Learning Embeddings

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
Mina Ghashami, Amazon
http://cs246.stanford.edu
What is Machine Learning?

- Machine learning is about *Optimization*
- Three key components:
  1. Training Data $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
  2. Loss function $\mathcal{L}$
  3. Model $f_\theta(x)$
- Optimize $f_\theta(x)$ on $D$ w.r.t loss function $\mathcal{L}$:
  - find the parameter $\theta$ that minimizes the expected loss on the training data

$$\min_f J(f) = \min_f \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f_\theta(x_i), y_i)$$
### Supervised learning:

- Given “labeled data” \( \{x, y\} \), learn \( f(x) = y \)
- Ex: classification, regression
  - In linear regression, the model \( f_\theta(x) = Wx + b \)
  - Parameters are \( \theta = \{W, b\} \)
  - The loss function is mean square error (MSE)

### Unsupervised learning:

- Given only “unlabeled data” \( \{x\} \), learn \( f(x) \)
- Ex: Dimensionality reduction, clustering
  - In SVD, the model is \( f(x) = \hat{x} = VVTx \) where \( V \) is right singular vectors of input matrix.
  - The loss function is L2 loss: \( L(x, \hat{x}) = \sum ||x - \hat{x}||^2 \)
All ML methods work with the input feature vectors \( \{x_1, x_2, \ldots, x_n\} \) and almost all of them require input features to be numerical.

From ML perspective, there are four types of features:

- Numerical (continues or discrete)
  - Continues: height
  - Discrete: age
- **Categorical** (ordinal or nominal)
  - Ordinal: level={beginner, intermediate, advanced}
  - Nominal: gender={male, female}, color={red, blue, green}
- Time series:
  - Average of home sale price over years
- Text
  - Bag of words
There are two ways to encode categorical var:

- Integer encoding
- One-hot encoding (and multi-hot encoding)

Consider the following movie dataset:

<table>
<thead>
<tr>
<th>Title</th>
<th>Provider</th>
<th>IMDB genres</th>
<th>Release Year</th>
<th>IMDb Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stranger Things</td>
<td>Netflix</td>
<td>drama, fantasy, horror</td>
<td>2016</td>
<td>8.7</td>
</tr>
<tr>
<td>Cocomelon</td>
<td>Prime Video</td>
<td>animation, comedy, family</td>
<td>2019</td>
<td>4.7</td>
</tr>
<tr>
<td>100 Foot Waves</td>
<td>HBO Max</td>
<td>documentary, sport</td>
<td>2021</td>
<td>8.1</td>
</tr>
<tr>
<td>I, Tonya</td>
<td>Hulu</td>
<td>biography, drama, comedy</td>
<td>2017</td>
<td>7.5</td>
</tr>
</tbody>
</table>
Assigns each category value with an integer

- \textit{provider :=\{Netflix, Prime Video, HBO Max, Hulu\}}, we assign them integers 1, 2, 3 and 4 respectively.

- **Pros**: dense representation
- **Cons**: It implies ordering between different categories: \textit{Netflix < Prime Video < HBO Max < Hulu}

<table>
<thead>
<tr>
<th>Title</th>
<th>provider</th>
<th>IMDB genres</th>
<th>Release year</th>
<th>IMDB rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stranger Things</td>
<td>1</td>
<td>drama, fantasy, horror</td>
<td>2016</td>
<td>8.7</td>
</tr>
<tr>
<td>Cocomelon</td>
<td>2</td>
<td>animation, comedy, family</td>
<td>2019</td>
<td>4.7</td>
</tr>
<tr>
<td>100 Foot Waves</td>
<td>3</td>
<td>documentary, sport</td>
<td>2021</td>
<td>8.1</td>
</tr>
<tr>
<td>I, Tonya</td>
<td>4</td>
<td>biography, drama, comedy</td>
<td>2017</td>
<td>7.5</td>
</tr>
</tbody>
</table>

- Makes more sense to use it for \textbf{ordinal variables}:
  - Such as “Education” = \{Diploma, Undergrad, Masters, PhD\}
  - But still it implies values are equally spaced out
First do integer encoding, then create a **binary vector** that represents the numerical values

- Ex: following integer encoding on provider:
  - Netflix -> 1, Prime Video -> 2, HBO Max -> 3, Hulu -> 4
- create a binary vector of length 4 for each value:

<table>
<thead>
<tr>
<th>Provider</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netflix</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>Prime Video</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>HBO Max</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>Hulu</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

The integer encoding is the index into the vector
## Multi-hot Encoding

- An extension of one-hot encoding when categorical variable can take **multiple** values at the same time
  - Ex: There are 28 distinct IMDB genres a movie can take multiple genres, e.g. *stranger things* is drama, fantasy, horror.

