Sparse and dense embeddings as input to neural networks

Input to feedforward neural networks

- A bag-of-word model where the input $\mathbf{x}$ is the count of each word (feature) x_i.
  - The connections from word (feature) i to each of the hidden units z_k form a vector $\theta_i^{(x \to z)}$ that is sometimes described as the embedding of word i.
  - With sufficient training data, word embeddings can be learned within the same network as the classification task.
- *Pretrained* word embeddings learned separately from unlabeled data, using techniques such as Word2Vec and GLOVE.
- *Contextualized* word embeddings (e.g., ELMO, BERT) that are computed dynamically for a word sequence. The requires more advanced architectures (Transformers) that we will talk about later in the course.

One-hot encodings for features

A *one-hot* encoding is one in which each dimension corresponds to a unique feature, and the resulting feature vector of a classification instance can be thought of as the sum of indicator feature vectors in which a single dimension has a value of one while all others have a value of zero.

Example:

When considering a bag-of-words representation over a vocabulary of 40000 words. A short document of 20 words will be represented with a very sparse 40000-dimensional vector in which **at most** 20 dimensions have non-zero values

Sparse vectors for text classification

Sparse vectors for text classification can be viewed as a summation of one-hot features for a text instance:

Assignment Project Exam Help

$$\begin{bmatrix}0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$$

$$+$$

https://powcoder.com

$$\begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

$$+$$

Add WeChat powcoder

$$\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\end{bmatrix}$$

$$=$$

$$\begin{bmatrix}0 & 1 & 0 & 1 & 0 & 0 & 1 & 0\end{bmatrix}$$

Shortcomings for sparse representations

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

- Each feature is a sparse vector in which one dimension is 1 and the rest are 0s (thus "one-hot")
- Dimensionality of one-hot vector is same as number of distinct features
- Features can be completely independent of one another. The feature "word is 'dog' " is as dissimilar to "word is 'thinking' " as it is to "word is 'cat"'
- Features for one classifying instance can only combined by summation.
- A recent trend is to use dense representations that can capture similarities between features, which lead to better generalizations to new data.

Dense vectors for text classification

- Extract a set of linguistic features $f_1, \cdots, f_k$ that are relevant for predicting the output class
- For each feature f_i of interest, retrieve the corresponding vector v_i, which can be pre-trained, pre-computed, or random initialized.
- Each core feature is embedded into a d-dimensional space (typically 50-600), and represented as a vector in that space.
- Combine the vectors (either by concatenation, summation, or a combination of both) into an input vector x for a classification instance.
  - Note: concatenation if we care about relative position, but doesn't work for variable-length vectors such as document classification
- Model training will cause similar features to have similar vectors - information is shared between similar features.

Relationship between one-hot and dense vectors

- Dense representations are typically pre-computed or pre-trained word embeddings.

- One-hot and dense representations may not be as different as one might think.

- In fact, using sparse, one-hot vectors as input when training a neural network amounts to dedicating the first layer of the network to learning a dense embedding vector [for each feature] based on training data.

- With task-specific word embedding, the training set is typically smaller, but the training objective for the embedding and the task objective are one and same
- With pre-trained word embeddings, the training data is easy to come by (just unannotated text), but the embedding objective and task objective may diverge.

Two ways of obtaining dense word vectors

- Count based methods, known in NLP as Distributional Semantic Models (DSM) or Vector Semantic Models (VSM)
- Predictive methods, originating from the neural network community, aimed at producing Distributed Representations for words, commonly known as word embeddings
  - Distributed word representations were initially a by-product of neural language models and later became a separate task on its own

Distributional semantics

- Based on the well-known observation of Z. Harris: Words are similar if they occur in the same context (Harris, 1954)
- Further summarized it as a slogan: "You shall know a word by the company it keeps." (J. R. Firth, 1957)
- A long history of using word-context matrices to represent word meaning where each row is a word and each column represents a context word it can occur with in a corpus
- Each word is represented as a sparse vector in a high-dimensional space
- Then word distances and similarities can be computed with such a matrix

Steps for building a distributional semantic model

- Preprocess a (large) corpus: tokenization at a minimum, possibly lemmatization, POS tagging, or syntactic parsing
- Define the "context" for a target term (words or phrases). The context can be a window centered on the target term, terms that are syntactically related to the target term (subject-of, object-of, etc.).
- Compute a term-context matrix where each row corresponds to a term and each column corresponds to a context term for the target term.
- Each target term is then represented with a high-dimensional vector of context terms.

