistical machine translation, done at IBM (Brown et al., 1993), the source language was French (f) and the target language was English (e), and we'll use those variables even for other language pairs. A slightly more general (but still Anglocentric) interpretation is that f stands for "foreign." In our example, f is Spanish and e is English.

Here is an example parallel text (Knight, 1999):

1. Garcia and associates
   García y asociados

2. his associates are not strong
   sus asociados no son fuertes

The original paper defined models for translating French to English as follows:

$$P(f, e) = P(e)\, P(f \mid e) \tag{3.1}$$

$$e^* = \arg\max_e P(e \mid f) \tag{3.2}$$

$$= \arg\max_e \frac{P(e, f)}{P(f)} \tag{3.3}$$

$$= \arg\max_e P(e, f) \tag{3.4}$$

$$= \arg\max_e P(e)\, P(f \mid e). \tag{3.5}$$

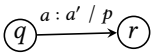
The term $P(e)$, called the language model, was responsible for ensuring that the output was fluent English, and the term $P(f \mid e)$, called the translation model,

was responsible for ensuring that the English was faithful to the French. This division of labor allows each part to do its job well. But neural networks are rather good at doing two jobs at the same time, and so modern MT systems don't take a noisy-channel approach. Instead, they directly model $P(e \mid f)$. Consequently, in this chapter, we will only consider models of $P(e \mid f)$.

3.2 How About...

3.2.1 Finite transducers? (no)

A finite-state transducer is like a finite-state automaton, but has both an input alphabet Σ and an output alphabet Σ'. The transitions look like this:



where $a \in \Sigma \cup \{\epsilon\}$, $a' \in \Sigma' \cup \{\epsilon\}$, and p is the weight. The ϵ stands for the empty string, so a transition $a : \epsilon$ means "delete input symbol a," and $\epsilon : a'$ means "insert output symbol a'."

Weighted finite transducers have been used with huge success in speech processing, morphology, and other tasks (Mohri, 1997; Mohri, Pereira, and Riley, 2002), and we'll have more to say about them later on when we talk about those tasks. Given their success, it might seem that finite transducers would be a great way to model translation. But a major limitation of transducers is that they only allow limited reordering. For example, there's no such thing as a transducer that inputs a string and outputs the reverse string. Despite valiant efforts to make them work for machine translation (Kumar and Byrne, 2003), they do not seem to be the right tool.

3.2.2 RNN language models? (also no)

Another idea would be to introduce a special symbol, say, SEP, and train a language model on strings of the form f SEP e. Then, when given a new source sentence f, use the language model to find the best completion:

$$e^* = \arg\max_e P(f \text{ SEP } e).$$

With an n-gram language model, this definitely won't work, but with an LSTM RNN, it kind of does work (Sutskever, Vinyals, and Le, 2014); a helpful trick is to reverse f. In such a model, the state of the model after reading SEP is a vector that can be thought of as representing the entire sentence f. But, as Ray Mooney likes to say, "You can't cram the meaning of a whole %&!$# sentence into a single $&!#* vector."[1]

3.3 IBM Models

Instead, we turn to a series of five models invented at IBM in their original work on statistical machine translation (Brown et al., 1993).

[1] https://www.cs.utexas.edu/~mooney/cramming.html

3.3.1 Word alignment

The IBM models are models of $P(e \mid f)$ that make the simplifying assumption that each English word depends on exactly one Spanish word. For example:

1. García y asociados EOS
   | | | |
   Garcia and associates EOS

2. sus asociados no son fuertes EOS
   | | \ | | |
   his associates are not strong EOS

(We've made some slight changes compared to the original paper. Originally, e did not end with EOS, and there was a different way to decide when to stop generating e. And f did have EOS, but it was called NULL.)

More formally: let Σ_f and Σ_e be the Spanish and English vocabularies, and

- $f = f_1 \cdots f_n$ range over Spanish sentences ($f_n = \texttt{EOS}$)
- $e = e_1 \cdots e_m$ range over English sentences ($e_m = \texttt{EOS}$)
- $a = (a_1, \ldots, a_m)$ range over possible many-to-one alignments, where each $1 \leq a_i \leq n$ and $a_i = j$ means that English word i is aligned to Spanish word j.

We will use these variable names throughout this chapter. Remember that e, i, and m come alphabetically before f, j, and n, respectively.

Thus, for our two example sentences, we have

1. f = García y asociados EOS $\quad n = 4$
   e = Garcia and associates EOS $\quad m = 4$
   $a = (1, 2, 3, 4)$

2. f = sus asociados no son fuertes EOS $\quad n = 6$
   e = his associates are not strong EOS $\quad m = 6$
   $a = (1, 2, 4, 3, 5, 6)$.

These alignments a will be included in our "story" of how a Spanish sentence f becomes an English sentence e. In other words, we are going to define a model of $P(e, a \mid f)$, not $P(e \mid f)$, and training this model will involve summing over all alignments a:

$$\text{maximize } L = \sum_{(f,e) \in \text{data}} \log P(e \mid f) \tag{3.6}$$

$$= \sum_{(f,e) \in \text{data}} \log \sum_{a} P(e, a \mid f). \tag{3.7}$$

(This is similar to training of NFAs in the previous chapter, where there could be more than one accepting path for a given training string.)

3.3.2 Model 1

IBM Model 1 goes like this.

1. Generate each alignment $a_1, \ldots, a_m$, each with uniform probability $\frac{1}{n}$.
2. Generate English words $e_1, \ldots, e_m$, each with probability $t(e_i \mid f_{a_i})$.