<table>
<thead>
<tr>
<th>IMDB genres</th>
</tr>
</thead>
<tbody>
<tr>
<td>[(1, 'Action'), (2, 'Comedy'), (3, 'Short'), (4, 'Western'), (5, 'Drama'), (6, 'Horror'), (7, 'Music'), (8, 'Thriller'), (9, 'Animation'), (10, 'Adventure'), (11, 'Family'), (12, 'Fantasy'), (13, 'Sport'), (14, 'Romance'), (15, 'Crime'), (16, 'Sci-Fi'), (17, 'Biography'), (18, 'Musical'), (19, 'Mystery'), (20, 'History'), (21, 'Documentary'), (22, 'Film-Noir'), (23, 'News'), (24, 'Game-Show'), (25, 'Reality-TV'), (26, 'War'), (27, 'Talk-Show'), (28, 'Adult')]</td>
</tr>
</tbody>
</table>

| Stranger things | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Cocomelon        | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 100 foot wave    | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| I, Tonya         | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
### Applying encodings on Movies dataset

<table>
<thead>
<tr>
<th>Provider</th>
<th>IMDB genres</th>
<th>Year</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stranger things</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>2016</td>
<td>8.7</td>
</tr>
<tr>
<td>cocomelon</td>
<td>0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>2019</td>
<td>4.7</td>
</tr>
<tr>
<td>100 foot waves</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0</td>
<td>2021</td>
<td>8.1</td>
</tr>
<tr>
<td>I, Tonya</td>
<td>0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0</td>
<td>2017</td>
<td>7.5</td>
</tr>
</tbody>
</table>

- Data dimensions increased from 5 to 35. It will blow up to thousands or a million if we multi-hot encode title!
- One-hot and multi-hot encodings are not practical for features with large value sets.
One-hot/multi-hot encodings:

- **pros**: simple, robust, the observation that simple models trained on huge amounts of data outperform complex systems trained on less data
- **cons**: sparse and high dimensional, does not capture semantic similarity

In a corpus of documents with one **million** distinct words:

- high dimensional: multi-hot encodings are 1-million dimensional
- Sparse: an average document contains 500 words therefore the multi-hot encodings are > 99.95% sparse
- lack of semantic: encoding of two words ‘good’ and ‘great’ are as different as encoding of ‘good’ and ‘bad’!

An embedding is a translation of a high-dim vector into a low-dim space. An embedding is a:

- Dense representation (floating-point value)
- Low-dimensional vector
- Captures semantic similarity
Standard dimensionality Reduction methods

- Singular value decompositions (SVD)

**A**: Input data matrix: $m \times n$ matrix (e.g., $m$ documents, $n$ terms)

$r$ : rank of the matrix $A$ – often $r < \min(m,n)$

**U**: Left singular vectors: $m \times r$ matrix ($m$ documents, $r$ concepts)

**Σ**: Singular values: $r \times r$ diagonal matrix (strength of each ‘concept’)

**V**: Right singular vectors: $n \times r$ matrix ($n$ terms, $r$ concepts)
Review: SVD as an embedder

- **$U$, $V$: column orthonormal**
  - $U^T U = I$; $V^T V = I$ ($I$: identity matrix)
  - Columns are orthogonal unit vectors hence they define an $r$-dimensional subspace
    - $U$ defines an $r$-dim subspace in $\mathbb{R}^m$
    - $V$ defines an $r$-dim subspace in $\mathbb{R}^n$

- Projecting $A$ onto $V$ and $U$ produces embeddings:
  - Since $A = U \Sigma V^T$ then $AV = U \Sigma$ are row embeddings
  - Since $A = U \Sigma V^T$ then $U^T A = \Sigma V^T$ are col embeddings
Ex: compute document & word embeddings

Step 1: given a corpus of documents convert it to BOW vectors \(\rightarrow\) get a term-document matrix

- Use term frequencies (tf), or normalize using tf-idf

<table>
<thead>
<tr>
<th></th>
<th>data</th>
<th>science</th>
<th>spark</th>
<th>Stanford</th>
<th>learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>document 1</td>
<td>10</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>document 2</td>
<td>0</td>
<td>9</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>document 3</td>
<td>1</td>
<td>2</td>
<td>20</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>document 4</td>
<td>14</td>
<td>11</td>
<td>1</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>document 5</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>document 6</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>document 7</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>
Step 2: apply SVD on the term-document matrix and pick a value $r \leq \text{rank}(A)$

Here, we set $r = 3$. 