Mathematical processing for building a DSM

- Weight the term-context matrix with association strength metrics such as Positive Pointwise Mutual Information (PPMI) to correct frequency bias

$$PPMI(x, y) = max(\log \frac{p(x, y)}{p(x)p(y)}, 0)$$

- Its dimensionality can also be reduced by matrix factorization techniques such as *singular value decomposition* (SVD)

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}$$

$$\boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{U} \in \mathbb{R}^{m \times k}, \boldsymbol{\Sigma} \in \mathbb{R}^{k \times k}, \boldsymbol{V} \in \mathbb{R}^{n \times k}, n >> k$$

- This will result in a matrix that has much lower dimension but retains most of the information of the original matrix.

Getting pre-trained word embeddings using predictive methods

- Learns word embeddings from large naturally occurring text, using various language model objectives.
  - Decide on the context window
  - Define the objective function that is used to predict the context words based on the target word or predict the target word based on context
  - Train the neural network
  - The resulting weight matrix will serve as the vector representation for the target word
- "Don't count, predict!" (Baroni et al, 2014) conducted systematic studies and found predict-based word embeddings outperform count-based embeddings.
- One of popular early word emdeddings are Word2vec embeddings.

Word2vec

- Word2vec is a software package that consists of two main models: CBOW (Continuous Bag of Words) and Skip-gram.
- It popularized the use of distributed representations as input to neural networks in natural language processing, and inspired many follow-on works, e.g., GLOVE, ELMO, BERT, XLNet)
- It has it roots in language modeling (the use of window-based context to predict the target word), but gives up the goal of getting good language models and focuses instead on getting good word embeddings.

Understanding word2vec: A simple CBOW model with only one context word input

- Input $\boldsymbol{x} \in \mathbb{R}^V$ is a one-hot vector where $x_k = 1$ and $x_{k'} = 0$ for $k' \neq k$. $\boldsymbol{\Theta} \in \mathbb{R}^{N \times V}$ is the weight matrix from the input layer to the hidden layer. Each column of $\boldsymbol{\Theta}$ is an N-dimensional vector representation $\boldsymbol{v}_w$ of the associated word of the input layer.

$$\boldsymbol{z} = \boldsymbol{\Theta}\boldsymbol{x} := \boldsymbol{v}_{w_i}$$

- $\boldsymbol{\Theta'} \in \mathbb{R}^{V \times N}$ is the matrix from the hidden layer to the output layer and $\boldsymbol{u}_{w_j}$ is the j-th row of $\boldsymbol{\Theta'}$. A "similarity" score o_j for each target word w_j and context word w_i can be computed as:

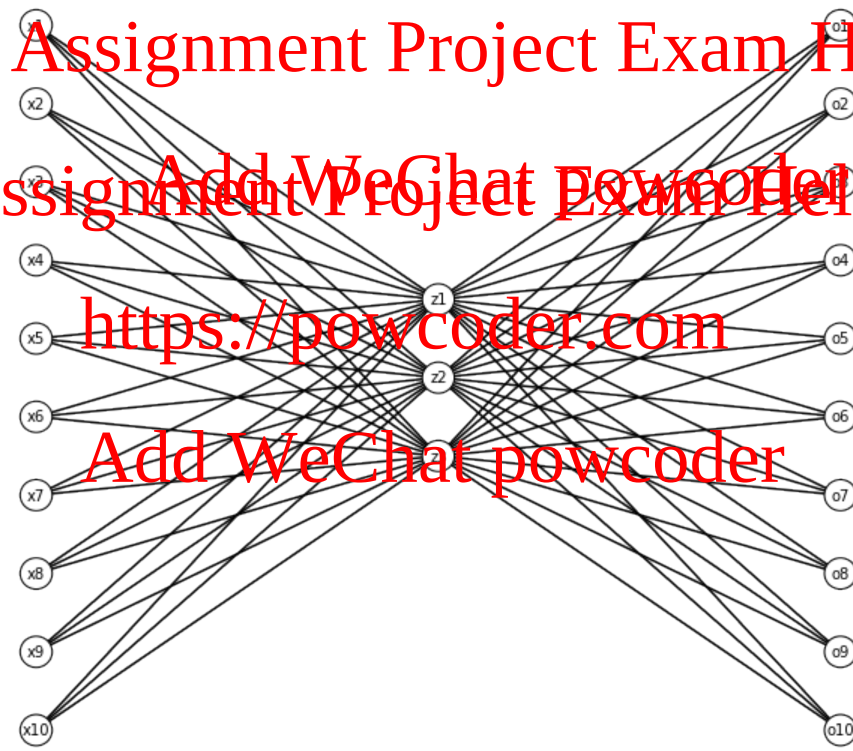
$$o_j = \boldsymbol{u}_{w_j}^\top \boldsymbol{v}_{w_i}$$

