1 Train a text embedding model

Let's talk about how to handle text in a deep model. Imagine that you want to classify a document. Is it talking about politics, or business, or science maybe? You're going to have to look at the words in that document to figure that out. Words are really difficult. There are lots of them and most of them you never, ever see. In fact the ones that you rarely see, tend to be the most important ones. If you know that your document contains duh, you don't know much at all. But if you know that it contains retinopathy, you know that it's probably a medical document. Retinopathy appears with a frequency of 0.0001It's even possible that you've never seen that word yourself before. For deep learning, rare events like that are a problem. We like to have lots of training examples to learn from.

2 Semantic Ambiguity

Another problem is that we often use different words to mean almost the same thing. You can say cat or you can say kitty. They're not the same, but they mean similar things. Remember that when we have things that are similar, we really, really would like to share parameters between them. But kitty is completely different from cat. So if we want to share anything between them. We're going to have to learn that they are related. So that's a very big problem for deep learning. We'd like to see those important words often enough to be able to learn the meanings automatically. And we'd also like to learn how words relate to each other so that we can share parameters between them. But that would mean collecting a lot of label data. In fact, it would require way too much label data for any task that really matters.

3 Unsupervised Learning

So to solve that problem we're going to turn to unsupervised learning. Unsupervised learning means training without any labels. It's easy to find texts, lots of texts, take the web for example, or even just Wikipedia. That's plenty of text to train on, if you can figure out what to learn from it. Here's a very very simple but very powerful idea that we're going to use. Similar words tend to occur in similar contexts.

4 Embeddings

¿¿ If I say the cat purrs or this cat hunts mice, it's perfectly reasonable to also say the kitty purrs or this kitty hunts mice. The context gives you a strong idea that those words are similar. You have to be catlike to purr and hunt mice. So, let's learn to predict a word's context. The hope is that a model that's good at predicting a word's context will have to treat cat and kitty similarly, and will tend to bring them closer together. The beauty of this approach is that you don't have to worry about what the words actually mean, giving further meaning directly by the company they keep. There are many way to use this idea that similar words occur in similar contexts. In our case, we're going to use it to map words to small vectors called embeddings which are going to be close to each other when words have similar meanings, and far apart when they don't. Embedding solves of the sparsity problem. Once you have embedded your word into this small vector, now you have a word representation where all the catlike things like cats, kitties, kittens, pets, lions, are all represented by vectors that are very similar. Your model no longer has to learn new things for every way there is to talk about a cat. It can generalize from this particular pattern of catlike things.

5 Word2Vec

Let's look at one way to learn those embeddings. It's called Word2Vec. Word2Vec is a surprisingly simple model that works very well. Imagine you had a corpus of text with a sentence say, the quick brown fox jumps over the lazy dog. For each word in this sentence, we're going to map it to an embedding. Initially a random one. And then we're going to use that embedding to try and predict the context of the word. In this model, the context is simply the words that are nearby. Pick a random word in a window around the original word, and that's your target. Then train your model exactly as if it were a supervised problem. The model you're going to use to predict this nearby word is a simple logistic regression. Nothing deep about it, just a simple linear model.

6 SNE

Now, it would be great to see for yourself that these embeddings are clustering together as you'd expect. One way to see it, is by doing a nearest neighbor lookup of the words that are closest to any given word. Another way is to try to reduce the dimensionality of the embedding space down to two dimensions, and to plug the two dimensional representation. If you do that the native way, for example using PCA, you basically get a mush. You lose too much information in the process. What you need is a way of projecting that preserves the neighborhood structure of your data. Things that are close in the embedding space should remain close to the ends, things that are far should be far away from each other. One very effective technique that does exactly that is called t-SNE. You'll get to play with this visualization technique in the assignment

7 Word2Vec Details

Two more technical details about Word 2 Vec and about methods to learn embeddings in general. First, because of the way embeddings are trained, it's often better to measure the closeness using a cosine distance instead of L2, for example. That's because the length of the embedding vector is not relevant to the classification. In fact, it's often better to normalize all embedding vectors to simply have unit norm. Second, we have the issue of trying to predict words, and there are lots of them. So in Word 2 Vec we have this set up, we have a word that we're going to embed into a small vector, then feed that into a simple linear model with weights and biases and that outputs a softmax probability. This is then compared to the target which is another word in the context of the input word. The problem of course is that there might be many, many words in our vocabulary. And computing the softmax function of all those words can be very inefficient. But you can use a trick. Instead of treating the softmax as if the label had probability of 1, and every other word had probability of 0, you can sample the words that are not the targets, pick only a handful of them and act as if the other words were not there. This idea of sampling the negative targets for each example is often called sampled softmax. And it makes things faster at no cost in performance. That's all there is to it. You take your large corpus of texts, look at each word, embed it, and predict its neighbors. What you get out of that is a mapping from words to vectors where similar words are going to be close to each other.