$A = \begin{bmatrix}
10 & 15 & 3 & 0 & 10 \\
0 & 9 & 2 & 8 & 2 \\
1 & 2 & 20 & 0 & 4 \\
14 & 11 & 1 & 32 & 2 \\
5 & 1 & 7 & 12 & 5 \\
6 & 3 & 5 & 1 & 1 \\
2 & 3 & 5 & 2 & 7
\end{bmatrix}$

$U = \begin{bmatrix}
-0.30 & 0.41 & -0.79 \\
-0.25 & 0.03 & -0.12 \\
-0.14 & 0.74 & 0.5 \\
-0.83 & -0.40 & 0.12 \\
-0.33 & 0.11 & 0.31 \\
-0.13 & 0.20 & -0.04 \\
-0.14 & 0.27 & -0.05
\end{bmatrix}$

$V^T = \begin{bmatrix}
0.42 & 0 & 0 \\
0 & 23.8 & 0 \\
0 & 0 & 16.7
\end{bmatrix}$

Here, the value $r = 3$. 

$X = \begin{bmatrix}
0.06 & 0.21 & 0.78 & -0.45 & 0.37 \\
-0.26 & -0.63 & 0.55 & 0.40 & -0.28
\end{bmatrix}$
Step 3: compute embedding of documents as

\[ \text{emb} = [\langle \text{doc, v1} \rangle, \langle \text{doc, v2} \rangle, \langle \text{doc, v3} \rangle] \]

- \( \langle \text{doc, v1} \rangle = \langle [10, 15, 3, 0, 10], v1 \rangle = -12.7 \)
**Step 3:** compute embedding of documents as

\[ \text{emb} = [\langle \text{doc}, v1 \rangle, \langle \text{doc}, v2 \rangle, \langle \text{doc}, v3 \rangle] \]

- \( \langle \text{doc}, v1 \rangle = \langle [10, 15, 3, 0, 10] \rangle, \quad v1 = -12.7 \)
- \( \langle \text{doc}, v2 \rangle = \langle [10, 15, 3, 0, 10] \rangle, \quad v2 = 9.79 \)
Step 3: compute embedding of documents as

$$\text{emb} = [\langle \text{doc}, \text{v1} \rangle , \langle \text{doc}, \text{v2} \rangle , \langle \text{doc}, \text{v3} \rangle]$$

- $$\langle \text{doc}, \text{v1} \rangle = \langle [10,15,3,0,10] , \text{v1} \rangle = -12.7$$
- $$\langle \text{doc}, \text{v2} \rangle = \langle [10,15,3,0,10] , \text{v2} \rangle = 9.79$$
- $$\langle \text{doc}, \text{v3} \rangle = \langle [10,15,3,0,10] , \text{v3} \rangle = -13.9$$
**Step 3**: compute embedding of documents as

\[ \text{emb} = [\langle \text{doc}, \text{v1} \rangle , \langle \text{doc}, \text{v2} \rangle , \langle \text{doc}, \text{v3} \rangle] \]

\[
\begin{align*}
\langle \text{doc}, \text{v1} \rangle &= \langle [10, 15, 3, 0, 10] , \text{v1} \rangle = -12.7 \\
\langle \text{doc}, \text{v2} \rangle &= \langle [10, 15, 3, 0, 10] , \text{v2} \rangle = 9.79 \\
\langle \text{doc}, \text{v3} \rangle &= \langle [10, 15, 3, 0, 10] , \text{v3} \rangle = -13.9
\end{align*}
\]

\[ \text{emb1} = [-12.7, 9.79, -13.9] \]
SVD is impractical on real-world datasets

- There are 0.5 billion wiki pages, and 4 billion words.
- SVD is computationally prohibitive, as it requires to load all data in memory
- SVD is a linear embedder
- Not utilizing sparsity
- Orthonormality constraint is an overkill
State of the art embedders are among neural networks

