AIE1902: Exploring Language Models with AI

Xiaopeng Li
CUHK(SZ)
SAI

January 19, 2026

Outline

Course Project

The most important thing: Start Now!

First thing to finish: Form teams of 4–5 students and designate one team leader. The team leader should submit the team roster by **23:59, Jan 20**. Students who have not joined a team by the deadline will be randomly assigned to a group.

Second thing to finish: Meet with your teammates to decide the agent topic and initial baseline. If you are unsure about what to do, schedule a discussion with the instructor or the TA.

Outline

Recap: N-grams Model

Goal: assign probabilities to sentences by predicting next words.

N-gram approximation: use a short context window.

$$p(x_t \mid x_{t-1}, \ldots) \approx p(x_t \mid x_{t-1}, \ldots, x_{t-n+1}).$$

Training: estimate probabilities from counts ratio.

Evaluation: perplexity measures how surprised the model is.

Improvement: interpolation or add-k smoothing to resolve zero probability issue.

What Still Feels Unsatisfying?

Even with smoothing, n-grams treat words as independent symbols.

But humans know: some words are similar and often interchangeable.

- "coffee" is closer to "tea" than to "airport".
- A good model should share knowledge across similar words.

Key idea: learn a vector for each word, where similarity is measurable.

A Concrete Example

Context: `"I want to drink a cup of"`

A good model should rank:

`coffee`, `tea`, `water`, `milk`, ...

Problem with n-gram:

- If `"cup of tea"` appears rarely (or never), an n-gram can assign it very low probability.
- Smoothing avoids zero, but still does not encode why tea is reasonable here.

We want: a model that learns "tea is similar to coffee" from data.

A New Representation

N-gram: use context window to predict next word.

New model: still use a context window, but now to learn word vectors.

Benefits:

- Similarity of two words is computable by relation of two vectors;
- Similar words get similar vectors, so the model can transfer what it learned from `coffee` to `tea` even if some n-grams are unseen.
- Vectors are numeric type (not strings), so we can learn them easily by optimization rather than hand-designed rules.

Question: how do we represent words as vectors?

Basic Linear Algebra

A vector is just an ordered list of numbers:

$$a = (a_1, a_2, \ldots, a_d) \in \mathbb{R}^d.$$

Dot product:

$$a^\top b = \langle a, b \rangle = a \cdot b = \sum_{i=1}^{d} a_i b_i.$$

- Similarity score: larger $a^\top b$ means two words are more aligned.

Length and cosine similarity:

$$\|a\| = \sqrt{\sum_{i=1}^{d} a_i^2}, \qquad S_C(a, b) = \frac{a^\top b}{\|a\| \, \|b\|}.$$

Small example: $a = (1, 2)$, $b = (2, 1)$

$$a^\top b = 1 \cdot 2 + 2 \cdot 1 = 4, \quad S_C(a, b) = \frac{4}{\sqrt{5}\sqrt{5}} = 0.8.$$

From String to Int

Ideally: these vectors have nice mathematical properties:

- If two words w_1, w_2 are related, their vectors are related in the sense that $a_{w_1}^{\mathrm{T}} a_{w_2}$ is large.
- Or, $\|a_{w_1} - a_{w_2}\|$ is small, then we can cluster similar words.
- Maybe $a_{w_1} - a_{w_2}$ has a meaning: $a_{\texttt{king}} - a_{\texttt{man}} \approx a_{\texttt{queen}} - a_{\texttt{woman}}$.

Build: a vocabulary set $\mathcal{V} = \{\texttt{a}, \texttt{and}, \texttt{the}, \texttt{cat}, \ldots\}$.

Assign each word a unique integer (ID):

$$\texttt{stoi} : w \mapsto i, \qquad \texttt{itos} : i \mapsto w.$$

Question: once we have IDs, what vector should represent each word?

