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

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

where θ 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 θ. Equivalently, the training process attempts to minimize the objective function

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

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 θ 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, σ 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 θ 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 ℓ, any activation function f, and regularization with constant λ.

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 β that initially consists of all the words of the sentence in order from left to right, and a stack σ 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 β and pushes it onto σ.
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 σ, 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 β 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

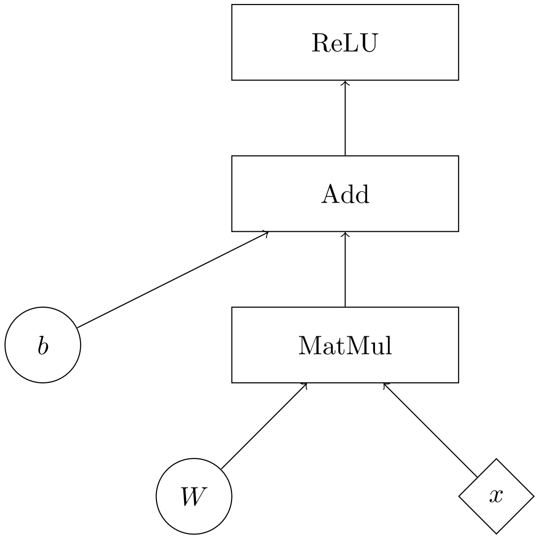

Figure 1: The expression $ReLU(Wx + b)$ as a TF computation graph.

- 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:

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

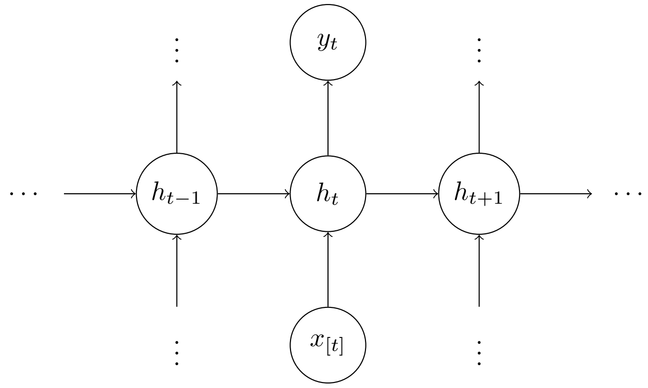

Figure 2: A common visualization of RNNs.

- 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 β_W and β_h be upper bounds on $\|W\|$ and $\max_t \|\text{diag}(f'(h_t))\|$, respectively, we have

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

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 of those words 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 τ, then we simply do

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

thus effectively capping $\|g\|$ at τ. 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:

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

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.)

9 Lecture 9: Machine translation, GRUs, and LSTMs

Machine translation (MT) is the problem of taking text in a source language (e.g. French) and translating it to a target language (e.g. English). MT models are mostly statistical models and are trained on parallel corpora, which are basically "copies" of "the same text" in multiple languages.

In a traditional MT model, you might have a sentence in a source language f. Given this, the model will attempt to find the most likely translation

$$\hat{e} = \text{argmax}_e p(e|f) = \text{argmax}_e p(f|e)p(e),$$

by Bayes' Theorem (we ignore the $p(f)$ factor because f is given). Here e represents a sentence in the target language. The MT model is then made up of two components:

- A translation model that trains $p(f|e)$ on a parallel corpus
- A language model that trains $p(e)$ on a single-language corpus.

The whole model might then take a source sentence f, calculate $p(f'|e)$ using the translation model for some phrases in e that might match up with phrases f' in the source sentence, calculate $p(e)$ for those phrases using the language model, and then combine and reorder everything in a decoder (a separate model).

The problem that the first part of such a model solves (finding translations for single words or short phrases in f) is called alignment. This problem is already hard for multiple reasons, which we will illustrate with examples in translating from English (source language) to French (target language):

- We might have *spurious words* which appear in the target sentence but have no counterpart in the source sentence: "Japan shaken by two new quakes" → "**Le** Japon secoué par deux nouveaux séismes"
- As in "**And** the program has been **implemented**" → "Le programme a été **mis en application**", we may have *zero fertility words* (such as "And"), which are present in the source sentence but have no counterpart in the target sentence, and *one-to-many alignments* ("implemented" → "mis en application"), where a single word in the source sentence maps to multiple words in the target sentence.