In equations, the model is:

$$P(e, a \mid f) = \prod_{i=1}^{m} \left(\frac{1}{n} t(e_i \mid f_{a_i}) \right). \tag{3.8}$$

The parameters of the model are the word-translation probabilities $t(e \mid f)$. We want to optimize these parameters to maximize the log-likelihood,

$$L = \sum_{(f,e) \in \text{data}} \log \sum_{a} P(e, a \mid f). \tag{3.9}$$

The summation over a is over an exponential number of alignments (n^m, to be exact), but we can rearrange it to make it efficiently computable:

$$\sum_{a} P(e, a \mid f) = \sum_{a} \prod_{i=1}^{m} \left(\frac{1}{n} t(e_i \mid f_{a_i}) \right) \tag{3.10}$$

$$= \sum_{a_1=1}^{n} \cdots \sum_{a_m=1}^{n} \frac{1}{n} t(e_1 \mid f_{a_1}) \cdots \frac{1}{n} t(e_m \mid f_{a_m}) \tag{3.11}$$

$$= \sum_{a_1=1}^{n} \frac{1}{n} t(e_1 \mid f_{a_1}) \cdots \sum_{a_m=1}^{n} \frac{1}{n} t(e_m \mid f_{a_m}) \tag{3.12}$$

$$= \prod_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} t(e_i \mid f_j). \tag{3.13}$$

The good news is that this objective function is *convex*, that is, every local maximum is a global maximum. The bad news is that there's no closed-form solution for this maximum, so we must use some iterative approximation. The classic way to do this is expectation-maximization, but we can also use stochastic gradient ascent. The trick is ensuring that the t probabilities sum to one. We do this by defining a matrix $\mathbf{T}$ with an element for every pair of Spanish and English words. The elements are unconstrained real numbers (called *logits*), and are the new parameters of the model. Then we can use the softmax function to change them into probabilities, which we use as the t probabilities.

$$\mathbf{T} \in \mathbb{R}^{|\Sigma_e| \times |\Sigma_f|} \tag{3.14}$$

$$t(e_i \mid f_j) = \left[\text{softmax}\, \mathbf{T}_{*,f_j} \right]_{e_i} \tag{3.15}$$

$$= \frac{\exp \mathbf{T}_{e_i,f_j}}{\sum_{e' \in \Sigma_e} \exp \mathbf{T}_{e',f_j}}. \tag{3.16}$$

For large datasets, the vast majority of (Spanish word, English word) pairs never cooccur (that is, in the same sentence pair), which means that the vast majority of entries of $\mathbf{T}$ would be $-\infty$. So to make this practical, we'd have to store $\mathbf{T}$ as a sparse matrix.

3.3.3 Model 2 and beyond

In Model 1, we chose each a_i with uniform probability $1/n$, which makes for a very weak model. For example, it's unable to learn that the first English word is more likely to depend on the first Spanish word than (say) the seventh Spanish word. In Model 2, we replace $1/n$ with a learnable parameter:

$$P(e, a \mid f) = \prod_{i=1}^{m} \left(\alpha(a_i \mid i)\, t(e_i \mid f_{a_i})\right).$$

where, for each j up to some maximum Spanish length N, and for each i up to some maximum English length M, the parameter $\alpha(j \mid i)$ must be learned. Then we can learn that (say) $\alpha(1 \mid 1)$ is high, but $\alpha(7 \mid 1)$ is low.

- In the original paper, α is called a, but I renamed it to avoid confusion with the random variable a.
- In the original paper, α was also conditioned on m and n. In fact, α technically has to depend on n in order to be a probability distribution ($\sum_{j=1}^{n} \alpha(j \mid i, n) = 1$). But I'm trying to keep things simple.

There are also Models 3, 4, and 5, which can learn dependencies between the a_i, like:

- Distortion: Even if the model gives low probability to $a_1 = 7$, it should be the case that given $a_1 = 7$, the probability that $a_2 = 8$ is high, because it's common for a block of words to move together.
- Fertility: It should be most common for one English word to align to one Spanish word, less common for zero or two English words to align to one Spanish word, and extremely rare for ten English words align to one Spanish word.

But for our purposes, it's good enough to stop here at Model 2.

To train Model 2 by stochastic gradient ascent, we again need to express the α probabilities in terms of unconstrained parameters:

$$\mathbf{A} \in \mathbb{R}^{N \times M} \tag{3.17}$$

$$\alpha(j \mid i) = [\operatorname{softmax} \mathbf{A}_{1:n,i}]_i \tag{3.18}$$

$$= \frac{\exp \mathbf{A}_{j,i}}{\sum_{j'=1}^{n} \exp \mathbf{A}_{j',i}}. \tag{3.19}$$

3.4 From Alignment to Attention

In this section, we are going to start to modify IBM Model 1/2 to make it more powerful and more like a neural network. See Figure 3.1a for a picture of IBM Model 1 drawn in the style of a neural network.

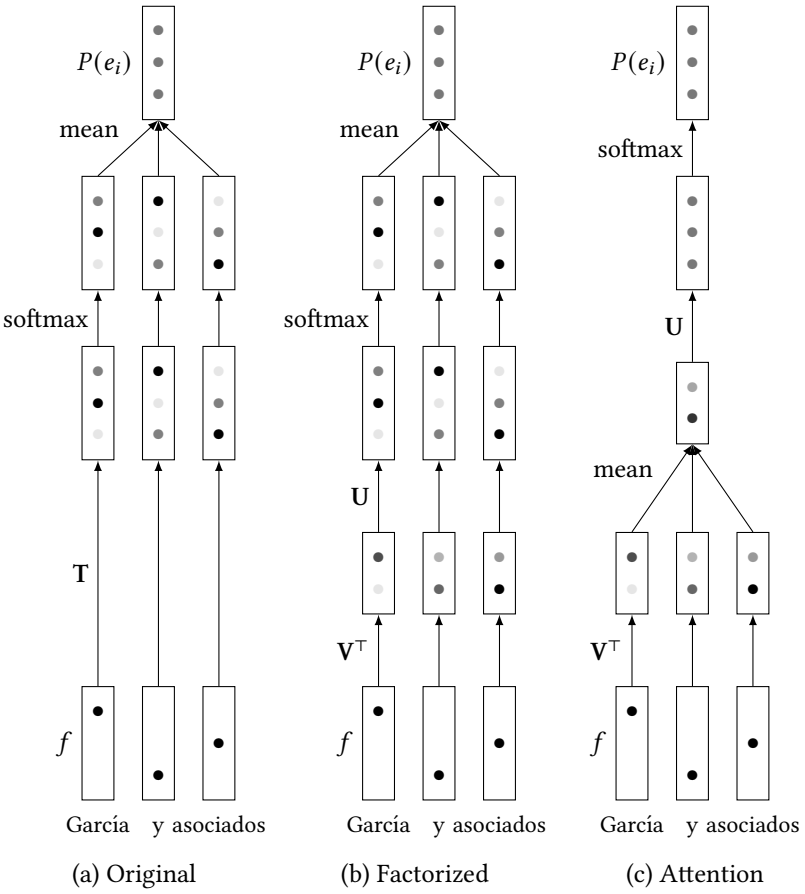


Figure 3.1: Variations of IBM Model 1, pictured as a neural network.

