Lecture 13: Recurrent Neural Nets

Roger Grosse

1 Introduction

Most of the prediction tasks we've looked at have involved pretty simple kinds of outputs, such as real values or discrete categories. But much of the time, we're interested in predicting more complex structures, such as images or sequences. The next three lectures are about producing sequences; we'll get to producing images later in the course. If the inputs and outputs are both sequences, we refer to this as **sequence-to-sequence prediction**. Here are a few examples of sequence prediction tasks:

- As we discussed in Lecture 10, language modeling is the task of modeling the distribution over English text. This isn't really a prediction task, since the model receives no input. But the output is a document, which is a sequence of words or characters.
- In speech-to-text, we'd like take an audio waveform of human speech and output the text that was spoken. In text-to-speech, we'd like to do the reverse.
- In caption generation, we get an image as input, and would like to produce a natural language description of the image.
- Machine translation is an especially important example of sequence-to-sequence prediction. We receive a sentence in one language (e.g. English) and would like to produce an equivalent sentence in another language (e.g. French).

We've already seen one architecture which generate sequences: the neural language model. Recall that we used the chain rule of conditional probability to decompose the probability of a sentence:

$$p(w_1, \dots, w_T) = \prod_{t=1}^{T} p(w_t \,|\, w_1, \dots, w_{t-1}), \tag{1}$$

and then made a Markov assumption so that we could focus on a short time window:

$$p(w_t \,|\, w_1, \dots, w_{t-1}) = p(w_t \,|\, w_{t-K}, \dots, w_{t-1}), \tag{2}$$

where K is the context length. This means the neural language model is **memoryless**: its predictions don't depend on anything before the context window. But sometimes long-term dependencies can be important.

Figure 1 shows a neural language model with context length 1 being used to generate a sentence. Let's say we modify the architecture slightly by adding connections between the hidden units. This gives it a long-term

For a neural language model, each set of hidden units would usually receive connections from the last K inputs, for $K > 1$. For RNNs, usually it only has connections from the current input. Why?

Figure 1: **Left:** A neural language model with context length of 1. **Right:** Turning this into a recurrent neural net by adding connections between the hidden units. Note that information can pass through the hidden units, allowing it to model long-distance dependencies.

memory: information about the first word can flow through the hidden units to affect the predictions about later words in the sentence. Such an architecture is called a **recurrent neural network (RNN)**. This seems like a simple change, but actually it makes the architecture much more powerful. RNNs are widely used today both in academia and in the technology industry; the state-of-the-art systems for all of the sequence prediction tasks listed above use RNNs.

1.1 Learning Goals

- Know what differentiates RNNs from multilayer perceptrons and memoryless models.
- Be able to design RNNs by hand to perform simple computations.
- Know how to compute the loss derivatives for an RNN using backprop through time.
- Know how RNN architectures can be applied to sequence prediction problems such as language modeling and machine translation.

2 Recurrent Neural Nets

We've already talked about RNNs as a kind of architecture which has a set of hidden units replicated at each time step, and connections between them. But we can alternatively look at RNNs as dynamical systems, i.e. systems which change over time. In this view, there's just a single set of input units, hidden units, and output units, and the hidden units feed into themselves. This means the graph of an RNN may have **self-loops**; this is in contrast to the graphs for feed-forward neural nets, which must be directed acyclic graphs (DAGs). What these self-loops really mean is that the values of the hidden units at one time step depend on their values at the previous time step.

We can understand more precisely the computation the RNN is performing by **unrolling** the network, i.e. explicitly representing the various units at all time steps, as well as the connections between them. For a given sequence length, the unrolled network is essentially just a feed-forward neural net, although the weights are shared between all time steps. See Figure 2 for an example.