Can we use neural networks to create non-linear embedding?
Neural Networks Fundamentals
A neural network is a collection of neurons that are connected in an *acyclic graph*. Outputs of some neurons are inputs to other neurons, and they are organized into layers.

credit: cs231
**Neural Network**

- **Fully-connected layer** is the most common layer type:
  - neurons between two adjacent layers are fully pairwise connected
  - neurons within a single layer share no connections

- Number of hidden layers and neurons in each hidden layer are hyperparameters of the network
A neuron is a classifier

- Input: \([x_0, x_1, x_2]\]
- Output = \(f(\sum w_i x_i + b)\)

\(f\) is the activation function. It takes a single number and performs an operation on it. Some choices are:

1. Sigmoid \(\sigma(x) = 1/(1 + e^{-x})\)
2. Tanh \(\tanh(x) = 2\sigma(2x) − 1\)
3. Relu \(f(x) = \max(0, x)\)

Each neuron performs a dot product with the input and its weights, adds the bias and applies the activation function.

Credit: cs231
Neural Network: A Layer

- Consider two neurons in a hidden layer

Each layer computes \( \mathbf{x}^{(l+1)} = \sigma(W_l \mathbf{x}^{(l)} + b^l) \)
- \( W_l \) is weight matrix that transforms representation at layer \( l \) to layer \( l + 1 \)
- \( b^l \) is bias at layer \( l \), and is added to the linear transformation of \( \mathbf{x} \)
- \( \sigma \) is sigmoid activation function
This network computes \( f(x) = W_1(\sigma(W_0x^{(0)} + b^0) + b^1) \)

- Notice without activation functions, \( f(x) \) will be linear in \( x \)!!

\[
 f(x) = W_1W_0x^{(0)} + W_1b^0 + b^1
\]
A loss function $\mathcal{L}$ is required to train the NN.

- Example: L2 loss
  \[ \mathcal{L}(y, f(x)) = \|y - f(x)\|_2 \]

- Common loss functions for **regression**:
  - L2 loss, L1 loss, huber loss, ...

- Common loss functions for **classification**:
  - Cross entropy, max margin (hinge loss), ...

- Example
Cross Entropy Loss

- **Common loss for classification tasks**
  - Defined between one-hot of true label and the predicted probability distribution over classes

- **Ex: Task = multi-class classification with 5 classes**
  - True label \( y \) belongs to class 3, so one-hot of \( y = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} \)
  - Predicted probability distribution \( \hat{y} = f(x) = \begin{bmatrix} 0.1 & 0.3 & 0.4 & 0.1 & 0.1 \end{bmatrix} \)

  - \( \text{CE}(y, \hat{y}) = - \sum_{i=1}^{C} (y_i \log \hat{y}_i) = - \log(\hat{y}_{\text{correct class}}) \)

  - \( y_i, \hat{y}_i \) are the actual and predicted value of the \( i \)-th class.

- **Intuition:** the lower the loss, the closer the prediction is to one-hot \( y \)

\[5/18/23\] Jure Leskovec & Mina Ghashami, Stanford University
Often model’s output is a score for each class, not a probability distribution.

To convert to probability distribution:

$$ f(x) = \text{Softmax}(g(x)) $$

It normalizes a vector into a probability distribution that sums to 1.

Cross entropy loss sometimes is referred to as softmax loss.
How to optimize the **Loss function**?

**Gradient descent:**

\[
\nabla_\Theta \mathcal{L} = \left( \frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \ldots \right)
\]

\(\Theta_1, \Theta_2 \ldots : \) components of \(\Theta\)

- repeatedly update weights in the (opposite) direction of gradients until convergence
  \[\Theta \leftarrow \Theta - \eta \nabla_\Theta \mathcal{L}\]

- **Learning rate (LR) \(\eta\):**
  - Hyperparameter that controls the size of gradient step

- **Ideal termination condition:** 0 gradient
  - In practice, we stop training if it no longer improves performance on the **validation set** (part of dataset we hold out from training)
There are much more about NN including:

- Minibatch Stochastic gradient descent
- Batch size, Epoch
- Learning rate scheduling
- Optimizers to improve over SGD

However they are not the focus of today’s lecture.

