Chapter 3

Machine Translation

3.1 Problem (again)

Remember that we motivated the language modeling problem by thinking about machine translation as “deciphering” the source language into the target language.

$$
\begin{aligned}
P(f, e) &= P(e)\, P(f \mid e) && (3.1)\\
e^* &= \arg\max_e P(e \mid f) && (3.2)\\
&= \arg\max_e \frac{P(e, f)}{P(f)} && (3.3)\\
&= \arg\max_e P(e, f) && (3.4)\\
&= \arg\max_e P(e)\, P(f \mid e). && (3.5)
\end{aligned}
$$

In this chapter, we start by focusing on $P(f \mid e)$ (the translation model). We will also consider so-called *direct* models that estimate $P(e \mid f)$, in particular neural networks.

All the models we’ll look at are trained on *parallel text*, which is a corpus of text that expresses the same meaning in two (or more) different languages. Usually we assume that a parallel text is already *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 (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. Our example uses Spanish and English.

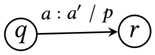
Here is an example parallel text (Knight, 1999):

1. Garcia and associates
   García y asociados

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

3.2 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 define a translation model $P(f \mid e)$. 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.

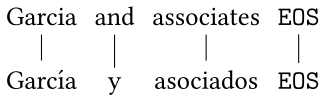
3.3 IBM Models

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

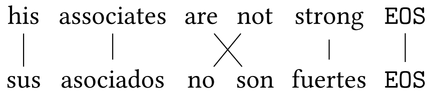
3.3.1 Word alignment

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

1.



2.



(We've made some slight changes compared to the original paper. Originally, f did not end with `EOS`, and there was a different way to decide when to stop generating f. And e 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 ($e_m = \texttt{EOS}$)

- $a = (a_1, \ldots, a_n)$ range over possible many-to-one alignments, where each $1 \leq a_j \leq m$ and $a_j = i$ means that Spanish word j is aligned to English word i.

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

$$\begin{aligned} \text{maximize } L &= \sum_{(f,e) \in \text{data}} \log P(f \mid e) && (3.6) \\ &= \sum_{(f,e) \in \text{data}} \log \sum_a P(f, a \mid e). && (3.7) \end{aligned}$$

(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 $a_1, \ldots, a_n$, each with uniform probability $\frac{1}{m}$.
2. Generate Spanish words $f_1, \ldots, f_n$, each with probability $t(f_j \mid e_{a_j})$.

In equations, the model is:

$$P(f, a \mid e) = \prod_{j=1}^{n} \left(\frac{1}{m} \, t(f_j \mid e_{a_j}) \right). \tag{3.8}$$

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

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

The summation over a is over an exponential number of alignments, but we can rearrange it to make it efficiently computable:

$$\sum_a P(f, a \mid e) = \sum_a \prod_{j=1}^{n} \left(\frac{1}{m} \, t(f_j \mid e_{a_j}) \right) \tag{3.10}$$

$$= \sum_{a_1=1}^{m} \cdots \sum_{a_n=1}^{m} \frac{1}{m} \, t(f_1 \mid e_{a_1}) \cdots \frac{1}{m} \, t(f_n \mid e_{a_n}) \tag{3.11}$$

$$= \sum_{a_1=1}^{m} \frac{1}{m} \, t(f_1 \mid e_{a_1}) \cdots \sum_{a_n=1}^{m} \frac{1}{m} \, t(f_n \mid e_{a_n}) \tag{3.12}$$

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

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 **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.

$$\mathbf{T} \in \mathbb{R}^{|\Sigma_\mathrm{f}| \times |\Sigma_\mathrm{e}|} \tag{3.14}$$

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

$$= \frac{\exp \mathbf{T}_{f_j,e_i}}{\sum_{f' \in \Sigma_\mathrm{f}} \exp \mathbf{T}_{f',e_i}}. \tag{3.16}$$

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 **T** would be $-\infty$. So to make this practical, we'd have to store **T** as a sparse matrix.

3.3.3 Model 2 and beyond

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

$$P(f, a \mid e) = \prod_{j=1}^{n} \left(\alpha(a_j \mid j, m, n) \, t(f_j \mid e_{a_j}) \right).$$

where for each i, j, m, n, the parameter $\alpha(i \mid j, m, n)$ must be learned. (In the original paper, α is called a but I renamed it to avoid confusion with the random variable a.) Then we can learn that (say) $\alpha(1 \mid 1, 10, 10)$ is high, but $\alpha(7 \mid 1, 10, 10)$ is low.