The trainable parameters for an RNN include the weights and biases for all of the layers; these are replicated at every time step. In addition, we

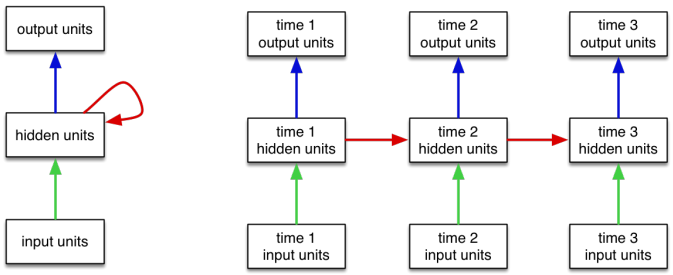


Figure 2: An example of an RNN and its unrolled representation. Note that each color corresponds to a weight matrix which is replicated at all time steps.

need some extra parameters to get the whole thing started, i.e. determine the values of the hidden units at the first time step. We can do this one of two ways:

- We can learn a separate set of biases for the hidden units in the first time step.
- We can start with a dummy time step which receives no inputs. We would then learn the initial values of the hidden units, i.e. their values during the dummy time step.

Really, these two approaches aren't very different. The signal from the $t = 0$ hiddens to the $t = 1$ hiddens is always the same, so we can just learn a set of biases which do the same thing.

Let's look at some simple examples of RNNs.

Example 1. *Figure 3 shows an example of an RNN which sums its inputs over time. All of the units are linear. Let's look at each of the three weights:*

- *The hidden-to-output weight is 1, which means the output unit just copies the hidden activation.*
- *The hidden-to-hidden weight is 1, which means that in the absence of any input, the hidden unit just remembers its previous value.*
- *The input-to-hidden weight is 1, which means the input gets added to the hidden activation in every time step.*

Example 2. *Figure 3 shows a slightly different RNN which receives two inputs at each time step, and which determines which of the two inputs has a larger sum over time steps. The hidden unit is linear, and the output unit is logistic. Let's look at what it's doing:*

- *The output unit is a logistic unit with a weight of 5. Recall that large weights squash the function, effectively making it a hard threshold at 0.*
- *The hidden-to-hidden weight is 1, so by default it remembers its previous value.*

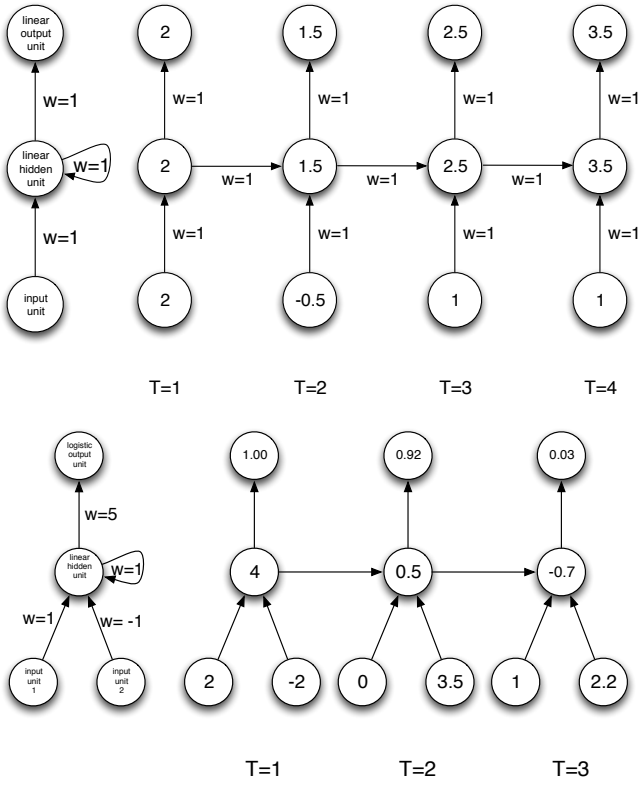


Figure 3: **Top:** the RNN for Example 1. **Bottom:** the RNN for Example 2.

- *The input-to-hidden weights rae 1 and -1, which means it adds one of the inputs and subtracts the other.*

Example 3. *Now let's consider how to get an RNN to perform a slightly more complex computation: the* **parity function**. *This function takes in a sequence of binary inputs, and returns 1 if the number of 1's is odd, and 0 if it is even. It can be computed sequentially by computing the parity of the initial subsequences. In particular, each parity bit is the XOR of the current input with the previous parity bit:*

$$\begin{aligned} \textit{Input:} &\quad 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1 \\ \textit{Parity bits:} &\quad 0\ 1\ 1\ 0\ 1\ 1 \longrightarrow \end{aligned}$$

This suggests a strategy: the output unit $y^{(t)}$ represents the parity bit, and it feeds into the computation at the next time step. In other words, we'd like to achieve the following relationship:

$y^{(t-1)}$	$x^{(t)}$	$y^{(t)}$
0	0	0
0	1	1
1	0	1
1	1	0

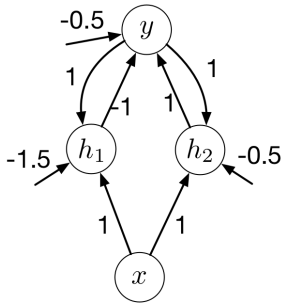


Figure 4: RNN which computes the parity function (Example 3).