A Naive Approach: One-Hot Vector

If $|\mathcal{V}|$ is the vocabulary size, represent each word w by

$$e_w \in \mathbb{R}^{|\mathcal{V}|}, \quad e_w[\texttt{stoi}(w)] = 1, \text{ others} = 0.$$

Example:

- Suppose the corpus $\mathcal{C} :=$ `"The cat was blue."`
- Suppose vocabulary $\mathcal{V} := \{\texttt{"the"}, \texttt{"cat"}, \texttt{"was"}, \texttt{"blue"}, \texttt{"."}\}$.
- Index `stoi` $:= \{\texttt{"the"} : 0, \texttt{"cat"} : 1, \texttt{"was"} : 2, \texttt{"blue"} : 3, \texttt{"."} : 4\}$.
- Then, a one-hot representation might be:

$$e_{\texttt{the}} = [1, 0, 0, 0, 0], \quad e_{\texttt{cat}} = [0, 1, 0, 0, 0],$$
$$e_{\texttt{was}} = [0, 0, 1, 0, 0], \quad e_{\texttt{blue}} = [0, 0, 0, 1, 0], \quad e_{\texttt{.}} = [0, 0, 0, 0, 1].$$

Pros: simple, unique, no ambiguity.

Problems of One-Hot Approach

Problem 1: $|\mathcal{V}|$ can be huge $\implies$ vectors are huge and sparse.

Problem 2: similarity is lost.
For two different words $w_i \neq w_j$:

$$e_{w_i}^\top e_{w_j} = 0.$$

So one-hot cannot express:

`coffee` is closer to `tea` than to `airport`.

Conclusion: we need smaller vectors where dot products or distances are meaningful.

A Better Idea: Word2Vec

Idea: Instead of computing global statistics, we create a model that learns at each iteration and refines what it learns as it progresses.

Reference: In 2013, a seminal paper from Google:

Distributed Representations of Words and Phrases and their Compositionality

---

Tomas Mikolov
Google Inc.
Mountain View
mikolov@google.com

Ilya Sutskever
Google Inc.
Mountain View
ilyasu@google.com

Kai Chen
Google Inc.
Mountain View
kai@google.com

Greg Corrado
Google Inc.
Mountain View
gcorrado@google.com

Jeffrey Dean
Google Inc.
Mountain View
jeff@google.com

Word2Vec At A High Level

Two basic models: CBOW and Skip-Gram

- CBOW: predict a center word given the words around it;
- Skip-Gram: predict words around a given center word.

One core idea: Words that are close together or used in the same way tend to mean the same thing.

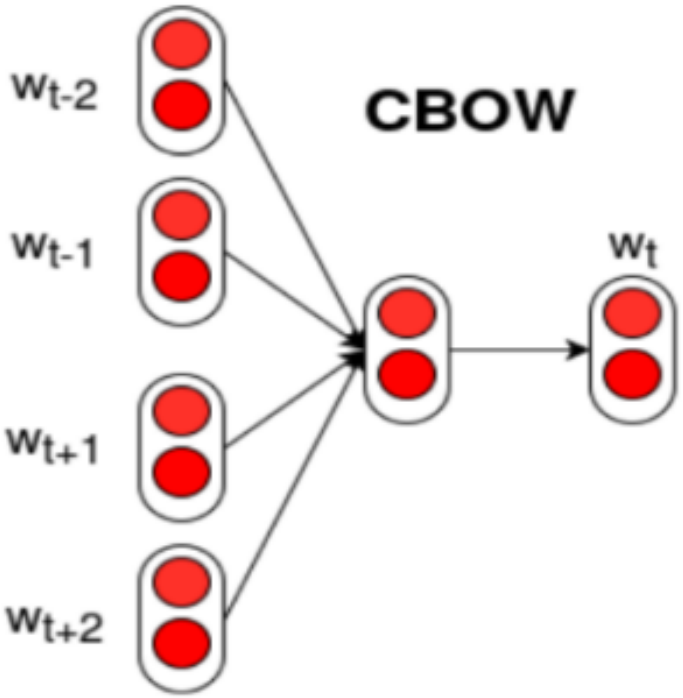
Examples:

- `"coffee"` often appears near: `"drink"`, `"cup"`, `"hot"`, `"morning"`;
- `"tea"` often appears near: `"drink"`, `"cup"`, `"hot"`, `"morning"`.

Outline

Word Embeddings: CBOW

Idea: The goal of CBOW is to predict the missing center word from the neighboring context words.



Word Embeddings: CBOW

Goal: we need to model $p(w_{\texttt{output}}|w_{\texttt{context}})$.

Example: given a sequence `"student studied NLP into the"` we can let `"NLP"` be the output word and `"student"`, `"studied"`, `"into"`, `"the"` be the context.