- We can also have *many-to-one alignments*, e.g. in “The balance was the territory of the aboriginal people” → “Le reste appartenait aux autochtones”, we have “was the territory” → “appartenait”; “of the” → “aux”; and “aboriginal people” → “autochtones”.
- Lastly, we can have *many-to-many alignments*; in “The poor don’t have any money” → “Les pauvres sont démunis”, “don’t have any money” becomes “sont démunis”.

We see that it’s pretty hard just to figure out which words are “standalones” and which words are parts of larger phrases. After doing this, we need to search over all the different sentence possibilities that arise as a result of the alignment step’s results (of which there can be many, given that there might be multiple candidate phrase structures and that phrases could be reordered in the target language).

Overall, traditional MT usually involves lots of manual feature engineering and independently training many different models.

9.1 RNNs?

We could try to train an RNN-based MT model:

- We start with an encoder. Supposing we have n input words, we need n time steps in the encoder RNN, and at each time step t, we have a hidden vector h and an input vector x, specified by

  $$h_t = f\left(W^{(hh)}h_{t-1} + W^{(hx)}x_t\right) = \phi(h_{t-1}, x_t),$$

  where ϕ just denotes an activation function applied to some linear combination of its inputs.
- We then have a decoder, which starts with h_n and outputs predicted words y_j using the relationships $(t > n)$

  $$\begin{aligned} h_t &= f(W^{(hh)}h_{t-1}) = \phi(h_{t-1}) \\ y_t &= \mathrm{softmax}(W^{(s)}h_t). \end{aligned}$$

  This decoder will output words until it decides to output a special “STOP” word.
- We have the model minimize the loss function

  $$-\frac{1}{N}\sum_{n=1}^{N} \log p_\theta(y^{(n)}|x^{(n)}).$$

However, this isn’t really powerful enough for MT; a major reason is that this model requires h_n to hold all the information needed to produce the output sentence. There are a few extensions we can make:

- We can train two different $W^{(hh)}$ matrices - one for encoding and one for decoding.
- We can feed more inputs into the computation of each decoding hidden unit. In the above model, h_t depends only on h_{t-1}, but we can make it depend on h_n (the last hidden unit from the encoding stage) and the previous predicted word y_{t-1} (converted to a single word, rather than the predicted probability distribution) as well.
- We can stack multiple RNNs to create a deep network.
- We can train a bidirectional encoder.
- We can input the source sentence in reverse order; this puts the initial words of the source sentence “close to” the initial output words in the architecture.

However, the key improvement is to create a more complex hidden unit.

9.2 Gated recurrent units (GRUs)

A classical RNN calculates a simple hidden unit at each time step t:

$$h_t = f\left(W^{(hh)}h_{t-1} + W^{(hx)}s_t\right).$$

A GRU is a little more complex. At each time step t, it calculates:

- An update gate:
$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$
- A reset gate, similar to the update gate but with a different set of weights:
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$
- A new memory:
$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right),$$
where $\circ$ denotes an elementwise product, and
- The next hidden layer:
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t.$$

Intuitively, the update gate allows the model to "forget" values from the previous timestamp if it decides that they are irrelevant to the future (if z_t is close to 0), or "focus" on the historical information while giving less importance to the new information. Similarly, the reset gate allows the model to "forget" old information in its memory unit if it deems it to be irrelevant for the future. The update gate can also help the model mitigate the vanishing gradient problem: if the model thinks a certain piece of information is important for the future, it can keep the values in z_t very close to 1, which will propagate gradient values close to 1.

