Machine Translation
Part 2

6 May 2021
Today we’ll see how to do machine translation using a *sequence-to-sequence* neural network architecture and how this is improved using *attention*. 
Background: Pre-neural statistical machine translation
The core idea is to learn a probabilistic model from data.

Suppose we’re translating French to English.

We want to find the best English sentence $y$ given French sentence $x$:

$$\text{argmax}_y P(y \mid x)$$
The core idea is to learn a probabilistic model from data.

Suppose we’re translating French to English.

We want to find the best English sentence $y$ given French sentence $x$:

$$\arg\max_y P(y \mid x)$$

We can use Bayes rule to break this down into two components to be learned separately:

$$\arg\max_y P(x \mid y) \cdot P(y)$$
The core idea is to learn a probabilistic model from data.

Suppose we’re translating French to English.

We want to find the best English sentence $y$ given French sentence $x$:

$$\text{argmax}_y P(y \mid x)$$

We can use Bayes rule to break this down into two components to be learned separately:

$$\text{argmax}_y \left( P(x \mid y) \cdot P(y) \right)$$

*Translation model*

*Models how words and phrases should be translated (fidelity).*

*Learned from parallel data*
The core idea is to learn a probabilistic model from data.

Suppose we’re translating French to English.

We want to find the best English sentence $y$ given French sentence $x$:

$$\text{argmax}_y \ P(y \mid x)$$

We can use Bayes rule to break this down into two components to be learned separately:

$$\text{argmax}_y \ (P(x \mid y) \cdot P(y))$$

- **Translation model**: Models how words and phrases should be translated (fidelity). Learned from parallel data.
- **Language model**: Models how to write good English (fluency). Learned from monolingual data.
The translation model $P(x \mid y)$ is learned from \textit{parallel data}, e.g., pairs of human-translated French/English sentences.

But knowing the sentences correspond isn’t enough. Really what we need to learn is

$$P(x, a \mid y)$$

where $a$ is the \textit{alignment}, i.e., word-level correspondence between French sentence $x$ and English sentence $y$. 
**Alignment** is the correspondence between particular words in the translated sentence pair.

```
<table>
<thead>
<tr>
<th></th>
<th>Le</th>
<th>Japon</th>
<th>Japan</th>
</tr>
</thead>
<tbody>
<tr>
<td>shaken</td>
<td></td>
<td>secoué</td>
<td>shaken</td>
</tr>
<tr>
<td>by</td>
<td></td>
<td>par</td>
<td>by</td>
</tr>
<tr>
<td>two</td>
<td></td>
<td>deux</td>
<td>two</td>
</tr>
<tr>
<td>new</td>
<td></td>
<td>nouveaux</td>
<td>new</td>
</tr>
<tr>
<td>quakes</td>
<td></td>
<td>séismes</td>
<td>quakes</td>
</tr>
</tbody>
</table>
```

Examples from Brown et al., 1993
Alignment can be *many-to-one*.

Examples from Brown et al., 1993
Alignment can be **one-to-many**.

Examples from Brown et al., 1993
Alignment can be *many-to-many* (phrase-level).

Examples from Brown et al., 1993
We learn $P(x, a | y)$ as a combination of many factors, including

- probability of particular words aligning (also depends on position in sent.)
- probability of particular words having particular fertility (number of corresponding words)
- etc.

These alignments $a$ are latent variables; they aren’t explicitly specified in the data!

It requires the use of special learning algorithms like expectation maximization (EM) for learning the parameters of distributions with latent variables.
$$\text{argmax}_y \quad P(x \mid y) \quad P(y)$$

- *Translation model*
- *Language model*
argmax_y P(x | y) P(y)

How to compute this argmax?
Translation model
Language model
argmax_y \ P(x \mid y) \ P(y)

Compute the argmax (the best English sentence $y$ for the French sentence $x$) using a **heuristic search algorithm**, discarding hypotheses that are too low probability.

This process is called **decoding**.
Decoding for statistical machine translation

Decoding for statistical machine translation

Koehn, Statistical Machine Translation, 2009
Decoding for statistical machine translation

The best statistical machine translation systems (1990s–2010s) were extremely complex.

Hundreds of important details we haven’t mentioned here.
Systems had many separately-designed subcomponents.
Lots of feature engineering aimed at capturing particular language phenomena
Lots of human effort required to compile and maintaining extra resources like tables of equivalent phrases for each language pair
Neural machine translation
Neural machine translation (NMT) is a way to do machine translation with a \textit{single neural network}.