- Finally we use softmax to obtain a posterior distribution

$$p(w_j | w_i) = y_j = \frac{exp(o_j)}{\sum_{j'=1}^{V} exp(o_{j'})}$$

where y_j is the output of the j-th unit in the output layer

A simple CBOW model



Computing the hidden layer is just embedding lookup

Hidden layer computation "retrieves" $\boldsymbol{v}_{w_i}$:

$$\boldsymbol{v}_{w_i} = \boldsymbol{z} = \boldsymbol{\Theta x} =$$

$$\begin{bmatrix} 0.1 & 0.3 & 0.5 & 0.4 & 0.6 & 0.1 & 0.3 & 0.5 & 0.4 & 0.6 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & 0.2 & 0.5 & 0.1 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & 0.2 & 0.5 & 0.1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.8 \\ 0.8 \end{bmatrix}$$

Note there is no activation at the hidden layer (or there is a linear activation function), so this is a "degenerate neural network".

Computing the output layer

$$\boldsymbol{o} = \boldsymbol{\Theta}'\boldsymbol{z} = \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \\ 0.6 & 0.5 & 0.6 \\ 0.3 & 0.1 & 0.1 \\ 0.2 & 0.4 & 0.8 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.6 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} \times \begin{bmatrix} 0.5 \\ 0.8 \\ 0.8 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.91 \\ 0.97 \\ 0.82 \\ 1.18 \\ 0.31 \\ 1.06 \\ 0.39 \\ 0.95 \\ 0.63 \end{bmatrix}$$

Each row of $\boldsymbol{\Theta}'$ correspond to vector for a target word w_j.

Taking the softmax over the output

$$\boldsymbol{y} = softmax(\boldsymbol{o}) = \begin{bmatrix} 0.11039215 \\ 0.10606362 \\ 0.11262222 \\ 0.09693485 \\ 0.13893957 \\ 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix}$$

The output $\boldsymbol{y}$ is a probabilistic distribution over the entire vocabulary.

Input vector and output vector

Since there is no activation function at the hidden layer, the output is really just the dot product of the vector of the input context word and the vector of the output target word.

$$p(w_j|w_i) = y_j = \frac{exp(o_j)}{\sum_{j'=1}^{V} exp(o_{j'})} = \frac{exp(\boldsymbol{u}_{w_j}^\top \boldsymbol{v}_{w_i})}{\sum_{j'=1}^{V} exp(\boldsymbol{u}_{w_{j'}}^\top \boldsymbol{v}_{w_i})}$$

where $\boldsymbol{v}_{w_i}$ from $\boldsymbol{\Theta}$ is the **input vector** for word w_i and $\boldsymbol{u}_{w_j}$ from $\boldsymbol{\Theta'}$ is the **output vector** for word w_j

Computing the gradient on the hidden-output weights

- Use the familiar cross-entropy loss

$$\ell = -\sum_{j}^{|V|} t_j \log y_j = -\log y_{j\star}$$

  where $j\star$ is the index of the target word

- Given y_j is the output of a softmax function, the gradient on the output is:

$$\frac{\partial \ell}{\partial o_j} = y_j - t_j$$

$$\frac{\partial \ell}{\partial \theta'_{ji}} = \frac{\partial \ell}{\partial o_j} \frac{\partial o_j}{\partial \theta'_{ji}} = (y_j - t_j) z_i$$

- Update the hidden→output weights

$$\theta'_{ji} = \theta'_{ji} - \eta (y_j - t_j) z_i$$

Updating input→hidden weights

- Compute the error at the hidden layer

$$\frac{\partial \ell}{\partial z_i} = \sum_{j=1}^{V} \frac{\partial \ell}{\partial o_j} \frac{\partial o_j}{\partial z_i} = \sum_{j=1}^{V} (y_j - t_j)\theta'_{ij}$$