3.4.1 Word embeddings

Above, we mentioned that matrix $\mathbf{T}$ is very large and sparse. We can overcome this by factoring it into two smaller matrices (see Figure 3.1b):

$$\mathbf{U} \in \mathbb{R}^{|\Sigma_e| \times d} \tag{3.20}$$

$$\mathbf{V} \in \mathbb{R}^{|\Sigma_f| \times d} \tag{3.21}$$

$$\mathbf{T} = \mathbf{U}\mathbf{V}^\top \tag{3.22}$$

where d is some number that we have to choose.

So the model now looks like

$$P(e \mid f) = \prod_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} \left[\text{softmax}\, \mathbf{U}\mathbf{V}_{f_j}\right]_{e_i} \tag{3.23}$$

If you think of $\mathbf{T}$ as transforming Spanish words into English words (more precisely, logits for English words), we're splitting this transformation into two steps. First, $\mathbf{V}$ maps the Spanish word into a size-d vector, called a *word embedding*. This transformation $\mathbf{V}$ is called an *embedding layer* because it embeds the Spanish vocabulary into the vector space $\mathbb{R}^d$ which is (somewhat sloppily) called the *embedding space*.

Second, $\mathbf{U}$ transforms the hidden vector into a vector of logits, one for each English word. This transformation $\mathbf{U}$, together with the softmax, are known as a *softmax layer*. The rows of $\mathbf{U}$ can also be thought of as embeddings of the English words.

In fact, we can think of $\mathbf{U}$ and $\mathbf{V}$ as embedding both the Spanish and English vocabularies into the *same* space. If e_i and f_j are translations of each other, then they should (hopefully) have a high logit $\mathbf{T}_{e_i,f_j}$, and $\mathbf{T}_{e_i,f_j} = \mathbf{U}_{e_i} \cdot \mathbf{V}_{f_j}$, so the embeddings of e_i and f_j will (hopefully) have a high dot-product. Recall that the dot product of two vectors is related to the angle between the two vectors, so e_i and f_j will (hopefully) have vectors that point in the same direction.

Figure 3.2 shows that if we run factored Model 1 on a tiny Spanish-English corpus (Knight, 1999) and normalize the Spanish and English word embeddings, words that are translations of each other do lie close to each other.

The choice of d matters. If d is large enough (at least as big as the smaller of the two vocabularies), then $\mathbf{U}\mathbf{V}^\top$ can compute any transformation that $\mathbf{T}$ can. But if d is smaller, then $\mathbf{U}\mathbf{V}^\top$ can only be an approximation of the full $\mathbf{T}$ (called a *low-rank approximation*). This is a good thing: not only does it solve the sparse-matrix problem, but it can also generalize better. Imagine that we have training examples

1. El perro es grande.
   The dog is big.

2. El perro es gigante.
   The dog is big.

3. El perro es gigante.
   The dog is large.

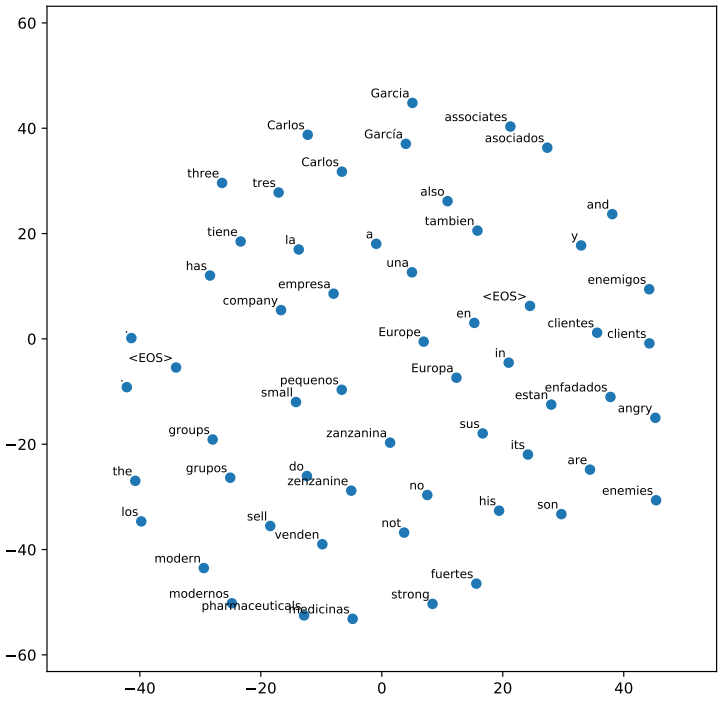


Figure 3.2: Two-dimensional visualization of the 64-dimensional word embeddings learned by the factored Model 1. The embeddings were normalized and then projected down to two dimensions using t-SNE (Maaten and Hinton, 2008). In most cases, the Spanish word embedding is close to its corresponding English word embedding.

Because *gigante* and *large* do not occur in the same sentence pair, the original Model 1 would not be able to learn a nonzero probability for $t(\text{gigante} \mid \text{large})$. But the factorized model would map both *grande* and *gigante* to nearby embeddings (because both translate to *big*), and map that region of the space to *large* (because *gigante* translates to *large*). Thus it would learn a nonzero probability for $t(\text{gigante} \mid \text{large})$.