The neural network architecture is called \textit{sequence-to-sequence} (seq2seq) and it’s made out of \textit{two} RNNs.
the cat likes to eat pizza

Lanners, 2019

el gato le gusta comer pizza
Encoding of the source sentence. Provides initial hidden state for Decoder RNN.

Encoder RNN

Source sentence (input)

Encoder RNN produces an encoding of the source sentence.
Encoder RNN produces an encoding of the source sentence.

Encoding of the source sentence. Provides initial hidden state for Decoder RNN.

Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*. 
The sequence-to-sequence model

Encoding of the source sentence. Provides initial hidden state for Decoder RNN.

Decoder RNN is a Language Model that generates target sentence, conditioned on encoding.

Source sentence (input)

<START> il a m' entarté

Target sentence (output)

he hit me with a pie

Encoder RNN produces an encoding of the source sentence.
The sequence-to-sequence model

Encoding of the source sentence. Provides initial hidden state for Decoder RNN.

Source sentence (input)

Decoder RNN is a Language Model that generates target sentence, conditioned on encoding.

Decoder RNN produces an encoding of the source sentence.
The sequence-to-sequence model

Encoding of the source sentence. Provides initial hidden state for Decoder RNN.

Source sentence (input)

Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*.

Encoder RNN produces an *encoding* of the source sentence.

Note: This diagram shows test time behavior: decoder output is fed in as next step’s input.
Sequence-to-sequence models are useful for more than just machine translation:

- Summarization (long text → short text)
- Dialogue (previous utterances → next utterance)
- Parsing (input text → output parse as sequence)
- Code generation (natural language → Python code)
The sequence-to-sequence model is an example of a *conditional language model*:

*Language model* because the decoder is predicting the next word of the target sentence $y$.

*Conditional* because its predictions are also conditioned on the source sentence $x$. 
Neural machine translation directly calculates $P(y \mid x)$ – no alignment!

$$P(y \mid x) = P(y_1 \mid x) \ P(y_2 \mid y_1, x) \ P(y_3 \mid y_1, y_2, x) \cdots \ P(y_T \mid y_1, \ldots, y_{T-1}, x)$$

*Probability of next target word, given the target words so far and the source sentence x*
How do we train a neural machine translation system?

Get a big parallel corpus!
il a m’entarté

Source sentence (from corpus)

<START> he hit me with a pie

Target sentence (from corpus)
Source sentence (from corpus)  

Target sentence (from corpus)

Encoder RNN
\[ J = \frac{1}{T} \sum_{t=1}^{T} J_t \]

\[
= J_1 + J_2 + J_3 + J_4 + J_5 + J_6 + J_7
\]

Encoder RNN

Decoder RNN

Source sentence (from corpus)

Target sentence (from corpus)
We saw how to generate (or “decode”) the target sentence by taking argmax on each step of the decoder:

![Diagram of greedy decoding](image)

This is **greedy decoding** – taking the most probable word on each step.
Greedy decoding has no way to undo decisions!

Input: *il a m’entarté*  [he hit me with a pie]

→ *he _____*

→ *he hit _____*  **Whoops! No going back now…**

→ *he hit a ______*
Exhaustive search decoding

Ideally we want to find a (length $T$) translation $y$ that maximizes

$$P(y | x) = P(y_1 | x) P(y_2 | y_1, x) P(y_3 | y_1, y_2, x) \cdots P(y_T | y_1, \ldots, y_{T-1}, x)$$

$$= \prod_{t=1}^{T} P(y_t | y_1, \ldots, y_{t-1}, x)$$
Exhaustive search decoding

Ideally we want to find a (length $T$) translation $y$ that maximizes

$$P(y | x) = P(y_1 | x) \ P(y_2 | y_1, x) \ P(y_3 | y_1, y_2, x) \cdots \ P(y_T | y_1, \ldots, y_{T-1}, x)$$

$$= \prod_{t=1}^{T} P(y_t | y_1, \ldots, y_{t-1}, x)$$

We could try computing all possible sequences $y$

This means that on each step $t$ of the decoder, we’re tracking $V^t$ possible partial translations, where $V$ is the vocabulary size

This $O(V^T)$ complexity is far too expensive!
Beam search decoding

On each step of decoder, keep track of the $k$ most probable partial translations (hypotheses), where $k$ is the “beam size” — in practice, around 5 to 10
Beam search decoding

On each step of decoder, keep track of the $k$ most probable partial translations (hypotheses),

where $k$ is the “beam size” – in practice, around 5 to 10

The score of a hypothesis $y_1, \ldots, y_t$ is its log probability:

$$
\text{score}(y_1, \ldots, y_t) = \log P_{LM}(y_1, \ldots, y_t|x) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)
$$

Scores are all negative; higher scores are better

We search for high-scoring hypotheses, tracking the top $k$ on each step
Beam search is not guaranteed to find an optimal solution.

