Language Models and Text Generation

Joseph Le Roux

18/08/26

Notes

Outline

Notes

1 Introduction

Notes

1 Language Models (LMs)

LMs are probabilistic models which assigns probabilities to strings dependening on their plausibility

$$\forall w_1 \dots w_n, \quad 0 \leq p_\theta(w_1 \dots w_n) \leq 1 \text{ and } \sum_{w_1 \dots w_n} p_\theta(w_1 \dots w_n) = 1$$

Plausibility?

- For French we want that:
  - $p_\theta(\text{dort}, \text{chat}, \text{le}) \approx 0$,
  - $p_\theta(\text{le}, \text{chat}, \text{dort}) > 0$,
  - $p_\theta(\text{le}, \text{chat}, \text{dort}) > p_\theta(\text{le}, \text{felin}, \text{domestique}, \text{sommeille})$.
- More generally:
  - *Incorrect* strings should have low probabilities
  - Fluent/natural sentences should be more probable than convoluted/awkward ones
- Remarks
  - Task-oriented definition e.g. for a chatbot we may want useful answers to be more probable than useless ones.

Notes

1 Applications

All NLP applications for which we want to compare candidate strings

Spell Correction

test whether an edit (change a letter…) increases string probability

Text Generation

Given a prefix (beginning of a sentence), sample following words which give high string probabilities

Assistant (chatbot)

Given a query (prompt/question) generate a useful answer with high string probabilities

Translation

Given sentences translated *word by word*, seek for the permutation of words with highest probability

Notes

2 Autoregressive LMs

Notes

2 Definition

We want a generative model

- *assigns* a probability to every string
- *explains* how the string is built

Auto-regressive (or conditional) Models

based on the chain rule

$$p_\theta(w_1 \dots w_t) = p_\theta(w_1) \times p_\theta(w_2 \mid w_1) \times p_\theta(w_3 \mid w_1, w_2) \times \cdots \times p_\theta(w_t \mid w_1, \dots, w_{t-1})$$

- *explanation*: we generate one word after the other, in order, based on previous words
- *implementation*: problem, we need an infinity of parameters (one table per prefix of arbitrary length)
  - let's have a look at implementation details, shall we?

Notes

2 Tokenization

What are the w_i to begin with?

Definition

- Texts come as big strings: Tokenization is the process of splitting text into small units called tokens
- Tokens are basic input for Language Models
  - Either word, a subword, or a single character (and special symbols, e.g. `<BOS>`/`<EOS>` see *infra*)

Remarks

- Vocabulary Size:
  - Large vocabulary ⇒ fewer unknowns or Out-Of-Vocabulary (OOV) tokens, but increases parameter count.
  - Small vocabulary ⇒ risk of high OOV rates, or reliance on subword tokens.
- Handling Unknown Words:
  - Use a special `<unk>` token, or use character-level tokens.
- Granularity:
  - Word-Level Simplest, but OOV issues can be severe.
  - Subword-Level (BPE, WordPiece, SentencePiece) Balances coverage and vocabulary size.
  - Character-Level No OOVs, but leads to longer sequences and sometimes slower training.

Notes

2 BPE: Byte-Pair Encoding [SHB16]

Goal Find the *optimal* set of tokens

- Generic data compression technique adapted for tokenization
- Iteratively merge most frequent pair of symbols (characters or subwords) into a single token.
- Produce vocabulary that reduces OOV issues while preventing vocabulary size explosion.

Algorithm

1. Initialize Vocabulary V_0
   - Each character is a token (e.g., `a`,`b`,`c`,`d`)
2. Count Pair Frequencies:
   - Scan training set for adjacent token pairs (e.g., `a`+`c`, `a`+`d`, etc.).
3. Merge Most Frequent Pair:
   - Combine pair as a single token (e.g., `a_c`).
   - Update text (i.e., each occurrence of subsequence `ac` becomes `a_c`).
   - Add this newly merged token to vocabulary.
4. Repeat for n merges or until desired vocabulary size is reached.

Notes

2 BPE: Byte-Pair Encoding, Example

Formal Language

- corpus `"aaabdaaaaabac"`, we start with 4 tokens `a`,`b`,`c`,`d`
- we want to have a vocabulary size of 7
  1. `"aaabdaaaaabac"` $V_0 = \{a, b, c, d\}$; $|V_0| < 7$ scan for more frequent bigram $\longrightarrow$ `aa`
  2. `"ZabdZZabac"` $V_1 = \{a, b, c, d, Z = aa\}$; $|V_1| < 7$ scan for more frequent bigram $\longrightarrow$ `ab`
  3. `"ZYdZZYac"` $V_2 = \{a, b, c, d, Z = aa, Y = ab\}$; $|V_2| < 7$ scan for more frequent bigram $\longrightarrow$ `ZY`
  4. `"XdZXac"` $V_2 = \{a, b, c, d, Z = aa, Y = ab, X = aaab\}$; $|V_2| = 7$ done!