There are also Models 3, 4, and 5, which can learn dependencies between the a_j, 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 Spanish word to align to one English word, less common for zero or two Spanish words to align to one English word, and extremely rare for ten Spanish words align to one English 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. Let M and N be the maximum English and Spanish sentence length, respectively. Then:

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

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

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

3.4 From Alignment to Attention

So far, we've been working in the noisy-channel framework,

$$P(f, e) = P(e)\, P(f \mid e). \tag{3.20}$$

One reason for doing this is to divide up the translation problem into two parts so each model (language model and translation model) can focus doing its part 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 $P(e \mid f)$. Let's start by rewriting Model 1 in the direct direction:

$$P(e \mid f) = \prod_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} \left[\operatorname{softmax} \mathbf{T}_{*,f_j}\right]_{e_i}. \tag{3.21}$$

See Figure 3.1a for a picture of this model, drawn in the style of a neural network.

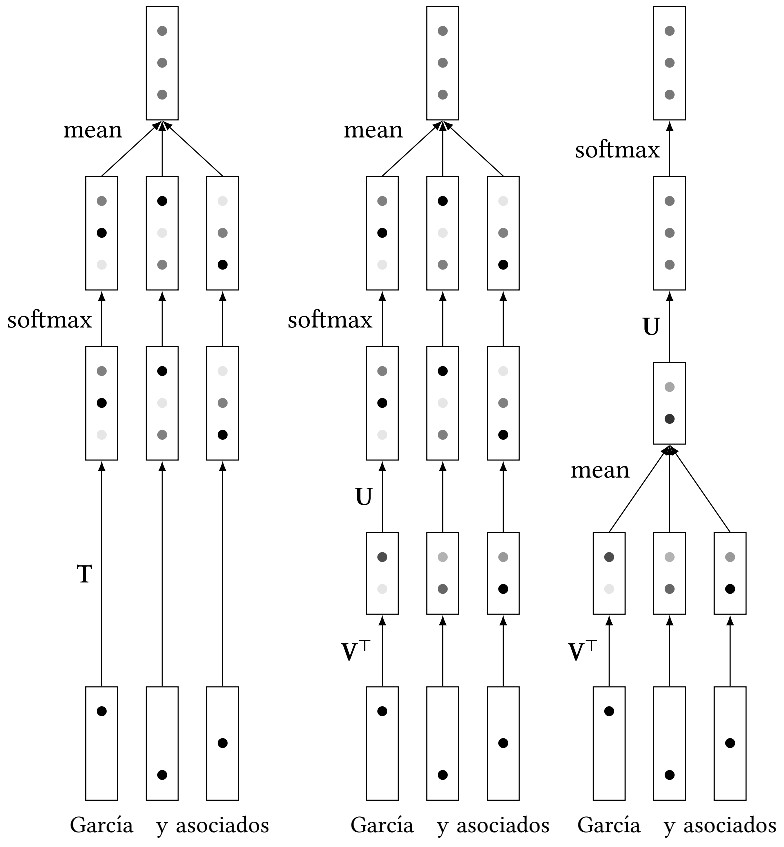
Factoring T. Above, we mentioned that matrix **T** is very large and sparse. We can overcome this by factoring it into two smaller matrices (see Figure 3.1b):

$$\mathbf{U} \in \mathbb{R}^{|\Sigma_\mathrm{e}| \times d} \tag{3.22}$$

$$\mathbf{V} \in \mathbb{R}^{|\Sigma_\mathrm{f}| \times d} \tag{3.23}$$

$$\mathbf{T} = \mathbf{U}\mathbf{V}^\top \tag{3.24}$$

where d is some number that we have to choose.



(a) Original (b) Factorized (c) Attention

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

So the model now looks like

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

If you think of **T** as transforming Spanish words into English words (more precisely, logits for English words), we're splitting this transformation into two steps. First, **V** maps the Spanish word into a size-d vector, called a *word embedding*. This transformation **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, **U** transforms the hidden vector into a vector of logits, one for each English word. This transformation **U**, together with the softmax, are known as a *softmax layer*. The rows of **U** can also be thought of as embeddings of the English words.

