Natural Language Processing with Deep Learning

neural machine translation
What is Neural Machine Translation?

• Neural Machine Translation (NMT) is a way to do Machine Translation with a *single neural network*

• The neural network architecture is called *sequence-to-sequence* (aka seq2seq) and it involves *two RNNs*. 
Neural Machine Translation (NMT)

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

Note: This diagram shows test time behavior: decoder output is fed in as next step’s input
Sequence-to-sequence is versatile!

• Sequence-to-sequence is useful for *more than just MT*

• Many NLP tasks can be phrased as sequence-to-sequence:
  • **Summarization** (long text $\rightarrow$ short text)
  • **Dialogue** (previous utterances $\rightarrow$ next utterance)
  • **Parsing** (input text $\rightarrow$ output parse as sequence)
  • **Code generation** (natural language $\rightarrow$ Python code)
Neural Machine Translation (NMT)

- 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 $x$

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

  Probability of next target word, given target words so far and source sentence $x$

- **Question**: How to *train* a NMT system?
- **Answer**: Get a big parallel corpus…


Training a Neural Machine Translation system

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

Seq2seq is optimized as a **single system.**
Backpropagation operates “end-to-end”.

Source sentence (from corpus)
Target sentence (from corpus)
Greedy decoding

- We saw how to generate (or “decode”) the target sentence by taking argmax on each step of the decoder.

- This is greedy decoding (take most probable word on each step).

- Problems with this method?
Problems with greedy decoding

• Greedy decoding has no way to undo decisions!
  • Input: *il a m’entarté*  
    
    *he hit me with a pie*
  • → *he _____*
  • → *he hit _____*
  • → *he hit a _____*  
    
    *(whoops! no going back now...)*

• How to fix this?
Exhaustive search decoding

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

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

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

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

- **Core idea:** On each step of decoder, keep track of the *k* most probable partial translations (which we call *hypotheses*).
  - *k* is the **beam size** (in practice around 5 to 10)

- A hypothesis \( y_1, \ldots, y_t \) has a **score** which is its log probability:

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

  - Scores are all negative, and higher score is better
  - We search for high-scoring hypotheses, tracking top *k* on each step

- Beam search is **not guaranteed** to find optimal solution
- But much more efficient than exhaustive search!
Beam search decoding: example

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

Backtrack to obtain the full hypothesis
Beam search decoding: stopping criterion

- In greedy decoding, usually we decode until the model produces a `<END>` token
  - For example: `<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.
  - Place it aside and continue exploring other hypotheses via beam search.

- Usually we continue beam search until:
  - We reach timestep $T$ (where $T$ is some pre-defined cutoff), or
  - We have at least $n$ completed hypotheses (where $n$ is pre-defined cutoff)
Beam search decoding: finishing up

- We have our list of completed hypotheses.
- How to select top one with highest score?

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

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

- **Problem with this:** longer hypotheses have lower scores

- **Fix:** Normalize by length. Use this to select top one instead:

$$\frac{1}{t} \sum_{i=1}^{t} \log P_{LM}(y_i | y_1, \ldots, y_{i-1}, x)$$
Advantages of NMT

Compared to SMT, NMT 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
Disadvantages of NMT?

Compared to SMT:

• NMT is less interpretable
  • Hard to debug

• NMT is difficult to control
  • For example, can’t easily specify rules or guidelines for translation
  • Safety concerns!
MT progress over time
[Edinburgh En-De WMT newstest2013 Cased BLEU; NMT 2015 from U. Montréal]

NMT: the biggest success story of NLP Deep Learning

Neural Machine Translation went from a fringe research activity in 2014 to the leading standard method in 2016

- **2014**: First seq2seq paper published
- **2016**: Google Translate switches from SMT to NMT
- This is amazing!
  - **SMT** systems, built by hundreds of engineers over many years, outperformed by NMT systems trained by a handful of engineers in a few months
So is Machine Translation solved?

• **Nope!**

• Many difficulties remain:
  • Out-of-vocabulary words
  • Domain mismatch between train and test data
  • Maintaining context over longer text
  • Low-resource language pairs

**Further reading:** “Has AI surpassed humans at translation? Not even close!”
https://www.skynettoday.com/editorials/state_of_nmt
Sequence-to-sequence: the bottleneck problem

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

Target sentence (output)

Target sentence (output)

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

Source sentence (input)

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

- **Attention** provides a solution to the bottleneck problem.