NLP

Algorithm 1 Learn BPE operations

```
import re, collections

def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

r · → r·
l o → lo
lo w → low
e r· → er·

Figure 1: BPE merge operations learned from dictionary {'low', 'lowest', 'newer', 'wider'}.

Notes

2 Parametrisation – Learning

Assume we have corpus $\mathcal{T}$ of sentences.

Parametrisation by MLE

maximum log-likelihood estimation on $\mathcal{T}$

$$\mathcal{L}(\theta) = \sum_{w_1 \dots w_t \in \mathcal{T}} -\log p_\theta(w_1 \dots w_t)$$
$$= \sum_{w_1 \dots w_t \in \mathcal{T}} \sum_{i=1}^{t} -\log p_\theta(w_i \mid w_{<i})$$

where $w_{<i}$ represents words before position i.

- same loss we saw for word2vec and image classification, adapted to task/dataset.
- depending on how p_θ is defined, the loss can be optimised in different ways (see n-gram and neural)
- 🤔 how is MLE aligned with plausibility?

Notes

3 Evaluation

Notes

3 Evaluation of a LM: *perplexity* (1)

Evaluation

- on an external corpus (yet similar to the training set)
- the (log)-likelihood depends on the length of the corpus → average by the number of words:

$$l = \frac{1}{N} log(L) = \frac{1}{N} \sum_{w_{1:n} \in \mathcal{T}} \sum_{i=1}^{n} \log p_\theta(w_i \mid w_{<i})$$

Assuming logs are in base 2, define the perplexity as:

$$\rho = 2^{-l}$$

Perplexity:

- is positive
- smaller as the log-likelihood is bigger

We want a LM with small perplexity.

Notes

3 Evaluation of a LM: *perplexity* (2)

Remark 1

For models with uniform distribution:

$$q(w|u,v) = \frac{1}{|V|+1}, \forall u, v, w$$

Show (as exercise) that in this case perplexity is always $|V|+1$, the number of classes

Remark 2

- If $q(w|u,v) = 0$ in the test corpus, then perplexity is ∞.
- Hence, must ensure that LM probabilities are never zero (smoothing...)

Notes

3 Evaluation of a LM: *perplexity* (3)

Some indicative values on English newspaper data ($|V| = 20\,000$):

Model	Perplexity
Uniform distribution	20 000
Unigram	$> 5\,000$
Bigram	700–1 000
Trigram	100–200
Neural n-gram (FFN)	$\approx$ 85–150
Transformer LM	$\ll 85$

- Smaller perplexity = better model.
- Modern Transformer LMs reach single-digit perplexity on some benchmarks.

Notes

4 Count-based n-gram LMs

Notes

4 Markov *n*-gram Models (1/2)

- The way to parametrise LMs before the adoption of NNs in NLP
- $n-grams$ are sequences of n consecutive words

  unigrams *"the"*, *"little"*, *"cat"*, *"sleeps"*
  bigrams *"the little"*, *"little cat"*, *"cat sleeps"*
  trigrams *"the little cat"*, *"little cat sleeps"*
  4-grams *"the little cat sleeps"*

Intuition

approximate prefixes of arbitrary lengths by $n-grams$

$$p_\theta(w_{i+1} \mid w_1, \dots, w_i) \approx p_\theta(w_{i+1} \mid \underbrace{w_{i-n+2} \dots, w_i}_{n-1 \text{ words}})$$

Notes

4 Markov *n*-gram Models (2/2)

$$p_\theta(w_1 \dots w_t) = \prod_{i=1}^{t} p_\theta(w_i \mid w_{i-n+1}, \dots, w_{i-1})$$

we add two ingredients:

- negative positions filled with special token `BOS` (beginning of sentence)
- final position always `EOS` (end of sentence)

Notes

4 Example

with a trigram model, decomposition of probability:

$$\begin{aligned} p(\text{the little cat sleeps}) = {} & p(\text{the} \mid \text{BOS, BOS}) \times p(\text{little} \mid \text{BOS, the}) \\ & \times p(\text{cat} \mid \text{the, litte}) \times p(\text{sleeps} \mid \text{little, cat}) \\ & \times p(\text{EOS} \mid \text{cat, sleeps}) \end{aligned}$$

Notes

4 Estimation

One distribution per context (that is *(n-1)*-gram): Loss minimisation has a simple closed form:

$$\begin{aligned} p(w \mid w_{i-n+1}, \ldots, w_{i-1}) &= \frac{\text{count}(w_{i-n+1}, \ldots, w_{i-1}, w)}{\text{count}(w_{i-n+1}, \ldots, w_{i-1})} \\ &= \text{prefix fixed, how many times } w \text{ is seen after} \end{aligned}$$

assuming that the denominator is positive:

- positive or zero
- sum to one

🤔 prove this

Notes

4 Estimation – Problems

If numerator is zero

- in practice, *smooth* counts
- many methods, some very involved
- simplest is *add-δ*

$$p(w \mid w_{i-n+1}, \ldots, w_{i-1}) = \frac{\text{count}(w_{i-n+1}, \ldots, w_{i-1}, w) + \delta}{\text{count}(w_{i-n+1}, \ldots, w_{i-1}) + |V|\delta}$$

If denominator is zero

use *(n-1)*-grams : *back-off* methods (see your favorite old NLP textbook)

Notes

5 Example

Notes

5 Toy Corpus, Tri-Gram Model from Nadi Tomeh

Corpus & Vocabulary

Toy Corpus consists of three sentences, each prepended with two <BOS>:

```
<BOS> <BOS> i love cats <EOS>
<BOS> <BOS> i love dogs <EOS>
<BOS> <BOS> cats chase mice <EOS>
```

Vocabulary {<BOS>, i, love, cats, dogs, chase, mice, <EOS>}.

Trigram Model Assumption

- For each position k, we model $p_\theta(w_k \mid w_{k-2}, w_{k-1})$.
- Example: In <BOS> <BOS> i love cats <EOS>, the third token i is predicted by $p_\theta(\texttt{i} \mid \texttt{<BOS>}, \texttt{<BOS>})$.
- We will collect all (2-token context, next token) counts from corpus and apply MLE:

$$\hat{\theta}_{\mathbf{c},w} = \frac{\text{count}(\mathbf{c}, w)}{\text{count}(\mathbf{c})}.$$

Notes

5 Toy Corpus, Tri-Gram Model from Nadi Tomeh

Context-Next Token Counts

Below is a partial table of trigram contexts and how often each next token appears:

Context (w_{k-2}, w_{k-1})	Next Token	Count	Sum Over Next Toks	MLE Probability
(<BOS>, <BOS>)	i	2	3	$\frac{2}{3} \approx 0.67$
(<BOS>, <BOS>)	cats	1		$\frac{1}{3} \approx 0.33$
(<BOS>, i)	love	2	2	$\frac{2}{2} = 1.0$
(i, love)	cats	1	2	$\frac{1}{2} = 0.5$
(i, love)	dogs	1		$\frac{1}{2} = 0.5$
(love, cats)	<EOS>	1	1	1.0
...				

🤔 Fill out this table for *all* observed 2-token contexts in the corpus (omitting zero-count contexts not observed, or using smoothing).

Notes

5 Toy Corpus, Tri-Gram Model from Nadi Tomeh

Cross-Entropy & Perplexity Computation

`<BOS> <BOS> i love mice <EOS>` {We deliberately chose this sentence to illustrate what happens when an n-gram is unseen in training.}

- Let N be total tokens in `<BOS> <BOS> i love mice <EOS>` (which is 6).
- Per-token cross-entropy $= \frac{1}{6}\sum_{k=1}^{5} \log p_\theta(w_k \mid w_{k-2}, w_{k-1})$.
- Perplexity $\exp(\text{cross-entropy})$
- Example: If $p_\theta(\texttt{mice} \mid \texttt{i}, \texttt{love}) = 0$, perplexity is infinite.

Notes

6 Neural Language Models

Notes

6 Neural *n*-gram Models [BDV00]

Key idea: replace count tables with a neural network

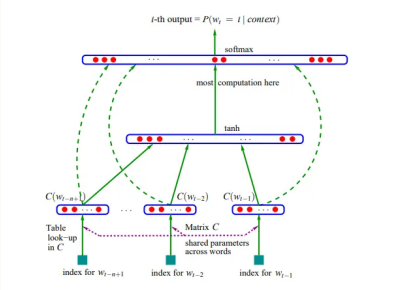
- Represent word by a vector $e(w) \in \mathbb{R}^d$ (embedding).
- Concatenate context embeddings (e.g. 2 for trigram models)

$$\text{input} = [e(w_{i-2}); e(w_{i-1})] \in \mathbb{R}^{2d}$$

- Pass through MLP then softmax:

$$p(w_i \mid w_{i-2}, w_{i-1}) = \text{softmax}\big(W_2 \tanh(W_1 [e(w_{i-2}); e(w_{i-1})] + b_1) + b_2\big)$$

- Output is a distribution over the full vocabulary $V \cup \{\text{EOS}\}$.



Advantages over count-based *n*-grams

1. Generalizes through word similarity: words with similar vectors share probability mass.
2. No need for smoothing heuristics: the network interpolates naturally.
3. Learned embeddings are a by-product (same spirit as word2vec).

Notes

6 Neural *n*-gram LM in PyTorch