Now that we know fundamentals, let’s use NN to learn embeddings
Learning Embeddings using Neural Networks
Agenda

- We will work with three examples:
  1. **Word embeddings**
     - Word embeddings produced by **Word2Vec** model
     - Converts one-hot encoding to dense embedding
     - Unsupervised mode
  2. **Video recommendation**
     - Converts one-hot encoding to embedding
     - Item-item collaborative filtering
  3. **Autoencoders:**
     - Learn representation by reconstructing input
     - Unsupervised mode
Word Embedding
  - Many techniques: Word2Vec, Glove, BERT, fastText
  - Today’s lecture: Word2Vec

Word2Vec was Developed at Google in 2013 (paper)

Word2Vec is a statistical method
  - Very efficiently in learning word embedding
  - It is unsupervised, and task independent.
Word2Vec

- Word2Vec comes in two architectures:
  - Continuous bag of words (CBOW)
  - Skip Gram
    (We will discuss skip-gram model today)

- The two methods are very similar, both use a shallow neural network (only 1 hidden layer) to learn word representations.

- The key idea of **word2Vec** is that words with similar context have similar meanings.
  - It learns embedding based on the usage of words.
**Word2Vec: target and context**

**The key idea:** The more often a word appears in the context of certain other words, the closer they are in meaning.

**This is how we define context:**
Given a document, set the window size=N. Window size is a hyperparameter.

For any given word (call it “target word”), \( N \) words to its left & \( N \) words to its right are the “context words”.

- Document “I read sci-fi books”, and window size = 2
  - Target = “I” → context words = “read”, “sci-fi”.
  - Target = “read” → context words = “I”, “sci-fi”, “books”

Given a document, we can slide the window from left to right and find all pairs of (target, context) words
Document: “I read sci-fi books and drink orange juice”. Let window size = 2
The highlighted word is the target word. Other words in the box are context words.
Word2Vec: architecture

- Word2Vec is a 2 layers NN

- Given the target word as input, it predicts context words
  - In our example, if target word = “I” → context = “read”, “sci-fi”

- Why is it called Skip-Gram?
  - window size = N, the model predicts the N-grams words except the current word as it is the input to the model, hence the name skip-gram.
**Word2Vec: high level architecture**

- Set window size = 2

Given the word at position $t$, it predicts the nearby context words both from past and future.

** For simplicity, in this lecture we predict only one context word.
Set window size = 2

Given the word at position $t$, it predicts the nearby context words both from past and future.

** For simplicity, in this lecture we predict only one context word.
Word2Vec: architecture

- Let $V =$ size of vocabulary
- Let $N =$ embedding dimension

one-hot vector of target word at position $t$: $w(t)$

The hidden layer is the embedding layer.

one-hot vector of context word: $w(t-1)$
A softmax function is applied on output layer.
Word2Vec: architecture

- The hidden layer are all linear neurons, no activation!
- Two different weight matrices:
  - $W_{V\times N}$: from input to hidden layer
  - $W'_{N\times V}$: from hidden to output layer
After training the network:

- The embedding of a word is obtained by $x^T W$ i.e. matrix multiplication between word’s one-hot vector and learned weights $W_{V \times N}$.
How is the network trained?
- There are no labels. It is an unsupervised task (it is a statistical method based on co-occurrence of words in one window).
- We therefore create a fake task!

Fake task = given a target word, predict its context words

Decisions to make:
- How to make training data?
- What is the loss function?
Word2Vec: Training data

- Take all your corpus of data = \{d_1, d_2, \ldots\}
  - Millions of documents, wiki pages, blot posts, etc.

- Tokenize all documents and build vocabulary
  - Many methods: wordpiece, BytePairEncoding (BPE)
  - Simplest method: k-gram

- If you take \( k=1 \) and do tokenization by words then 1-gram is equivalent to split sentences by words
Word2Vec: Training data

- Move sliding window over tokenized documents and collect training data
- Ex: Document = {I read sci-fi books and drink orange juice}