- Since

$$z_i = \sum_{k=1}^{V} \theta_{ik} x_k$$

The derivative of ℓ on the input→ hidden weights:

$$\frac{\partial \ell}{\partial \theta_{ik}} = \frac{\partial \ell}{\partial z_i} \frac{\partial z_i}{\partial \theta_{ik}} = \sum_{j=1}^{V} (y_j - t_j)\theta'_{ji} x_k$$

- Update the input→hidden weights

$$\theta_{ki} = \theta_{ki} - \eta \sum_{j=1}^{V} (y_j - t_j)\theta'_{ij} x_k$$

Gradient computation in matrix form

$$
\boldsymbol{D_o} = \boldsymbol{Y} - \boldsymbol{T} = \begin{bmatrix} 0.11039215 \\ 0.10606362 \\ 0.11262222 \\ 0.09693485 \\ 0.13893957 \\ 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.11039215 \\ 0.10606362 \\ -0.88737778 \\ 0.09693485 \\ 0.13893957 \\ 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix}
$$

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Computing the errors at the hidden layer

$$\boldsymbol{D_z} = \boldsymbol{D_o}^\top \boldsymbol{\Theta'} =$$

$$\begin{bmatrix} 0.110 & 0.106 & -0.887 & 0.097 & 0.139 & 0.058 & 0.123 & 0.063 & 0.110 & 0.080 \end{bmatrix}$$

$$\times \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \\ 0.6 & 0.5 & 0.6 \\ 0.3 & 0.1 & 0.1 \\ 0.2 & 0.4 & 0.8 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.6 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} = \begin{bmatrix} -0.115 & 0.157 & -0.194 \end{bmatrix}$$

Computing the updates to $\boldsymbol{\Theta}'$

$$\nabla_{\boldsymbol{\Theta}'} = \boldsymbol{D_o} \boldsymbol{z}^{\top} = \begin{bmatrix} 0.110 \\ 0.106 \\ -0.887 \\ 0.097 \\ 0.139 \\ 0.058 \\ 0.123 \\ 0.063 \\ 0.110 \\ 0.080 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.8 & 0.8 \end{bmatrix} = \begin{bmatrix} 0.055 & 0.088 & 0.088 \\ 0.053 & 0.085 & 0.085 \\ -0.443 & -0.710 & -0.710 \\ 0.048 & 0.078 & 0.0778 \\ 0.069 & 0.111 & 0.111 \\ 0.029 & 0.047 & 0.0467 \\ 0.062 & 0.099 & 0.099 \\ 0.032 & 0.050 & 0.050 \\ 0.055 & 0.088 & 0.088 \\ 0.040 & 0.064 & 0.064 \end{bmatrix}$$

Computing the update to $\mathbf{\Theta}$

$$\nabla_{\mathbf{\Theta}} = \boldsymbol{x}\boldsymbol{D} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} -0.11538456 & 0.15687943 & -0.19388611 \end{bmatrix}$$

$$= \begin{bmatrix} 0. & 0. & 0. \\ 0. & 0. & 0. \\ -0.11538456 & 0.15687943 & -0.19388611 \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \end{bmatrix}$$

CBOW for multiple context words

$$
\begin{aligned}
\boldsymbol{z} &= \frac{1}{M}\boldsymbol{\Theta}(\boldsymbol{x}_1 + \boldsymbol{x}_2 + \cdots \boldsymbol{x}_M) \\
&= \frac{1}{M}(\boldsymbol{v}_{w_1} + \boldsymbol{v}_{w_2} + \cdots + \boldsymbol{v}_{w_M})
\end{aligned}
$$

where M is the number of words in the context, $w_1, w_2, \cdots, w_M$ are the words in the context and $\boldsymbol{v}_w$ is an input vector. The loss function is

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

$$
\begin{aligned}
\ell &= -\log p(w_j | w_1, w_2, \cdots, w_M) \\
&= -o_{j^\star} + \log \sum_{j'=1}^{V} \exp(o_{j'}) \\
&= -\boldsymbol{u}_{w_j}^\top \boldsymbol{z} + \log \sum_{j'=1}^{V} \exp(\boldsymbol{u}_{w_{j'}}^\top \boldsymbol{z})
\end{aligned}
$$

Computing the hidden layer for multiple context words

$$\boldsymbol{z} = \boldsymbol{\Theta x} =$$

$$\begin{bmatrix} 0.1 & 0.3 & 0.5 & 0.4 & 0.6 & 0.1 & 0.3 & 0.5 & 0.4 & 0.6 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & 0.2 & 0.5 & 0.1 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & 0.2 & 0.5 & 0.1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 3.0 \\ 3.0 \end{bmatrix}$$

During backprop, update vectors for four words instead of just one.