In fact, we can think of **U** and **V** as embedding both the Spanish and English vocabularies into the same space. In equation (3.25), if we expand the definition of softmax, we get:

$$P(e \mid f) = \prod_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} \frac{\exp[\mathbf{U}\mathbf{V}_{f_j}]_{e_i}}{\sum_{\sigma\in\Sigma_\mathrm{e}} \exp[\mathbf{U}\mathbf{V}_{f_j}]_\sigma} \tag{3.26}$$

$$= \prod_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} \frac{\exp(\mathbf{U}_{e_i} \cdot \mathbf{V}_{f_j})}{\sum_{\sigma\in\Sigma_\mathrm{e}} \exp(\mathbf{U}_\sigma \cdot \mathbf{V}_{f_j})}. \tag{3.27}$$

Here, $\mathbf{U}_{e_i}$ is the embedding of e_i, and we can also think of $\mathbf{V}_{f_j}$ as the embedding of f_j. If the model is trained well, then an English word and a Spanish word that are translations of each other will have a high dot-product. Recall that the dot product of two vectors is related to the angle between the two vectors (thought of as arrows in Euclidean space), so we hope that English and Spanish words that are translation of each other will 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 **T** can. But if d is smaller, then $\mathbf{U}\mathbf{V}^\top$ can only be an approximation of the full **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.

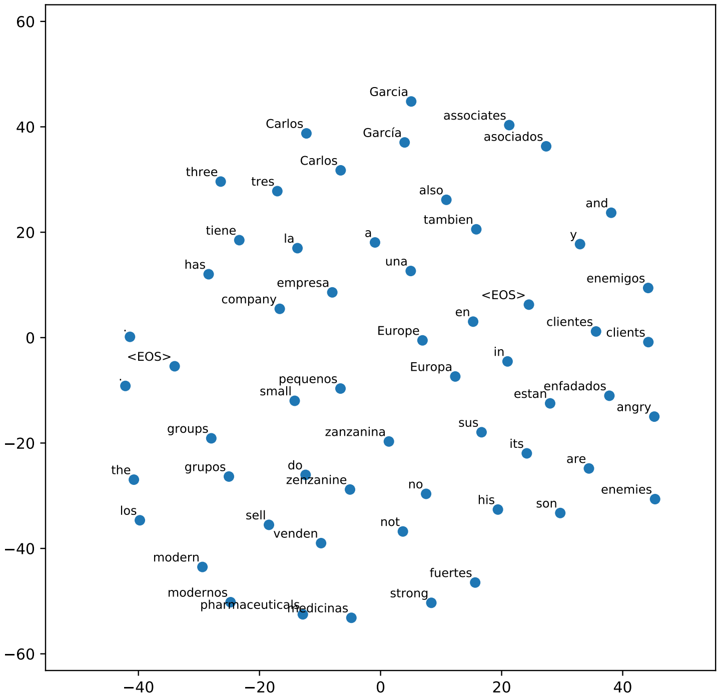


Figure 3.2: Two-dimensional visualization of the 64-dimensional word embeddings learned by the factored Model 1. The embeddings were normalized and then projected down to two dimensions using t-SNE (Maaten and Hinton, 2008). In most cases, the Spanish word embedding is close to its corresponding English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

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 $t(\text{gigante} \mid \text{large})$.

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 probabiity for both $t(\text{why} \mid \text{por})$ and $t(\text{why} \mid \text{qué})$. In fact, 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*. Nor does it have any way to learn that the absence of *por* or the absence of *qué* should "veto" the translation *why*. Remember that something similar happened when we took the union of two NFAs, and the solution here is also kind of similar.

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

$$P(e \mid f) = \prod_{i=1}^{m} \left[\text{softmax} \left(\sum_{j=1}^{n} \frac{1}{n} \mathbf{U} \mathbf{V}_{f_j} \right) \right]_{e_i}. \tag{3.28}$$

How does this help? Here's a near-optimal solution for the logits $(\mathbf{UV})$:

	por	qué	EOS
why	20	20	0
for	30	0	0
what	0	30	0
EOS	10	10	20

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	20	10	0
for	15	15	0
what	0	0	15
EOS	20	15	15

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 20, which is the best translation by a lot (5).