3.4.2 Attention

To motivate the next change, consider the Spanish-English sentence pairs

1. por qué EOS
   why EOS
2. por qué EOS
   why EOS
3. por qué EOS
   why EOS
4. por EOS
   for EOS
5. qué EOS
   what EOS

Here are the probabilities that Model 1 learns:

	por	qué	EOS
why	0.49	0.49	0
for	0.33	0	0
what	0	0.33	0
EOS	0.18	0.18	1

It learns a high probability for both $t(\text{why} \mid \text{por})$ and $t(\text{why} \mid \text{qué})$. Remember that Model 1 averages probabilities uniformly. If we average the columns for each Spanish sentence, we get: added 2023-09-18

	por qué EOS	por EOS	qué EOS
why	0.33	0.25	0.25
for	0.11	0.17	0
what	0.11	0	0.17
EOS	0.45	0.59	0.59

These probabilities are high enough that if we ask the model to re-translate *qué* EOS, it will prefer the translation *why* over *what*. What went wrong?

When Model 1 looks at the first sentence, it imagines that there are two variants of this sentence, one in which *why* is translated from *por* and one in which *why* is translated from *qué*. It has no notion of *why* being translated from both *por* and *qué*. Remember that something similar happened when we tried to write

an NFA for pangrams: we wanted a word that contained *a* and *b* and *c*, etc., but could only write an NFA that contained *a* or *b* or *c*, etc.

We can fix this if we move the average ($\sum_{j=1}^{n} \frac{1}{n}(\cdot)$) inside the softmax:

$$P(e \mid f) = \prod_{i=1}^{m} \left[\text{softmax}\left(\sum_{j=1}^{n} \frac{1}{n} \mathbf{U}\mathbf{V}_{f_j} \right) \right]_{e_i} . \tag{3.24}$$

Note that whereas factoring **T** was an approximation to the original model, this is a drastic change to the model. How does it help? Here's a near-optimal solution for the logits (**UV**):

Corrected 2023-09-18

	por	qué	EOS
why	60	60	0
for	90	0	0
what	0	90	0
EOS	30	30	60

Because we're now averaging logits, not probabilities, there's a lot more room for words to influence one another's translations. If we average the columns for each Spanish sentence, we get:

	por qué EOS	por EOS	qué EOS
why	40	40	0
for	30	45	0
what	0	30	45
EOS	40	45	45

If *por* is by itself, then *for* is the best translation by a lot (5). Similarly if *qué* is by itself. But if *por* and *qué* occur together, the score for *why* goes up to 40, which is the best translation by a lot (10).

Why is a margin of 5 "a lot"? Because the softmax has an exp in it, so a margin of 5 becomes a factor of $\exp 5 \approx 150$. After taking the softmax, we get something very close to

	por qué EOS	por EOS	qué EOS
why	0.5	0.0	0.0
for	0.0	0.5	0.0
what	0.0	0.0	0.5
EOS	0.5	0.5	0.5

Since everything inside the softmax is linear, we can move the average to wherever we want. Let's move it to in between **U** and **V**:

$$P(e \mid f) = \prod_{i=1}^{m} \left[\text{softmax}\left(\mathbf{U} \sum_{j=1}^{n} \frac{1}{n} \mathbf{V}_{f_j} \right) \right]_{e_i} . \tag{3.25}$$

This model is shown in Figure 3.1c. The averaging of the Spanish word embeddings $\mathbf{V}_{f_j}$ is an example of *attention*, and each English word is said to *attend* to

the Spanish words. The vectors being averaged are called *value* vectors. The resulting average is sometimes called a *context* vector, and can be thought of as a vector representation of the whole sentence f. Are we once again cramming the meaning of a whole %&!$# sentence into a single $&!#* vector? Yes. But moving to Model 2 will change that.

3.4.3 Position embeddings

Recall that the difference between Model 1 and Model 2 is that we changed the uniform average into a weighted average, weighted by the parameters $\alpha(j \mid i)$. Similarly, here, we can make the uniform average into a weighted average:

$$P(e \mid f) = \prod_{i=1}^{m} \left[\text{softmax}\left(\mathbf{U} \sum_{j=1}^{n} \alpha(j \mid i)\, \mathbf{V}_{f_j} \right) \right]_{e_i}. \tag{3.26}$$

The weighted average $\sum_j \alpha(j \mid i)\, \mathbf{V}_{f_j}$ is once again a vector representation of f, but now, this representation is different for each time step i.

The weights $\alpha(j \mid i)$ are called *attention weights*. In Model 2, we expressed them in terms of logits $\mathbf{A}$, which were learned as parameters of the model. Here, we factor $\mathbf{A}$ just as we factored $\mathbf{T}$:

Equation (3.29) corrected 2023-09-20

$$\mathbf{Q} \in \mathbb{R}^{M \times d} \tag{3.27}$$

$$\mathbf{K} \in \mathbb{R}^{N \times d} \tag{3.28}$$

$$\alpha(j \mid i) = \left[\text{softmax}\, \mathbf{A}_{1:n,i} \right]_j \tag{3.29}$$

$$\mathbf{A} = \mathbf{K}\mathbf{Q}^\top \tag{3.30}$$

The vectors $\mathbf{Q}_i$ are learned as parameters of the model. They can be thought of as vector representations of the positions i, or *position embeddings* (Gehring et al., 2017). Similarly, the vectors $\mathbf{K}_j$, which are also learned parameters, can be thought of as embeddings of positions j. Because $\mathbf{A}_{ij} = \mathbf{K}_i \cdot \mathbf{Q}_j$, the model makes English position i attend most strongly to Spanish positions j where the dot-product between the embeddings of i and j are highest. Thus (3.30) is known as *dot-product attention*.

The vectors $\mathbf{Q}_i$ are also known as *query* vectors, and the $\mathbf{K}_j$ as *key* vectors. Here, the queries and keys are position embeddings, and recall that the value vectors are Spanish word embeddings. Below, we will see that other choices for queries, keys, and values are possible.

3.5 Neural Machine Translation

Our modified Model 2 (eqs. 3.26–3.30) is still not a credible machine translation system. Its ability to model context on both the source side and target side is very weak. But there have been two very successful extensions of this model, which we describe in this section.