Embedding Matrix: Define two matrices $A \in \mathbb{R}^{|\mathcal{V}| \times d}$ (context) and $B \in \mathbb{R}^{|\mathcal{V}| \times d}$ (output) where d represents the dimension of embedding space.

Word Vector: If we have the one-hot representation of a word $e_w \in \mathbb{R}^{|\mathcal{V}|}$, then $B^\top e_w = b_w$ and $A^\top e_w = a_w$.

CBOW Algorithm

Step 1: given a set of documents $\mathcal{C}$, break up $\mathcal{C}$ into sentences and tokenize $\mathcal{C}$ to get the vocabulary $\mathcal{V}$.

Tokenize: cutting raw text into a sequence of basic units. For one sentence, there may be many different ways to tokenize.

Example: word-level tokenization

- Raw text:

```
The cat is on the mat.
```

- Tokenized:

```
[ "The", "cat", "is", "on", "the", "mat", "." ]
```

CBOW Algorithm

Step 2: fix a window size m so for each word we consider m words to the right and left.

Example: set $m = 1$ and $s =$ `"This is a short sentence."`,

- `["this","is","a"]`
- `["is","a","short"]`
- `["a","short","sentence"]`
- `["short","sentence","."]`

CBOW Algorithm

Step 3: for each center out word w_o and context words, gather pairs of vector (x, y) like the one-hot encoding of

$$((w_{o-m}, \ldots, w_{o-1}, w_{o+1}, \ldots, w_{o+m}), w_o)$$

Example: `["a","short","sentence"]` becomes `[("a", "sentence"), "short"]` and $w_o =$ `"short"`.

Step 4: for each context word, get the one-hot representations $e_w \in \mathbb{R}^{|\mathcal{V}|}$.

Step 5: for each context word, get the vector representations

$$(A^\top e_{w_{o-m}}, \ldots, A^\top e_{w_{o-1}}, A^\top e_{w_{o+1}}, \ldots, A^\top e_{w_{o+m}})$$

which is $(a_{w_{o-m}}, \ldots, a_{w_{o-1}}, a_{w_{o+1}}, \ldots, a_{w_{o+m}})$.

Basic Linear Algebra

A matrix $M \in \mathbb{R}^{n \times d}$ has n rows and d columns:

$$M = \begin{bmatrix} M_{11} & \cdots & M_{1d} \\ \vdots & \ddots & \vdots \\ M_{n1} & \cdots & M_{nd} \end{bmatrix}.$$

Matrix-vector multiplication: if $x \in \mathbb{R}^d$, then

$$y = Mx \in \mathbb{R}^n, \qquad y_i = \sum_{j=1}^{d} M_{ij} x_j.$$

One-hot lookup: Let $e_w \in \mathbb{R}^{|\mathcal{V}|}$ be the one-hot vector of word w. Then

$$A^\top e_w = a_w \in \mathbb{R}^d.$$

CBOW Algorithm

Step 6: Average the vectors to obtain a_{avg}:

$$a_{\text{avg}} = \frac{a_{w_{o-m}} + \cdots + a_{w_{o-1}} + a_{w_{o+1}} + \cdots + a_{w_{o+m}}}{2m} \in \mathbb{R}^d.$$

Step 7: Generate logit scores $z = Ba_{\text{avg}} \in \mathbb{R}^{|\mathcal{V}|}$.

Step 8: Turn the scores into probabilities:

$$p = \text{softmax}(z) \in \mathbb{R}^{|\mathcal{V}|}, \quad p_i = \frac{\exp(z_i)}{\sum_{j=1}^{|\mathcal{V}|} \exp(z_j)}, \ i = 1, \ldots, |\mathcal{V}|.$$

Requirement: We hope p match vector y, i.e., $p_{w_o} \approx 1$, because y is the one-hot representation of the output word w_o.

CBOW Algorithm

To achieve $p \approx y$, we can minimize the cross entropy loss:

$$\mathcal{L}(A, B) := -\sum_{i=1}^{|\mathcal{V}|} e_{w_o}[i] \log p_i.$$

- $\mathcal{L}(A, B) = -\log p_j$ where j is the index of w_o in $\mathcal{V}$: $j = \texttt{stoi}[w_o]$.
- If $p_j \approx 1$, as it should be, we get a 0 loss.
- If $p_j \approx 0$, then we get a $+\infty$ loss.
- If $p_j \approx 0.1$, we get a loss of $-\log 0.1 \approx 4.605$.

