Encoder-Decoder Models

Jindřich Libovíký, Jindřich Helcl
Model Concept
Neural model with a sequence of discrete symbols as an input that generates another sequence of discrete symbols as an output.

- pre-process source sentence (tokenize, split into smaller units)
- convert input into vocabulary indices
- run the encoder to get an intermediate representation (vector/matrix)
- run the decoder
- postprocess the output (detokenize)
Language Models and Decoders
What is a Language Model

LM = an estimator of a sentence probability given a language

- From now on: sentence = sequence of words $w_1, \ldots, w_n$
- Factorize the probability by word
  i.e., no grammar, no hierarchical structure

\[
\Pr(w_1, \ldots, w_n) = \Pr(w_1) \cdot \Pr(w_2|w_1) \cdot \Pr(w_3|w_2, w_1) \cdot \ldots \\
= \prod_{i=1}^{n} \Pr(w_i|w_{i-1}, \ldots, w_1)
\]
What is it good for?

• Substitute for grammar: tells what is a good sentence in a language
• Used in ASR, and statistical MT to select more probable outputs
• Being able to predict next word = proxy for knowing the language
  • language modeling is training objective for word2vec
  • BERT is a masked language model

• Neural decoder is a conditional language model.
\(n\)-gram vs. Neural LMs

**n-gram**
cool from 1990 to 2013

- Limited history \(=\) Markov assumption
- Transparent: estimated from \(n\)-gram counts in a corpus

\[
P(w_i|w_{i-1}, w_{i-2}, \ldots, w_{i-n}) \approx \sum_{j=0}^{n} \lambda_j \frac{c(w_i|w_{i-1}, \ldots, w_{i-j})}{c(w_i|w_{i-1}, \ldots, w_{i-j+1})}
\]

**Neural**
cool since 2013

- Conditioned on RNN state which gather potentially unlimited history
- Trained by back-propagation to maximize probability of the training data
- Opaque, but works better (as usual with deep learning)
**Reminder: Recurrent Neural Networks**

**RNN = pipeline for information**

In every step some information goes in and some information goes out.

Technically: A “for” loop applying the same function $A$ on input vectors $x_i$.

At training time unrolled in time: technically just a very deep network.

Sequence Labeling

- Assign a label to each word in a sentence.
- Tasks formulated as sequence labeling:
  - Part-of-Speech Tagging
  - Named Entity Recognition
  - Filling missing punctuation

MLP = Multilayer perceptron
\[ n \times \text{layer: } \sigma(Wx + b) \]

Softmax for \( K \) classes with logits
\[ z = (z_1, \ldots, z_K): \]
\[
\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}
\]
Detour: Why is softmax a good choice

Output layer with softmax (with parameters $W$, $b$) — gets categorical distribution:

$$P_y = \text{softmax}(\mathbf{x}) = \Pr(y \mid \mathbf{x}) = \frac{\exp\left\{ \mathbf{x}^\top W \right\} + b}{\sum \exp\left\{ \mathbf{x}^\top W \right\} + b}$$

Network error = cross-entropy between estimated distribution and one-hot ground-truth distribution $T = \mathbf{1}(y^*) = (0, 0, \ldots, 1, 0, \ldots, 0)$:

$$L(P_y, y^*) = H(P, T) = -\mathbb{E}_{i \sim T} \log P(i)$$

$$= - \sum_i T(i) \log P(i)$$

$$= - \log P(y^*)$$
Derivative of Cross-Entropy

Let $l = x^T W + b$, $l_{y^*}$ corresponds to the correct one.

\[
\frac{\partial L(\mathbf{P}_y, y^*)}{\partial l} = - \frac{\partial}{\partial l} \log \frac{\exp l_{y^*}}{\sum_j \exp l_j} = - \frac{\partial}{\partial l} \left( l_{y^*} - \log \sum \exp l \right)
\]

\[
= \mathbf{1}_{y^*} + \frac{\partial}{\partial l} - \log \sum \exp l = \mathbf{1}_{y^*} - \frac{\sum \mathbf{1}_{y^*} \exp l}{\sum \exp l} = \mathbf{1}_{y^*} - \mathbf{P}_y(y^*)
\]

Interpretation: Reinforce the correct logit, suppress the rest.
Language Model as Sequence Labeling

- **Input symbol**: one-hot vectors
- **Embedding lookup**
- **RNN cell** (more layers)
- **Classifier**
- **Normalization**
- **Distribution for the next symbol**

```
<s> ... w_1 ... w_2 ... ...
```

`P(w_1 | <s>)`  
`P(w_1 | ...)`  
`P(w_2 | ...)`

**Encoder-Decoder Models**
Sampling from a Language Model

$\text{Pr}(w_1|<s>)$  

$\text{Pr}(w_1|\ldots)$  

$\text{Pr}(w_2|\ldots)$  

$\text{Pr}(w_3|\ldots)$  

Encoder-Decoder Models
Sampling from a Language Model: Pseudocode

```
last_w = "<s>"
state = initial_state
while last_w != "</s>":
    last_w_embeding = target_embeddings[last_w]
    state = rnn(state, last_w_embeding)
    logits = output_projection(state)
    last_w = vocabulary[np.random.multinomial(1, logits)]
    yield last_w
```
Training objective: negative-log likelihood:

$$\text{NLL} = - \sum_{i}^{n} \log \Pr (w_i | w_{i-1}, \ldots, w_1)$$

I.e., maximize probability of the correct word.