<table>
<thead>
<tr>
<th>Input document (window size = 2)</th>
<th>Training data (target, context)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="" /></td>
<td>(I, read) (I, sci-fi)</td>
</tr>
<tr>
<td><img src="image2" alt="" /></td>
<td>(read, I) (read, sci-fi) (read, books)</td>
</tr>
<tr>
<td><img src="image3" alt="" /></td>
<td>(sci-fi, I) (sci-fi, read) (sci-fi, books)</td>
</tr>
<tr>
<td><img src="image4" alt="" /></td>
<td>(books, read) (books, sci-fi) ...</td>
</tr>
<tr>
<td><img src="image5" alt="" /></td>
<td>(and, sci-fi) (and, books) (and, drink)</td>
</tr>
<tr>
<td><img src="image6" alt="" /></td>
<td>(drink, books) (drink, and) ....</td>
</tr>
<tr>
<td><img src="image7" alt="" /></td>
<td>(orange, drink) (orange, juice)</td>
</tr>
<tr>
<td><img src="image8" alt="" /></td>
<td>(juice, drink) (juice, orange)</td>
</tr>
</tbody>
</table>
Word2Vec: Loss function

- Given the topology of the network, if \( x \) is the input and \( y \) is the output, then

\[
y = \text{softmax}(W^TW^Tx)
\]

- We train against target-context pairs \((w_t, w_c)\)
  - The context word \( w_c \) represents the ideal prediction, given the target word \( w_t \)
  - \( W_c \) is represented as one-hot, i.e. it has value 1 at some position \( j \) and other positions are 0

\[
w_c = [0,0,0,\ldots,1,0,0,0\ldots,0]
\]
The loss function needs to evaluate the output layer at the same position \( j \), i.e. \( y_j \) (remember \( y \) is a probability distribution; ideal value of \( y_j \) is being 1)

We use cross-entropy loss function:

\[
CE(w_c, y) = -\log(y_{\text{correct class}})
\]

Since \( w_c = [0,0,0,...,1,0,0,0..,0] \)
And \( y = [0.02, 0.11,....,0.8, 0, 0.031, ...] \)

the loss value would be \( L = -\log(0.8) \)
Now that loss function is clear, we want to find the values of $W$ and $W'$ that minimize it.
- We want our model to learn the weights.

We use gradient descent to tackle this
- We find derivatives $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial W'}$ and update weights as
  $$W_{\text{new}} = W_{\text{old}} - \mu \frac{\partial L}{\partial W}$$
Document = {I read sci-fi books and drink orange juice}

Since this is only doc in our corpus, our vocab is
vocabulary = ["I", "read", "sci-fi", "books", "and", "drink", "orange", "juice"] and V = 8

We execute one forward pass using above document

Step 1: assign one-hot vectors to words

<table>
<thead>
<tr>
<th>Word</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>[1, 0, 0, 0, 0, 0, 0, 0]</td>
</tr>
<tr>
<td>read</td>
<td>[0, 1, 0, 0, 0, 0, 0, 0]</td>
</tr>
<tr>
<td>sci-fi</td>
<td>[0, 0, 1, 0, 0, 0, 0, 0]</td>
</tr>
<tr>
<td>books</td>
<td>[0, 0, 0, 1, 0, 0, 0, 0]</td>
</tr>
<tr>
<td>and</td>
<td>[0, 0, 0, 0, 1, 0, 0, 0]</td>
</tr>
<tr>
<td>drink</td>
<td>[0, 0, 0, 0, 0, 1, 0, 0]</td>
</tr>
<tr>
<td>orange</td>
<td>[0, 0, 0, 0, 0, 0, 1, 0]</td>
</tr>
<tr>
<td>juice</td>
<td>[0, 0, 0, 0, 0, 0, 0, 1]</td>
</tr>
</tbody>
</table>
- size of vocabulary = 8, Let’s set embedding dim = 3

- Input layer: one-hot vector of target word
- 3-dim embedding
- Output layer: probability of each word being in context
Word2Vec: Example

- If target word = books and weight matrix $W_{V \times N}$ be as following:

$$W_{V \times N} = \begin{bmatrix}
1 & 2 & 2 \\
-1.2 & -3 & -2 \\
1.2 & 1.1 & 0.5 \\
0.5 & 2.3 & 2 \\
-1.1 & 0.6 & -1 \\
1 & -1 & 2 \\
0.3 & 1.2 & 0.7
\end{bmatrix}$$

- Then

$$[0,0,0,1,0,0,0,0] \times \begin{bmatrix}
1 & 2 & 2 \\
-1.2 & -3 & -2 \\
1.2 & 1.1 & 0.5 \\
0.5 & 2.3 & 2 \\
-1.1 & 0.6 & -1 \\
1 & -1 & 2 \\
0.3 & 1.2 & 0.7
\end{bmatrix} = [0.5, 2.3, 2.2]$$
Word2Vec: Example

