# CS224n

Thomas Lu

Note: statements that I am unsure about are enclosed in double square brackets, e.g. [[Every even integer greater than 4 can be expressed as the sum of two primes.]]

## 1 Lecture 1: Introduction

This lecture basically just gives a high level overview of the fields of Natural Language Processing (NLP) and Deep Learning (DL). Some highlights from the lecture:

- NLP is a study that aims for computers to understand natural (human) language well enough to perform useful tasks.
- NLP has a lot of useful applications (virtual assistant, customer support automation, etc.) and has taken off commercially in the past few years.
- What is special about NLP (vs. other subfields of ML)?
  - Language is specifically constructed to convey information. It's not an environmental signal (e.g. an Amazon purchase, which I probably made for a reason other than conveying information).
  - Language is (mostly) a discrete/symbolic/categorical system, but our brains, which process language, are continuous systems.
- What is special about DL (vs. other ML techniques)?
  - Traditionally, learning an ML model requires an engineer/researcher to design and build relatively human-interpretable features. The machine then basically does some numerical optimization on these features to produce something useful. Note that this still requires a lot of intuitive human work to interpret data and figure out what parts of it might be useful.
  - In contrast, deep learning just feeds "raw data" or very simplistic representations of data to a deep neural net, which then learns its own effective representations of the data, obviating the need for feature engineering.
  - DL has performed very well recently, e.g. speech recognition, ImageNet.
- Why is NLP hard?
  - Human language contains lots of contextual information. The meaning of a sentence may depend on something that was said a few sentences ago, a piece of information from another document, current events, or something observable in the environment ("Did you see that dog?" refers presumably to a dog in the speaker's field of vision).
  - Human language is also ambiguous by design; since speech is pretty slow at transmitting information, people will almost always leave "relatively obvious" things unsaid in order to improve communication efficiency. Listeners are expected to infer these things. This is very different from computer languages, which must specify every posibility.

## 2 Lecture 2: word2vec

One part of NLP is word meaning representation. How do we represent the definition of a word so that a computer can do something useful with that representation?

One method is WordNet, which is a database of words that groups words with similar meaning together into synsets, which are also arranged in a kind of hierarchy. But this has some problems:

- Synonym/hypernym relationships miss a lot of nuance (e.g. "expert", "proficient", "good" don't really mean the same thing).
- It's hard to keep the database up-to-date when new words are added or existing words gain/lose usages and meanings.
- Creating the database is subjective and labor-intensive.
- Given two words, there isn't a good way to get a measure of how similar they are.

This is an example of a discrete/categorical/symbolic representation of words, where each word represents a single concept. Words might be related, but separate words are generally regarded as separate entities. If these words were vectors, each one would be a $V$-dimensional vector of zeroes with a single 1, where $V$ is the vocabulary size. This is sometimes referred to as a "localist" or "one-hot" representation.

A problem with these representations is that they lack similarity measures; we could try to manually encode some (as with WordNet), but they don't exist in the representations themselves (e.g. any two distinct word vectors are orthogonal). We can instead try to encode words so that the include similarity information, where a dot product of two words can give you a measure of their similarity. This gives us representations of words as shorter vectors, and this is known as a "distributed" representation: instead of the meaning of a word being "localized" to a single component of a vector, it's "distributed" across multiple components.

One idea for generating distributed representations is the idea of distributional similarity, where we figure out the meaning of a word based on the contexts it appears in, i.e. its distribution in text. (The word "distributional" in "distributional similarity" is not related to the word "distributed" in "distributed representation".) This idea is summed up in a well-known quote: "You shall know a word by the company it keeps." (John Rupert Firth)

word2vec is a well-known distributed representation model. Its key idea is to predict **outer words** that will appear in the context of a **center word**. More specifically, given a center word $w_c$, it attempts to predict the probabilities $p(w_o|w_c)$ that various outer words $w_o$ will appear in the context of $w_c$. Supposing we have a training corpus of $T$ words, the training process will attempt to maximize

$$\prod_{t=1}^{T} \prod_{\substack{-m\leq j\leq m,\\ j\neq 0}} p(w_{t+j}|w_t;\theta),$$

where $\theta$ is the parametrization of the model, the context of a word is defined as the $m$ words before and after it, $w_i$ is the $i$-th word in the corpus, and $p(w_i|w_j;\theta)$ denotes the model's predicted probability of $w_i$ appearing in the context of $w_j$ given its parametrization $\theta$. Equivalently, the training process attempts to minimize the objective function

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

More concretely, the model is parametrized by $2|V|$ word vectors, where $V$ is the vocabulary. Each word $w \in V$ has two vector representations: a center word representation $v_w$ and an outer word representation

$u_w$. The model then predicts

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

More intuitively, this means that the model predicts that $o$ is likely to appear in the context of $c$ if the outer word representation of $o$ is similar to the center word representation of $c$.

Because the training corpus for a word2vec model is typically very large, the model is usually trained using stochastic gradient descent, regarding each context window as a single training example. Now the gradient of $J$ with respect to $\theta$ is a little tricky to notate, but since $J$ is just a sum of terms of the form $\log p(w_o|w_c)$, the partial derivatives of that term will be quite informative. We have:

$$\begin{aligned}
\frac{\partial}{\partial u_o} \log p(o|c) &= \frac{\partial}{\partial u_o} \log \frac{\exp\left(u_o^T v_c\right)}{\sum_{w \in V} \exp\left(u_w^T v_c\right)} \\
&= \frac{\partial}{\partial u_o} \left[ \log \exp\left(u_o^T v_c\right) - \log \left( \sum_{w \in V} \exp\left(u_w^T v_c\right) \right) \right] \\
&= v_c - \frac{1}{\sum_{w \in V} \exp\left(u_w^T v_c\right)} \frac{\partial}{\partial u_o} \sum_{w \in V} \exp\left(u_w^T v_c\right) \\
&= v_c - \frac{1}{\sum_{w \in V} \exp\left(u_w^T v_c\right)} \frac{\partial}{\partial u_o} \exp\left(u_o^T v_c\right) \\
&= v_c - \frac{\exp\left(u_o^T v_c\right)}{\sum_{w \in V} \exp\left(u_w^T v_c\right)} v_c \\
&= v_c \left(1 - p(o|c)\right) \ .
\end{aligned}$$