- Cross-entropy between the predicted distribution and one-hot “true” distribution
- Error from word is backpropagated into the rest of network unrolled in time
- Prone to exposure bias: during training only well-behaved sequences, it can break when we sample something weird at inference time
Generating from a Language Model

Neural Machine Translation is

- a new technology developed by a team at the University
- a technology that uses neural networks and machine learning to
- a powerful tool for understanding the spoken language.

(Example from GPT-2, a Transformer-based English language model, screenshot from https://transformer.huggingface.co/doc/gpt2-large)

Cool, but where is the source language?
Conditioning the Language Model & Attention
Conditional Language Model

Formally it is simple, condition distribution of

- target sequence \( y = (y_1, \ldots, y_T) \) on
- source sequence \( x = (x_1, \ldots, x_T) \)

\[
\text{Pr}(y_1, \ldots, y_n | x) = \prod_{i} \text{Pr}(y_i | y_{i-1}, \ldots, y_1, x)
\]

We need an encoder to get a representation of \( x \)!

What about just continuing an RNN...
The interface between encoder and decoder is a single vector regardless the sentence length.
state = np.zeros(rnn_size)

for w in input_words:
    input_embedding = source_embeddings[w]
    state = enc_cell(encoder_state, input_embedding)

last_w = "<s>"

while last_w != "</s>":
    last_w_embedding = target_embeddings[last_w]
    state = dec_cell(state, last_w_embedding)
    logits = output_projection(state)
    last_w = vocabulary[np.argmax(logits)]
    yield last_w
Vanilla Seq2Seq: Information Bottleneck

Bottleneck all information needs to run through.
A single vector must represent the entire source sentence.

Main weakness and the reason for introducing the attention.
The Attention Model

• Motivation: It would be nice to have variable length input representation
• RNN returns one state per word …
• …what if we were able to get only information from words we need to generate a word.

Attention = probabilistic retrieval of encoder states for estimating probability of target words.

Query = hidden states of the decoder
Values = encoder hidden states
Sequence-to-Sequence Model With Attention

- Encoder = bidirectional RNN states \( \langle h_i \rangle \approx \text{retrieved values} \)
- Decoder step starts as usual state \( s_0 \approx \text{retrieval query} \)
- Decoder state \( s_0 \) used to compute distribution the over encoder states
- Weighted average of encoder states = context vector
- Decoder state & context concatenated MLP + Softmax predicts next word
Attention Model in Equations (1)

**Inputs:**
- decoder state \( s_i \)
- encoder states \( h_j = [\widehat{h}_j; \widehat{h}_j] \) \( \forall i = 1 \ldots T_x \)

**Attention energies:**
\[
e_{ij} = v_a^\top \tanh (W_a s_{i-1} + U_a h_j + b_a)
\]

**Attention distribution:**
\[
\alpha_{ij} = \frac{\exp (e_{ij})}{\sum_{k=1}^{T_x} \exp (e_{ik})}
\]

**Context vector:**
\[
c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j
\]
Attention Model in Equations (2)

Output projection:

\[ t_i = \text{MLP} \left( s_{i-1} \oplus v_{y_{i-1}} \oplus c_i \right) \]

...attention is mixed with the hidden state
(different in different models)

Output distribution:

\[ p \left( y_i = k | s_i, y_{i-1}, c_i \right) \propto \exp \left( W_o t_i + b_k \right)_k \]

(usual trick: use transposed embeddings as \( W_o \))

- Different version of attentive decoders exist
- Alternative: keep the context vector as input for the next step
- Multilayer RNNs: attention between/after layers
Workings of the Attentive Seq2Seq model

Ich habe den Walros gesehen

⟩⟩⟩ RNN ⟩⟩⟩ RNN ⟩⟩⟩ RNN ⟩⟩⟩

⟩ ⟨ ⟨ ⟨ RNN ⟨ ⟨ ⟨ RNN ⟨ ⟨ ⟨ RNN ⟨ ⟨ ⟨

⟩⟩⟩ RNN ⟩⟩⟩ RNN ⟩⟩⟩ RNN ⟩⟩⟩

<h₁> h₂ h₃ h₄ h₅
<h₁> h₂ h₃ h₄ h₅
<h₁> h₂ h₃ h₄ h₅
<h₁> h₂ h₃ h₄ h₅
<h₁> h₂ h₃ h₄ h₅