Skip-gram: model

$$p(w_{c,j} = w_{O,c} | w_I) = y_{c,j} = \frac{exp(o_{c,j})}{\sum_{j'=1}^{V} exp(o_{j'})}$$

Assignment Project Exam Help

where $w_{c,j}$ is the j-th word on the c-th panel of the output layer, $w_{O,c}$ is the actual c-th word in the output context words; w_I is the only input word, $y_{c,j}$ is the output of the j-th unit on the c-th panel of the output layer; $o_{c,j}$ is the net input of the j-th unit on the c-th panel of the output layer.

https://powcoder.com
Add WeChat powcoder

$$o_{c,j} = o_j = \boldsymbol{u}_{w_j} \cdot \boldsymbol{z}, \; for \; c = 1, 2, \cdots, C$$

Skip-gram: loss function

$$
\begin{aligned}
\ell &= -\log p(w_{O,1}, w_{O,2}, \cdots, w_{O,C} | w_I) \\
&= -\log \prod_{c=1}^{C} \frac{exp(o_{c,j_c^\star})}{\sum_{j'=1}^{V} exp(o_{j'})} \\
&= -\sum_{c=1}^{C} o_{c,j_c^\star} + C \times \log \sum_{j'=1}^{V} exp(o_{j'})
\end{aligned}
$$

where $j_c^\star$ is the index of the actual c-th output context word.

Combine the loss of C context words with multiplication. Note: $o_{j'}$ is the same for all C panels

Skip-gram: updating the weights

- We take the derivative of ℓ with regard to the net input of every unit on every panel of the output layer, $o_{c,j}$, and obtain

$$e_{c,j} = \frac{\partial \ell}{\partial o_{c,j}} = y_{c,j} - t_{c,j}$$

  which is the prediction error of the unit.

- We define a V-dimensional vector $E = E_1, \cdots, E_V$ as the sum of the prediction errors of the context word: $E_j = \sum_{c=1}^{C} e_{c,j}$

$$\frac{\partial \ell}{\partial \theta'_{ji}} = \sum_{c=1}^{C} \frac{\partial \ell}{\partial o_{c,j}} \cdot \frac{\partial o_{c,j}}{\partial \theta'_{ji}} = E_j \cdot z_i$$

- Updating the hidden$\rightarrow$output weight matrix:

$$\theta'_{ji} = \theta'_{ji} - \eta \cdot E_j \cdot z_i$$

- No change in how the input$\rightarrow$hidden weights are updated.

Additional sources on the skip-gram model

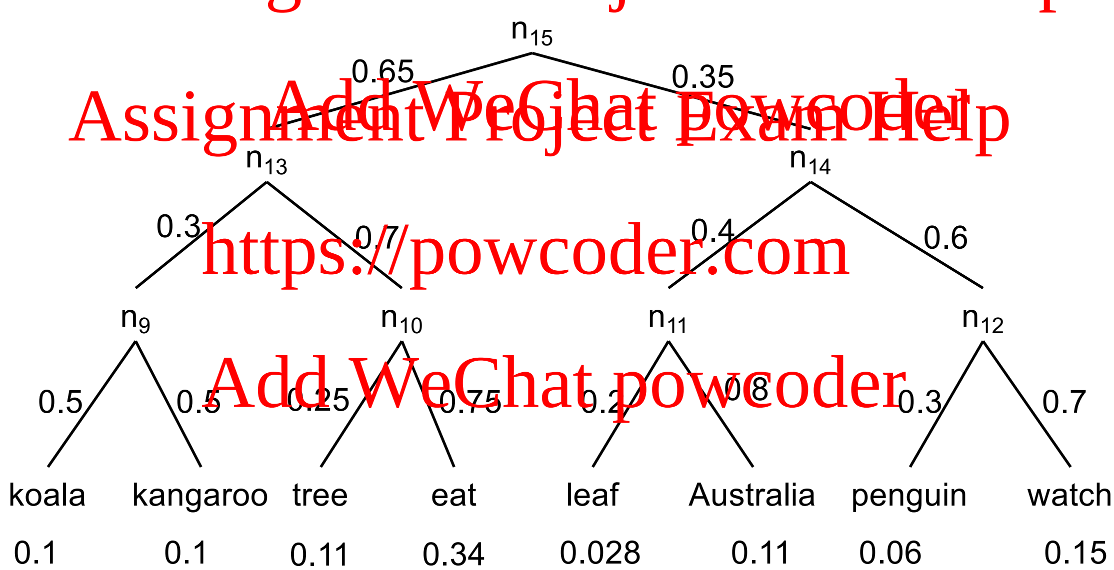
- For step-by-step derivation of the Skip-gram model, here is an excellent tutorial: https://aegis4048.github.io/demystifying_neural_network_in_skip_gram_language_modeling