If we give too low probability to something that should have a high probability then the loss will be large.

CBOW Algorithm

The loss of a general output word w_o and fixed window around it is:

$$\begin{aligned}\mathcal{L}(A, B) &:= -\log p(w_o | w_{o-m}, \ldots, w_{o-1}, w_{o+1}, \ldots, w_{o+m}) \\ &= -\log p(w_o | a_{\mathsf{avg}}) \\ &= -\log \frac{\exp b_{w_o}^\top a_{\mathsf{avg}}}{\sum_{w \in \mathcal{V}} \exp b_w^\top a_{\mathsf{avg}}} \\ &= -b_{w_o}^\top a_{\mathsf{avg}} - \log \sum_{w \in \mathcal{V}} \exp b_w^\top a_{\mathsf{avg}}.\end{aligned}$$

Generally, the total loss of this model is accumulating the above loss across the entire training data and averaging over the number of data.

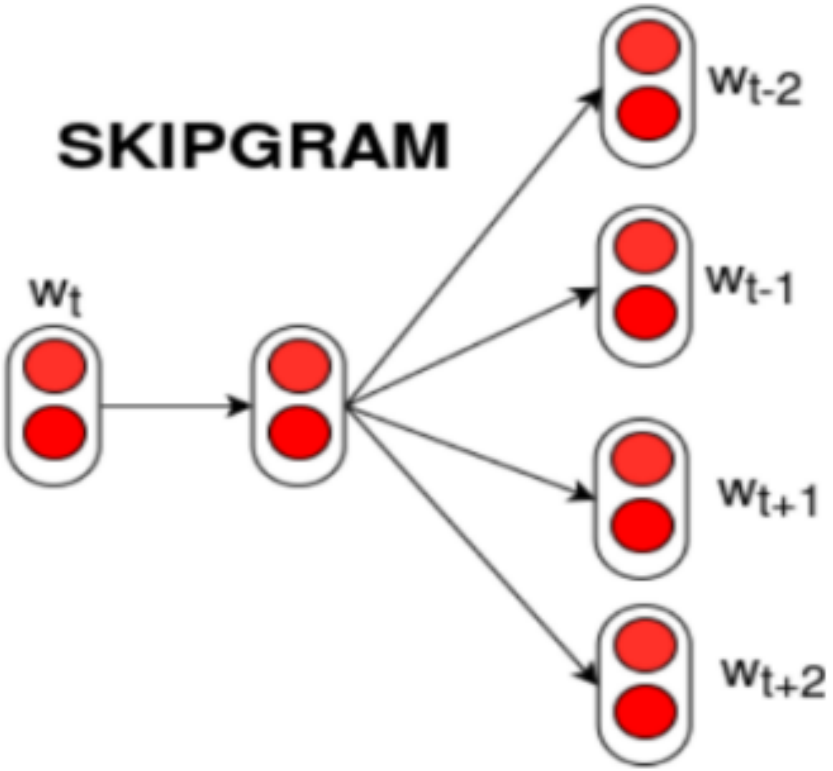
The parameters of this model are A and B. We can use gradient descent (GD) to update the parameters of this model:

- $A_{\text{new}} \leftarrow A_{\text{old}} - \eta \nabla_A \mathcal{L}(A_{\text{old}}, B_{\text{old}})$;
- $B_{\text{new}} \leftarrow B_{\text{old}} - \eta \nabla_B \mathcal{L}(A_{\text{old}}, B_{\text{old}})$.

Outline

Word Embeddings: Skip-Gram

Another approach takes the reverse of the CBOW model: given a center word, predict the words around this word.



Skip-Gram Algorithm

Step 1: given a set of sentences $\mathcal{C}$, tokenize at the word level and get the vocabulary $\mathcal{V}$ and fix a window size m.

Step 2: From all sentences, get all center (context) input words c and output words o pairs like

$$(w_c, w_{c-m}), \ldots, (w_c, w_{c-1}), (w_c, w_{c+1}), \ldots, (w_c, w_{c+m})$$

so that output words are within m distance of the context word w_c.

- Note that in this case the center word is a feature x and the output word is the target y.

Step 3: For each w_c, get the one-hot representations $e_{w_c} \in \mathbb{R}^{|\mathcal{V}|}$.

