Analysis Summary

Title

Building Machine Learning Systems for Text Data with Recurrent Neural Networks

Overview

The goal of building machine learning systems that can operate on text data is a complex task that requires a deep understanding of the relationships between words in a sentence or document. In the previous lecture, embeddings were introduced as a way to convert words into vectorized representations with a large but not crazy number of dimensions. These embeddings can be used as a starting point for building machine learning systems that can operate on text data. However, for simple tasks like classifying the sentiment of a product review, it might suffice to break up the document into words, embed each word, and then pass that sequence of embeddings as input to a neural network. But for more challenging tasks like conversational replies or machine translation, a more sophisticated approach is needed.

Main Sections

There are two broad approaches to operating on text data with neural networks. The first approach involves giving an entire document as input to a neural network. This approach has several limitations, including the fact that it gives us a lot fewer training examples to work with, and we also have to deal with the fact that different documents that we might be trying to operate on could be of wildly varying sizes. The second approach involves giving a single word as input to a neural network. This approach also has its limitations, as the network will be missing out on all of the surrounding context that is important to understanding a particular word's meaning in a sentence or document.

To address these limitations, a compromise between the two extremes is needed. The first approach to achieving this compromise is to operate on one word at a time, but set up the architecture of the network so that it can remember some of its previous inputs and therefore retain some of the important context. This can be achieved by feeding the network's output back to its input. For example, if we give the neural network the first word of a sentence, it can make a guess about the next word, but then when we give it the second word of the sentence, it will also receive as input its own activations on the first word in the sentence. These activations can serve as a summary of what it saw before.

The result of this approach is a recurrent neural network. To reason about recurrent neural networks, it helps to unroll them over time. Each box in the unrolled network represents the same neural network, but at different points in time. Since we're inputting the different words one at a time and feeding the network's output back to its input, we represent feeding the output as arrows forward from time one to time two. On each time step, the network receives a word as input and tries to predict what the next word will be. The target for each of these time steps is the same as the input for the next time step.

Note that all of the inputs will be vectors from our embedding, but the outputs can't actually be embedding vectors because we need to allow the network to express uncertainty over different possible outputs. And so the outputs will actually be in our one-hot dictionary encoding. This might seem rather complicated, but in terms of computing the predictions, it's not really all that bad. We're giving an embedded word as input, and we're concatenating that with the previous activations from the last time step.

The tricky part, and the reason for unwinding the time steps, is computing the gradients. Broadly speaking, to take a gradient descent step, we want to figure out how the weights within the neural network influence the error on each of the words in the sentence. So our gradient will be an average of the losses on each of the different outputs at different time steps, but that by itself is not all that different from averaging loss gradients over a batch as we've been doing before. The insidious part is that for later time steps, the weights of the network influence the loss, not just through how they were applied to the input at that time step, but also how those weights affected the inputs from all of the previous time steps, because that information gets carried all the way into the eventual computation of output at a later time step.

Key Takeaways

• Recurrent neural networks are a type of neural network that can remember previous inputs and retain important context.

• They are achieved by feeding the network's output back to its input, allowing it to make guesses about the next word based on its previous inputs.

• The unrolled recurrent neural network consists of a series of boxes, each representing the same neural network at different points in time.

• The network receives a word as input, tries to predict the next word, and feeds its output back to its input for the next time step.

• The outputs are in a one-hot dictionary encoding, allowing the network to express uncertainty over different possible outputs.

• Recurrent neural networks often deliberately use sigmoid or tanh activation functions, which are prone to the vanishing gradient problem.

• The vanishing gradient problem makes it difficult to propagate information backwards through a large number of time steps, limiting the effectiveness of plain recurrent neural networks for tasks that require long-term memory.

Conclusion

Recurrent neural networks are a powerful tool for building machine learning systems that can operate on text data. By feeding the network's output back to its input, we can create a system that can remember previous inputs and retain important context. However, the vanishing gradient problem limits the effectiveness of plain recurrent neural networks for tasks that require long-term memory. To address this challenge, variants of recurrent neural networks have been developed, which are designed to achieve longer-term memory and are more effective at text and language processing tasks. These variants will be discussed in the next topic, and they have the potential to revolutionize the field of natural language processing.

Full Video Script

In the last lecture, we talked about embeddings, which we can train to convert words into vectorized representations with a large but not crazy number of dimensions.

And now we'd like to use those embeddings as a starting point for building machine learning systems that can operate on text data.

And for some simple tasks like classifying the sentiment of a product review, it might suffice to break up the document into words, embed each word, and then pass that sequence of embeddings as input to a neural network.