9.3 LSTMs

Long short-term memory (LSTM) units are a generalization of GRUs. They have a few more definitions at each time step:

- An input gate:
$$i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$$
Intuitively, this tells the model whether to continue remembering the new memory.
- A forget gate:
$$f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$$
Intuitively, this tells the model whether to continue remembering the previous memory.
- An output gate:
$$o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$$
Intuitively, this tells the model whether to expose the final memory at this time step to the part of the unit actually calculating the ouptut. (Note that, regardless of the value of this gate, the final memory formed at each time step will still be used to compute the final memory formed at the next time step.)

- A new memory cell:
$$\tilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$$

- A final memory cell:
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

- A hidden state:
$$h_t = o_t \circ \tanh(c_t)$$

Both GRUs and LSTMs offer far more power than vanilla RNNs, and LSTMs are state-of-the-art at machine translation (at the time the class was taught).

10 Lecture 10: More machine translation, Attention

Translation is a huge industry - USD 40B/year. Getting good automated translation can be hugely impactful, even if that impact is less clear to English speakers.

Neural Machine Translation (NMT) models are models that perform machine translation end-to-end with a single large neural network. Typically these models have an encoder that takes input text and transforms it into some encoding Y and a decoder that generates output text from Y.

- The encoding part of an NMT model will be a recurrent neural network that reads in one source word at a time and passes them through some recurrent activation unit (tanh, GRU, LSTM, convolutional unit). The last hidden state produced by this RNN will typically be taken as the encoding Y.
- The decoder part of the NMT can be thought of as a conditional recurrent language model: it's basically a language model that's conditioned on Y to nudge it towards producing a specific output.

In 2014, NMT was this crazy idea that someone thought up but nobody was realy taking seriously, but by 2016 it was working much better than all the previous methods. There's a number of reasons why it works well:

- End-to-end training that allows the entire model to be optimized at once on a single loss function is really powerful. This is in contrast to having a bunch of separate models for reordering, transliteration, etc.
- They make good use od distributed word representations, which are very powerful. Previous methods had largely used one-hot representations, and the sparseness of the data is very limiting to the model.
- Neural models use context much more effectively than traditional phrase-based models.
- Deep neural nets are just better at generating fluent text, even if the "translation quality" isn't improved much.

However, there are still some shortcomings of NMT:

- NMT models don't make good explicity use of syntactic and semantic structures.
- NMT is still bad with higher-level properties of text, like discourse structure, anaphora, and clause linking.

10.1 Attention

One issue with decoding an encoding Y into a target language is that you might end up forgetting a bunch of the information in Y as you pass through time steps. A simple way to counter this is to pass Y directly into your hidden unit at each time step. However, there's a larger, higher-level issue here: Y itself might have already forgotten a lot of the information from the earlier parts of the source sentence. In practice this is ok for short sentences, but for longer sentences (¿30 words), NMT translations get significantly worse.

Attention is a concept that combats this difficulty. The key idea is to maintain a pool of the hidden units at all time steps during the encoding step, and let the decoder draw from this pool of source states as needed. This means that the decoder needs some way to know which elements from the pool are important, and some way to act on this knowledge. This is usually done at each decoding time step by scoring each source state against the most recent hidden state, creating a context vector by summing the source states weighted by their scores (passed through a softmax to turn them into a probability distribution), and passing this context vector as an input in creating the next hidden state. There are multiple ways to do this scoring between a source state h_s and a target hidden state h_t:

- A simple dot product: $h_s^T h_t$
- A bilinear form with a trained weight matrix W_a: $h_s^T W_a h_t$
- A single-layer neural network with trained weights v_a and W_a and activation function f: $v_a^T f\left(W_a[h_s; h_t]\right)$, where $[h_s; h_t]$ denotes concatenation.
- More complex neural networks.

In practice, the bilinear form has been found to work quite well, and one advantage it has over the single-layer neural net is that it allows for direct interaction between h_t and h_s, whereas the neural network is just a sum of independent weighted sums of transformations of h_t and h_s by an activation function.