```python
import torch, torch.nn as nn

class NgramLM(nn.Module):
    def __init__(self, vocab_size, emb_dim, hidden_dim, n=3):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, emb_dim)
        self.fc1 = nn.Linear((n-1) * emb_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, vocab_size)

    def forward(self, context):   # context: (batch, n-1) word indices
        x = self.emb(context).view(context.size(0), -1)  # flatten
        x = torch.tanh(self.fc1(x))
        return self.fc2(x)   # logits; use CrossEntropyLoss

model = NgramLM(vocab_size=20000, emb_dim=128, hidden_dim=256, n=3)
loss_fn = nn.CrossEntropyLoss()
```


Notes

7 Generation

Notes

7 Generation with Language Models

Sampling with temperature

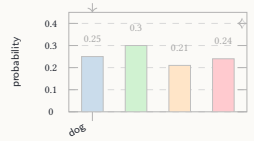
Input

- LM (probabilities) $p_\theta(w_k \mid w_1, \ldots, w_{k-1})$ + Initial context $\mathbf{c}$ (e.g., <BOS>, <BOS> for trigrams).
- (Optional) Temperature T

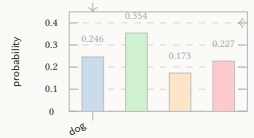
Algorithm

1. Initialize *sequence* ← []
2. Repeat
   - 2.1 Compute $p(w \mid \mathbf{c})$ for all tokens w.
   - 2.2 Choose
   $$w^* \sim p^{(T)}(w \mid \mathbf{c}) = \frac{p(w \mid \mathbf{c})^{1/T}}{\sum_{w'} p(w' \mid \mathbf{c})^{1/T}}.$$
   - 2.3 Append w^* to *sequence* and update context $\mathbf{c}$.
3. Until w^* = <EOS>.
4. Return *sequence* (without special tokens <BOS> and <EOS>).

$T = 1$ — **unchanged**



$T = 0.5$ — **peaked (more deterministic)**



Notes

7 Generation with Language Models

Sampling

- At each step, sample next token
- Pros Can produce diverse, creative outputs.
- Cons May generate nonsensical or low-probability tokens if distribution is broad.

Temperature Scaling

- Effects of$T$
  - $T > 1$: Flattens distribution, increase randomness.
  - $T < 1$: Sharpens distribution, increase greed
- Pros: Fine-grained control over output randomness.
- Cons: Requires careful tuning of T.

Other Sampling Strategies

- Top-k Restrict sampling to the k most probable tokens at each step.
- Nucleus (Top-p) Sample from the smallest set of tokens whose cumulative probability exceeds p.

Notes

8 Conclusion

Notes

8 Limitation of Fixed-Context Models

The context window is fixed

- An n-gram model (neural or not) only sees the last $n-1$ words.
- Long-range dependencies are invisible: {"The keys that the minister lost **were** on the table."}
- Increasing n helps but the input size grows linearly, and data sparsity returns.

What we really want

- A model that can attend to *any* word in the prefix, weighted by relevance.
- Computation should be parallelizable (not sequential like RNNs).
- $\Rightarrow$ Attention and Transformers the topic of the next lecture.

Notes

9 References

[BDV00] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. "A Neural Probabilistic Language Model." In: *Advances in Neural Information Processing Systems*. Ed. by T. Leen, T. Dietterich, and V. Tresp. Vol. 13. MIT Press, 2000. URL: https://proceedings.neurips.cc/paper_files/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf.

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units." In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Katrin Erk and Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162. URL: https://aclanthology.org/P16-1162/.

Notes