But with text data, there are many much more challenging tasks that we might be interested in, like conversational replies or machine translation.

And for those sorts of problems, we're going to need to do something much more sophisticated.

So as our running example here, we will use the task of predicting the next word in a sentence, given the words so far.

This is harder than just classifying text as positive or negative reviews, but it's also much easier than machine translation.

Thinking for a moment about how we want to set up neural networks in general to operate on any text processing task, there are, in a sense, two broad approaches.

One extreme would be to give an entire document as input to a neural network.

And the other extreme would be to give a single word as input to a neural network.

If we're operating on entire documents, that gives us a lot fewer training examples to work with.

And we also have to deal with the fact that different documents that we might be trying to operate on could be of wildly varying sizes.

But if we try to operate on one word at a time, then our network will be missing out on all of the surrounding context that is important to understanding a particular words, meaning in a sentence or a document.

And this causes problems if we're trying to do translations or trying to represent concepts that don't map one to one onto words.

And so both of these broad approaches make the learning problem way too difficult.

In one case, because we're trying to operate on way too much data at a time and not getting enough learning feedback.

And in the other case, because we're trying to operate on way too little data at a time and not having the information that we need to make good inferences.

And so each of the methods that we will actually consider will in fact be some sort of compromise between these two extremes.

The first approach will consider achieves this compromise by operating on one word at a time, but setting up the architecture of the network so that it can remember some of its previous inputs and therefore retain some of the important context.

So thinking in terms of our predict the next word problem, how the heck can we set up the architecture of a network to give it memory of inputs that it was given in the past? Well, the basic idea is to feed the network's output back to its input.

And so then if we give the neural network the first word of the sentence, it can make a guess about the next word, but then when we give it the second word of the sentence, it will also receive as input its own activations on the first word in the sentence.

And so those activations can serve as a summary of what it saw before.

When it outputs its guess for the third word, that will be fed back in along with the actual third word when it is trying to guess the fourth word.

And so the output or the activations of a hidden layer at the end of the network can serve as the summary of all of the previous words of the sentence.

And the result is what's known as a recurrent neural network.

To reason about recurrent neural networks, it helps to unroll them over time.

Each of these boxes represents the same neural network, but at different points in time.

Since we're inputting the different words one at a time and feeding the network's output back to its input, we represent feeding the output as these arrows forward from time one to time two.

And on each time step, the network receives a word as input and tries to predict what the next word will be.

And so the target for each of these time steps is the same as the input for the next time step.

Note that all of the inputs will be vectors from our embedding, but the outputs can't actually be embedding vectors because we need to allow the network to express uncertainty over different possible outputs.

And so the outputs will actually be in our one-hot dictionary encoding.

This might seem rather complicated, but in terms of computing the predictions, it's not really all that bad.

We're giving an embedded word as input, and we're concatenating that with the previous activations from the last time step.

The tricky part, and the reason for unwinding the time steps is computing the gradients.

Broadly speaking, to take a gradient descent step, we want to figure out how do the weights within the neural network influence the error on each of the words in the sentence.

So our gradient will be an average of the losses on each of the different outputs at different time steps, but that by itself is not all that different from averaging loss gradients over a batch as we've been doing before.

The insidious part is that for later time steps, the weights of the network influence the loss, not just through how they were applied to the input at that time step, but also how those weights affected the inputs from all of the previous time steps, because that information gets carried all the way into the eventual computation of output at a later time step.

So to do a gradient descent step, we wanted to average the loss over all of these time steps, but to compute the effect that the neural networks weights had on the loss at a particular time step, we have to compute not only how the weights affected the input from that time step, but also how the repeated applications of weights affected the earlier time steps and their contribution to error made on this time step.

I'm not going to go through a full derivation of those gradients, but I will post a reading that does.

But one thing that's noteworthy here is that recurrent neural networks often deliberately use sigmoid or 10H activation functions, because those tend to be prone to the vanishing gradient problem, and with a vanishing gradient problem, we don't expect gradients to propagate too far backwards through the network, and so we get things less wrong if we cut off these gradient propagations after a relatively small number of steps.

But that inability to propagate information backwards through a large number of time steps is part of the reason that I've been talking about this in terms of sentences and not documents.

And for exactly this reason of gradients not being able to propagate very far backwards, plain recurrent neural networks are not very good at solving tasks that require a lot of context and therefore long-term memory.

And so the topic for next time will be a variant of recurrent neural networks that are designed specifically to achieve longer-term memory and therefore be much more effective at text and language processing tasks.