There are also other ideas that have been laid on top of attention as well:

- The idea of coverage is an idea to make sure that we don't neglect any information from the source. In an attention model, you end up with an attention weight α_{st} for each source word s and target word t. Scoring and applying a softmax will ensure that $\sum_s \alpha_{st} = 1$ for each t, but coverage biases the model towards making sure that $\sum_t \alpha_{st}$ is close to 1 for each s as well. This helps to prevent the attention model from always failing to pay attention to a particular word or piece of information from the source. It does this by adding a term to the loss function equal to

$$\lambda \sum_s \left(1 - \sum_t \alpha_{st}\right)^2 .$$

- The attention model can also be augmented to be aware of position - words in the beginning or end of a source sentence are more likely to matter for the beginning or end of the target sentence, respectively.
- Another thing to consider is fertility - if a single source state is getting too much attention, then maybe the resulting translation isn't that accurate.

10.2 Decoding

Once you've produced an encoding Y, you need to decode this into an actual sequence of words in the target language. A RNN (or GRU, or LSTM, or what have you) will produce some probability distribution over the set of possible target words at each time step, but ultimately we need to select a single sentence that we think is the best translation. There are multiple ways to do this:

- The most naive way is to enumerate all possible sequences of words up to some length and select the single one with the highest probability, but this is massively exponential and clearly impossible.
- A “statistically correct” way to do this is to randomly sample a word from the probability distribution produced by the neural net at each time step. However, this is pretty high-variance and will get you a different translation on the same sentence each time you run it, so it’s not ideal.
- An intuitively obvious way is to do a greedy search - at each time step, just pick the word with highest probability. This is simple and efficient, but often gives suboptimal results.
- We can improve on greedy search by doing beam search. In beam search, we:
  - Keep K hypotheses at each time step.
  - For each of those K hypotheses, consider the K highest-probability words for the next time step. This yields K^2 total possibilities.
  - Filter those K^2 possibilities down to the K hypotheses with the highest probability. These are your K hypotheses for the next time step.
  - Repeat.

It turns out that beam search with pretty small K (about 5) works quite well, and that is what’s most commonly used today (at the time the lecture was given).

11 Lecture 11: Gated RNN review, Machine translation evaluation

11.1 Gated RNN review

In RNNs, you often have the vanishing gradient problem, where the gradient of the loss at time $t+n$ with respect to the parameters at time t are close to 0. When this happens, it’s often unclear whether this is because there actually isn’t a dependency or because your weights are just causing the gradients to vanish. This is an issue with the naive transition function:

$$h_{t+1} = \tanh\left(Wx_{t+1} + Uh_t + b\right)$$

This gives

$$\frac{\partial h_{t+1}}{\partial h_t} = U\frac{\partial \tanh a}{\partial a},$$

where $a = Wx_{t+1} + Uh_t + b$. We see that the loss must backpropagate through every single intermediate node, and this is what causes the vanishing gradient problem.

Gated units allow us to create “shortcut” connections, which pass information (and gradients) from one time step to another much later one. One such gated unit is the GRU, which has at each time step, in addition to the input x_t and hidden state h_t, a reset gate r_t, an update gate u_t, and a memory $\tilde{h}_t$. GRUs are governed by the equations

$$\begin{aligned}
u_t &= \sigma\left(W^{(u)}x_t + U^{(u)}h_{t-1} + b_u\right)\\
r_t &= \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1} + b_r\right)\\
\tilde{h}_t &= \tanh\left(Wx_t + r_t \circ Uh_{t-1} + b\right)\\
h_t &= u_t \circ \tilde{h}_t + (1-u_t) \circ h_{t-1}.
\end{aligned}$$

These gates allow the GRU to learn adaptive shortcut connections and learn to prune unnecessary connections.

If we roughly think of the naive RNN as a comptational unit with a memory register h, it's basically reading the whole register, doing an update $h \leftarrow \tanh\left(Wx_{t+1} + Uh_t + b\right)$, and writing all of that back to h. A GRU instead:

- Uses the reset gate r to select a readable subset of registers,
- Reads the subset $r \circ h$ and combines it with the input x to produce a new memory $\tilde{h}$,
- Uses the update gate u to select a writeable subset of registers, and
- Updates the subset of h with $h \leftarrow u \circ \tilde{h} + (1 - u) \circ h$.

