# word2vec & simple tasks with backprop

- the basis of a lot of methods has been to map words to word meaning vectors → like 25-d vectors or 300-d vectors or 1000-d vectors if we have a lot of data and are going for state of the art performance
- what are these vectors and what do the dimensions mean? its kind of upto the system and gradient descent to assign whatever meaning to each dimension that makes the representation better for the task at hand. but some vector dimensions have been shown to have meanings that mean something semantic to humans. also 2D and 3D clustering using PCA/t-SNE (which we'll see later) gives some intuitions to us humans who only work well in 2 or 3 D spaces
- another approach is to map parts of words to vectors so like un-natural is un + natural and then learn networks that take in these representation of part of words and do some task (like MT or suggesting responses to messages like google Inbox etc)
- machine translation actually was started in wars to help understand enemy's language!
- the curious thing is that a almost everything is getting turned into representations → parts of words, words, sentences, paragraphs, sounds, everything!
- a lot of NLP before deep learning used features and also used WordNet which is a taxonomy system that has a bunch of information about each word like what are the meanings in different senses and what are the synonyms for each sense.
- the problem with the WordNet approach is that context matters a lot. so given a word and given a sense, each synonym still means lots of different things → there is a lot of nuance that is hard to capture just by looking up WordNet
- so using these discrete representations is a bit like using a very very long vector where everything is 0 except for one slot corresponding to this word that has a 1 → one hot encoding
- the problem with one hot encoding might be that it gives us very little information. dot product between motel and hotel would be 0 → no notion of relationships between words
- the key idea in modern NLP and deep learning is that we can "know a word by the company it keeps" → the environments and the contexts it appears in. **distributional similarity. "**what other words usually occur in these contexts?"
- "if you can predict which textual context the word appears in, then you understand the word." → has turned out to be a successful approach.
- so lets learn vector representations that are good at predicting in what contexts words appear.
- so we want to predict context given word. a loss function we can use is $1 - p(context|w_t)$. Notice that if we can predict context perfectly, our loss will be 0. We'll keep adjusting the vector representation of words until this loss is minimized.
- This idea of learning representations by back propagating errors was floated around by Yoshua Bengio in 2003 and made popular quite recently by Tomas Mikolov and team at google
- Mikolov's is called **word2vec:**

## Predict between every word and its context words!

Two algorithms

1. **Skip-grams (SG)**
   Predict context words given target (position independent)
2. Continuous Bag of Words (CBOW)
   Predict target word from bag-of-words context

Two (moderately efficient) training methods

1. Hierarchical softmax
2. Negative sampling

**Naïve softmax**

# Details of word2vec

For each word $t = 1 \ldots T$, predict surrounding words in a window of "radius" $m$ of every word.

Objective function: Maximize the probability of any context word given the current center word:

$$J'(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} p(w_{t+j} | w_t ; \theta)$$

Negative Log Likelihood

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \neq 0}} \log p(w_{t+j} | w_t)$$

Where $\theta$ represents all variables we will optimize

- $m$ is the window size, that is a hyperparameter of the model

- we use log because it turns into sums instead of products and the math becomes easier. we put a negative sign so that we can call it a loss function and minimize instead of maximize.
- actually it turns out if each word has two vectors → a word vector and a context vector it works out better in practice.

Predict surrounding words in a window of radius $m$ of every word

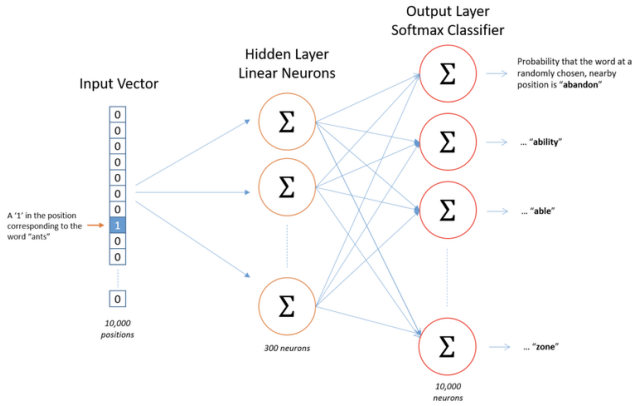
For $p(w_{t+j}|w_t)$ the simplest first formulation is