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 ≈ 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**:

$$P(e \mid f) = \prod_{i=1}^{m} \left[\operatorname{softmax} \left(\mathbf{U} \sum_{j=1}^{n} \frac{1}{n} \mathbf{V}_{f_j} \right) \right]_{e_i}. \tag{3.29}$$

This model is shown in Figure 3.1c. If the $\mathbf{V}_{f_j}$ can be thought of as vector representations of words, then the average $\sum_j \frac{1}{n}\mathbf{V}_{f_j}$ can be thought of as a vector representation of the whole sentence f. So the model has two parts, an encoder (**V**, then average) which converts f to a vector representation of f, and a decoder (**U**, then softmax) which converts the vector representation to English words.

Now let's do the same thing to Model 2. 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

$$P(e \mid f) = \prod_{i=1}^{m} \left[\operatorname{softmax} \left(\mathbf{U} \sum_{j=1}^{n} \alpha(j \mid i) \, \mathbf{V}_{f_j} \right) \right]_{e_i}. \tag{3.30}$$

At each time step i, the weights $\alpha(j \mid i)$, which must sum to one ($\sum_j \alpha(j \mid i) = 1$), provide a different "view" of f. This mechanism is known as *attention*, and the network is said to *attend* to different parts of the sentence at different times. The weights $\alpha(j \mid i)$ are called *attention weights*. These days, they are usually computed using *dot-product attention*, which factors $\alpha(\cdot \mid \cdot)$ like we did for $t(\cdot \mid \cdot)$ earlier:

$$\mathbf{Q} \in \mathbb{R}^{m \times d} \tag{3.31}$$

$$\mathbf{K} \in \mathbb{R}^{n \times d} \tag{3.32}$$

$$\alpha(j \mid i) = [\operatorname{softmax} \mathbf{K}\mathbf{Q}_i]_j \tag{3.33}$$

For each Spanish word f_j, the network computes a vector $\mathbf{K}_j$, called a *key*. This vector could depend on the position j, the word f_j, or any other words in f.

Then, at time step i, the network computes a vector $\mathbf{Q}_i$, called a *query*. This vector could depend on the position i, or the words $e_1, \ldots, e_{i-1}$. The above definition makes the network attend most strongly to Spanish words f_j whose keys $\mathbf{K}_j$ are most similar to the query $\mathbf{Q}_i$. This is more apparent if we expand the definition of softmax:

$$\alpha(j \mid i) = \frac{\exp[\mathbf{K}\mathbf{Q}_i]_j}{\sum_{j'} \exp[\mathbf{K}\mathbf{Q}_i]_{j'}} \tag{3.34}$$

$$= \frac{\exp(\mathbf{K}_j \cdot \mathbf{Q}_i)}{\sum_{j'} \exp(\mathbf{K}_{j'} \cdot \mathbf{Q}_i)}. \tag{3.35}$$

The vectors that are averaged together (here, the $\mathbf{V}_{f_j}$) are called the *values*. They are frequently (but not always) the same as the keys. And the resulting weighted average is sometimes called the *context vector*.

To get something similar to Model 2, we would let $\mathbf{Q}$ and $\mathbf{K}$ be learnable parameters. More precisely, let M and N be the maximum length of any English or Spanish or English sentence, respectively, and define learnable parameters

$$\bar{\mathbf{Q}} \in \mathbb{R}^{M \times d} \tag{3.36}$$

$$\bar{\mathbf{K}} \in \mathbb{R}^{N \times d}. \tag{3.37}$$

The rows of $\bar{\mathbf{Q}}$ and $\bar{\mathbf{K}}$ are called *position embeddings* (Gehring et al., 2017). Then for a given Spanish-English sentence pair with lengths n and m, let the queries and keys be the first m and n rows of $\bar{\mathbf{Q}}$ and $\bar{\mathbf{K}}$, respectively:

$$\mathbf{Q} = \begin{bmatrix} \bar{\mathbf{Q}}_{1,*} \\ \vdots \\ \bar{\mathbf{Q}}_{m,*} \end{bmatrix} \tag{3.38}$$

$$\mathbf{K} = \begin{bmatrix} \bar{\mathbf{K}}_{1,*} \\ \vdots \\ \bar{\mathbf{K}}_{n,*} \end{bmatrix}. \tag{3.39}$$