However, it is much more efficient than an exhaustive search!
Beam search decoding example

Beam size = \( k = 2 \). **Blue numbers** = \( \text{score}(y_1, \ldots, y_t) = \sum_{t=1}^{t} \log P_{LM}(y_t | y_1, \ldots, y_{t-1}, x) \)
Beam search decoding example

Beam size = k = 2. **Blue numbers** = \( \text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x) \)

\[
<\text{START}> \\
\text{Calculate prob dist of next word}
\]
Beam search decoding example

Beam size = k = 2. Blue numbers = \( \text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x) \)

-0.7 = \log P_{LM}(he|<START>)

-0.9 = \log P_{LM}(l|<START>)

Take top \( k \) words and compute scores
Beam search decoding example

Beam size = k = 2. Blue numbers = \text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)

-1.7 = \log P_{LM}(\text{hit} | \langle \text{START} \rangle \text{ he}) + 0.7

-2.9 = \log P_{LM}(\text{struck} | \langle \text{START} \rangle \text{ he}) + 0.7

-1.6 = \log P_{LM}(\text{was} | \langle \text{START} \rangle \text{ I}) + 0.9

-1.8 = \log P_{LM}(\text{got} | \langle \text{START} \rangle \text{ I}) + 0.9

For each of the k hypotheses, find top k next words and calculate scores
Beam search decoding example

Beam size = $k = 2$. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_M(y_i | y_1, \ldots, y_{i-1}, x)$

Of these $k^2$ hypotheses, just keep $k$ with highest scores
Beam search decoding example

Beam size = $k = 2$. **Blue numbers** = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)$

For each of the $k$ hypotheses, find top $k$ next words and calculate scores.
Beam size = $k = 2$. Blue numbers = score($y_1, \ldots, y_t$) = $\sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)$

Of these $k^2$ hypotheses, just keep $k$ with highest scores.
Beam search decoding example

Beam size = k = 2. Blue numbers = \( \text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x) \)

For each of the \( k \) hypotheses, find top \( k \) next words and calculate scores
Beam search decoding example

Beam size = k = 2. **Blue numbers** = \( \text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x) \)

Of these \( k^2 \) hypotheses, just keep \( k \) with highest scores
Beam search decoding example

Beam size = k = 2. Blue numbers = score(y₁, ..., yᵢ) = \sum_{i=1}^{r} \log P_{LM}(y_i|y_1, ..., y_{i-1}, x)

For each of the k hypotheses, find top k next words and calculate scores.
Beam search decoding example

Beam size = k = 2. Blue numbers = \( \text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_t|y_1, \ldots, y_{t-1}, x) \)

Of these \( k^2 \) hypotheses, just keep \( k \) with highest scores
Beam search decoding example

Beam size = $k = 2$. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)$

For each of the $k$ hypotheses, find top $k$ next words and calculate scores
Beam search decoding example

Beam size = k = 2. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)$

This is the top-scoring hypothesis!
Beam search decoding example

Beam size = $k = 2$. Blue numbers = $\text{score}(y_1, \ldots, y_t) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)$
Beam search decoding: stopping criterion

In *greedy decoding*, usually we decode until the model produces an `<END>` token, e.g.

```plaintext
<START> he hit me with a pie <END>
```

In *beam search decoding*, different hypotheses may produce `<END>` tokens on different timesteps.

- When a hypothesis produces `<END>`, that hypothesis is complete.
- Put it aside and continue exploring the other hypotheses via beam search.

Usually we continue beam search until either

- we reach some pre-defined cutoff for the number of timesteps or
- we produce some pre-defined number of complete hypotheses
Beam search decoding: finishing up

So, we have our list of completed hypotheses. How do we select the best one?

Each hypothesis $y_1, \ldots, y_t$ on our list has a score:

$$\text{score}(y_1, \ldots, y_t) = \log P_{LM}(y_1, \ldots, y_t | x) = \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)$$
Beam search decoding: finishing up

So, we have our list of completed hypotheses. How do we select the best one?

Each hypothesis $y_1, \ldots, y_t$ on our list has a score:

$$
\text{score}(y_1, \ldots, y_t) = \log P_{LM}(y_1, \ldots, y_t|x) = \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)
$$

But the longer hypotheses have lower scores!