8 Word Analogy Game Quiz

You actually get a little more than vector that just tell you when things are alike. And it's really fun. You have vectors. Imagine that you could do math with words now. Let's try an experiment. What do you think the results of puppy- dog + cat should be intuitively? How about taller- tall + short?

9 Word Analogy Game Solution

Let's see, you take a puppy, you remove what makes it a dog, and add what makes it a cat, and you get a kitten. Or you take taller, remove what pertains to being tall and add what pertains to being short, and you get shorter.

10 Analogies

Saying that a kitten is to cat what a puppy is to a dog is a semantic analogy. Just like saying that shorter is to short what taller is to tall is a syntactic analogy. Here's a fun emergent properties of those embedding factors. They let you express analogies like this in terms of mathematical operations. If you have a good model, and you take the vector for puppy, subtract the vector for dog and then add back the vector for cat, then you get another vector. If you look for what that vector is closest to in imbedding space, chances are that it will be kitten.

11 Sequences of Varying Length

You obviously need a very good model train on a lot of data to get this kind of result. And it's likely that the model that you're training in the assignments is just not trained with enough data to show this. But you can try. There are many other ways to learn embeddings that I won't cover. Now, that we have models for individual words, how do we deal with the fact that text is actually a sequence of words? So far, your models have only looked at inputs that were a fixed size. Fixed size means you can always turn things into a vector, and feed it to your neural network. When you have sequences of varying length, like speech or text, you can no longer do that. Now what?

12 RNNs

You've already seen how convolutional network uses shared parameters across space to extract patterns over an image. Now you're going to do the same thing but over time instead of space. This is the idea behind recurrent neural networks. Imagine that you have a sequence of events, at each point in time you want to make a decision about what's happened so far in this sequence. Imagine that you have a sequence of events, at each point in time you want to make a decision about what's happened so far in the sequence. If your sequence is reasonably stationary, you can use the same classifier at each point in time. That simplifies things a lot already. But since this is a sequence, you also want to take into account the past. Everything that happened before that point. One natural thing to do here is to use the state of the previous classifier as a summary of what happened before, recursively. Now, you would need a very deep neural network to remember far in the past. Imagine that this sequence could have hundreds, thousands of steps. It would basically mean to have a deep network with hundreds or thousands of layers. But instead, we're going to use tying again and have a single model responsible for summarizing the past and providing that information to your classifier. What you end up with is a network with a relatively simple repeating pattern with part of your classifier connecting to the input at each time step and another part called the recurrent connection connecting you to the past at each step.

13 Backprop Through time

To compute the parameter updates of a recurrent network, we need to backpropagate the derivative through time, all the way to the beginning of the sequence. Or in practice, more often, for as many steps as we can afford. All these derivatives are going to be applied to the same parameters. That's a lot of correlated updates all at once, for the same weights. This is bad for stochastic gradient descent. Stochastic gradient descent, prefers to have uncorrelated updates, to its parameters, for the stability of the training. This makes the math very unstable. Either the gradients grow exponentially and you end up with infinities. Or they go down to zero very quickly. And you end up not training anything.

14 Vanishing Exploding Gradients

These problems are often called the exploding and vanishing gradient problems. And we're going to fix them, surprisingly in very different ways. One using a very simple hack, and the other one with a very elegant but slightly complicated change to the model. The simple hack is called gradient clipping. In order to prevent the gradients from growing inbounded, you can compute their norm and shrink their step, when the norm grows too big. It's hacky, but it's cheap and effective. The more difficult thing to fix is gradient vanishing. Vanishing gradients make your model only remember recent events and forget the more distant past, which means recurrent neural nets tend to not work well past a few time steps.

15 LSTM

¿¿ This is where LSTMs come in. LSTM stands for long short-term memory. Now, conceptually, a recurrent neural network consists of a repetition of simple little units like this, which take as an input the past, a new inputs, and produces a new prediction and connects to the future. Now, what's in the middle of that is typically a simple set of layers. Some weights and some linearities.A typical neural network. With LSTMs, we're going to replace this little module with a strange little machine. It will look complicated, but it's still functionally a neural net. And you can drop it into your RNN and not worry about it. Everything else about the architecture remains the same, and it greatly reduces the vanishing gradient problem.

16 Memory Cell

So, what does this little machine look like? Remember, we're doing this to help R and Ns memorize things better. So let's forget about neural networks for just a moment. Imagine that you want a system to have memory. You need to do three things. First, you need to write the data into the memory. Then you will be able to read it back, and then you also want to be able to erase it, or forget. Here is one way to write this very same idea down as a diagram. You have a memory, which is maybe a matrix of values. You also have inputs coming into the memory, and a little instruction gate that tells whether you are going to write to that memory or not. You also have another gate that's says whether you are going to able to read from that memory or not. And then you have one final gate that says forget the data or write it back.