3.5 Neural Machine Translation

Our modified Model 2 (eqs. 3.30–3.39) 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

$$P(e_2 = \text{river} \mid \text{el río Jordan}) = 0.5 \quad P(e_3 = \text{Jordan} \mid \text{el río Jordan}) = 0.5 \tag{3.40}$$

$$P(e_2 = \text{river} \mid \text{el río Jordan}) = 0.5 \quad P(e_3 = \text{Jordan} \mid \text{el río Jordan}) = 0.5 \tag{3.41}$$

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 $f^{\boxed{\ell}}$, then its definition may contain a parameter with the same superscript, like $x^{\boxed{\ell}}$. The ℓ stands for $1, 2$, etc., so if we call $f^{\boxed{1}}$ twice, the same parameter $x^{\boxed{1}}$ is shared across both calls. But if we call $f^{\boxed{1}}$ and $f^{\boxed{2}}$, they have two different parameters $x^{\boxed{1}}$ and $x^{\boxed{2}}$. (In PyTorch, such functions would be implemented as modules.)

So, we can define some functions:

$$\begin{aligned}
\text{Embedding}^{\boxed{\ell}}(k) &= \mathbf{E}^{\boxed{\ell}}_k && (3.42)\\
\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) &= \sum_j [\text{softmax}\, \mathbf{Kq}]_j\, \mathbf{V}_j && (3.43)\\
\text{SoftmaxLayer}^{\boxed{\ell}}(\mathbf{x}) &= \text{softmax}(\mathbf{W}^{\boxed{\ell}} \mathbf{x}) && (3.44)
\end{aligned}$$

And now our modified Model 2 (eqs. 3.30–3.39) can be written as:

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

$$\begin{aligned}
\mathbf{V}_j &= \text{Embedding}^{\boxed{1}}(f_j) && (3.45)\\
\mathbf{K}_j &= \text{Embedding}^{\boxed{2}}(j) && (3.46)
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{q}^{(i)} &= \text{Embedding}^{\boxed{3}}(i) && (3.47)\\
\mathbf{c}^{(i)} &= \text{Attention}(\mathbf{q}^{(i)}, \mathbf{K}, \mathbf{V}) && (3.48)\\
P(e_i) &= \text{SoftmaxLayer}^{\boxed{4}}(\mathbf{c}^{(i)}). && (3.49)
\end{aligned}$$

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 couple of new ones. First, a tanh layer:

$$\text{TanhLayer}^{\boxed{\ell}}(\mathbf{x}) = \tanh(\mathbf{W}^{\boxed{\ell}} \mathbf{x} + \mathbf{b}^{\boxed{\ell}}). \tag{3.50}$$

To compute one step of an RNN:

$$\text{RNNCell}^{\boxed{\ell}}(\mathbf{h}, \mathbf{x}) = \tanh(\mathbf{A}^{\boxed{\ell}} \mathbf{h} + \mathbf{B}^{\boxed{\ell}} \mathbf{x} + \mathbf{c}^{\boxed{\ell}}). \tag{3.51}$$

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 $\mathbf{h}^{(j)} \in \mathbb{R}^d$:

$$\begin{aligned}
\mathbf{v}^{(j)} &= \text{Embedding}^{\boxed{1}}(f_j) && j = 1, \ldots, n && (3.52)\\
\mathbf{h}^{(j)} &= \text{RNNCell}^{\boxed{\ell}}(\mathbf{h}^{(j-1)}, \mathbf{v}^{(j)}) && j = 1, \ldots, n && (3.53)
\end{aligned}$$

where $\mathbf{h}^{(0)}$ is a parameter to be learned. It will be convenient to pack the rest of the $\mathbf{h}^{(j)}$ into a single matrix,

$$\begin{aligned}
&\mathbf{H} \in \mathbb{R}^{n \times d}\\
&\mathbf{H} = [\mathbf{h}^{(1)} \cdots \mathbf{h}^{(n)}]^\top. && (3.54)
\end{aligned}$$

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.