Optimizing computational efficiency

- Computing softmax at the output layer is expensive. It involves iterative over the entire vocabulary.

- Two methods for optimizing computational efficiency
  - **Hierarchical softmax**: an alternative way to compute the probability of a word that reduces the computation complexity from $|V|$ to $\log |V|$.
  - **Negative sampling**: Instead of updating the weights for all the words in the vocabulary, only sample a small number of words that are not actual context words in the training corpus

Hierarchical softmax



Computing the probabilities of the leaf nodes

$$P(\text{“Kangaroo”}|z) = P_{n_1}(Left|\boldsymbol{z}) \times P_{n_3}(Left|\boldsymbol{z}) \times P_{n_6}(Right|\boldsymbol{z})$$

$$P_n(Right|\boldsymbol{z}) = 1 - P_n(Left|\boldsymbol{z})$$

$$P_n(Left|\boldsymbol{z}) = \sigma(\boldsymbol{\gamma}_n^\top \boldsymbol{z})$$

where $\boldsymbol{\gamma}_n$ is a vector from a set of new parameters that replace $\boldsymbol{\Theta}$

Huffman Tree Building

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

A simple algorithm:

- Prepare a collection of n initial Huffman trees, each of which is a single leaf node. Put the n trees onto a priority queue organized by weight (frequency).
- Remove the first two trees (the ones with lowest weight). Join these two trees to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two children trees. Put this new tree into the priority queue.
- Repeat steps 2-3 until all of the partial Huffman trees have been combined into one.

Negative sampling

- Computing softmax over the vocabulary is expensive. Another alternative is to approximate softmax by only updating a small sample of (context) words at a time.
- Given a pair of words (w, c), let $P(D = 1|w, c)$ be the probability of the pair of words came from the training corpus, and $P(D = 0|w, c)$ be the probability that the pair did not come from the corpus.
- This probability can be modeled as a sigmoid function:

$$P(D = 1|w, c) = \sigma(\boldsymbol{u}_w{}^\top \boldsymbol{v}_c) = \frac{1}{1 + e^{-\boldsymbol{u}_w{}^\top \boldsymbol{v}_c}}$$

New learning objective for negative sampling

- We need a new objective for negative sampling, which is to minimize the following loss function:

$$\mathcal{L} = -\sum_{w_j \in D} \log \sigma(o_{w_j}) - \sum_{w_j \in D'} \log \sigma(-o_{w_j})$$

  where D is a set of correct context - target word pairs and D' is a set of incorrect context - target word pairs.

- Note that we use the sum for positive samples as well as negative samples. In the skip-gram algorithm, there will be multiple positive context words in the output. In the CBOW algorithm, there will be only one positive target word.

- The derivative of the loss function with respect to the output word will be:

$$\frac{\partial \ell}{\partial o_{w_j}} = \sigma(o_{w_j}) - t_{w_j}$$

  where $t_{w_j} = 1$ if $w_j \in D$ and $t_{w_j} = 0$ if $w_j \in D'$

Updates to the hidden→output weights

- Compute the gradient on the output weights

$$\frac{\partial \ell}{\partial \boldsymbol{u}_{w_j}} = (\sigma(o_{w_j}) - t_{w_j})\boldsymbol{z}$$

- When updating the output weights, only the weight vectors for words in the positive sample and negative sample need to be updated:

Updates to the input→hidden weights

- Computing the derivative of the loss function with respect to the hidden layer

Assignment Project Exam Help

$$\frac{\partial \ell}{\partial \boldsymbol{z}} = \sum_{w_j \in D \cup D'} (\sigma(o_{w_j}) - t_{w_j}) \boldsymbol{u}_{w_j}$$

https://powcoder.com

- In the CBOW algorithm, the weights for all input context words will be updated. In the Skip-gram algorithm, the weight for the target word will be updated.