A more complex model is the LSTM, which has, in addition to the input x_t and hidden state h_t, an input gate i_t, an output gate o_t, a forget gate f_t, an intermediate cell $\tilde{c}_t$, and a final cell c_t, governed by the equations

$$\begin{aligned}
i_t &= \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1} + b_i\right) \\
o_t &= \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1} + b_o\right) \\
f_t &= \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1} + b_f\right) \\
\tilde{c}_t &= \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1} + b\right) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= c_t \circ o_t.
\end{aligned}$$

This is pretty similar to the GRU, with c in the LSTM acting similarly to h in the GRU.

The addition:

$$\begin{aligned}
h_t &= u_t \circ \tilde{h}_t + (1 - u_t) \circ h_{t-1} && \text{(GRU)} \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t && \text{(LSTM)}
\end{aligned}$$

is what allows these gated units to overcome the issues of vanishing gradients, as they give the long backpropagation multiplication chains a friendly "constant" factor. In practice, this allows GRUs and LSTMs to remember information up to approximately 100 time steps back, while naive RNNs pretty much forget everything after 10 time steps.

Some practical tips for training a gated RNN:

- Use a LSTM or GRU; it will make your life easier.
- Initialize the recurrent matrices (the ones that are being multiplied by something from the previous time step, i.e. the various flavors of U in the equations above) to be orthogonal. This seems to prevent issues at the beginning.
- Initialize the other matrices with sensibly small values.
- Initialize the biases for the reset/forget gates to be around 1 or 2 so that the NNs default to remembering at the beginning of the first batch of training sentences.
- Use adaptive learning rate algorithms (e.g. Adam, AdaDelta) instead of vanilla SGD.
- Clip gradient norms to prevent exploding gradients.

- If using dropout regularization, don't randomly dropout horizontally (from one time step to the next), as this will mean that after a few time steps you would have forgotten most of the information from the past. Random dropout on the vertical connections (from one layer to the next within a time step) is fine.
- To improve performance, train an ensemble of around 10 NNs and average their predictions.

11.2 Machine translation evaluation

Evaluating MT is hard: for any given sentence there are often many valid translations into any other language, so it's not obvious how to determine if any one translation is correct or not. One way to do MT evaluation is manually: we can ask bilingual humans whether the translation is correct or incorrect, or to rate the translation on a 5- or 7-point scale for adequacy (correctness) and fluency (whether it actually sounds like a well-formed sentence in the target language). We can also have them do comparative ranking - given two translations of a sentence, select the better one. However, this is slow and expensive.

Another idea people had was to test the translations in some easier-to-evaluate downstream application that uses MT as a component. However, the downstream application often doesn't depend heavily on the quality of the translation, and a much better translation might not perform much better on the downstream application than a very rough one.

The community finally settled on the BLEU (bilingual evaluation understudy) metric, which compares a machine-generated translation to a human-generated reference translation. It does this by computing n-gram precision (what percent of MT-produced n-grams are also found in the reference translation?) and using that as a metric. A few details prevent the metric from being "gamed":

- A brevity penalty penalizes machine translations that are much shorter than the reference translation. This prevents the metric from being gamed by the machine producing very short translations like "The." which are almost guaranteed to have perfect precision.
- If an n-gram appears m_m times in the machine translation but $m_r < m_m$ times in the reference translation, only m_r of the instances of that n-gram in the machine translation are considered as correct. This prevents the machine from gaming the metric by finding an n-gram that's very likely to appear in the reference translation and repeating it many times to subvert the brevity penalty.

To compute the score:

- We consider n-grams up to some given length, commonly $N = 4$.
- For each n, we compute p_n, which is the number of matched MT n-grams divided by the total number of n-grams produced by the MT.
- We also specify a weight w_n for each n.
- We compute the brevity penalty $BP = \exp\left(\min\left(0, 1 - \ell_r/\ell_m\right)\right)$, where ℓ_r is the length of the reference translation and ℓ_m is the length of the machine translation.
- We compute the final score
$$BLEU = BP \prod_{n=1}^{N} p_n^{w_n}.$$

When BLEU was first introduced, it was shown to correlate very well with human judgments of translation quality. However, when researchers started optimizing heavily for BLEU, this correlation dropped off dramatically. Today, BLEU scores produced by MT systems are comparable to those produced by human translators, but there is still a lot of work to be done in MT.