3.5.1 Remaining problems

The most glaring problem with our modified Model 2 is that it outputs probability distributions for each English word, $P(e_i \mid f)$, but the English words are all independent of one another. The string *el río Jordan* can be translated as *the river Jordan* or *the Jordan river*, so if

$$P(e_2 = \text{river} \mid \text{el río Jordan}) = 0.5 \quad P(e_3 = \text{Jordan} \mid \text{el río Jordan}) = 0.5 \tag{3.31}$$

$$P(e_2 = \text{river} \mid \text{el río Jordan}) = 0.5 \quad P(e_3 = \text{Jordan} \mid \text{el río Jordan}) = 0.5 \tag{3.32}$$

then the translations *the river river* and *the Jordan Jordan* will be just as probable as *the river Jordan* and *the Jordan river*. To fix this problem, we need to make the generation of e_i depend on the previous English words. In the original noisy-channel approach ($P(f \mid e)\, P(e)$), modeling dependencies between English words was the job of the language model ($P(e)$), but we threw the language model out when we switched to a direct approach ($P(e \mid f)$).

Likewise, on the source side, although we've argued that our modified Model 2 can, to a certain extent, translate multiple words like *por qué* at once, it's not very sensitive to word order. Indeed, if the model attends equally to both words, it cannot distinguish at all between *por qué* and *qué por*. So we'd like to make the encoding of a Spanish word also take into account its surrounding context.

3.5.2 Preliminaries

Please note that my descriptions of these models are highly simplified. They're good enough to get the main idea and to do the homework assignment on machine translation, but if you should ever need to implement a full-strength translation model, please consult the original papers or the many online tutorials about them.

Even simplified, these networks get rather large. To make their definitions more manageable, we break them up into functions. These functions usually have learnable parameters, and to make it unambiguous which function calls share parameters with which, we introduce the following notation. If a function's name has a superscript that looks like $f^{[\ell]}$, then its definition may contain a parameter with the same superscript, like $x^{[\ell]}$. The ℓ stands for 1, 2, etc., so if we call $f^{[1]}$ twice, the same parameter $x^{[1]}$ is shared across both calls. But if we call $f^{[1]}$ and $f^{[2]}$, they have two different parameters $x^{[1]}$ and $x^{[2]}$. (In PyTorch, such functions would be implemented as `Modules`.)

So, we can define some functions:

$$\text{Embedding}^{[\ell]}(k) = \mathbf{E}^{[\ell]}_k \tag{3.33}$$

$$\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \sum_j \left[\text{softmax}\, \mathbf{Kq}\right]_j \mathbf{V}_j \tag{3.34}$$

$$\text{LinearLayer}^{[\ell]}(\mathbf{x}) = \mathbf{W}^{[\ell]}\mathbf{x} + \mathbf{b}^{[\ell]}. \tag{3.35}$$

And now our modified Model 2 (eqs. 3.26–3.30) can be written as:

For $j = 1, \ldots, n$:

$$\mathbf{V}_j = \text{Embedding}^{\boxed{1}}(f_j) \tag{3.36}$$
$$\mathbf{K}_j = \text{Embedding}^{\boxed{2}}(j) \tag{3.37}$$

For $i = 1, \ldots, m$:

$$\mathbf{q}^{(i)} = \text{Embedding}^{\boxed{3}}(i) \tag{3.38}$$
$$\mathbf{c}^{(i)} = \text{Attention}(\mathbf{q}^{(i)}, \mathbf{K}, \mathbf{V}) \tag{3.39}$$
$$P(e_i) = \text{softmax}\left(\text{LinearLayer}^{\boxed{4}}(\mathbf{c}^{(i)})\right). \tag{3.40}$$

It will be useful later to wrap Attention inside a CrossAttentionLayer that first applies three linear transformations to the queries, keys, and values: Updated 2023-09-22

$$\text{CrossAttentionLayer}^{\boxed{\ell}}(\mathbf{X}, \mathbf{y}) = \text{Attention}(\mathbf{W}_{\text{Q}}^{\boxed{\ell}}\mathbf{y}, \mathbf{K}, \mathbf{V}) \tag{3.41}$$
$$\text{where } \mathbf{K}_j = \mathbf{W}_{\text{K}}^{\boxed{\ell}}\mathbf{X}_j \tag{3.42}$$
$$\mathbf{V}_j = \mathbf{W}_{\text{V}}^{\boxed{\ell}}\mathbf{X}_j. \tag{3.43}$$

3.5.3 Using RNNs

The first way to introduce more context sensitivity (Bahdanau, Cho, and Bengio, 2015) is to insert an RNN on both the source and target side (see Figure 3.3). These RNNs are called the *encoder* and *decoder*, respectively.

In addition to the functions defined above, we need a function to compute one step of an RNN:

$$\text{RNNCell}^{\boxed{\ell}}(\mathbf{h}, \mathbf{x}) = \tanh(\mathbf{A}^{\boxed{\ell}}\mathbf{h} + \mathbf{B}^{\boxed{\ell}}\mathbf{x} + \mathbf{c}^{\boxed{\ell}}). \tag{3.44}$$

Now, the model is defined as follows. For $j = 1, \ldots, n$, we compute a sequence of source word embeddings $\mathbf{v}^{(j)} \in \mathbb{R}^d$, and use an RNN to compute a sequence of vectors $\mathbf{h}^{(j)} \in \mathbb{R}^d$:

$$\mathbf{h}^{(0)} = \text{RNNCell}^{\boxed{2}}(\mathbf{0}, \text{Embedding}^{\boxed{1}}(\texttt{BOS})) \tag{3.45}$$
$$\mathbf{v}^{(j)} = \text{Embedding}^{\boxed{1}}(f_j) \qquad j = 1, \ldots, n \tag{3.46}$$
$$\mathbf{h}^{(j)} = \text{RNNCell}^{\boxed{2}}(\mathbf{h}^{(j-1)}, \mathbf{v}^{(j)}) \qquad j = 1, \ldots, n \tag{3.47}$$

It will be convenient to pack the rest of the $\mathbf{h}^{(j)}$ into a single matrix,

$$\mathbf{H} \in \mathbb{R}^{n \times d}$$
$$\mathbf{H} = [\mathbf{h}^{(1)} \cdots \mathbf{h}^{(n)}]^\top. \tag{3.48}$$

Usually fancier RNNs (using GRUs or LSTMs) are used instead of a simple RNN as shown here. Also, it's quite common to stack up several RNNs, with the output of one feeding into the input of the next.