Add WeChat powcoder

$$\boldsymbol{v}_{w_i} = \boldsymbol{v}_{w_i} - \eta(\sigma(o_{w_j}) - t_{w_j}) \boldsymbol{u}_{w_j} x_i$$

How to pick the negative samples?

- If we just randomly pick a word from a corpus, the probability of any given word w_i getting picked is:

$$p(w_i) = \frac{freq(w_i)}{\sum_{j=0}^{V} freq(w_j)}$$

Assignment Project Exam Help

  More frequent words will be more likely to be picked and this may not be ideal.

https://powcoder.com

- Adjust the formula to give the less frequent words a bit more chance to get picked:

Add WeChat powcoder

$$p(w_i) = \frac{freq(w_i)^{\frac{3}{4}}}{\sum_{j=0}^{V} freq(w_j)^{\frac{3}{4}}}$$

- Generate a sequence of words using the adjusted probability, and randomly pick $n_{D'}$ words

Use of embeddings: word and short document similarity

- Word embeddings can be used to compute word similarity with cosine similarity

$$sim_{cos} = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|_2 \|\boldsymbol{v}\|_2}$$

- How accurately can they be used to compute word similarity can also be used to evaluate word embeddings
- They can also be used to compute the similarity of short documents

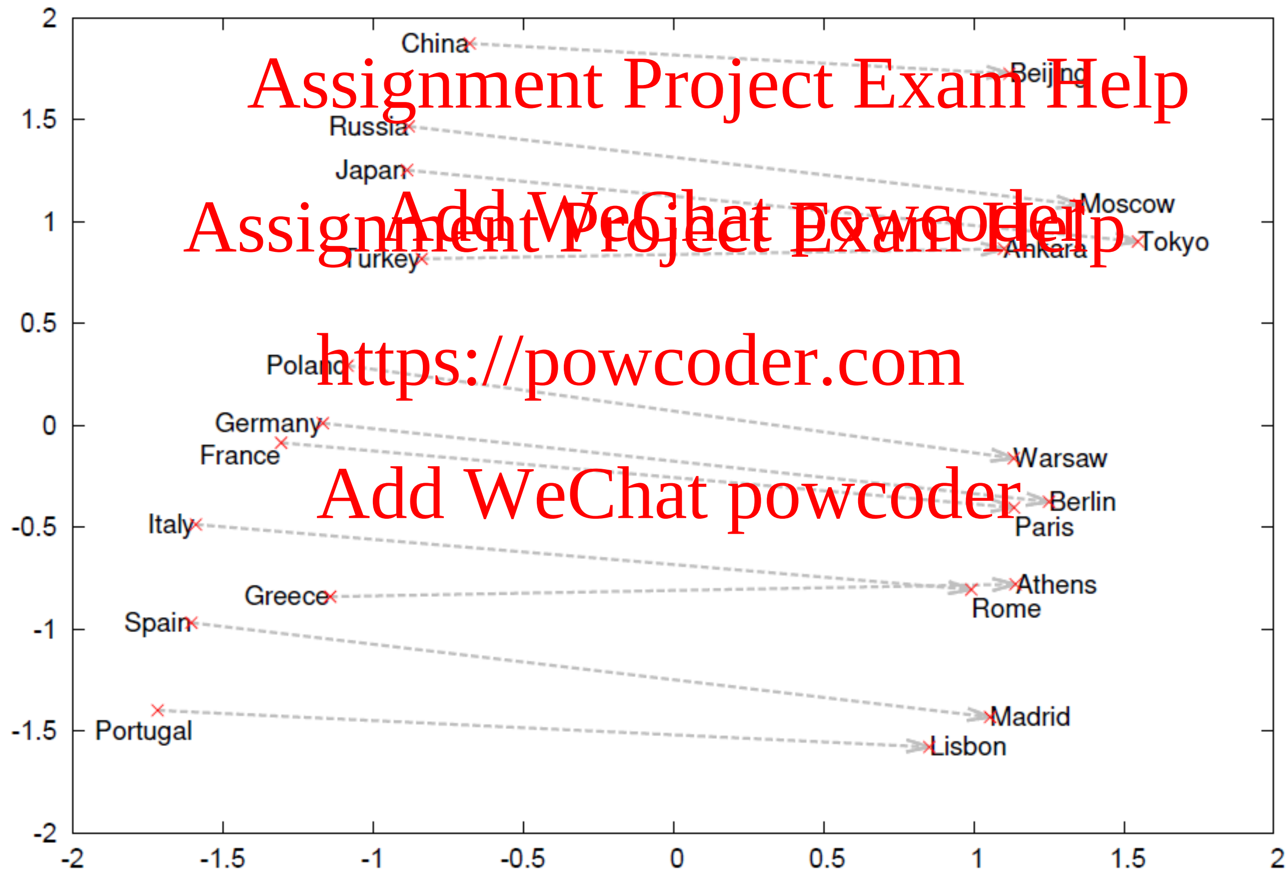
$$sim_{doc}(D_1, D_2) = \sum_{i=1}^{m} \sum_{j=1}^{n} cos(\boldsymbol{w_i^1}, \boldsymbol{w_j^2})$$