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{V} \exp(u_w^T v_c)}$$

where $o$ is the outside (or output) word index, $c$ is the center word index, $v_c$ and $u_o$ are "center" and "outside" vectors of indices $c$ and $o$

**Softmax** using word $c$ to obtain probability of word $o$

- the details of training word2vec → its a network with just hidden layer. our goal is to extract that matrix that transforms the one hot vector into the middle hidden layer of 300 neurons. this matrix is basically the matrix of word vectors.
- so we want the matrix that maps the input layer to the hidden layer




- the above is trained as usual with stochastic gradient descent as usual in neural nets → batch gradient descent with huge corpuses is not practical
- because computing the output layer involves a very large matrix multiplication to produce the output the size of the vocabulary, we can avoid this by sampling a few words to produce outputs for (the labels are 0 for all words except the 1 word that does appear in the k-radius.) → this optimization to get an estimate forms the skip gram model

- in continuous bag of words method, the task is to predict the center word given the sum of word vectors of the context words. that is, given the sum of word vectors of k-radius words.
- CBOW is kind of the opposite of skip gram task.
- While training word vectors using the skip gram model, note that the transformation from the first layer to the hidden layer basically "selects" the word vector of the input word from the matrix of word vectors. this is called the center word matrix. in the second step we go from this word vector to the output probabilities using the softmax activation function. for this second transformation, we actually use a different word vector matrix → turns out this makes the training more stable. so given these two word vector matrices how do we get the final word vector → the average word vector across these two matrices turns out to work well for downstream tasks.
- in softmax classification if you have a huge vocabulary size, the matrix becomes crazy big. in order to overcome this, there are two strategies:
  - hierarchical softmax: divide vocab into hierarchy tree and predict probability for each node in tree until you go down to the leaf
  - noise contrastive estimation (also called negative sampling): instead of training over full vocab, compute loss over correct target and over a sampled batch of incorrect targets. this makes the matrix multiplication more practical, rather than computing negative loss over all wrong answers.
- now, instead of looking at these co-occurrences batch by batch using windows, what if we compute global co-occurrence matrices like

## Window based co-occurrence matrix

- Example corpus:
  - I like deep learning.
  - I like NLP.
  - I enjoy flying.

| counts | I | like | enjoy | deep | learning | NLP | flying | . |
|---|---|---|---|---|---|---|---|---|
| I | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| like | 2 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| enjoy | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| deep | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| learning | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| NLP | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| flying | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| . | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |

- can we use the vector from this matrix as a word vector? its high dimensional (as big as the size of the vocabulary) and sparse → turns out not to work so well. for example, doesn't really capture associations/co-occurrences with synonyms if it doesn't exist in the text (→ doesn't do well in taking advantage from distributed representations power of deep learning?)

- so one option is to do SVD of these and reduce to a lower dimensional set of vectors from this matrix.
- can we combine word2vec that tends to perform well on downstream tasks but also take advantage of these global counts? yes, enter global word vectors or GloVe:

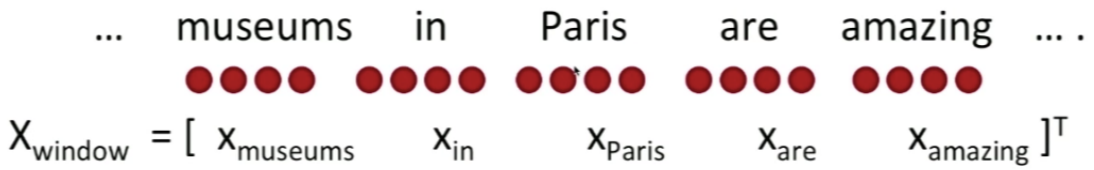
$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2$$