The decoder RNN varies more from model to model; the one shown here is most similar to that of Luong, Pham, and Manning (2015). Like the encoder, it

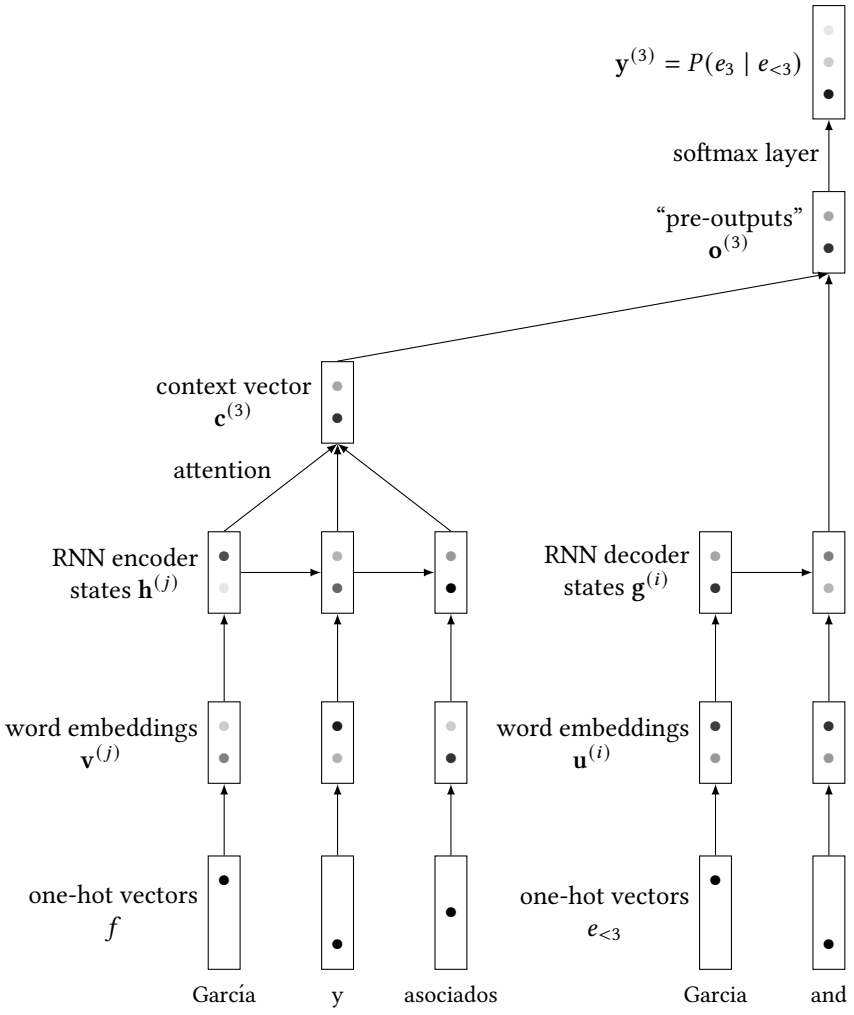


Figure 3.3: Simplified diagram of an RNN translation model (Bahdanau, Cho, and Bengio, 2015; Luong, Pham, and Manning, 2015), at time step $i = 3$. Some details (BOS, queries/keys) are omitted.

computes a sequence of vectors $\mathbf{g}^{(i)}$:

$$\mathbf{g}^{(0)} = \text{RNNCell}^{[4]}(\mathbf{0}, \text{Embedding}^{[3]}(\texttt{BOS})) \tag{3.49}$$

$$\mathbf{u}^{(i)} = \text{Embedding}^{[3]}(e_i) \qquad i = 1, \ldots, m \tag{3.50}$$

$$\mathbf{g}^{(i)} = \text{RNNCell}^{[4]}(\mathbf{g}^{(i-1)}, \mathbf{u}^{(i)}) \qquad i = 1, \ldots, m. \tag{3.51}$$

Rather than immediately trying to predict an output word, we first use an attention layer to compute a context vector:

$$\mathbf{c}^{(i)} \in \mathbb{R}^d$$
$$\mathbf{c}^{(i)} = \text{CrossAttentionLayer}^{[5]}(\mathbf{H}, \mathbf{g}^{(i-1)}). \tag{3.52}$$

Using the Spanish encodings ($\mathbf{H}$) as the keys and values is very standard, whereas the choice of queries varies. For simplicity, we're using the most recent English word's encoding ($\mathbf{g}^{(i-1)}$).

So we have an English encoding $\mathbf{g}^{(i-1)}$ that summarizes the English sentence so far ($e_1 \cdots e_{i-1}$), and a context vector $\mathbf{c}^{(i)}$ that summarizes the Spanish sentence. We just add them to get a single vector: Modified 2023-09-18

$$\mathbf{o}^{(i)} \in \mathbb{R}^d$$
$$\mathbf{o}^{(i)} = \mathbf{c}^{(i)} + \mathbf{g}^{(i-1)} \tag{3.53}$$

And finally we predict an English word:

$$P(e_i) = \text{softmax}\left(\text{LinearLayer}^{[6]}(\mathbf{o}^{(i)})\right). \tag{3.54}$$

Important implementation note: Whereas the encoder can be written either left-to-right or bottom-up, the decoder has to be written left-to-right; that is, the order of computation must be: $\mathbf{c}^{(1)}, \mathbf{o}^{(1)}, P(e_1), \mathbf{u}^{(1)}, \mathbf{g}^{(1)}, \mathbf{c}^{(2)}$, etc.