- **Core idea**: on each step of the decoder, use *direct connection to the encoder* to *focus on a particular part* of the source sequence.

- First we will show via diagram (no equations), then we will show with equations.
Sequence-to-sequence with attention

Encoder RNN

Attention scores

Attention distribution

Attention output

Decoder RNN

Source sentence (input):

il a m’ entarté

<START> he hit me with a

pie

\( \hat{y}_6 \)
Attention: in equations

- 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 (this is a probability distribution and sums to 1)
  $$\alpha_t = \text{softmax}(e_t) \in \mathbb{R}^N$$
- We use $\alpha_t$ to take a weighted sum of the encoder hidden states to get the attention output $a_t$
  $$a_t = \sum_{i=1}^{N} \alpha_i h_i \in \mathbb{R}^h$$
- Finally we concatenate the attention output $a_t$ with the decoder hidden state $s_t$ and proceed as in the non-attention seq2seq model
  $$[a_t; s_t] \in \mathbb{R}^{2h}$$
Attention is great

- Attention significantly **improves NMT performance**
  - It’s very useful to allow decoder to focus on certain parts of the source
- Attention **solves the bottleneck problem**
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention **helps with vanishing gradient problem**
  - Provides shortcut to faraway states
- Attention provides **some interpretability**
  - By inspecting attention distribution, we can see what the decoder was focusing on
  - We get (soft) **alignment for free**!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself
Attention is a *general* Deep Learning technique

- We’ve seen that attention is a great way to improve the sequence-to-sequence model for Machine Translation.
- **However:** You can use attention in many architectures (not just seq2seq) and many tasks (not just MT)

**More general definition of attention:**
- Given a set of vector *values*, and a vector *query*, **attention** is a technique to compute a weighted sum of the values, dependent on the query.

- We sometimes say that the *query* *attends to* the values.
- For example, in the seq2seq + attention model, each decoder hidden state (query) *attends to* all the encoder hidden states (values).
Attention is a *general* Deep Learning technique

<table>
<thead>
<tr>
<th>More general definition of attention:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a set of vector <em>values</em>, and a vector <em>query</em>, <strong>attention</strong> is a technique to compute a weighted sum of the values, dependent on the query.</td>
</tr>
</tbody>
</table>

**Intuition:**

- The weighted sum is a *selective summary* of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a *fixed-size representation of an arbitrary set of representations* (the values), dependent on some other representation (the query).
There are several attention variants

- We have some \( v_{values} h_1, \ldots, h_N \in \mathbb{R}^{d_1} \) and a \( query \ s \in \mathbb{R}^{d_2} \)

- Attention always involves:
  1. Computing the \textit{attention scores} \( e \in \mathbb{R}^N \)
  2. Taking softmax to get \textit{attention distribution} \( \alpha \):
     \[
     \alpha = \text{softmax}(e) \in \mathbb{R}^N
     \]
  3. Using attention distribution to take weighted sum of values:
     \[
     a = \sum_{i=1}^{N} \alpha_i h_i \in \mathbb{R}^{d_1}
     \]
     thus obtaining the \textit{attention output} \( a \) (sometimes called the \textit{context vector})
Attention variants

There are several ways you can compute $e \in \mathbb{R}^N$ from $h_1, \ldots, h_N \in \mathbb{R}^{d_1}$ and $s \in \mathbb{R}^{d_2}$:

- **Basic dot-product attention:** $e_i = s^T h_i \in \mathbb{R}$
  - Note: this assumes $d_1 = d_2$
  - This is the version we saw earlier

- **Multiplicative attention:** $e_i = s^T W h_i \in \mathbb{R}$
  - Where $W \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix

- **Additive attention:** $e_i = v^T \tanh(W_1 h_i + W_2 s) \in \mathbb{R}$
  - Where $W_1 \in \mathbb{R}^{d_3 \times d_1}$, $W_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $v \in \mathbb{R}^{d_3}$ is a weight vector.
  - $d_3$ (the attention dimensionality) is a hyperparameter

More information:
