Lecture 17: ResNets and Attention

Roger Grosse

1 Introduction

We have two unrelated agenda items for today. First, we'll revisit image classification in light of what we've learned about RNNs. In particular, we saw that one way to make it easier for RNNs to learn long-distance dependencies is to make it easy for each layer to represent the identity function, which lets them pass information unmodified through many layers. This is a useful thing for the network to do, and it also helps keep the gradients from exploding or vanishing. If we want to train image classifiers with a ridiculously large number of layers, we need to use these sorts of tricks. The deep residual network (ResNet) is a particularly elegant architecture which lets information pass directly through; it can be used to train networks with hundreds, or even thousands of layers, and is the current state-of-the-art for a variety of computer vision tasks.

Our second agenda item is attention. The problem with the encoder-decoder architecture for translation is that all the information about the input sentence needs to be stored in the vector of hidden activations. This has a fixed dimension (typically on the order of 1000), i.e. it doesn't grow with the length of the sentence. It's pretty neat that summarizing the meaning of a sentence as a vector works at all, but this strategy hits is limits once the sentences are about 20 words or so, a fairly typical sentence length. Attention-based architectures allow the network to refer back to the input sentence as they produce their output, thereby reducing the pressure on the hidden units and allowing them to easily handle very long sentences.

2 ResNets

Before 2015, the GoogLeNet (Inception) architecture set the standard for a deep conv net. It was about 20 layers deep, not counting pooling. In 2015, the new state-of-the-art on ImageNet was the deep residual network (ResNet), which had the distinction that that it was 150 layers deep. When we discussed image classification, I promised we'd come back to ResNets once we covered a key conceptual idea. That idea was exploding and vanishing gradients.

Recall that the Jacobian $\partial \mathbf{h}^{(T)}/\partial \mathbf{h}^{(1)}$ for an RNN is the product of the Jacobians of individual layers:

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}$$

Multiplying together lots of matrices causes the Jacobian to explode or vanish unless we're careful about keeping all of them close to the identity.

But notice that this same formula applies to the Jacobian for a feed-forward network (e.g. MLP or conv net). How come we never talked about exploding and vanishing gradients until we got to RNNs? The reason is that until recently, feed-forward nets were at most tens of layers deep, whereas RNNs would often be unrolled for hundreds of time steps. Hence, we'd be doing lots more steps of backprop (i.e. multiplying lots of Jacobians together), making things more likely to explode or vanish. This means if we want to train feed-forward nets with hundreds of layers, we need to figure out how to keep the backprop computations stable.

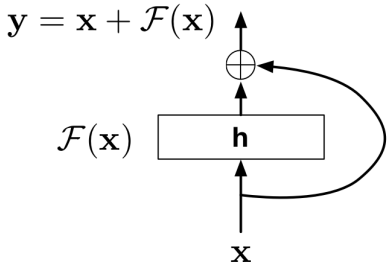
In Homework 3, you derived the backprop equations for the following architecture, where the inputs get added to the outputs:

$$\begin{aligned} \mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{h} &= \phi(\mathbf{z}) \\ \mathbf{y} &= \mathbf{x} + \mathbf{W}^{(2)}\mathbf{h} \end{aligned} \tag{1}$$

This is a special case of a more general architectural primitive called the **residual block**:

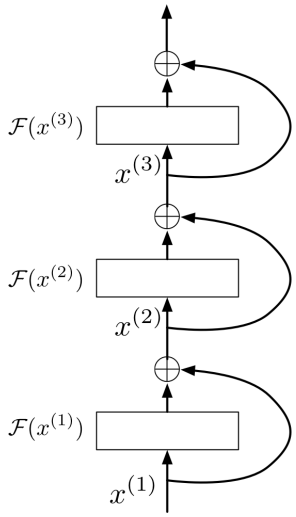
$$\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x}), \tag{2}$$

where $\mathcal{F}$ is a function called the **residual function**. In the above example, $\mathcal{F}$ is an MLP with one hidden layer. In general, it's typically a shallow neural net, with 1–3 hidden layers. We can represent the residual block graphically as follows:



Here, $\oplus$ denotes addition of the values.

We can string together multiple residual blocks in series to get a **deep residual network**, or **ResNet**:



(Each layer computes a separate residual function, with separate trainable parameters.) Last lecture, we noted two architectures that make it easy to represent the identity function: identity RNNs and LSTMs. The ResNet is a third such architecture. Observe that if each $\mathcal{F}$ returns zero (e.g. because all the weights are 0), then this architecture simply passes the input $\mathbf{x}$ through unmodified. I.e., it computes the identity function.

We can also see this algebraically in terms of the backprop equation for a residual block:

$$\begin{aligned}\overline{\mathbf{x}^{(\ell)}} &= \overline{\mathbf{x}^{(\ell+1)}} + \overline{\mathbf{x}^{(\ell+1)}}\,\frac{\partial \mathcal{F}}{\partial \mathbf{x}} \\ &= \overline{\mathbf{x}^{(\ell+1)}}\left(\mathbf{I} + \frac{\partial \mathcal{F}}{\partial \mathbf{x}}\right)\end{aligned} \tag{3}$$

Hence, if $\partial \mathcal{F}/\partial \mathbf{x} = 0$, the error signals are simply passed through unmodified. As long as $\partial \mathcal{F}/\partial \mathbf{x}$ is small, the Jacobian for the residual block will be close to the identity, and the error signals won't explode or vanish.

So that's the one big idea behind ResNets. If people say they are using ResNets for a vision task, they're probably referring to particular architectures based on the ones in this paper[1]. This paper achieved state-of-the-art on ImageNet in 2015, and since then, the state-of-the-art on many computer vision tasks has consisted of variants of these ResNet architectures. There's one important detail that needs to be mentioned: the input and output to a residual block clearly need to be the same size, because the output is the sum of the input and the residual function. But for conv nets, it's important to shrink the images (e.g. using pooling) in order to expand the number of feature maps. ResNets typically achieve this by having a few convolution layers with a stride of 2, so that the dimension of the image is reduced by a factor of 2 along each dimension.

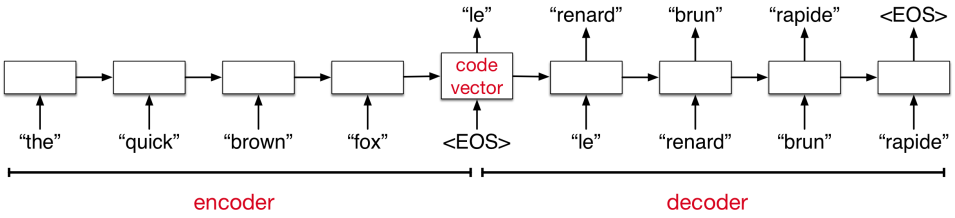
The benefit of the ResNet architecture is that it's possible to train absurdly large numbers of layers. The state-of-the-art ImageNet classifier from the above paper had 50 residual blocks, and the residual function for each was a 3-layer conv net, so the network as a whole had about 150 layers. Hardly anybody had been expecting it to be useful to train 150 layers. On a smaller object recognition benchmark called CIFAR, they were actually able to train a ResNet with 1000 layers, though it didn't work any better than their 100-layer network.

What on earth are all these layers doing? When we visualized the Inception activations, we found pretty good evidence that higher layers were learning more abstract and high-level features. But the idea that there are 150 meaningfully different levels of abstraction seems pretty fishy. We actually don't have a good explanation for why 150 layers works better than 50.