- the above is the loss function used for training. it tries to make the dot product of two vectors proportional to the log of their co-occurrence count using the loss function above (weighted by some function of count)
- This GloVe tends to work quite well on both intrinsic and extrinsic evaluations:
  - intrinsic is quick tests to check if word vectors make sense. like word vector analogy test (zuckerberg - facebook + msft = nadella) → the test is actually implemented by finding the vector with least cosine angle. so equality actually means nearest cosine angle vector, not strict equality.
  - extrinsic evaluations are things humans actually care about like maybe performance on a machine translation task or on named entity recognition task (NER)
- in word vector analogy task, syntactic similarity means like "dancing : danced :: decreasing : decreased". semantic is like “man : woman :: king : queen”
- **simple classification task**: now lets say you want to train these word vectors to perform simple classification tasks. we will classify each word as positive or negative based on some dataset that is given to us. we can frame this as a simple logistic regression problem or a input to output layer neural net and then optimize the weights that map word vector to positive/negative using log likelihood loss (or cross entropy loss in the binary case)
- now, we could allow the gradients in the above classification task to flow back to the word vectors. should we do this? if you have a lot of training data, you can retrain the word vectors for the task at hand, it can improve performance and generalization. if you don’t have a lot of data, such retraining might cause overfitting → you’ll do well on the training set but won’t do well on the test set.
- but, classifying single words can be ambiguous and hard

- Example: auto-antonyms:
  - "To sanction" can mean "to permit" or "to punish.”
  - "To seed" can mean "to place seeds" or "to remove seeds."

- Example: ambiguous named entities:
  - Paris → Paris, France vs Paris Hilton
  - Hathaway → Berkshire Hathaway vs Anne Hathaway

- so lets look at window classification. in this task, we will classify a phrase of center word + k neighbors on either side as positive or negative. we will represent the input as concatenation of word vectors. once we have this input, it is straightforward training with backprop as usual.




- we could do direct input → output softmax/logistic regression style. the downside of this is that this gives us a linear decision boundary in the word vector space → there is no non-linearities in this network once the word vectors are computed. we can introduce non-linearities by introducing more hidden layers. this allows us to learn more complex decision boundaries.
- training neural nets with examples that are "hard": we can use a maximum margin objective function like the one below to train a network on the above window classification task. the point of the maximum margin objective definition is that it is only trained on the "hard" examples. that is, examples where those from the opposite classes are incorrectly classified by the classifier. this makes it develop a robust margin that does well on borderline or "hard" examples → similar idea to maximizing the support vector in SVMs

- Objective for a single window:

$$J = \max(0, 1 - s + s_c)$$

- Each window with a location at its center should have a score +1 higher than any window without a location at its center

- note in the above objective function, as the model gets better and better at distinguishing examples, only the "hard" examples are taken into account for future training, ignoring the ones we clearly get right.
- the stochasticity introduced by SGD (vs whole batch gradient descent) helps us escape local minima → richard hypothesizes this. also "local minima aren't really a problem, they are quite close to what you might think is global minima"
- during training, we should store the activations during forward propagation. this way, while backpropagating, the chain rule formula for gradients involves activations that we would have already computed and can just use without recomputing. for deep networks and for RNNs and stuff, this needs to be traded off with how many activations can be kept in memory.
- for equations of backprop see "how the backprop algorithm works" notes in the michael nielsen book notes.
- one last takeaway: even though neural net based word embeddings are often considered superior nowadays, a research snippet presented in the class showed a paper that showed that if you choose the

right hyperparameters, SVD based embeddings can be made to perform equally well on downstream high level NLP tasks.

- another new idea is to learn character level embeddings instead of word level embeddings. while this seems unintuitive at first (what meaning do characters carry anyway), in languages like German where it is common to pull together a ton of phrases to make one word, turns out that even though single character embeddings may not strictly mean a lot, the composition (stacking) of character level embeddings helps compose representations that do a good job of representing the symantics  of a long compound word composed of many subphrases, like “unfortunately” (but some languages have it a lot longer)