Step 4: For each w_c, get the vector representations $a_{w_c} = A^\top e_{w_c} \in \mathbb{R}^d$.

Skip-Gram Algorithm

Step 5: generate a score $z = Ba_{w_c} \in \mathbb{R}^{|\mathcal{V}|}$.

Step 6: turn the scores into probabilities $p = \text{softmax}(z) \in \mathbb{R}^{|\mathcal{V}|}$.

Requirement: we want the probabilities $p_{w_{c-m}}, p_{w_{c-m+1}}, \ldots, p_{w_{c+m-1}}, p_{w_{c+m}}$ to match the one-hot vectors $e_{w_{c-m}}, e_{w_{c-m+1}}, \ldots, e_{w_{c+m-1}}, e_{w_{c+m}}$.

Cross Entropy Loss:

$$\mathcal{L}(A, B) := -\sum_{\substack{i=-m \\ i \neq 0}}^{m} e_{w_{c+i}}^{\top} \log p.$$

Skip-Gram Algorithm

Explicitly:

$$\begin{aligned}
\mathcal{L}(A, B) &:= -\sum_{\substack{i=-m \\ i\neq 0}}^{m} \log p_{c+i} \\
&= -\log \prod_{\substack{i=-m \\ i\neq 0}}^{m} p(w_{c+i}|w_c) \\
&= -\log \prod_{\substack{i=-m \\ i\neq 0}}^{m} \frac{\exp b_{w_{c+i}}^\top a_{w_c}}{\sum_{j=1}^{|\mathcal{V}|} \exp b_{w_j}^\top a_{w_c}} \\
&= -\sum_{\substack{i=-m \\ i\neq 0}}^{m} b_{w_{c+i}}^\top a_{w_c} + 2m \log \sum_{j=1}^{|\mathcal{V}|} \exp b_{w_j}^\top a_{w_c}.
\end{aligned}$$

Outline

Problems

Recall Softmax computation:

$$p(w_{c+i} \mid w_c) = \frac{\exp b_{w_{c+i}}^\top a_{w_c}}{\sum_{j=1}^{|\mathcal{V}|} \exp b_{w_j}^\top a_{w_c}}.$$

Problem: the denominator sums over all words in the vocabulary. So one update costs $O(|\mathcal{V}|)$, which is too expensive when $|\mathcal{V}|$ is large.

Idea: approximate this using only a few "wrong" words each time.

- training is about comparison, not absolute scores.
- We want the true context to score higher than most other words:

$$b_{w_{c+i}}^\top a_{w_c} \quad > \quad b_w^\top a_{w_c} \quad \text{for many } w \neq w_{c+i}.$$

- If the true word beats a few random wrong words in many updates, it will also beat most words overall.

Negative Sampling

Softmax loss for one training pair (w_c, w_o)**:**

$$-\log p(w_o|w_c) = -b_{w_o}^\top a_{w_c} + \log \sum_{w \in \mathcal{V}} \exp\left(b_w^\top a_{w_c}\right).$$

Negative sampling: draw K negatives $w_1^-, \ldots, w_K^- \overset{\text{i.i.d.}}{\sim} P_n(w)$,

$$\mathcal{L}_{\mathsf{NS}}(w_c, w_o) = -\log \sigma\left(b_{w_o}^\top a_{w_c}\right) - \sum_{k=1}^{K} \log \sigma\left(-b_{w_k^-}^\top a_{w_c}\right),$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid function.

Complexity: reduced from $\propto |\mathcal{V}|$ to $\propto K$ per training pair.

Reference Value: $K \in [5, 20]$ for small data set and $K \in [2, 5]$ for large data set. Context length $C = 5, 10$ and embedding dimension $d = 50 \sim 300$. The vocabulary size $|\mathcal{V}| = 5k \sim 150k$.

Thank you for your attendance!

Questions?

Next week (Lab session): Bring your own laptop!

- **Environment:** remote server, GPU, PyTorch.
- **Embeddings:** query the most similar words using pretrained embeddings.
- **Training data:** tokenize raw text $\rightarrow$ build vocabulary $\mathcal{V} \rightarrow$ generate (w_c, w_o) window pairs; construct the negative-sampling distribution.
- **Skip-Gram with negative sampling:** implement the training loop and sampled negatives; monitor loss and qualitative neighbors; try different parameters.