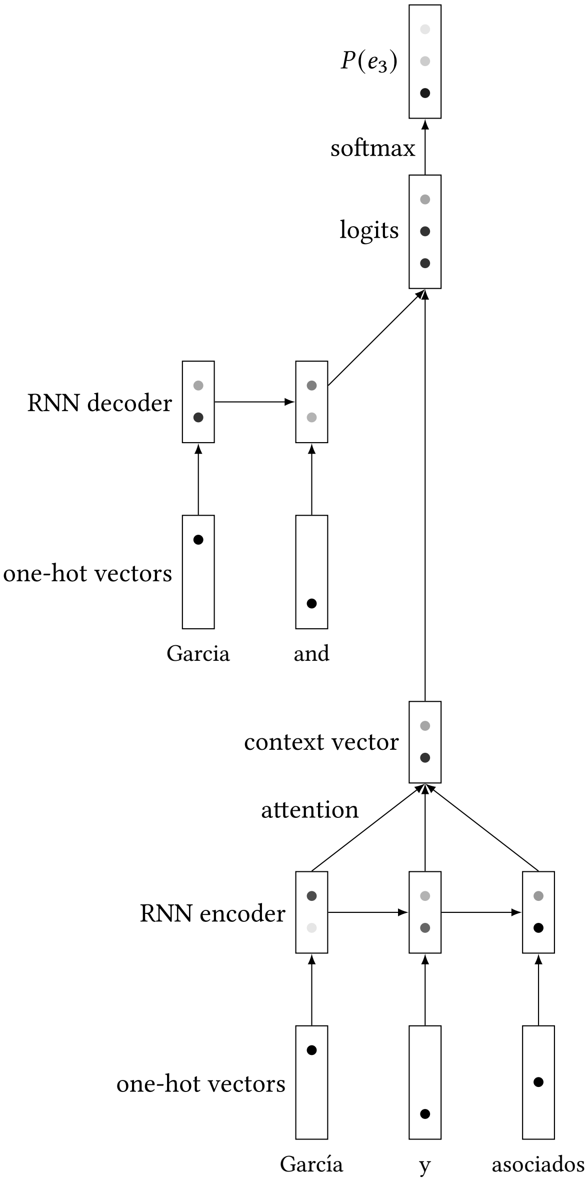


Figure 3.3: Simplified diagram of an RNN translation model (Bahdanau, Cho, and Bengio, 2015; Luong, Pham, and Manning, 2015).

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 has an initial vector $\mathbf{g}^{(0)} \in \mathbb{R}^d$, which is a parameter to be learned, and computes a sequence of vectors $\mathbf{g}^{(i)}$:

$$\begin{aligned}
\mathbf{u}^{(i)} &= \text{Embedding}^{\boxed{3}}(e_i) && i = 1, \ldots, m && (3.55)\\
\mathbf{g}^{(i)} &= \text{RNNCell}^{\boxed{4}}(\mathbf{g}^{(i-1)}, \mathbf{u}^{(i)}) && i = 1, \ldots, m. && (3.56)
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{c}^{(i)} &\in \mathbb{R}^d \\
\mathbf{c}^{(i)} &= \text{Attention}(\mathbf{g}^{(i-1)}, \mathbf{H}, \mathbf{H}) && (3.57)
\end{aligned}$$

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 concatenate the two and apply a tanh layer to get a single vector:

$$\begin{aligned}
\mathbf{o}^{(i)} &\in \mathbb{R}^d \\
\mathbf{o}^{(i)} &= \text{TanhLayer}^{\boxed{5}}\left(\begin{bmatrix} \mathbf{c}^{(i)} \\ \mathbf{g}^{(i-1)} \end{bmatrix}\right) && (3.58)
\end{aligned}$$

And finally we predict an English word:

$$P(e_i) = \text{SoftmaxLayer}^{\boxed{6}}(\mathbf{o}^{(i)}). \qquad (3.59)$$

Important implementation note: Whereas the encoder could be written using many loops over j, the decoder has to be written as a single loop over $i = 1, \ldots, n$. The order of computation is: $\mathbf{c}^{(1)}, \mathbf{o}^{(1)}, P(e_1), \mathbf{u}^{(1)}, \mathbf{g}^{(1)}, \mathbf{c}^{(2)}$, 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.