- If target word = books, and $W_{VxN}$ given:

```
Target word = books

3-dim embedding

Output layer: probability of each word being in context
```

```
Word2Vec: Example

- If the weight matrix $W'_{N \times V}$ be as following:

$$W'_{N \times V} = \begin{bmatrix}
1 & 2 & 2 & 0 & 0.7 & 1.3 & -1 & -0.1 \\
1.2 & 0.5 & -1 & 1 & 0.3 & 2 & 0.6 & 1 \\
-1 & 1.6 & -0.5 & 1.4 & 2.3 & 1 & 1 & 0.6
\end{bmatrix}$$

- Then

$$[0.5, 2.3, 2.2] \times \begin{bmatrix}
1 & 2 & 2 & 0 & 0.7 & 1.3 & -1 & -0.1 \\
1.2 & 0.5 & -1 & 1 & 0.3 & 2 & 0.6 & 1 \\
-1 & 1.6 & -0.5 & 1.4 & 2.3 & 1 & 1 & 0.6
\end{bmatrix} = [1.0, 5.6, -2.4, 5.3, 6.1, 7.4, 3.0, 3.5]$$
If $W'_{N \times V}$ given:

Target word = books

3-dim embedding
If $W'_{NxV}$ given:

- Target word = books

We apply softmax functions to turn them into probabilities.

Each output neuron $x = \frac{e^x}{\sum e^x}$
If $W'_{NxV}$ given:

Target word = books

3-dim embedding

Probability distribution of context word

- $W_{VxN}$
- $W'_{NxV}$

- 0.5
- 2.3
- 2.2

- 1.0
- 5.6
- -2.4
- 5.3
- 6.1
- 7.4
- 3
- 3.5

- 0.001
- 0.104
- 0
- 0.077
- 0.177
- 0.627
- 0.008
- 0.013
If $W'_N \times V$ given:

Target word = books

3-dim embedding
If $W'_N \times V$ given:

Target word = books

3-dim embedding

probabilities

One-hot of context

$W_{V \times N}$
The network learns by comparing softmax vector to the one-hot of true context word.

In our example, \text{target} = “books”, one correct context = “read” but we predicted “drink”

- Predicted vector = 
  \[0.001, 0.104, 0, 0.077, 0.177, 0.627, 0.008, 0.013]\n
- One-hot of “read” = \[0, 1, 0, 0, 0, 0, 0, 0, 0\]

\[L = -\ln(0.104) = 2.26\]

Then
Word2Vec: Summary

- Word2Vec comes in two architecture:
  - CBOW: Given context words, it predicts target word
  - Skip-Gram: Given target word, predicts context words

- Skip Gram method:
  - works well with small amount of data and is found to represent rare words well

- CBOW method:
  - is faster and more suitable for large data, it has better representations for more frequent words.
Word2Vec: Summary

- Word2vec assigns an embedding to every word in the vocabulary
- Embedding dimension << size of the vocabulary
Task independent vs Task specific

- So far we worked with unsupervised data
  - A corpus of documents
- We learned embeddings that was not tied to any classification or regression task
  - We created a fake task of predicting nearby words
- Alternatively, we can learn embeddings for a specific tasks such as classification
Video Recommendation
Example: Recommending movies

- **Input**: 1 million movies, and 500k users who have watched some of these movies
- **Task**: recommend movies to users

- We solved this problem before using collaborative filtering, and latent factor models

- Here, we formulate it as multi-class classification where each movie is a class. We use neural network to learn embeddings for movies such that similar movies have similar embeddings.
**Train-Test split**: First split data into train and test. For every user, randomly hold out few movies they have watched as test and use the rest to build train data.

<table>
<thead>
<tr>
<th>Full data</th>
<th>Train</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice -&gt; m1, m2, m3, m4, m5</td>
<td>Alice -&gt; m4, m5</td>
<td>Alice -&gt; m3, m2</td>
</tr>
<tr>
<td>Bob -&gt; m8, m9, m21</td>
<td>Bob -&gt; m8, m9</td>
<td>Bob -&gt; m21</td>
</tr>
<tr>
<td>Sam -&gt; m2, m6, m10</td>
<td>Sam -&gt; m6, m10</td>
<td>Sam -&gt; m2</td>
</tr>
</tbody>
</table>
Example: Recommending movies

- **Build train data**: We then build train data as pairs (movie1, movie2) where both movies are watched by same user

<table>
<thead>
<tr>
<th>Train</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>m1, m4, m5</td>
</tr>
<tr>
<td>Bob</td>
<td>m8, m9</td>
</tr>
<tr>
<td>Sam</td>
<td>m6, m10</td>
</tr>
</tbody>
</table>

Alice -> m1, m4, m5

<table>
<thead>
<tr>
<th>Alice</th>