But remember that a linear model can't compute XOR, so we're going to need hidden units to help us. Just like in Lecture 5, we can let one hidden unit represent the AND of its inputs and the other hidden unit represent the OR. This gives us the following relationship:

$y^{(t-1)}$	$x^{(t)}$	$h_1^{(t)}$	$h_2^{(t)}$	$y^{(t)}$
0	*0*	*0*	*0*	*0*
0	*1*	*0*	*1*	*1*
1	*0*	*0*	*1*	*1*
1	*1*	*1*	*1*	*0*

Based on these computations, the hidden units should receive connections from the input units and the previous time step's output, and the output unit should receive connections from the hidden units. Such an architecture is shown in Figure 4. This is a bit unusual, since output units don't usually feed back into the hiddens, but it's perfectly legal.

The activation functions will all be hard thresholds at 0. Based on this table of relationships, we can pick weights and biases using the same techniques from Lecture 5. This is shown in Figure 4.

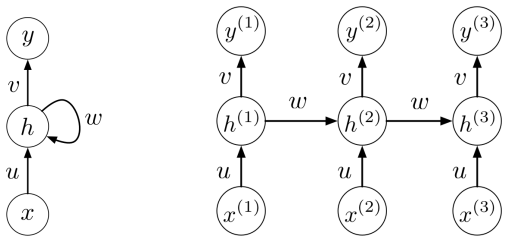
How would you modify this solution to use logistics instead of hard thresholds?

For the first time step, the parity bit should equal the input. This can be achieved by postulating a dummy output $y^{(0)} = 0$.

3 Backprop Through Time

As you can guess, we don't normally set the weights and biases by hand; instead, we learn them using backprop. There actually isn't much new here. All we need to do is run ordinary backprop on the unrolled graph, and account for the weight sharing. Despite the conceptual simplicity, the algorithm gets a special name: **backprop through time**.

Consider the following RNN:



It performs the following computations in the forward pass:

All of these equations are basically like the feed-forward case except $z^{(t)}$.

$$z^{(t)} = ux^{(t)} + wh^{(t-1)} \tag{3}$$
$$h^{(t)} = \phi(z^{(t)}) \tag{4}$$
$$r^{(t)} = vh^{(t)} \tag{5}$$
$$y^{(t)} = \phi(r^{(t)}). \tag{6}$$

Figure 5 shows the unrolled computation graph. Note the weight sharing. Now we just need to do backprop on this graph, which is hopefully a completely mechanical procedure by now:

Pay attention to the rules for $\overline{h^{(t)}}$, $\overline{u}$, $\overline{v}$, and $\overline{w}$.

$$\overline{\mathcal{L}} = 1 \tag{7}$$
$$\overline{y^{(t)}} = \overline{\mathcal{L}}\,\frac{\partial \mathcal{L}}{\partial y^{(t)}} \tag{8}$$
$$\overline{r^{(t)}} = \overline{y^{(t)}}\,\phi'(r^{(t)}) \tag{9}$$
$$\overline{h^{(t)}} = \overline{r^{(t)}}\,v + \overline{z^{(t+1)}}\,w \tag{10}$$
$$\overline{z^{(t)}} = \overline{h^{(t)}}\,\phi'(z^{(t)}) \tag{11}$$
$$\overline{u} = \sum_{t=1}^{T} \overline{z^{(t)}}\,x^{(t)} \tag{12}$$
$$\overline{v} = \sum_{t=1}^{T} \overline{r^{(t)}}\,h^{(t)} \tag{13}$$
$$\overline{w} = \sum_{t=1}^{T-1} \overline{z^{(t+1)}}\,h^{(t)} \tag{14}$$

These update rules are basically like the ones for an MLP, except that the weight updates are summed over all time steps.