Similarly, we can calculate

$$\begin{aligned}
\frac{\partial}{\partial v_c} \log p(o|c) &= \frac{\partial}{\partial v_c} \log \frac{\exp\left(u_o^T v_c\right)}{\sum_{w \in V} \exp\left(u_w^T v_c\right)} \\
&= \frac{\partial}{\partial v_c} \left[ \log \exp\left(u_o^T v_c\right) - \log \left( \sum_{w \in V} \exp\left(u_w^T v_c\right) \right) \right] \\
&= u_o - \frac{1}{\sum_{w \in V} \exp\left(u_w^T v_c\right)} \frac{\partial}{\partial v_c} \sum_{w \in V} \exp\left(u_w^T v_c\right) \\
&= u_o - \sum_{w \in V} \frac{\exp\left(u_w^T v_c\right)}{\sum_{w' \in V} \exp\left(u_{w'}^T v_c\right)} u_w \\
&= u_o - \sum_{w \in V} p(w|c) u_w \\
&= u_o \left(1 - p(o|c)\right) - \sum_{\substack{w \in V \\ w \neq o}} p(w|c) u_w \ .
\end{aligned}$$

Intuitively, this means that when we encounter a word $o$ in the context of another word $c$, we should:

- Move the outer word representation $u_o$ of $o$ towards the center word representation $v_c$ of $c$ by an amount negatively correlated with the prior prediction $p(o|c)$. This means that we should make a larger update for more unexpected events.
- Move the center word representation $v_c$ of $c$:
  - towards the outer word representation $u_o$ of $o$ by an amount negatively correlated with the prior prediction $p(o|c)$, meaning that we should make a larger update for more unexpected events; and

  – away from the outer word representations $u_w$ for each $w_o$ by an amount positively correlated with the prior prediction $p(w|c)$, meaning that we should make a larger update for more unexpected events (since observing $o$ in the context of $c$ means that we did not observe $w$ in the location of $o$).

This matches our intuition that we expect to see $o$ in the context of $c$ if $u_o$ is similar to $v_c$.

# 3 Lecture 3: GloVe

## 3.1 word2vec revisited

Recall that the skip-gram (word2vec) model is typically trained using stochastic gradient descent. Because the parametrization of the model is very large ($2Vd$ terms, where $V$ is the vocabulary size and $d$ is the dimensionality of the vector representations of the words) and the calculated gradient at each iteration is comparatively sparse, we should avoid computing and passing around complete gradient vectors when performing SGD.

Previously, we used the following probability in the skip-gram model:

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

The numerator of this is easy to compute, but the denominator is pretty expensive, as the sum has to be taken across the entire vocabulary. Instead of computing the full sum, the actual word2vec model uses a slightly different objective function that allows it to randomly sample from the vocabulary instead of iterating over the whole vocabulary:

$$J(\theta) = \sum_{t=1}^{T} J_t(\theta) = \sum_{t=1}^{T} \left[ \log \sigma\left(u_o^T v_c\right) + \sum_{\substack{j \sim P(V) \\ k \text{ samples}}} \log \sigma\left(-u_j^T v_c\right) \right],$$

where $T$ represents our corpus, $\sigma$ is the sigmoid function, $\sigma(z) = (1 + e^{-z})^{-1}$, $P(V)$ is a probability distribution over the vocabulary $V$, $\sim$ indicates random sampling, and $k$ is a constant, usually around 10. Furthermore, instead of minimizing this objective function, the model actually seeks to maximize it. Intuitively, this means that the model attempts to maximize $u_o^T v_c$ when $o$ occurs in the context of $c$ and minimize $u_w^T v_c$ for $w$ that do not appear in the context of $c$, which is pretty similar to what we were doing with the previous objective function.

One detail on the objective function: the probability distribution $P(V)$ that is usually used is $P(w) = U(w)^{3/4}/Z$, where $U(w)$ is the unigram frequency of $w \in V$ and $Z$ is a normalizing factor to make $\sum_{w \in V} P(w) = 1$. The exponent of 3/4 "smoothes out" the distribution a bit, so that more frequent items are less likely to be selected than less frequent items in proportion to their frequencies.

An alternative to the skip-gram model is the similar Continuous Bag-Of-Words (CBOW) model. While the skip-gram model attempts to predict the probability of an outside word appearing given a center word, the CBOW model attempts to predict the probability of the center word occuring given the sum of the (vector representation of the) outside words.

## 3.2 Co-occurrence matrices

Co-occurrence matrices are another way to construct vector representations of words. To start, we can create a $|V| \times |V|$ matrix $X$ and iterate through our corpus, incrementing $X_i j$ every time we see $i, j \in V$

occurring together. We can define "occurring together", or co-occurrence, in various ways. Some common ones are:

- Window-based co-occurrence. This is similar to what the skip-gram model does: we say that two words co-occur if one appears in a window around the other (e.g. the window from 8 words before to 8 words after).
- [[Full-document co-occurrence. Here we say that two words co-occur if they both appear in some document in our corpus.]]
- [[Word-document co-occurrence. In this definition, we need to change our matrix $X$. We will still have a row for each word, but instead of having a column for each word as well, we have a column for each document in our corpus, and we say that a word and a document co-occur if the word appears in that document.]]