$$\begin{aligned}
\text{SelfAttentionCell}^{\boxed{\ell}}(\mathbf{X}, i) &= \text{Attention}(\mathbf{W}_{\text{Q}}^{\boxed{\ell}}\mathbf{X}_i, \mathbf{K}, \mathbf{V}) && (3.60)\\
\text{where } \mathbf{K}_j &= \mathbf{W}_{\text{K}}^{\boxed{\ell}}\mathbf{X}_j && (3.61)\\
\mathbf{V}_j &= \mathbf{W}_{\text{V}}^{\boxed{\ell}}\mathbf{X}_j && (3.62)\\
\text{SelfAttention}^{\boxed{\ell}}(\mathbf{X}) &= \mathbf{C} && (3.63)\\
\text{where } \mathbf{C}_i &= \text{SelfAttentionCell}^{\boxed{\ell}}(\mathbf{X}, i). && (3.64)
\end{aligned}$$

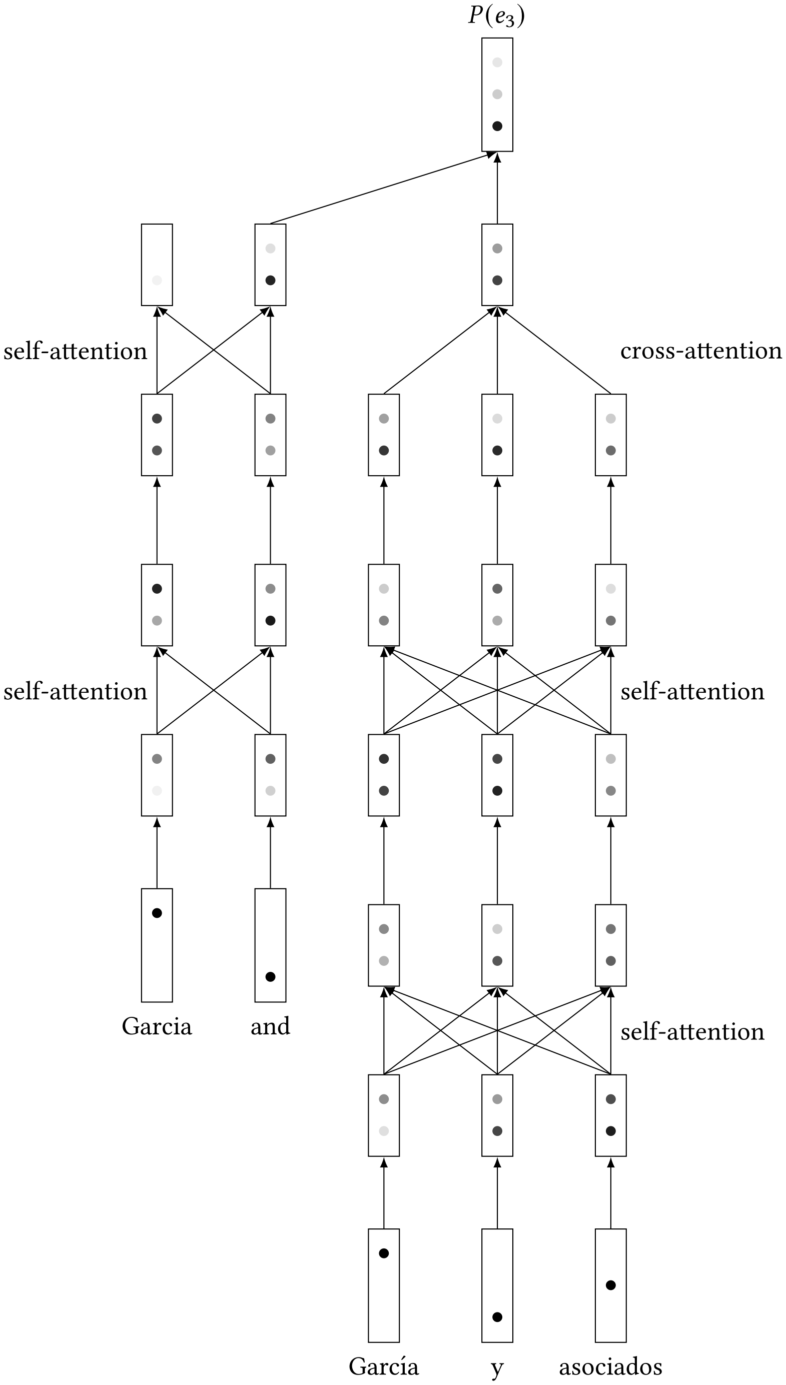


Figure 3.4: Simplified diagram of a Transformer translation model (Vaswani et al., 2017).