Fix: Normalize by length before selecting the top-scoring hypothesis:

$$
\frac{1}{t} \sum_{i=1}^{t} \log P_{LM}(y_i|y_1, \ldots, y_{i-1}, x)
$$
Compared to traditional statistical machine translation, neural machine translation has many advantages:

Better *performance*

More *fluent*

Better use of *context*

Better use of *phrase similarities*

A *single neural network* to be optimized end-to-end

No subcomponents to be individually optimized

Requires much *less human engineering effort*

No feature engineering

Same method for all language pairs
There are some disadvantages though!

Neural machine translation models are less interpretable, making them harder to debug.

They’re also difficult to control. You can’t, for example, easily specify rules or guidelines for translation.
Uninterpretable systems do strange things:

Christian, “Why is Google Translate Spitting Out Sinister Religious Prophecies?”, 2018
Uninterpretable systems do strange things:

Christian, “Why is Google Translate Spitting Out Sinister Religious Prophecies?”, 2018
MT progress over time
Neural machine translation is the biggest success story of deep learning in NLP.

In 2014 the first seq2seq research paper was published,

By 2016 neural machine translation was the leading standard method and Google Translate switched to using it.

NMT systems trained by a *handful* of engineers in a few *months* outperformed SMT systems built by *hundreds* of engineers over many *years*. 
But there’s still lots of room for improvement in machine translation, including

- out-of-vocabulary words
- domain mismatch between train and test data
- maintaining context over longer text
- low-resource language pairs
Neural machine translation research continues to thrive, and researchers have found many improvements to the “vanilla” seq2seq NMT system we’ve considered.

One improvement is so integral it’s the new vanilla: attention.
Attention
Problems with this architecture?
Sequence-to-sequence: the bottleneck problem

Encoding of the source sentence. This needs to capture all information about the source sentence. Information bottleneck!

Source sentence (input)

il a m' entarté

Target sentence (output)

<START> he hit me with a pie <END>

Decoder RNN
Attention provides a solution to the bottleneck problem:

On each step of the decoder, use *direct connection to the encoder* to *focus on a particular part* of the source sequence.
dot product

Encoder RNN

Attention scores

Decoder RNN

il a m’ entarté <START>

Source sentence (input)
On this decoder timestep, we’re mostly focusing on the first encoder hidden state (“he”).

Take softmax to turn the scores into a probability distribution.

Source sentence (input): il a m’ entarté <START>
Use the attention distribution to take a **weighted sum** of the encoder hidden states.

The attention output mostly contains information from the hidden states that received high attention.
Concatenate attention output with decoder hidden state, then use to compute $\hat{y}_1$ as before.
Sometimes we take the attention output from the previous step, and also feed it into the decoder (along with the usual decoder input).
We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$.

On timestep $t$, we have decoder hidden state $s_t \in \mathbb{R}^h$.

We get the attention scores $e^t$ for this step:

$$e^t = [s_t^T h_1, \ldots, s_t^T h_N] \in \mathbb{R}^N$$

We take softmax to get the attention distribution $\alpha^t$ for this step:

$$\alpha^t = \text{softmax}(e^t) \in \mathbb{R}^N$$

(This is a probability distribution and sums to 1.)

We use $\alpha^t$ to take a weighted sum of the encoder hidden states to get the attention output $a_t$:

$$a_t = \sum_{i=1}^{N} \alpha^t_i h_i \in \mathbb{R}^h$$

Finally we concatenate the attention output $a_t$ with the decoder hidden state $s_t$ and proceed as in the normal seq2seq model.

$$[a_t; s_t] \in \mathbb{R}^{2h}$$
Attention significantly improves NMT performance

It’s very useful to allow the decoder to focus on certain parts of the source

Attention solves the bottleneck problem

It allows the decoder to look directly at the source, bypassing the bottleneck

Attention provides some interpretability

By inspecting the attention distribution, we can see what the decoder was focusing on!

We get (soft) alignment for free!

We never explicitly trained an alignment system, but the network learns it by itself
The idea of attention can be used for other neural architectures and other tasks, not just seq2seq and machine translation.
Summary

Since 2014, neural machine translation (NMT) has rapidly replaced intricate statistical machine translation (SMT) systems.

Sequence-to-sequence is the architecture for NMT, using two RNNs.

Attention is a way to focus on particular parts of the input, which improves sequence-to-sequence a lot!
Acknowledgments

The lecture incorporates material from:

Daniel Jurafsky and James Martin, *Speech and Language Processing*

Quinn Lanners

Abigail See, Stanford University