3 Attention

Our second topic for today is attention. Recall the encoder-decoder model for machine translation from last lecture:

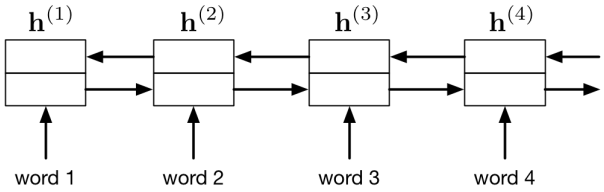
[1] K. He, X. Zhang, S. Ren, and J. Sun, 2016. Deep residual learning for image recognition



All the information the decoder receives about the input sentence is stored in a single code vector, which is the final hidden state of the encoder. This means the code vector needs to store all the relevant information about the input sentence — and since we're translating the whole sentence, that effectively means it must have memorized the sentence. It's a bit surprising that this is possible, though not implausible: it may require about 1000 bits to store the ASCII characters in a 20-word sentence, so it should be possible in principle to store the same information in a vector of length 5000 (a typical hidden dimension for this architecture). But still, this is putting a lot of pressure on the RNN's memory.

Attention-based modeling fixes this problem by allowing the decoder to look at the input sentence as it generates text. This removes the need for the hidden units to store the whole input sentence. Instead, they'll just have to remember a little bit of context about things like where it is in the input sentence and what part of speech it's looking for next. The original attention-based translation paper was Bahdanau et al., "Neural machine translation by jointly learning to align and translate"[2], and we'll be focusing on their architecture here.

This model has both an encoder and a decoder. Let's consider both in sequence. First, the encoder. The encoder's job is to compute an **annotation vector** for each word in the sentence; these vectors are what the decoder will see when it focuses on a word. One seemingly obvious choice would be to use a lookup table of word representations, just like the neural language model from Lecture 7. But words can have multiple meanings, and it often requires information from the rest of the sentence to disambiguate the meaning. The relevant information could be either earlier or later in the sentence. So instead, we use an architecture called a **bidirectional RNN**:



The original bidirectional RNN uses a kind of architecture called the gated recurrent unit (GRU), which is similar to the LSTM. You could use an LSTM instead if you want.

This is really just a fancy term for two completely separate RNNs, one of which processes the words in forward order, and the other of which processes them in reverse order. The hidden states of the two RNNs are concatenated at each time step to give the annotation vector.

The decoder architecture is shown in Figure 1. It is very similar to the RNN language models we've looked at, in that it is an RNN which predicts a

[2] D. Bahdanau, K. Cho, and Y. Bengio, Neural machine translation by jointly learning to align and translate. ICLR, 2015. `https://arxiv.org/abs/1409.0473`

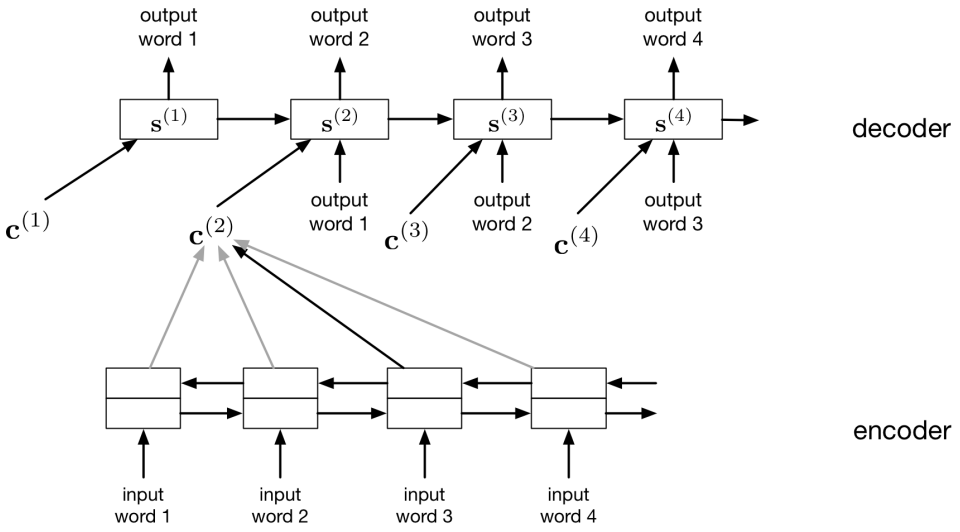


Figure 1: The decoder architecture from Bahdanau et al.'s attention-based translation model.

distribution over words at each time step, and receives the words as input. Like the RNN language models, it's trained using teacher forcing, so it receives the words of the actual target sentence as inputs. To sample from the model at test time, the words are sampled from the model's predictions and fed back in as inputs. So far, nothing new.

In practice, we don't actually sample from the model's predictions, since that has too high a chance of producing a silly sentence. Instead, we search for the most probable output sentence using a technique called beam search. But this is beyond the scope of the class.

The difference is that the decoder uses attention to compute a **context vector** $\mathbf{c}^{(i)}$ for each output time step i. (We're using i rather than t to keep separate the time steps of the encoder and decoder.) This is a **soft attention** model, which means that it has the ability to spread its attention across all the input words. More precisely, it computes a weighted average of the annotation vectors for all the input words:

Hard attention models only look at one part of the input at a time, but we won't consider those in this class.

$$\mathbf{c}^{(i)} = \sum_j \alpha_{ij} h^{(j)}, \tag{4}$$

where the attention weights α_{ij} are computed as a softmax over all the input words:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})} \tag{5}$$

$$e_{ij} = a(\mathbf{s}^{(i-1)}, \mathbf{h}^{(j)}). \tag{6}$$

Notice that the logits e_{ij} are a function of both the decoder's previous hidden state $\mathbf{s}^{(i-1)}$ and the annotation vector $\mathbf{h}^{(j)}$. The previous hidden state is clearly needed, since the decoder needs to remember some context about what it has already generated (such as the part of speech of the previous word) in order to know where to look. Using the annotation vectors themselves as inputs to the attention weights is an interesting approach, as it lets the attention mechanism implement **content-based addressing**, which looks up words according to their semantics rather than their position in the sentence. For instance, if the decoder has just produced an adjective,

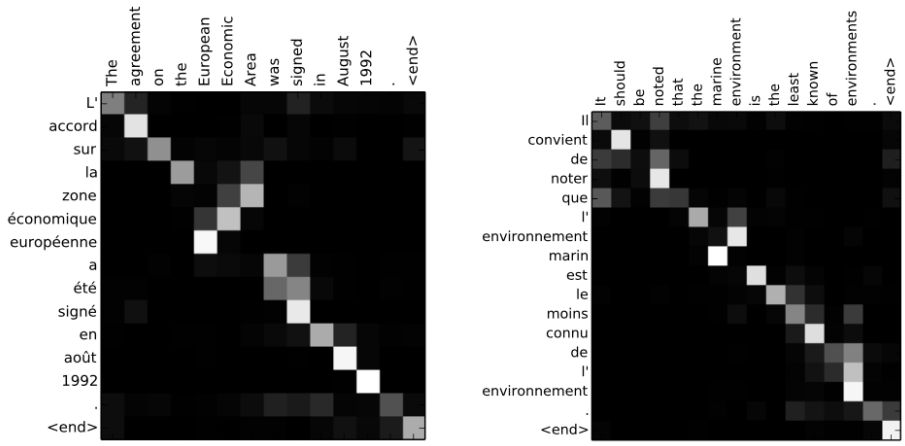


Figure 2: Visualizations of where the attention model is looking as it generates each output word. Each row corresponds to the attention vector (the α_{ij}'s) for one word in the output (French) sentence. Figure from Bahdanau et al.

then it may want to attend to nouns next. Of course, it is likely to use positional information as well. This isn't specified explicitly as part of the attention mechanism, but it can do it implicitly because it's pretty easy for the encoder RNN to count the position in the sentence as part of the annotation vectors.

Even though the soft attention mechansim has the ability to spread the attention over the entire sentence, in practice it usually chooses to look only at one input word or a handful of input words. Figure 2 shows some visualizations of where an English-to-French translation model is attending to as it generates each output word. For the most part, it marches in order through the input, looking at one word at a time. But it's able to reorder words as appropriate, such as when it translates "the European Economic Area" to "la zone économique européenne."