Why are the bounds different in the summations over t?

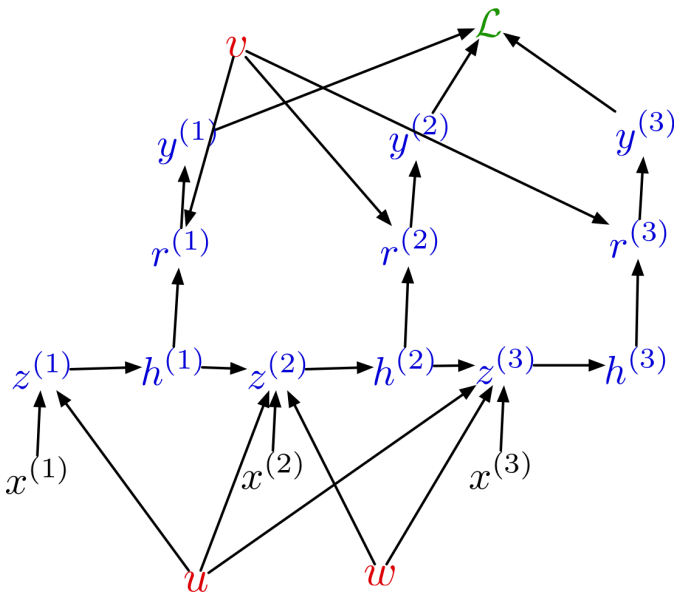


Figure 5: The unrolled computation graph.

The vectorized backprop rules are analogous:

Remember that for all the activation matrices, rows correspond to training examples and columns corresopnd to units, and N is the number of data points (or mini-batch size).

$$\overline{\mathcal{L}} = 1 \tag{15}$$

$$\overline{\mathbf{Y}^{(t)}} = \overline{\mathcal{L}} \frac{\partial \mathcal{L}}{\partial \mathbf{Y}^{(t)}} \tag{16}$$

$$\overline{\mathbf{R}^{(t)}} = \overline{\mathbf{Y}^{(t)}} \circ \phi'(\mathbf{R}^{(t)}) \tag{17}$$

$$\overline{\mathbf{H}^{(t)}} = \overline{\mathbf{R}^{(t)}} \, \mathbf{V}^\top + \overline{\mathbf{Z}^{(t+1)}} \, \mathbf{W}^\top \tag{18}$$

$$\overline{\mathbf{Z}^{(t)}} = \overline{\mathbf{H}^{(t)}} \circ \phi'(\mathbf{Z}^{(t)}) \tag{19}$$

$$\overline{\mathbf{U}} = \frac{1}{N} \sum_{t=1}^{T} \overline{\mathbf{Z}^{(t)}}^\top \mathbf{X}^{(t)} \tag{20}$$

$$\overline{\mathbf{V}} = \frac{1}{N} \sum_{t=1}^{T} \overline{\mathbf{R}^{(t)}}^\top \mathbf{H}^{(t)} \tag{21}$$

$$\overline{\mathbf{W}} = \frac{1}{N} \sum_{t=1}^{T-1} \overline{\mathbf{Z}^{(t+1)}}^\top \mathbf{H}^{(t)} \tag{22}$$

When implementing RNNs, we generally do an explicit summation over time steps, since there's no easy way to vectorize over time. However, we still vectorize over training examples and units, just as with MLPs.

That's all there is to it. Now you know how to compute cost derivatives for an RNN. The tricky part is how to use these derivatives in optimization. Unless you design the architecture carefully, the gradient descent updates will be unstable because the derivatives explode or vanish over time. Dealing with exploding and vanishing gradients will take us all of next lecture.

4 Sequence Modeling

Now let's look at some ways RNNs can be applied to sequence modeling.