Use of embeddings: word analogy

- What's even more impressive is that they can be used to compute word analogy

$$
\begin{aligned}
analogy(m : w \rightarrow k :?) &= \underset{v \in V \setminus m,k,w}{\operatorname{argmax}} \; cos(\boldsymbol{v}, \boldsymbol{k} - \boldsymbol{m} + \boldsymbol{w}) \\
analogy(m : w \rightarrow k :?) &= \underset{v \in V \setminus m,k,w}{\operatorname{argmax}} \; cos(\boldsymbol{v}, \boldsymbol{k}) - cos(\boldsymbol{v}, \boldsymbol{m}) \\
&\quad + cos(\boldsymbol{v}, \boldsymbol{w}) \\
analogy(m : w \rightarrow k :?) &= \underset{v \in V \setminus m,k,w}{\operatorname{argmax}} \; \frac{cos(\boldsymbol{v}, \boldsymbol{k}) cos(\boldsymbol{v}, \boldsymbol{w})}{cos(\boldsymbol{v}, \boldsymbol{m}) + \epsilon}
\end{aligned}
$$

Word analogy



Use of word embeddings

- Computing word similarities is not a "real" problem in the eyes of many
- The most important use word embeddings is as input to predict the outcome of tasks that have real-world applications
- Many follow-on work in develop more effective word embeddings, e.g., GLOVE
  - word2vec: http://vectors.nlpl.eu/repository
  - fasttext: https://fasttext.cc/docs/en/english-vectors.html
  - GLOVE: https://nlp.stanford.edu/projects/glove

Shortcoming of “per-type” word embeddings

- “Per-type” word embeddings learned via models like word2vec do not account for the meanings of words in context:
  - “Work out the **solution** in your head.”
  - “Heat the **solution** to 75° Celsius.”
  - Having the same embedding for both instances of “solution” doesn’t make sense.
- The solution is Contextualized word embeddings, which are generated on the fly given the entire sentence as input. The same word will have different embeddings if they occur in different sentences.
  - ELMO: https://allennlp.org/elmo
  - BERT: https://github.com/google-research/bert
  - Roberta: https://pytorch.org/hub/pytorch_fairseq_roberta
- The contextualized word embeddings can be fine-tuned when used in a new classification task in a process called *transfer learning*.
- This turns out to be a very powerful idea that leads to many breakthroughs.

Embeddings in Pytorch

In [39]:
```python
from torch import nn
embedding = nn.Embedding(10, 3)
print(embedding)
input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
embedding(input)
```

```
Embedding(10, 3)
```

Out[39]:
```
tensor([[[-0.9538,  0.3385, -1.6404],
         [ 1.7206,  1.4395,  0.2744],
         [-2.9429,  0.9432, -0.4569],
         [-0.6187, -0.5479,  0.2746]],

        [[-2.9429,  0.9432, -0.4569],
         [ 1.2738,  1.1245,  0.6983],
         [ 1.7206,  1.4395,  0.2744],
         [ 0.6431, -1.2324, -1.3246]]], grad_fn=<EmbeddingBackward>)
```

In [38]:
```python
weight = torch.FloatTensor([[1, 2.3, 3], [4, 5.1, 6.3]])
embedding = nn.Embedding.from_pretrained(weight)
input = torch.LongTensor([0,1,1])
embedding(input)
```

Out[38]:
```
tensor([[1.0000, 2.3000, 3.0000],
        [4.0000, 5.1000, 6.3000],
        [4.0000, 5.1000, 6.3000]])
```