It turns out that window-based co-occurrence allows us to represent both syntactic and semantic information in the vector representations, while [[full-document co-occurrence and word-document co-occurrence allow us to capture topics and perform Latent Semantic Analysis.]] Unless otherwise specified, further discussion in this section will be about window-based co-occurrence matrices.

These "raw" co-occurrence matrices, however, are extremely storage intensive; a 100,000 word vocabulary would yield a matrix with 10 billion entries when using window-based co-occurrence. These matrices are also rather sparse, which negatively impacts the robustness of models trained on them. We can solve both of these problems by reducing the column space using singular value decomposition (SVD). Other improvements that we can make to this model include:

- To prevent stopwords from creating very spiky counts, we can cap frequency counts when assembling the pre-SVD matrix or ignore stopwords altogether.
- We can weight co-occurrences; e.g. maybe we should weight the co-occurrence of two words more if they are adjacent rather than 5 words apart (e.g. by adding a larger or smaller increment to the corresponding entry in the pre-SVD co-occurrence matrix).

However, co-occurrence matries still have some issues:

- Adding words to the vocabulary is difficult - we have to grow both the row space and the column space.
- SVD is an expensive operation; on an $m \times n$ matrix with $n < m$, it has runtime $O(mn^2)$.
- The model only captures word similarity and doesn't really capture any additional patterns.
- It's hard to make the model not give excessive significance to large counts.

On the other hand, co-occurrence matrices have some advantages as well:

- It's relatively fast to train.
- It uses the statistics of the corpus efficiently - after a single training pass, we can perform whatever further transformation/analysis we want on the co-occurrence counts.

As for the skip-gram model:

- The representations it produces are often better suited for downstream tasks.
- It captures patterns beyond just word similarity - it also gets things like syntactical relationships.
- The training time scales with the corpus size. It can take a very long time to train on a large corpus.
- It doesn't really let us use corpus statistics well, as there isn't really any intermediate representation of the data besides the trained model itself.

## 3.3 GloVe

Global Vectors, or GloVe, is another method for training vector representations of words. It compiles a co-occurrence matrix $P$, and then seeks to minimize the objective function

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

$f$ here is a function that is increasing until a cutoff point, after which it becomes constant. The parameters of the model here are the word vectors $u_i$ and $v_j$, similar to the skip-gram model.

Intuitively, we weight common co-occurrences more heavily in this objective function, but we have a cap on how heavily any one can be weighted. We also want $u_i^T v_j$ to be larger, i.e. $u_i$ and $v_j$ to be more similar, when $P_{ij}$ is larger (i.e. when $i$ and $j$ co-occur more frequently). At the end when we've trained our $u_i$ and $v_j$, the final representation $x_w$ that we choose for each word $w \in V$ is simply $x_w = u_w + v_w$.

This method scales to large corpora but has been shown to also work well for small corpora. It's like a best-of-both-worlds between word2vec and co-occurrence matrices: it's fast to train and uses statistics efficiently, but also captures information beyond word similarity and caps the influence of large co-occurrence counts. Furthermore, it avoids the computational cost of performing SVD on a large matrix.

## 3.4 Evaluation of vector representations of words

Once we have a model, we'd like to evaluate how well it performs. In general, there are two categories of evaluation metrics for any model: intrinsic and extrinsic.

- Intrinsic metrics measure how well the model performs some specific or intermediate subtask, such as how well vector similarities of words map to human judgments of similarities of words. They:
  - are typically easy to compute and measure and
  - often can give you more insight into your own system by showing you which hyperparameters affect your system, but
  - oftentimes don't clearly show whether the model is actually good at some more externally useful task.
- Extrinsic metrics measure how well the model performs some more useful downstream task, such as machine translation. Extrinsic metrics:
  - are often slow to compute (e.g. it might take a while to train the machine translation model from your word representations) and
  - can be unclear on whether your component (e.g. word representations) of the larger system (e.g. machine translation system) specifically improved performance on its own if other parts of the system have also changed.

One intrinsic metric is word vector analogy accuracy. For example, if we give the analogy task

`Paris:France::Rome:??`

to the model, the model should be able to determine that the correct answer is Italy..Typically word representation models do this by finding the word $w$ that maximizes

$$\frac{(x_b - x_a + x_c)^T x_w}{||x_b - x_a + x_c||||x_w||},$$

where the analogy is given as `a:b:::c:???`. We can then measure how well the model works by feeding it many such analogies and examining how many it gets correct.

## 4 Lecture 4: Word Window Classification and Neural Networks

Previously we have discussed unsupervised models for learning word representations, but eventually we want to use these representations to train supervised predictive models.

In a typical classification problem, we'll have a training dataset $(x_i, y_i), 1 \leq i \leq N$, where the $x_i$ are our inputs and the $y_i$ are labels that we want to train our model to predict. A legacy technique for training these models is logistic regression, a.k.a. softmax. In this method, we train a weight matrix $W$ with one row per class, and then attempt to adjust it so that our predictions

$$p(y|x) = \frac{\exp(W_y x)}{\sum_c \exp(W_c x)}$$

match the training examples as closely as possible. Here $W_c$ represents the row of weights in $W$ corresponding to the specific class $c$. A common loss function for softmax is the cross-entropy loss:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} -\log \frac{\exp(W_{y_i} x_i)}{\sum_c \exp(W_c x_i)},$$

where $\theta$ represents all the parameters of the model (in this case $W$). Oftentimes in practice we also add a regularization term $\lambda\theta^T\theta$, which helps prevent overfitting. Note that this method treats the inputs $X = [x_1^T, x_2^T, ..., X_N^T]^T$ as constant. In constrast, when we use deep learning models, we usually update our word vectors (i.e. $X$) as well. However, because this vastly increases our parameter space, we need to be careful not to overfit. Often if we have a small training set, we shouldn't update word vectors, as overfitting would be a big problem.