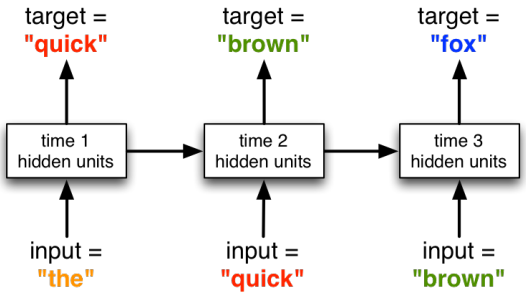
4.1 Language Modeling

We can use RNNs to do language modeling, i.e. model the distribution over English sentences. Just like with n-gram models and the neural language model, we'll use the Chain Rule of Conditional Probability to decompose the distribution into a sequence of conditional probabilities:

$$p(w_1, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_1, \ldots, w_{t-1}), \tag{23}$$

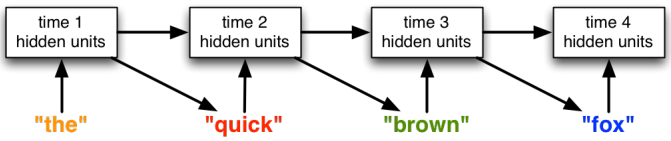
However, unlike with the other two models, we *won't* make a Markov assumption. In other words, the distribution over each word depends on *all* the previous words. We'll make the predictions using an RNN; each of the conditional distributions will be predicted using the output units at a given time step. As usual, we'll use a softmax activation function for the output units, and cross-entropy loss.

At training time, the words of a training sentence are used as both the inputs and the targets to the network, as follows:



It may seem funny to use the sentence as both input and output — isn't it easy to predict a sentence from itself? But each word appears as a target *before* it appears as an input, so there's no way for information to flow from the word-as-input to the word-as-target. That means the network can't cheat by just copying its input.

To generate from the RNN, we sample each of the words in sequence from its predictive distribution. This means we compute the output units for a given time step, sample the word from the corresponding distribution, and then feed the sampled word back in as an input in the next time step. We can represent this as follows:



Remember that vocabularies can get very large, especially once you include proper nouns. As we saw in Lecture 10, it's computationally difficult to predict distributions over millions of words. In the context of a

neural language model, one has to deal with this by changing the scheme for predicting the distribution (e.g. using hierarchical softmax or negative sampling). But RNNs have memory, which gives us another option: we can model text one *character* at a time! In addition to the computational problems of large vocabularies, there are additional advantages to modling text as sequences of characters:

- Any words that don't appear in the vocabulary are implicitly assigned probability 0. But with a character-based language model, there's only a finite set of ASCII characters to consider.
- In some languages, it's hard to define what counts as a word. It's not always as simple as "a contiguous sequence of alphabetical symbols." E.g., in German, words can be built compositionally in terms of simpler parts, so you can create perfectly meaningful words which haven't been said before.

Here's an example from Geoffrey Hinton's Coursera course of a paragraph generated by a character-level RNN which was trained on Wikipedia back in 2011.[1] (Things have improved significantly since then.)

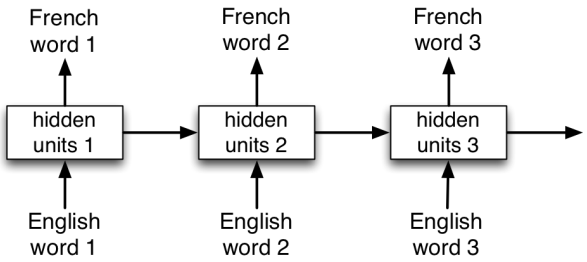
> He was elected President during the Revolutionary War and forgave Opus Paul at Rome. The regime of his crew of England, is now Arab women's icons in and the demons that use something between the characters' sisters in lower coil trains were always operated on the line of the ephemerable street, respectively, the graphic or other facility for deformation of a given proportion of large segments at RTUS). The B every chord was a "strongly cold internal palette pour even the white blade."

The first thing to notice is that the text isn't globally coherent, so it's clearly not just memorized in its entirety from Wikipedia. But the model produces mostly English words and some grammatical sentences, which is a nontrivial achievement given that it works at the character level. It even produces a plausible non-word, "ephemerable", meaning it has picked up some morphological structure. The text is also locally coherent, in that it starts by talking about politics, and then transportation.