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 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.

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

$$\begin{aligned} &\mathbf{V} \in \mathbb{R}^{n \times d} \\ &\mathbf{V}_j = \text{Embedding}^{\boxed{1}}(f_j) + \text{Embedding}^{\boxed{2}}(j) \qquad j = 1, \ldots, n \end{aligned} \tag{3.65}$$

Next comes a self-attention layer:

$$\begin{aligned} &\mathbf{H} \in \mathbb{R}^{n \times d} \\ &\mathbf{H} = \text{SelfAttention}^{\boxed{3}}(\mathbf{V}). \end{aligned} \tag{3.66}$$

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

$$\begin{aligned} &\mathbf{H}' \in \mathbb{R}^{n \times d} \\ &\mathbf{H}'_j = \text{TanhLayer}^{\boxed{4}}(\mathbf{H}_j) \qquad j = 1, \ldots, n. \end{aligned} \tag{3.67}$$

Then, steps (3.66–3.67) are repeated: SelfAttention$^{\boxed{5}}$, TanhLayer$^{\boxed{6}}$, and so on, usually with 4 or 6 repetitions in total. To avoid running out of letters of the alphabet, though, we don't write equations for any more repetitions.

The decoder is also a stack of self-attention layers, and again we need to write the equations using a single iteration over i. For each time step $i = 1, \ldots, n-1$, we want to predict the next English word, $P(e_{i+1})$. Start by computing the vector representation of e_i:

$$\begin{aligned} &\mathbf{u}^{(i)} \in \mathbb{R}^{d} \\ &\mathbf{u}^{(i)} = \text{Embedding}^{\boxed{7}}(e_i) + \text{Embedding}^{\boxed{8}}(i). \end{aligned} \tag{3.68}$$

Then self-attention and feedforward layers, but note that at each time step i, self-attention only operates on $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(i)}$ because it can't see the future:

$$\begin{aligned} &\mathbf{g}^{(i)} \in \mathbb{R}^{d} \\ &\mathbf{g}^{(i)} = \text{SelfAttentionCell}^{\boxed{9}}([\mathbf{u}^{(1)} \cdots \mathbf{u}^{(i)}]^\top, i) \end{aligned} \tag{3.69}$$

$$\begin{aligned} &\mathbf{g}'^{(i)} \in \mathbb{R}^{d} \\ &\mathbf{g}'^{(i)} = \text{TanhLayer}^{\boxed{10}}(\mathbf{g}^{(i)}). \end{aligned} \tag{3.70}$$

Now, just as in the RNN-based model, we have a sequence of source encodings and a sequence of target encodings, and the rest of (our simplified version of) the model proceeds as before (cf. eqs. 3.57–3.59).

$$\begin{aligned}
\mathbf{c}^{(i)} &\in \mathbb{R}^d \\
\mathbf{c}^{(i)} &= \text{Attention}(\mathbf{g}'^{(i-1)}, \mathbf{H}', \mathbf{H}') && (3.71) \\
\mathbf{o}^{(i)} &\in \mathbb{R}^d \\
\mathbf{o}^{(i)} &= \text{TanhLayer}^{\boxed{5}}\left(\begin{bmatrix} \mathbf{c}^{(i)} \\ \mathbf{g}'^{(i-1)} \end{bmatrix}\right) && (3.72) \\
P(e_i) &= \text{SoftmaxLayer}^{\boxed{6}}(\mathbf{o}^{(i)}). && (3.73)
\end{aligned}$$

Vector $\mathbf{g}'^{(0)}$ is a parameter to be learned.[1]

The real Transformer is more complicated – in particular, there are actually multiple cross-attentions, one after 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.

[1] The standard formulation doesn't need this because it prepends BOS to every sentence. I decided not to prepend BOS in order to get a more unified treatment of automata, RNNs, the IBM models, and NMT, and painted myself into this corner.

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`.