### 4.1 Tips for vector calculus

There will be a lot of vector calculus involved in calculating gradients for neural network (NN) models. Some tips for when it gets tricky:

- Carefully define all variables and keep track of their dimensionality. Checking dimensionality is a great sanity check for whether you've made any obvious mistakes.
- When using the chain rule, keep track of variable dependencies to make sure you haven't forgotten a necessary step of differentiation.
- Casework can sometimes make a calculation more intuitive (e.g. gradients for the correct class vs. the incorrect classes).
- It can sometimes also be easier to calculate derivatives element-by-element instead of trying to differentiate the whole vector function at once.
- After doing casework and elementwise differentiation, try to vectorize your notation again. It's important to vectorize your implementations in code, so you'll need to vectorize the notation at some point.

### 4.2 Window classification

A common prediction task is to predict whether a center word in a window of words is a named person/-place/organization or not a named entity. Typically the input to the model is the concatenation of the word vectors in the window (so if we had $d$-dimensional word vectors and used a window with $m$ words before and after the center word, our input would be a vector of dimensionality $(2m + 1)d$).

We could approach this problem with the softmax technique, but softmax is only capable of learning linear decision boundaries. NNs, on the other hand, can learn more complex nonlinear boundaries.

## 4.3 Neural networks

A neural network is made up of many neurons, arranged in sequential layers. We call the first layer the input layer, since it just contains the input values, and the final layer the output layer. The intermediate layers between them are commonly referred to as hidden layers. Each neuron has a weight vector $w$, an input vector $x$, a bias unit $b$, and an activation function $f$, and produces output $f(wx+b)$. A common activation function is the sigmoid function $f(z) = (1+e^{-z})^{-1}$.

The reason that a neural network can learn more complex decision boundaries is that it can feed the outputs of neurons into another layer of neurons: instead of being constrained to immediately predict the output, it can learn an intermediate representation that might be more helpful for predicting the output. Note however that this requires a nonlinear activation function like the sigmoid; if we had a linear activation function, like the identity function, our neural network would still just be a linear classifier at the end of the day.

Back to our example problem - imagine that we're using a simple one-hidden-layer NN to predict whether the center word in a window is a named entity location or not. We have an input layer $x_j$ which together with a weight matrix $W$ produces a hidden layer $a_i$, which finally produces a score $s$. For simplicity, we will use a simple model where $s$ is a linear score: $s = U^T a$ for some weight vector $U$. We will use the max-margin loss function:

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

where $s$ is predicted score of a positive example (where the center word is a named location) and $s_c$ is the predicted score of a negative example (where the center word is not a named location). In practice, for a single positive example, we usually randomly sample some $s_c$ rather than computing the max over the whole training corpus. Intuitively, this loss function tries to find a decision boundary that separates positive and negative samples by a "sufficiently large" margin ("sufficiently large" is defined here as 1, but it can be tuned as a hyperparameter).

Note that it is very common to write $a = f(z)$, so that $a = f(Wx+b)$ and $z = Wx+b$. (Here $f$ is evaluated element-wise.)

We now compute the gradient of $J$ in parts. When $J = 0$ we have $\vec{\nabla} J = 0$, so assume $J > 0$. Then $\vec{\nabla} J = -\vec{\nabla} s + \vec{\nabla} s_c$. We will just compute $\vec{\nabla} s$ for illustration. For the partial derivatives with respect to $U$, we have:

$$\frac{\partial s}{\partial U} = \frac{\partial}{\partial U} U^T a = a.$$

For the partial derivatives with respect to $W$, we have:

$$\frac{\partial s}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} U^T a = \frac{\partial}{\partial W_{ij}} U^T f(z) = \frac{\partial}{\partial W_{ij}} U^T f(Wx+b).$$

Note that only the $i$-th entry of $f(Wx+b)$ is nonconstant with respect to $W_{ij}$, so we have

$$\begin{aligned}\frac{\partial}{\partial W_{ij}} U^T f(Wx+b) &= \frac{\partial}{\partial W_{ij}} U_i f(W_i x + b_i) \\ &= U_i f'(z_i) x_j \\ &= \delta_i x_j,\end{aligned}$$

where we have written $\delta_i = U_i f'(z_i)$. Now note that by combining all of these entries, we can write:

$$\frac{\partial s}{\partial W} = \delta x^T.$$

To calculate the partials with respect to $b$, we have:

$$\begin{aligned}
\frac{\partial s}{\partial b_i} &= \frac{\partial}{\partial b_i} U^T f(Wx+b) = \frac{\partial}{\partial b_i} U_i f(W_i x + b_i) \\
&= U_i f'(W_i x + b_i) \\
&= \delta_i \\
\Rightarrow \frac{\partial s}{\partial b} &= \delta.
\end{aligned}$$

Finally it remains to find the partials with respect to $x$. We have:

$$\begin{aligned}
\frac{\partial s}{\partial x_j} &= \frac{\partial}{\partial x_j} U^T a \\
&= \sum_i \frac{\partial}{\partial x_j} U_i a_i \\
&= \sum_i U_i \frac{\partial}{\partial x_j} f(W_i x + b_i) \\
&= \sum_i U_i f'(W_i x + b_i) \frac{\partial}{\partial x_j} W_i x \\
&= \sum_i \delta_i W_{ij} \\
\Rightarrow \frac{\partial s}{\partial x} &= W^T \delta.
\end{aligned}$$

## 5 Lecture 5: Backpropagation

Let's now imagine a neural network with two hidden layers, still using the same output layer and cost function as before. We now have:

$$s = U^T a^{(3)}$$

$$\begin{aligned}
a^{(3)} &= f(z^{(3)}) & z^{(3)} &= W(2) a^{(2)} + b^{(2)} \\
a^{(2)} &= f(z^{(2)}) & z^{(2)} &= W^{(1)} x + b^{(1)}
\end{aligned}$$

We can easily derive $s$ with respect to $U$ and $W^{(2)}$ just as we did before to get

$$\begin{aligned}
\frac{\partial s}{\partial U} &= a^{(3)} \\
\frac{\partial s}{\partial W^{(2)}} &= \delta^{(3)} a^{(3)T},
\end{aligned}$$

where $\delta^{(3)} = U \circ f'(z^{(3)})$, with $\circ$ denoting the element-wise product of two vectors. We can calculate $\partial s / \partial b^{(2)} = \delta^{(3)}$ similarly as well.

It now remains to calculate the partials of $s$ with respect to $W^{(1)}$, $b^{(1)}$, and $x$. We will demonstrate the computation for $W^{(1)}$:

$$\begin{aligned}
\frac{\partial s}{\partial W_{ij}^{(1)}} &= \frac{\partial}{\partial W_{ij}^{(1)}} U^T f(W^{(2)}a^{(2)} + b^{(2)}) \\
&= \sum_k \frac{\partial}{\partial W_{ij}^{(1)}} \left[ U_k f(W_k^{(2)} a^{(2)} + b_k^{(2)}) \right] \\
&= \sum_k U_k f'(W_k^{(2)} a^{(2)} + b_k^{(2)}) \frac{\partial}{\partial W_{ij}^{(1)}} W_k^{(2)} a^{(2)} \\
&= \sum_k \delta_k^{(3)} \frac{\partial}{\partial W_{ij}^{(1)}} W_k^{(2)} f(W^{(1)}x + b^{(1)}) \\
&= \sum_k \delta_k^{(3)} \frac{\partial}{\partial W_{ij}^{(1)}} W_{ki}^{(2)} f(W_i^{(1)}x + b_i^{(1)}) \\
&= \sum_k \delta_k^{(3)} W_{ki}^{(2)} f'(W_i^{(1)}x + b_i^{(1)}) \frac{\partial}{\partial W_{ij}^{(1)}} \left( W_i^{(1)}x + b_i^{(1)} \right) \\
&= \sum_k \delta_k^{(3)} W_{ki}^{(2)} f'(z_i^{(2)}) x_j \\
&= x_j f'(z_i^{(2)}) \left( W_{.i}^{(2)T} \delta^{(3)} \right) \\
\Rightarrow \frac{\partial s}{\partial W^{(1)}} &= \left( f'(z^{(2)}) \left( \circ W^{(2)T} \delta^{(3)} \right) \right) x^T = \delta^{(2)} x^T,
\end{aligned}$$

where

$$\delta^{(2)} = \left( W^{(2)T} \delta^{(3)} \right) \circ f'(z^{(2)}).$$

It's easy to see from the previous derivation that $\partial s / \partial b^{(1)} = \delta^{(2)}$, and the partial with respect to $x$ can be found in a similar way.

In general, we have

$$\begin{aligned}
\delta^{(\ell)} &= \left( W^{(\ell)T} \delta^{(\ell+1)} \right) \circ f'(z^{(\ell)}) \\
\frac{\partial E_R}{\partial W^{(\ell)}} &= \delta^{(\ell+1)} a^{(\ell)T} + \lambda W^{(\ell)},
\end{aligned}$$

for any intermediate layer $\ell$, any activation function $f$, and regularization with constant $\lambda$.

## 6 Lecture 6: Dependency Parsing

Two main models of linguistic structure are used these days.

- One is the constituency model, also commonly referred to as phrase structure grammar model or the context free grammar model. This attempts to organize words into nested constituents (noun phrase, verb phrase, prepositional phrase, etc.) and define how these constituents can be constructed (e.g. a noun phrase is a determinant, followed by an arbitrary number of optional adjectives, followed by a noun, folllowed by an arbitrary number of optional prepositional phrases).
- The other is the dependency model, which the rest of this section will focus on. In this model, we typically just care about which words depend on (modify or are arguments of) which other words. Oftentimes dependency structures are depicted with dependency diagrams, in which dependencies between words in a sentence are indicated with arrows pointing from words to the words dependent on them.

Sometimes, the meaning of a sentence can be ambiguous, e.g. "Scientists study whales from space." Ambiguities such as these often come from ambiguous dependency structures (does "from space" depend on "whales" or "study"?).

To train sentence-parsing models using dependency structures, there are "treebanks," which are datasets of dependency-annotated sentences. While intuitively it may seem advantageous for the field to spend time designing grammars instead of manually annotating sentences (grammars generalize, annotating sentences is very slow, etc.), it turns out that human-created grammars differ a lot and that the work involved in designing grammars isn't very reusable, while treebanks are highly reusable and provide a useful ground truth for future work.

In a dependency syntax, we represent sentences as directed graphs of words, where words are nodes and edges point from "head" nodes to "dependent" nodes (dependent nodes depend on head nodes). Edges are often also annotated with the type of dependence (e.g. auxiliary, case, subject, etc.). Normally, this graph is a tree: it has a single root, is acyclic, and is connected. Usually there will also be a pseudoword ROOT in the sentence so that every other word in the sentence depends on one word.

When evaluating the plausibility of a dependence from one word to another in a sentence, there are various information sources that we can draw from:

- Bilexical affinities (is it reasonable, based on the definitions of the word, for word A to depend on word B?)
- Distance (words are generally more likely to depend on nearby words)
- Intervening material (e.g. an adjective and the noun it's modifying are unlikely to have a verb between them)
- Valency of heads (certain words are likely to have certain numbers of dependents on the left and/or right side; e.g. determinants are unlikely to have any dependents; nouns might have many dependents but determiners and adjectives typically are on the left, while prepositional phrases are likely to be on the right).

To parse a sentence, we must decide for each word which other word it depends on. Usually we constrain the total space of possibilities by enforcing that only one word depends on ROOT, and that dependencies cannot form cycles, which guarantees that our dependency graph will be a tree. We also say that a dependency graph is projective if, when we draw it with the words in order and all edges as "straight" arcs (i.e. never change their left-right direction) going over the row of words, we can draw it in such a way that the arcs don't cross.

There are various methods for doing dependency parsing:

- Dynamic programming (runs in $O(n^3)$)
- Graph algorithms (minimum spanning tree)
- Constraint Satisfaction: eliminate edges that don't satisfy hard constraints
- "Transition-based parsing." Slightly less accurate than graph algorithms, but is a greedy algorithm that runs in linear time. This is what's most commonly used today.

## 6.1 Arc-standard transition-based parser

The arc-standard transition-based parser is one form of transition-based parsing (there are others). In this algorithm, we have a buffer $\beta$ that initially consists of all the words of the sentence in order from left to right, and a stack $\sigma$ that initially contains one element: ROOT. At each step, we have three operations that we can perform:

1. Shift. This pops the leftmost word from $\beta$ and pushes it onto $\sigma$.
2. Left Arc. This sets the second-from-the-top element of the stack as dependent on the top element of the stack and removes the second-from-the-top element.
3. Right Arc. The same as left arc, but the top element is the dependent instead of the head.

As an example, we perform this algorithm on the simple sentence "I ate fish." In this example, the right side of the stack is the top.

1. At the beginning, we have $\sigma = [\text{ROOT}], \beta = [\text{I}, \text{ate}, \text{fish}]$.
2. We perform two shift operations, and now we have $\sigma = [\text{ROOT}, \text{I}, \text{ate}], \beta = [\text{fish}]$.
3. We perform a left-arc, so now 'I' depends on 'ate' and is removed from $\sigma$, so we have $\sigma = [\text{ROOT}, \text{ate}], \beta = [\text{fish}]$.
4. We now perform another shift, giving $\sigma = [\text{ROOT}, \text{ate}, \text{fish}]$, and $\beta$ is now empty.
5. We follow this up with a right-arc, so 'fish' depends on 'ate', and now $\sigma = [\text{ROOT}, \text{ate}]$.
6. Finally, we perform another right-arc, and $\sigma = [\text{ROOT}]$, 'ate' depends on ROOT, and 'I' and 'fish' depend on 'ate'.

To train a model to actually do this, we can use the a treebank - given the dependency structure, we can figure out which sequence of shifts and left/right-arcs will give us the correct structure. This allows us to train a classifier on which operation is correct at each step. In the model above, we have 3 possible operations at each step, but in practice we also want to label each dependency in a dependency structure with a type (i.e. not just that 'fish' depends on 'ate', but that it is specifically the object of 'ate'). To accommodate this, we have a different left-arc and right-arc operation for each dependency type, giving $2R + 1$ total distinct operations, where $R$ is the number of types of dependencies. These dependencies are often scored using two metrics: UAS, which counts the percentage of dependency arrows that are correct, and LAS, which counts the percentage of depdency arrows that are both correct *and* correctly labeled.

These classifiers were commonly built with sparse feature representations at first, where we use binary vectors where an element is 1 if some specific position on either the stack or the buffer contains some specific word in some specific part of speech (e.g. 'set' can be multiple parts of speech depending on context). However, you often end up with $O(10^7)$ features, which causes issues where there are too many possible configurations of features, some of which you will almost certainly never see in your training data. The vast feature space also makes for rather expensive computation.

So instead, we can learn a dense feature representation. We use $d$-dimensional distributed representations of words, and also smaller-dimensioned distributed representations of parts of speech. We can then extract features by looking at certain stack/buffer positions and concatenating the word/POS vectors. We can then learn on these features with a neural network; it turns out a single hidden layer with a ReLU activation function and an output layer with a softmax activation function works pretty well.

# 7 Lecture 7: Intro to TensorFlow

TensorFlow (TF) is a deep learning framework from Google. There are various reasons why these frameworks are useful:

- Provides simple ways to make ML code to run at massive scales
- Compute gradients automatically
- Standardizes ML so that work can be shared more easily

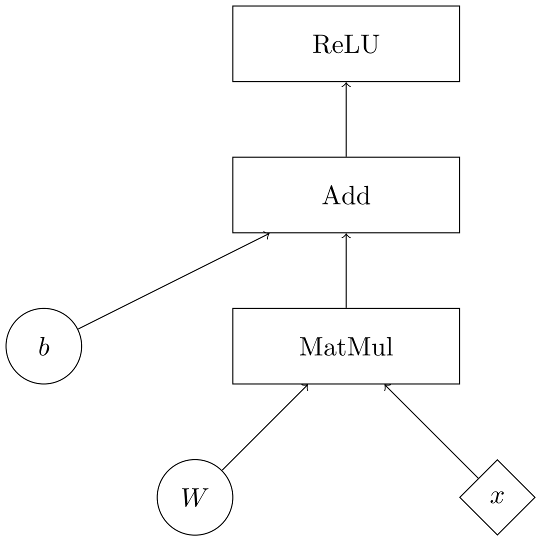

Figure 1: The expression $ReLU(Wx + b)$ as a TF computation graph.

- Allows easy interfacing with GPUs

The core idea of TF is that numerical computation is a graph, where nodes are operations, and they take inputs and transmit outputs via edges. In addition to "normal" operations such as addition or matrix multiplication, there are also two important special types of operations:

- Variables, which output their current value. The current value is stored as state of the node. Gradient descent updates will also by default update all variables in the graph. Usually these will be the parameters of your model.
- Placeholders, which are nodes whose values are fed in at execution time. Usually these will be the training inputs/labels of your model.

As an example, the equation $h = ReLU(Wx + b)$ (which could represent a one-layer neural network with a ReLU activation function) would be expressed as graph in Figure 1. In code, this might be (assuming that we have 784-dimensional training examples in batches of 2000, and that our output is 100-dimensional):

```
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((100, 784), -1, 1))

x = tf.placeholder(tf.float32, (784, 2000))

h = tf.nn.relu(tf.matmul(x, W) + b)
```

Now to actually run this computation, we need to deploy it in a TF session, which binds to a particular execution context (e.g. CPU, GPU, TPU). When running a session, we need to provide two arguments:

- Fetches. This is the list of graph nodes whose values we care about. These will be returned by the `run` function.
- Feeds. This maps graph nodes to specified concrete values, and e.g. allows us to specify the values of placeholders.

If we extend our code example from earlier to run the graph, it might look like this:

```
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((100, 784), -1, 1))

x = tf.placeholder(tf.float32, (784, 2000))

h = tf.nn.relu(tf.matmul(x, W) + b)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
sess.run(h, {x: np.random.random(784, 2000)})
```

In the last line, `h` is our fetch and `x` is our feed. Here we're just initializing our training data randomly; in an actual application we'd probably load it in from somewhere. The `Session` constructor could optionally also take a computation environment as a function parameter. (There's probably other function parameters as well.)

We now need to define a loss function. To do this, we create a placeholder for the training labels and create a loss node using those and our prediction values:

```
prediction = tf.nn.softmax(h)
label = tf.placeholder(tf.float32, [100, 2000])
# cross entropy per example
cross_entropy = -tf.reduce_sum(label * tf.log(prediction), axis=1)
```

Then to actually train the model, we need to perform gradient descent. We can use an Optimizer object, which allows us to add an optimization operation to our computation graph. For example, to perform gradient descent, we can do:

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

Here 0.5 is a learning rate, and `train_step` now refers to a optimization operation node. When we run `train_step` in a session, it does two things: it calculates the gradients with respect to the variables in the graph, and it also adjusts the values of the variables according to those gradients. TF is able to calculate these gradients because each graph node has an attached gradient operation, and the graph structure of the computation allows TF to easily apply the chain rule.

To tie this all together, our final code might look like this:

```
import tensorflow as tf
from . import data  # some data loading code you defined

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((100, 784), -1, 1))

x = tf.placeholder(tf.float32, (784, 2000))
h = tf.nn.relu(tf.matmul(x, W) + b)

prediction = tf.nn.softmax(h)
label = tf.placeholder(tf.float32, [100, 2000])
# cross entropy per example
cross_entropy = -tf.reduce_sum(label * tf.log(prediction), axis=1)

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

for i in range(1000):  # train on 1000 batches
    batch_x, batch_label = data.next_batch()
    sess.run(
        train_step,
        feed_dict={
            x: batch_x,
            label: batch_label
        }
    )
```

### 7.1 Variable sharing

Sometimes we want to use multiple machines to train our models, and we need to share variables between them. TF has a concept called variable sharing that allows us to do this. In TF, we can enter a variable scope using `with tf.variable_scope('foo')`, and within that scope, we can call `tf.get_variable('v')`.

We can use `get_variable` to create new variables or access the values of existing ones. Its exact behavior depends on the value of the `reuse` keyword argument passed to the enclosing `variable_scope` (details can be found on the official TF documentation).

## 8 Lecture 8: Recurrent Neural Networks and Language Models

A language model is a model that predicts the probability of a sequence of words: $p(w_1, w_2, \ldots, w_T)$. Language models have applications in machine translation (e.g. $p(\text{the cat is small}) > p(\text{small is the cat})$) and speed recognition (e.g. $p(\text{walking home after school}) > p(\text{walking hone after school})$), among other things.

### 8.1 Traditional Language Models

Traditional language models are typically set up so that the probability of a word is conditioned on a window of $n$ previous words. For reasons of computational complexity, $n$ is usually small; as $n$ increases, performance improves significantly, but the storage and training time requirements grow extremely quickly. Thus these models are pretty limited, since words in a language often depend on other words that occurred quite long ago.

In a traditional language model, if $n = 2$, we might estimate probabilities based on the last word and the last two words like so:

$$p(w_2|w_1) = \frac{\text{count}(w_1 w_2)}{\text{count}(w_1)}$$

$$p(w_3|w_1, w_2) = \frac{\text{count}(w_1 w_2 w_3)}{\text{count}(w_1 w_2)}$$

### 8.2 Recurrent Neural Networks

A recurrent neural network (RNN) is basically a neural network that produces an output at each time step $t$, and where the hidden layer at time $t-1$ feeds into the hidden layer at time $t$. Mathematically, suppose that we have a vocabulary $V$ and a corpus/document represented as a list of word vectors $x_{[1]}, x_{[2]}, \ldots, x_{[t-1]}, x_{[t]}, x_{[t+1]}, \ldots, x_{[T]}$. Then at each time step $t$, we perform:

- $h_t = \sigma(W^{(hh)} h_{t-1} + W^{(hx)} x_{[t]})$
- $\hat{y}_t = \text{softmax}(W^{(S)} h_t)$

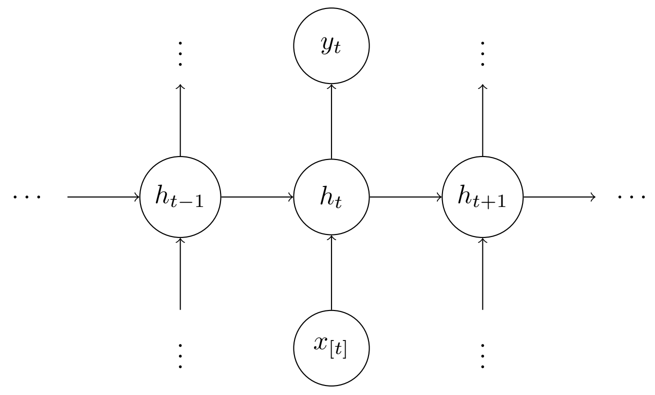

Figure 2: A common visualization of RNNs.