11.3 Computational complexity of word generation

In the last step of any language model, we need to multiply the hidden layer by a $|V| \times d_h$ matrix and compute a softmax over the result to generate an actual word prediction. This is incredibly computationally intensive, and people have been looking for ways to reduce the complexity.

One idea is to truncate the vocabulary to only include the most common words, but this actually doesn't work very well, and the model hits unknown words a lot. Other ideas were to use a hierarchical softmax or noise-contrastive estimation, but these techniques aren't very GPU-friendly, as they require a relatively large amount of sequential computation.

A more promising idea is to split up the training set into many smaller individual training sets, and to be selective about the words allowed at test time. Partitioning the training set into subsets with similar vocabularies allowed a group of researchers (On Using Very Large Target Vocabulary for Neural Machine Translation) to reduce the vocabulary in each partition by an order of magnitude (500k to 30k or 50k). At testing time, we restrict the vocabulary to the K most common words overall, unioned with the K' most likely translations of each of the words in the test set. This also reduced $|V|$ by about an order of magnitude by using K' around 10 or 20, and K around 15000.

An issue that this doesn't address is that new words will almost always be encountered at test time anyways (numbers, proper nouns, etc.). Dealing with this is another ongoing area of research.

12 Lecture 12: End-to-end Models for Speech Processing

Automatic speech recognition (ASR) refers to the task of converting an audio signal to the underlying textual representation. ASR is important because it provides a natural interface for human communication, and because it has a lot of possible applications (talking to cars, Siri, call center chatbots).

Classical approaches to ASR involve building generative models, starting with a statistical language model (usually N-gram models) and then deriving pronunciation models (usually just a mapping from words to pronunciations) and acoustic models (usually Gaussian mixture models) from it. They then take an audio signal and compute audio features from it by transforming the signal into the frequency domain (classical signal processing). Basically, when given an audio signal presenting features X, it just uses the model to predict the sequence of words

$$\hat{Y} = \text{argmax}_Y p(X|Y)p(Y).$$

As work in speech recognition progressed, all the subcomponents of the model were replaced by neural models:

- N-gram models were replaced with neural language models.
- Pronunciation lookup tables were replaced with neural network based pronunciation models. (Traditional pronunciation tables were unable to generalize to new words, e.g. proper nouns.)
- Gaussian mixture models were replaced with DNN-HMMs and LSTM-HMMs.
- Classical signal processing was replaced with convolutional signals on raw signals.

However, this still leaves us in a state where we have multiple independently trained models, which might not cooperate well with each other to perform the actual end task of speech recognition. This led to end-to-end models being developed, e.g.

- Connectionist Temporal Classification (CTC)
- Sequence to sequence (e.g. Listen Attend and Spell)

In an end-to-end model, the goal is to go directly from an audio signal $X = x_1x_2 \dots x_t$ (either raw audio or a minimally processed spectrogram) to an output text $Y = y_1y_2 \dots y_\ell$, where each y_i is a character (or maybe a phoneme; this was not specified in the lecture) in some language. The model should learn a direct probabilistic model $p(Y|X)$.

12.1 Connectionist Temporal Classification (CTC)

CTC is one example of an end-to-end model, and it takes spectrogram as its input X. The spectrogram is sliced along the time axis, and the time slices are fed into a bidirectional RNN which outputs a softmax prediction over a vocabulary of characters (phonemes?), including a special blank character, at each time step. At the end, repeated vocabulary tokens are deduplicated to create a final output text. To generate a final prediction, we simply go through the predictions at each time step and find the sequence of predicted tokens that maximizes

$$\sum_{t=0}^{T} s(k_t, t),$$

where k_t is a vocabulary class and $s(k_t, t) = \log p(k_t, t|X).$, subject to the condition that a pair of consecutive tokens must either be identical, or one of them must be the blank symbol.