<s> I saw the walrus </s>
state = np.zeros(emb_size)
fw_states = []
for w in input_words:
    input_embedding = source_embeddings[w]
    state, _ = fw_enc_cell(encoder_state, input_embedding)
    fw_states.append(state)

bw_states = []
state = np.zeros(emb_size)
for w in reversed(input_words):
    input_embedding = source_embeddings[w]
    state, _ = bw_enc_cell(encoder_state, input_embedding)
    bw_states.append(state)

enc_states = [np.concatenate(fw, bw) for fw, bw in zip(fw_states, reversed(bw_states))]
last_w = "<s>"
while last_w != "</s>":
    last_w_embedding = target_embeddings[last_w]
    state = dec_cell(state, last_w_embedding)
    alphas = attention(state, enc_states)
    context = sum(a * state for a, state in zip(alphas, enc_states))
    logits = output_projection(np.concatenate(state, context, last_w_embedding))
    last_w = np.argmax(logits)
    yield last_w
Attention Visualization (1)

Image source: ?, Fig. 3
Attention Visualization (2)

Encoder-Decoder Models
Differences between attention model and word alignment used for phrase table generation:

**attention (NMT)**
- probabilistic
- declarative
- LM generates

**alignment (SMT)**
- discrete
- imperative
- LM discriminates
Optimize negative log-likelihood of parallel data, backpropagation does the rest.
If you choose a right optimizer, learning rate, model hyper-parameters, prepare data, do back-translation, monolingual pre-training …

Confusion: decoder inputs vs. output

\[
\begin{array}{cccccc}
\text{inputs } y[:{-1}] & <s> & y_1 & y_2 & y_3 & y_4 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\text{Decoder} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\text{outputs } y[1:] & y_1 & y_2 & y_3 & y_4 & </s> \\
\end{array}
\]
Inference
• Encoder-decoder is a conditional language model
• For a pair $x$ and $y$, we can compute:

$$\Pr(y|x) = \prod_{i=1}^{T_y} \Pr(y_i|y_{:i}, x)$$

• When decoding we want to get

$$y^* = \arg\max_{y'} \Pr(y'|x)$$

Enumerating all $y$’s is computationally intractable ☠
In each step, take the maximum probable word.

\[
y^*_i = \arg\max_{y_i} \Pr(y_i | y^*_{i-1}, \ldots, <s>)
\]

```python
last_w = "<s>"
state = initial_state
while last_w != "</s>":
    last_w_embedding = target_embeddings[last_w]
    state = dec_cell(state, last_w_embedding)
    logits = output_projection(state)
    last_w = vocabulary[np.argmax(logits)]
yield last_w
```
What if...

⚠ Greedy decoding can easily miss the best option. △
Beam Search

Keep a small $k$ of hypothesis (typically 4–20).

1. Begin with a single empty hypothesis in the beam.
2. In each time step:
   2.1 Extend all hypotheses in the beam by all (or the most probable) from the output distribution (we call these candidate hypotheses)
   2.2 Score the candidate hypotheses
   2.3 Keep only $k$ best of them.
3. Finish if all $k$-best hypotheses end with $</s>$
4. Sort the hypotheses by their score and output the best one.
Beam Search: Example

<s>
- Hey
- Hi
- Hello
- world
- there
- !
- World
- hello

Encoder-Decoder Models

33/38
beam = [["<s>", initial_state, 1.0]]
while any(hyp[-1] != "</s>" for hyp, _, _ in beam):
    candidates = []

    for hyp, state, score in beam:
        distribution, new_state = decoder_step(hyp[-1], state, encoder_states)
        for i, prob in enumerate(distribution):
            candidates.append(hyp + [vocabulary[i]], new_state, score * prob)

    beam = take_best(k, candidates)
Implementation issues

• Multiplying of too many small numbers $\rightarrow$ float underflow
  need to compute in log domain and add logarithms

• Sentences can have different lengths
  This is a good long sentence.

  $$0.7 \times 0.6 \times 0.9 \times 0.1 \times 0.4 \times 0.4 \times 0.8 \times 0.9 = 0.004$$

  This

  $$0.7 \times 0.01 = 0.007$$

$\Rightarrow$ use the geometric mean instead of probabilities directly

• Sorting candidates is expensive, asymptotically $|V| \log |V|$: $k$-best can be found in linear time, $|V| \sim 10^4 - 10^5$
Final Remarks
Brief history of the architectures

• 2013 First encoder-decoder model (?)
• 2014 First really usable encoder-decoder model (?)
• 2014/2015 Added attention (crucial innovation in NLP) (?)
• 2016/2017 WMT winners used RNN-based neural systems (?)
• 2017 Transformers invented (outperformed RNN) (?)

The development of architectures still goes on...

Document context, non-autoregressive models, multilingual models, ...
Summary

- Encoder-decoder architecture = major paradigm in MT
- Encoder-decoder architecture = conditional language model
- Attention = way of conditioning the decoder on the encoder
- Attention = probabilistic vector retrieval
- We model probability, but need heuristics to get a good sentence from the model