3.5.4 Using self-attention: Transformers

The other successful neural translation model, which is the current state of the art, is called the Transformer (Vaswani et al., 2017). The key idea here is to recognize that attention is not just useful for linking the source and target sides of the model; it can transform a sequence into a sequence of the same length, and therefore be used as a replacement for RNNs (Figure 3.4).

We define a new *self-attention* layer, which applies three different linear transformations to the same sequence of vectors to get queries, keys, and values. Then it uses attention to compute a sequence of context vectors.

$$\text{SelfAttentionLayer}^{[\ell]}(\mathbf{X}) = \mathbf{C} \tag{3.55}$$

$$\text{where } \mathbf{C}_i = \text{CrossAttentionLayer}^{[\ell]}(\mathbf{X}, \mathbf{X}_i). \tag{3.56}$$

Like an RNN, it maps a sequence of n vectors to a sequence of n vectors, and so it can, in principle, be used as a drop-in replacement for an RNN.

They're not the same, though. Self-attention is better at learning long-distance dependencies, but (like Model 1) it knows nothing about word order. The solution

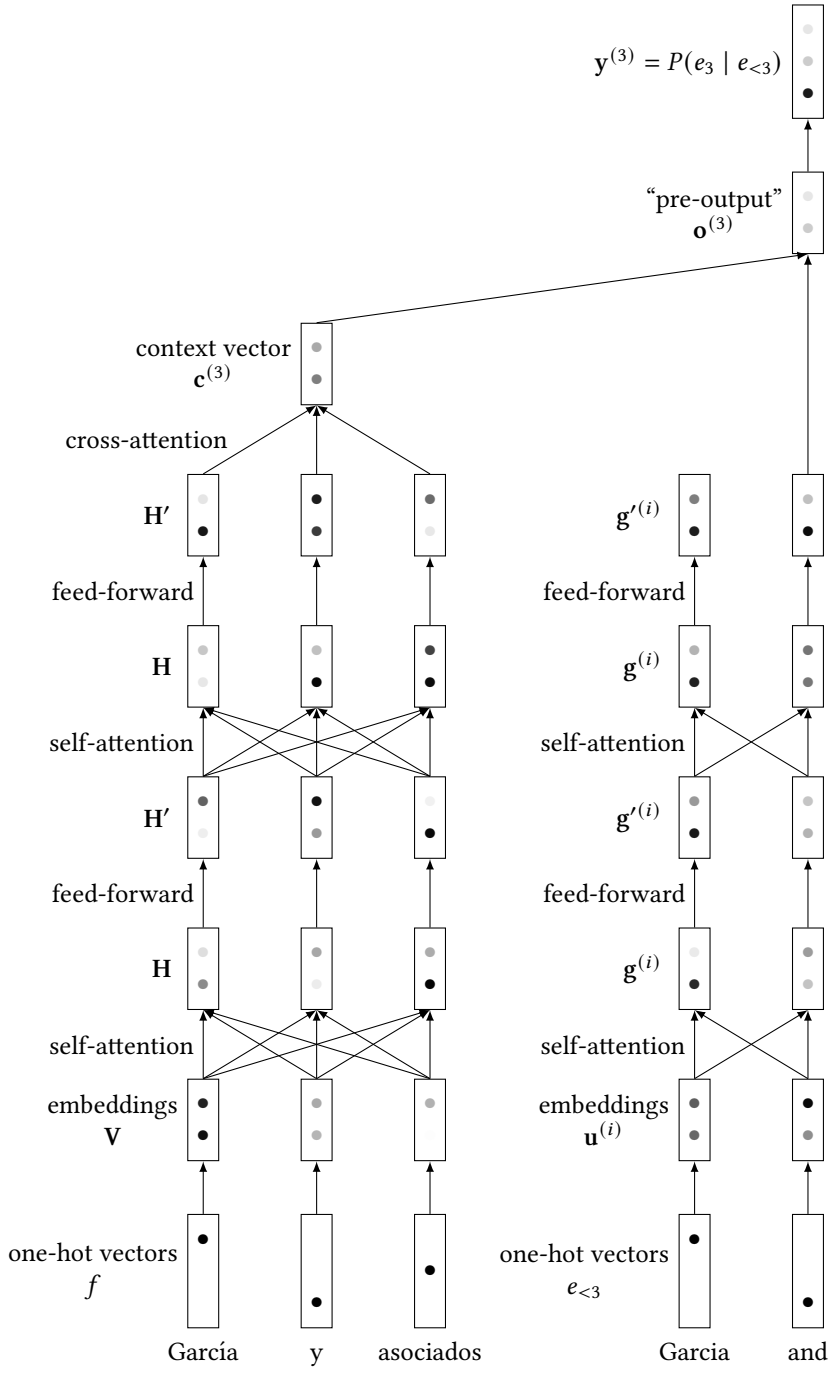


Figure 3.4: Simplified diagram of a Transformer translation model (Vaswani et al., 2017). Some details (BOS, queries/keys) are omitted. Because the encoder and decoder have two layers, some variable names get reused.

is surprisingly simple: augment word embeddings with position embeddings. Then the vector representation of a word token will depend both on the word type and its position, and the model has the potential to be sensitive to word order. Added 2023-09-22

We're also going to need a version where each word attends only to itself and to the left, known as *masked* self-attention:

$$\begin{aligned}\text{MaskedSelfAttentionLayer}^{\boxed{\ell}}(\mathbf{X}) &= \mathbf{C} \\ &\text{where } \mathbf{C}_i = \text{CrossAttentionLayer}^{\boxed{\ell}}(\mathbf{X}_{1:i}, \mathbf{X}_i).\end{aligned}$$

The model is defined as follows. We represent the source words as word embeddings plus position embeddings:

$$\begin{aligned}&\mathbf{V} \in \mathbb{R}^{n \times d} \\ &\mathbf{V}_j = \text{Embedding}^{\boxed{1}}(f_j) + \text{Embedding}^{\boxed{2}}(j) \qquad j = 1, \ldots, n \end{aligned} \tag{3.57}$$

Next comes a self-attention layer:

$$\begin{aligned}&\mathbf{H} \in \mathbb{R}^{n \times d} \\ &\mathbf{H} = \text{SelfAttentionLayer}^{\boxed{3}}(\mathbf{V}).\end{aligned} \tag{3.58}$$

The self-attention layer is always followed by a *position-wise feedforward network*:

$$\text{FFN}^{\boxed{\ell}}(\mathbf{x}) = \text{LinearLayer}^{\boxed{\ell}}(\text{ReLU}(\text{LinearLayer}^{\boxed{\ell'}}(\mathbf{x}))) \tag{3.59}$$

$$\text{ReLU}(z) = \max\{0, z\} \tag{3.60}$$

$$\begin{aligned}&\mathbf{H}' \in \mathbb{R}^{n \times d} \\ &\mathbf{H}'_j = \text{FFN}^{\boxed{4}}(\mathbf{H}_j) + \mathbf{H}_j \qquad j = 1, \ldots, n.\end{aligned} \tag{3.61}$$

The ReLU is a *rectified linear unit*, which is yet another alternative to sigmoid and tanh. The $+\,\mathbf{H}_j$ term in red is known as a *residual connection*.

Then, steps (3.58–3.61) are repeated: self-attention, position-wise feedforward, and so on, usually with 4 or 6 repetitions in total. In the equations, we've only shown 1 repetition to avoid writing $\mathbf{H}''$, $\mathbf{H}'''$, etc. Figure 3.4 shows 2 repetitions.

The decoder is also a stack of self-attention layers, and, just as with the RNN decoder, we need to write the equations as a single left-to-right pass, that is, as a single iteration over i.

Assume that the English sentence begins with $e_0 = \texttt{BOS}$ and ends with $e_m = \texttt{EOS}$. For each time step $i = 1, \ldots, n$, we have seen $e_{<i}$ (which includes BOS) and want to predict the next English word, $P(e_i \mid e_{<i})$. At the previous time step, $i-1$, we generated the English word e_{i-1}, and at the current time step, i, it becomes part of the input. So we start by computing the vector representation of e_{i-1}:

$$\begin{aligned}&\mathbf{u}^{(i-1)} \in \mathbb{R}^d \\ &\mathbf{u}^{(i-1)} = \text{Embedding}^{\boxed{5}}(e_{i-1}) + \text{Embedding}^{\boxed{6}}(i-1).\end{aligned} \tag{3.62}$$

Then this representation goes through self-attention and feedforward layers as before, but note that self-attention only operates on $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(i-1)}$, because it can't see the future. Modified 2023-09-28

$$\mathbf{g}^{(i-1)} \in \mathbb{R}^d$$
$$\mathbf{g}^{(i-1)} = \text{CrossAttentionLayer}^{[7]}([\mathbf{u}^{(0)} \cdots \mathbf{u}^{(i-1)}]^\top, \mathbf{u}^{(i-1)}) \tag{3.63}$$
$$\mathbf{g'}^{(i-1)} \in \mathbb{R}^d$$
$$\mathbf{g'}^{(i-1)} = \text{FFN}^{[8]}(\mathbf{g}^{(i-1)}) + \mathbf{g}^{(i-1)}. \tag{3.64}$$

Again, the self-attention and feedforward layers repeat, usually for the same number of times as in the encoder.

Now, just as in the RNN-based model, we have a sequence of source encodings, $\mathbf{H'}$, and a target encoding, $\mathbf{g'}^{(i-1)}$, and the rest of (our simplified version of) the model proceeds as before (cf. eqs. 3.52–3.54).

$$\mathbf{c}^{(i)} \in \mathbb{R}^d$$
$$\mathbf{c}^{(i)} = \text{CrossAttentionLayer}^{[9]}(\mathbf{H'}, \mathbf{g'}^{(i-1)}) \tag{3.65}$$
$$\mathbf{o}^{(i)} \in \mathbb{R}^d$$
$$\mathbf{o}^{(i)} = \mathbf{c}^{(i)} + \mathbf{g'}^{(i-1)} \tag{3.66}$$
$$P(e_i) = \text{softmax}(\text{LinearLayer}^{[10]}(\mathbf{o}^{(i)})). \tag{3.67}$$

The attention here is called a *cross-attention*, and in the real Transformer, there are actually multiple cross-attentions, one before each decoder self-attention. But hopefully this suffices to get the main idea across.

References

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *Proc. ICLR*. URL: https://arxiv.org/abs/1409.0473.

Brown, Peter F. et al. (1993). "The Mathematics of Statistical Machine Translation: Parameter Estimation". In: *Computational Linguistics* 19, pp. 263–311.

Gehring, Jonas et al. (2017). "Convolutional Sequence to Sequence Learning". In: *Proc. ICML*.

Knight, Kevin (1999). *A Statistical MT Tutorial Workbook*. Notes for the JHU CLSP Summer Workshop. URL: https://kevincrawfordknight.github.io/papers/wkbk.pdf.

Kumar, Shankar and William Byrne (2003). "A Weighted Finite State Transducer Implementation of the Alignment Template Model for Statistical Machine Translation". In: *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 142–149. URL: https://aclanthology.org/N03-1019.

Luong, Thang, Hieu Pham, and Christopher D. Manning (Sept. 2015). "Effective Approaches to Attention-based Neural Machine Translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, pp. 1412–1421. DOI: 10.18653/v1/D15-1166. URL: https://www.aclweb.org/anthology/D15-1166.

Maaten, Laurens van der and Geoffrey Hinton (2008). "Visualizing High-Dimensional Data Using t-SNE". In: *Journal of Machine Learning Research* 9, pp. 2579–2605.

Mohri, Mehryar (1997). *Finite-State Transducers and Language and Speech Processing*.

Mohri, Mehryar, Fernando Pereira, and Michael Riley (2002). "Weighted finite-state transducers in speech recognition". In: *Computer Speech and Language* 16, pp. 69–88.

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le (2014). "Sequence to Sequence Learning with Neural Networks". In: *Proc. NeurIPS*.

Vaswani, Ashish et al. (2017). "Attention is All You Need". In: *Proc. NeurIPS*, pp. 5998–6008. URL: https://papers.nips.cc/paper/7181-attention-is-all-you-need.