The first CTC model produced transcripts that sounded correct when read, but had spelling and grammar issues. Adding a language model to rescore outputs at the end reduced word error rate from 30.1% to 8.7%. Google's CTC addressed these issues by integrating a language model into the CTC itself during training, and having the final model predict a sequence of words.

12.2 Listen Attend and Spell (LAS)

LAS is a different end-to-end model. While CTC makes predictions going forward and looks at one time step of input at a time, LAS has a decoder that repeatedly looks at the entire input, uses attention to determine where the most relevant information is (based on previous predictions), and spits out a token. It is an example of a sequence to sequence (seq2seq) model, which takes a sequence (here, the audio signal) as input, and produces a sequence (the text transcription) as output.

LAS has two parts: an encoder and a decoder. The encoder produces a hidden state h_t at each frame of the input, and the decoder produces a state s at each time step. The attention part of the model works by concatenating s to h_t for each t, passing h_ts through a neural network f to produce a score e_t, taking the softmax of all the values e_t to produce attention coefficients a_t, and creating a context vector $c = \sum a_t h_t$. The model then uses this context vector c to make its prediction about the next token. One notable aspect of this model is that it doesn't include a blank token in its voacbulary: since it looks at the whole input sequence at each step, it shouldn't ever need to predict a blank token.

The encoder is actually structured as a hierarchical encoder. This is because the number of audio frames in a typical input was too many (often something like 100 frames per second), and the attention model was not able to learn anything useful. The hierarchical encoder combines adjacent frames at each layer of the hierarchy, so that the final encoding does not have so many time slices.

The LAS model has a few limitations:

- The model isn't online - it cannot start producing output until it has received the entire input.
- The attention model is also a large computational bottleneck: each output token must compute attention with respect to each input frame.
- The word error rate increases significantly as input length increases.

12.3 Online Sequence to Sequence Models

Online seq2seq models attempt to address the first two of the above limitations by producing outputs as inputs arrive. Such a model must figure out when it has enough information to be confident about outputting symbols.

One such model is a neural transducer, which is conceptually very similar to a normal seq2seq model: it simply splits the input up into blocks, and runs a seq2seq model on each block. Each block also produces some state that is fed into the next block, which allows the model to maintain causality: the output at one frame can still influence the output at future frames. However, as there might be nothing at the end of a block that is in the middle of an input, we need some sort of blank token again - we introduce it in this model as the end-of-block symbol.

The blocks, however, introduce some alignment problems. First, when the model is trying to predict the probability of a prediction $\hat{y}_1\hat{y}_2\dots\hat{y}_s$ from an input X, there are $\binom{s+b-1}{b-1}$ ways it can arrive at that prediction, where b is the number of blocks, since the concatenation of the transducer outputs will also include $b-1$ end-of-block tokens. (Actually b end-of-block tokens, but the location of the last one is fixed.) This means that the probability of a final prediction $p(\hat{Y} = \hat{y}_1\hat{y}_2\dots\hat{y}_s|X)$ is actually a sum over a combinatorial number of terms:

$$p(\hat{Y} = \hat{y}_1\hat{y}_2\dots\hat{y}_s|X) = \sum_{\tilde{y}_1\tilde{y}_2\cdots\tilde{y}_{s+b-1}\in\hat{y}_1\hat{y}_2\cdots\hat{y}_s} p(\tilde{y}_1\tilde{y}_2\dots\tilde{y}_{s+b-1}|X).$$

Obviously this sum is impractical to compute, since each block depends on the previous blocks, so there actually isn't any way to "shortcut" this computation, i.e. you'd actually have to compute each one of the probabilities. In practice, we just try to find the complete sequence of $s+b-1$ tokens $\tilde{y}_1\tilde{y}_2\dots\tilde{y}_{s+b-1}$ with the highest probability using beam search, and take the corresponding $\hat{y}_1\hat{y}_2\dots\hat{y}_s$ as our prediction.