- Predict $\hat{P}(x_{[t+1]} = v_j \in V | x_{[t]}, \ldots, x_{[1]}) = \hat{y}_{t,j}$.

Here $W^{(hh)} \in \mathbb{R}^{D_h \times D_h}, W^{(hx)} \in \mathbb{R}^{D_h \times d}, W^{(S)} \in \mathbb{R}^{|V| \times D_h}$ are three separate weight matrices (shared across all time steps) that are used in different parts of the model. Sharing these weights between time steps effectively allows the model to "remember" what happened more than a few words ago, so that we can build a more effective language model.

As a minor detail of implementation, we typically initialize $h_0$ to a vector of all zeros so that there is some defined starting value.

With this RNN, we can define a loss function. Since $\hat{y}_t$ gives us a probability distribution over $V$, we can calculate the cross entropy loss at each time step:

$$J^{(t)} = \sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

After calculating these, we typically use perplexity as a loss function (lower is better):

$$E = \text{perplexity} = 2^{\sum_t J^{(t)}}.$$

However, training these RNNs is very hard, as inputs from many time steps ago can influence the current prediction/gradient. The next section, while focused on a different issue of RNNs, will illustrate the complexity in computing the gradients of RNNs.

## 8.3 The vanishing gradient problem

An issue with RNNs is that the gradients of $E$ with respect to the weight matrices tend to vanish or explode after a certain number of time steps. To see why this is, consider a simpler RNN with:

$$h_t = Wf(h_{t-1}) + W^{(hx)} x_{[t]}$$

$$\hat{y}_t = W^{(S)} f(h_t),$$

where $f$ is some nonlinear activation function. Let $E_t$ be the loss at time step $t$ (i.e. the loss for the prediction of $y_t$). Then the total error $E$ is the sum of all the $E_t$, so

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W}.$$

Then we have

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}.$$

(Note that this is already getting very complex.) We then have

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}.$$

It should be noted that $\partial h_j / \partial h_{j-1}$ is a Jacobian, i.e.

$$\frac{\partial h_j}{\partial h_j - 1} = \begin{bmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,1}}{\partial h_{j-1,n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{j,n}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,n}}{\partial h_{j-1,n}} \end{bmatrix}$$

Let

$$\|A\| = \sum \sum a_{ij}^2$$

be the Frobenius norm of a matrix $A$. It is well known that $\|AB\| \leq \|A\|\|B\|$. Then letting $\beta_W$ and $\beta_h$ be upper bounds on $\|W\|$ and $\max_t \|\mathrm{diag}(f'(h_t))\|$, respectively, we have

$$\begin{aligned} \left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| &= \left\| \frac{\partial}{\partial h_j - 1} \left( W f(h_{j-1}) \right) \right\| \\ &= \|W \mathrm{diag}\left( f'(h_{j-1}) \right)\| \\ &\leq \|W\| \, \|\mathrm{diag}\left( f'(h_{j-1}) \right)\| \\ &\leq \beta_W \beta_h. \end{aligned}$$

This implies

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right| \leq (\beta_W \beta_h)^{t-k}.$$

This last expression is exponential, so it will either explode or vanish rapidly.

The issue with vanishing gradients is that it makes it difficult for the model to use words from the relatively distant past to make predictions about a word at time $t$, as any contribution to the gradient of $E$ with respect to $W$ will be vanishingly small (so the model can't update on that information). An example of a simple prediction where an RNN suffering from vanishing gradients would have trouble on is:

- Jane walked into the room. John walked in too. It was late in the day. Jane said hi to ______. (It's very easy to see that the next word should probably be "John".)

The exploding gradient problem is relatively easy to solve - if the gradient $g = \partial E / \partial W$ satisfies $\|g\| > \tau$ for some threshold $\tau$, then we simply do

$$g := \frac{\tau}{\|g\|} g,$$

thus effectively capping $\|g\|$ at $\tau$. This basically prevents us from getting too far away from our previous model if we encounter a point in our parameter space where the gradient is very large. However, an analogous solution for vanishing gradients would not work, since it would add too much random noise. But One simple solution for vanishing gradients is to initialize the $W$ matrices to the identity and to use the ReLU activation function (won't go into why this works but it does).

As presented, RNNs are incredibly computationally intensive to train. However, there are a few optimizations that we can do in practice:

- Instead of computing the softmax over the entire vocabulary, we can first compute a softmax over categories of words, and then perform a second softmax over the words within that category. This is similar to hierarchical softmax.
- To update each $E_t$, we only need to do one backwards propagation at the end, instead of performing a backward pass on the neural net at every single time step.

Common extensions of RNNs include bidirectional RNNs and deep RNNs. The equations for a bidirectional RNN are slightly different - instead of one hidden representation layer $h$ that only depends on past words, we have two hidden representations - $\overrightarrow{h}$ that depends on past words and $\overleftarrow{h}$ that depends on future words. Then the equations look like:

$$\overrightarrow{h}_t = f(\overrightarrow{W}x_t + \overrightarrow{V}h_{t-1} + \overrightarrow{b})$$
$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}h_{t+1} + \overleftarrow{b})$$
$$y_t = g(U[\overrightarrow{h}_t; \overleftarrow{h}_t] + c)$$

Deep RNNs are similar, except that they might have multiple hidden layers $h^{(1)}, h^{(2)}, etc.$ with every hidden layer feeding into the next at each timestamp. (Deep RNNs can also be bidirectional.)