4.2 Neural Machine Translation

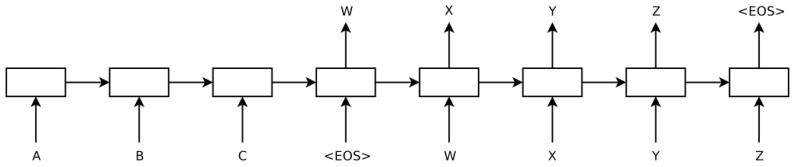
Machine translation is probably the canonical sequence-to-sequence prediction task. Here, the input is a sentence in one language (e.g. English), and the output is a sentence in another language (e.g. French). One could try applying an RNN to this task as follows:

[1] J. Martens. Generating text with recurrent neural networks. ICML, 2011



But this has some clear problems: the sentences might not even be the same length, and even if they were, the words wouldn't be perfectly aligned because different languages have different grammar. There might also be some ambiguous words early on in the sentence which can only be resolved using information from later in the sentence.

Another approach, which was done successfully in 2014[2], is to have the RNN first read the English sentence, remember as much as it can in its hidden units, and then generate the French sentence. This is represented as follows:



The special end-of-sentence token `<EOS>` marks the end of the input. The part of the network which reads the English sentence is known as the **encoder**, and the part that reads the French sentence is known as the **decoder**, and they don't share parameters.

Interestingly, remembering the English sentence is a nontrivial subproblem in itself. We can defined a simplified task called **memorization**, where the network gets an English sentence as input, and has to output the same sentence. Memorization can be a useful testbed for experimenting with RNN algorithms, just as MNIST is a useful testbed for experimenting with classification algorithms.

Before RNNs took over, most machine translation was done by algorithms which tried to transform one sentence into another. The RNN approach described above is pretty different, in that it converts the whole sentence into an abstract semantic representation, and then uses that to generate the French sentence. This is a powerful approach, because the encoders and decoders can be *shared* between different languages. Inputs of any language would be mapped to a common semantic space (which ought to capture the "meaning"), and then any other langage could be generated from that semantic representation. This has actually been made to work, and RNNs are able to perform machine translation on pairs of languages for which there were no aligned pairs in the training set!

[2] I. Sutskever. Sequence to sequence learning with neural networks. 2014

Input:

```
j=8584
for x in range(8):
  j+=920
b=(1500+j)
print((b+7567))
```

Target: 25011.

Input:

```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else
    5308)
```

Target: 1218.

Input:

```
vqppkn
sqdvfljmnc
y2vxdddsepnimcbvubkomhrpliibtwztbljipcc
```

Target: hkhpg

Figure 6: **Left:** Example inputs for the "learning to execute" task. **Right:** An input with scrambled characters, to highlight the difficulty of the task.

Input:

```
print(6652).
```

Target:	6652.
"Baseline" prediction:	6652.
"Naive" prediction:	6652.
"Mix" prediction:	6652.
"Combined" prediction:	6652.

Input:

```
d=5446
for x in range(8):d+=(2678 if 4803<2829 else 9848)
print((d if 5935<4845 else 3043)).
```

Target:	3043.
"Baseline" prediction:	3043.
"Naive" prediction:	3043.
"Mix" prediction:	3043.
"Combined" prediction:	3043.

```
print((5997-738)).
```

Target:	5259.
"Baseline" prediction:	5101.
"Naive" prediction:	5101.
"Mix" prediction:	5249.
"Combined" prediction:	5229.

Input:

```
print(((1090-3305)+9466)).
```

Target:	7251.
"Baseline" prediction:	7111.
"Naive" prediction:	7099.
"Mix" prediction:	7595.
"Combined" prediction:	7699.

Figure 7: Examples of outputs of the RNNs from the "Learning to execute" paper.

4.3 Learning to Execute Programs

A particularly impressive example of the capabilities of RNNs is that they are able to learn to execute simple programs. This was demonstrated by Wojciech Zaremba and Ilya Sutskever, then at Google.[3] Here, the input to the RNN was a simple Python program consisting of simple arithmetic and control flow, and the target was the result of executing the program. Both the inputs and the targets were fed to the RNN one *character* at a time. Examples are shown in Figure 6.

Their RNN architecture was able to learn to do this fairly well. Some examples of outputs of various versions of their system are shown in Figure 7. It's interesting to look at the pattern of mistakes and try to guess what the networks do or don't understand. For instance, the networks don't really seem to understand carrying: they know that something unusual needs to be done, but it appears they're probably just making a guess.

[3] W. Zaremba and I. Sutskever. Learning to Execute. ICLR, 2015.