<th>m3, m2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>m21</td>
</tr>
<tr>
<td>Sam</td>
<td>m2</td>
</tr>
</tbody>
</table>

Prepare train data for NN

<table>
<thead>
<tr>
<th>Train</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>(m1, m4)</td>
<td>m3, m2</td>
</tr>
<tr>
<td>(m4, m5)</td>
<td></td>
</tr>
<tr>
<td>(m8, m9)</td>
<td>m21</td>
</tr>
<tr>
<td>(m9, m8)</td>
<td></td>
</tr>
<tr>
<td>(m6, m10)</td>
<td>m2</td>
</tr>
<tr>
<td>(m10, m6)</td>
<td></td>
</tr>
</tbody>
</table>

5/18/23
We then build a neural network that performs item-item collaborative filtering while learning 3-dim embeddings.
Example: Recommending movies

- How to recommend a movie to a user e.g. Alice?
  - Alice has watched m1, m4, m5 in train
  - Find movies that have similar embeddings to m1
    - Similarity score = \(<\text{emb}(m1), \text{emb}(v)>\) for any movie v
    - Find top 5 movies with highest similarity score
    - Recommend them to Alice

- Or even better: repeat above for m1, m4 and m5
  - Recommend movies in intersection of above sets
So far...

- So far we have seen examples of converting one-hot encodings to embeddings
  - word2Vec
  - Supervised NN with one-hot input vector

- We can use NN to learn embedding from dense feature vectors
  - What other method does the same? SVD, PCA
Autoencoders
Autoencoder

Autoencoder are an extension of PCA to non-linear space

They are a special type of neural network that is trained to copy input to output except that it has to go through a bottleneck

- They are unsupervised too

- It learns to compress the data while minimizing the reconstruction error.
**Input layer**: is input feature vector. It does not need to be one-hot vectors. Here, input data is 6-dim vectors.

**Bottleneck layer**: is the bottleneck as it projects down 6-dim vector to 3-dim space. It constrains the amount of information that traverses the network.

**Output layer**: is the reconstructed input from 3-dim to 6-dim.
There can be multiple hidden layers between Input layer and bottleneck layer, similarly between bottleneck layer and output layer.
Two main components in their architecture:

- **Encoder**: a function $f$ that compresses the input into a latent-space representation
  - $f(x) = h$ such that $\text{dimension}(h) < \text{dimension}(x)$

- **Decoder**: a function $g$ that reconstructs the input from the latent space representation
  - $g(h) \sim x$, i.e. bring $h$ back to the original space
The bottleneck is the key:
- Without an information bottleneck, autoencoder could learn to memorize the input data!!

There are different types of autoencoders:
- Undercomplete, denoising, sparse, variational
- Today, we talk about undercomplete autoencoder
  - i.e bottleneck dimension < input dimension
The loss function to train an undercomplete AE is *reconstruction loss*:

\[
L(x, \hat{x}) = \|x - \hat{x}\|_{1,2}
\]

No regularization term is needed in undercomplete AE. To ensure the model is not memorizing the input data we regulate:

- size of the bottleneck layer
- number of hidden layers
A neuron has activation functions. As long as activation function is not Identity, we learn non-linear embedding.

If we use Identity activation functions in hidden layers we convert back to PCA and produce similar dimensionality reduction as PCA.
Autoencoder: summary

- Non-linear PCA
- A neural network that is trained to copy input to output
  - it passes data through a bottleneck
  - Reconstruction loss function: L1, KL divergence
  - Unsupervised

- There are different types of autoencoders:
  - Undercomplete, denoising, sparse, variational
  - We studied undercomplete AE.
Today’s lecture

- Categorical variables
  - Integer encoding
  - One-hot encoding
  - Multi-hot
- How to transform encodings to embeddings
  - SVD
  - Neural networks
- Task independent vs task specific embedding
  - Word2Vec architecture
  - Autoencoder architecture