17 LSTM Cell

You can also write at those gates as multiplying the inputs either by 0, when the gate is closed, or by 1, when the gate is open. That's all fine, but what does it have to do with neural networks? Imagine that instead of having binary decisions at each gate, you had continuous decisions. For example, instead of yes-no gate at the input, you take the input and multiply it by a value that's between 0 and 1. If it's exactly 0, no input comes in. If it's exactly 1, you write it entirely to memory. Anything in between can be added partially to the memory. Now that becomes very interesting for us, because if that multiplicative factor is a continuous function that's also differentiable, that means we can take derivatives of it. And that also means we can back propagate through it. That's exactly what an LSTM is. We take this simple model of memory, we replace everything with continuous functions. And make that the new Lowell machine that's at the heart of a recurrent neural network.

18 LSTM Cell 2

The gating values for each gate get controlled by a tiny logistic regression on the input parameters. Each of them has its own set of shared parameters. And there's an additional hyperbolic tension sprinkled in here to keep the outputs between -1 and 1. It looks complicated but once you write down the math It's literally five lines of code, as you'll see in the assignment. And it's well-behaved, continues, and differentiable all the way, which means we can optimize those parameters very easily. So why do LSTMs work? Without going into too many details, all these little gates help the model keep its memory longer when it needs to, and ignore things when it should. As a result, the optimization is much easier, and the gradient vanishing, vanishes.

19 Regularization

There is one more thing to discuss about LSTMs. It's regularization. You can always use L2, that works. Dropout works well on your LSTMs as well, as long as you use it on the inputs or on the outputs, not on the recurrent connections. Now you know.

20 Beam Search

What can you do with an? Lots of things. Imagine that you have a model that predicts the next step of your sequence, for example. You can use that to generate sequences. For example, text, either one word at a time or one character at a time. For that, you take your sequence at time, t, you generate the predictions from your RNN, and then you sample from the predicted distribution. And then you pick one element based on its probability. And feed that sample to the next step and go on. Do this repeatedly, predict, sample, predict, sample, and you can generate a pretty good sequence of whatever your RNN models. There is a more sophisticated way to do this, that gives better results. Because just sampling the next prediction every time, is very greedy. Instead of just sampling once at each step, you could imagine sampling multiple times. Here we pick O but also A for example. When we have multiple sequences, often call that hypotheses, that you could continue predicting from at every step. You can choose the best sequence out of those, by computing the total probability of all the characters that you generated so far. By looking at the total probability of the multiple time steps at a time. You prevent the sampling from accidentally making one by choice, and being stuck with that one bad decision forever. For example, we could have just picked A by chance here, and never explored the hypothesis that O could lead to a very sensible next word. Of course, if you do this the number of sequences that you need to consider grows exponentially. There is a smarter way to do that, which is to do what's called a Beam Search. You only keep, say, the most likely few candidate sequences at every time step, and then simply prune the rest. In practice, this works very well for generating very good sequences from your model.

21 Play Legos

Let's look at some other fun use cases for sequence models. Remember, we've seen that one of the most entertaining aspects of deep models is that you can play Legos with them. You can mix them and attach them together, and then you can use back prop to optimize the whole thing directly end to end.

22 Captioning and Translation

If you think about RNNs in general, they are a very general, trainable model that maps variable-length sequences to fixed-length vectors. In fact, thanks to the sequence generation and Bing search that we've just discussed, they can also be made to map fixed-length vectors to sequences. Start from a vector, map it to the state of your RNN, then produce a prediction. Then sample from it, or use Bing search, and feed them back into the RNN to get the next one. Now you have those two building blocks, and you can stitch them together. It gives you a new model that maps sequences of arbitrary lengths to other sequences of arbitrary length. And it's fully trainable. What can you do with? Many things. Imagine that your input is a sequence of English words and you output a sequence of French words. You've just built a machine translation system. All you need is some parallel text. Imagine that your input is sounds and your output words. You've just built an end-to-end speech recognition system. Real systems based on variations on this design exist and are very competitive. In practice, they do require a lot of data and a lot of compute to work very well. We've also talked about conv nets, which basically map images into vectors that represent the content of that image. So picture what would happen if you took a conv net and connected it to an RNN. You have an image going in, and a sequence of things coming out. A sequence of words maybe. Maybe the caption for that image. To do that, you need training images and captions. There are a few data sets out there that you can use for that. Most notably the coco data set. It does images and crowdsourced captions for them. You can train a model that uses a cov net to analyze the image and generate captions from them. And it works. You can get great captions, generated completely automatically, and it sometimes fails in very, very funny ways.

23 Course Outro

What's really cool about those examples, is that you don't have to know much about the problem that you're trying to solve. All you need is data. Lot's of data. And computers. Lot's of them. And off you go. You can build a machine or any system that often does much better than specialized, handcrafted approaches to the problem. So, here we are. With those tools in your back pockets, you can now train models that can predict from images, texts, sequences, or any kind of data, for that matter. If you're curious, there's a ton more to learn in the field of deep learning. Things also change every day as new techniques get developed and people find new ways to use those models. Even with just what you've learned in this class you have the tools to design and train real models on real data, so what are you going to try first?