Similarly, during training, the gradient of the log likelihood (of the correct output) generated by an example will be the sum of $\binom{s+b-1}{b-1}$ terms:

$$\frac{\partial J}{\partial\theta} = \sum_{\tilde{y}_1\tilde{y}_2\cdots\tilde{y}_{s+b-1}\in\hat{y}_1\hat{y}_2\cdots\hat{y}_s} p(\tilde{y}_1\tilde{y}_2\dots\tilde{y}_{s+b-1}|X)\frac{\partial}{\partial\theta}\log p(\tilde{y}_1\tilde{y}_2\dots\tilde{y}_{s+b-1}|X).$$

We approach this the same way: we simplify this to

$$\frac{\partial}{\partial\theta}\log p(\tilde{y}_1\tilde{y}_2\dots\tilde{y}_{s+b-1}|X)$$

for the highest-probability complete sequence $\tilde{y}_1\tilde{y}_2\dots\tilde{y}_{s+b-1}$. However, beam search doesn't work well in practice in this case, so we use a pseudo-DP approach where for each block b (startin from the first path), we find the best paths that end b at token j for some set of possible j. We then use these pseudo-optima at block b to calculate the optima at the next block. Note however that this isn't true DP, since the best alignment of the first $b+1$ blocks might not start with the best alignment of the first b blocks, but it works well enough in practice.

12.4 Sequence to Sequence Improvements and Shortcomings

In recent times, there have been a number of improvements made to seq2seq models, summarized below:

- In the original LAS, the hierarchical encoder just did some pretty simple linear transformations (the exact transformations were not mentioned in the lecture) to reduce the resolution of the audio input. One improvement on this is by replacing that simple transformation with a CNN, which reduced word error rate substantially.

- Originally, output sequences were represented by output vocabularies of words, characters, or some combination thereof. However, none of these structures provide very good correspondence to the actual sound of English speech, so later work improved upon this by introducing an N-gram vocabulary. However, there are many ways to represent a word as a sequence of N-grams, so it was not obvious how to actually represent each word.
  - One possibility is to greedily choose the longest possible N-gram from the vocabulary at each step starting from the beginning of the word.
  - Another one is to choose the most common N-grams, i.e. select for the greatest compression factor.
  - The approach that the authors (in the paper mentioned in lecture) used was just to calculate the predicted probabilities of all possible representations and calculate a gradient with respect to all of them during training (using a probabilistic estimate from the calculated probabilities). This performed better than the other two approaches.
- Seq2seq models often have trouble predicting the correct token at the beginnings of words. It makes sense that there's less ambiguity later on in the word (there's only a few possible words once the first few characters are given).
  - This is especially problematic when the model is overconfident about an incorrect prefix, since the wrong word prefix will force a lot of other errors, which will lead to issues in the integrated language model.
  - Entropy regularization, which penalizes the model for making too-confident softmax predictions, has shown to address the overconfidence problem pretty effectively.
- Seq2seq models sometimes will ignore much of the input. Each output token produced by the model incurs some additive cost (i.e. in the objective function), so if the input is very long, the model may get into a state where ignoring the rest of the input will be less costly than even making a perfect prediction. This is because there's nothing encoded in the model that says it must "explain" the entire input.
  - This can be addressed by adding a coverage reward which adds a reward for each frame of the input whose total attention across all output token predictions is greater than some threshold.
- There's much more textual language data than audio data, so better integration of language models has a lot of potential to improve speech recognition performance. There's simply not enough audio data to create a good language model without introducing something that was trained on textual data; without a language model to help it, a speech recognition model is bound to make errors.
- A lot of speech recognition models predict one time step at a time. However, they don't consider the long-term impact of that prediction; for example, an incorrect word prefix can have heavy consequences on the predictions for the rest of the sentence. There's some work going on to better optimize these models with longer sequences in mind.

There's also exciting directions for future work. A couple of examples:

- Multispeaker/multichannel speech recognition. Most work today has been focused on the case of one speaker and one microphone, but settings with multiple speakers and multiple microphones haven't gotten much attention lately.
- Direct translation of speech in a source language to text in a different target language.