6 Language Models 4: Recurrent Neural Network Language Models

The neural-network models presented in the previous chapter were essentially more powerful and generalizable versions of n-gram models. In this section, we talk about language models based on recurrent neural networks (RNNs), which have the additional ability to capture long-distance dependencies in language.

6.1 Long Distance Dependencies in Language

He doesn't have very much confidence in himself
She doesn't have very much confidence in herself

Figure 14: An example of long-distance dependencies in language.

Before speaking about RNNs in general, it's a good idea to think about the various reasons a model with a limited history would not be sufficient to properly model all phenomena in language.

One example of a long-range grammatical constraint is shown in Figure 14. In this example, there is a strong constraint that the starting "he" or "her" and the final "himself" or "herself" must match in gender. Similarly, based on the subject of the sentence, the conjugation of the verb will change. These sorts of dependencies exist regardless of the number of intervening words, and models with a finite history e_{i-n+1}^{i-1}, like the one mentioned in the previous chapter, will never be able to appropriately capture this. These dependencies are frequent in English but even more prevalent in languages such as Russian, which has a large number of forms for each word, which must match in case and gender with other words in the sentence.[21]

Another example where long-term dependencies exist is in **selectional preferences** [13]. In a nutshell, selectional preferences are basically common sense knowledge of "what will do what to what". For example, "I ate salad with a fork" is perfectly sensible with "a fork" being a tool, and "I ate salad with my friend" also makes sense, with "my friend" being a companion. On the other hand, "I ate salad with a backpack" doesn't make much sense because a backpack is neither a tool for eating nor a companion. These selectional preference violations lead to nonsensical sentences and can also span across an arbitrary length due to the fact that subjects, verbs, and objects can be separated by a great distance in sentences across many languages.

Finally, there are also dependencies regarding the **topic** or **register** of the sentence or document. For example, it would be strange if a document that was discussing a technical subject suddenly started going on about sports – a violation of topic consistency. It would also be unnatural for a scientific paper to suddenly use informal or profane language – a lack of consistency in register.

These and other examples describe why we need to model long-distance dependencies to create workable applications.

[21] See `https://en.wikipedia.org/wiki/Russian_grammar` for an overview.

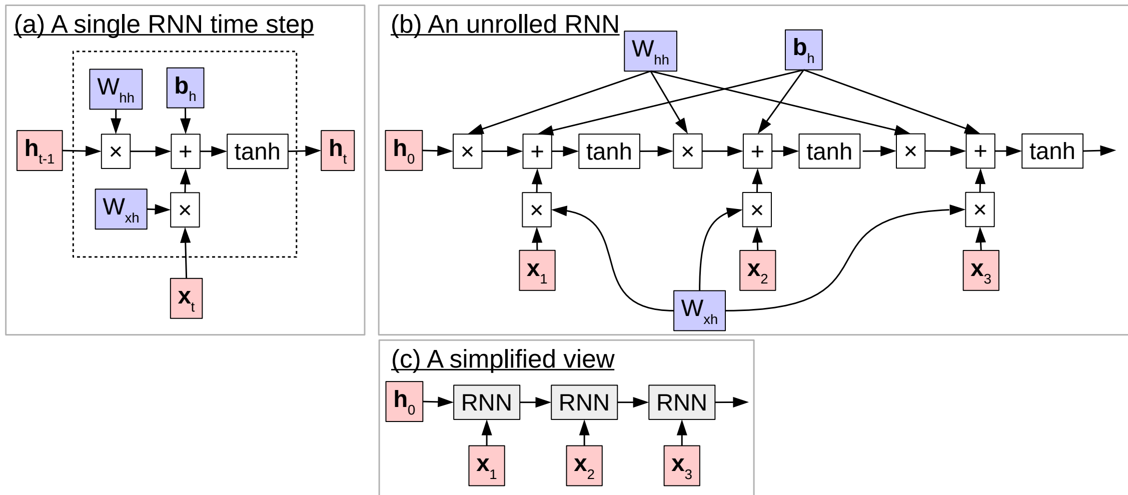

Figure 15: Examples of computation graphs for neural networks. (a) shows a single time step. (b) is the unrolled network. (c) is a simplified version of the unrolled network, where gray boxes indicate a function that is parameterized (in this case by W_{xh}, W_{hh}, and $\boldsymbol{b}_h$).

6.2 Recurrent Neural Networks

Recurrent neural networks (RNNs; [3]) are a variety of neural network that makes it possible to model these long-distance dependencies. The idea is simply to add a connection that references the previous hidden state $\boldsymbol{h}_{t-1}$ when calculating hidden state $\boldsymbol{h}$, written in equations as:

$$\boldsymbol{h}_t = \begin{cases} \tanh(W_{xh}\boldsymbol{x}_t + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h) & t \geq 1, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{44}$$

As we can see, for time steps $t \geq 1$, the only difference from the hidden layer in a standard neural network is the addition of the connection $W_{hh}\boldsymbol{h}_{t-1}$ from the hidden state at time step $t-1$ connecting to that at time step t. As this is a recursive equation that uses $\boldsymbol{h}_{t-1}$ from the previous time step, we also define $\boldsymbol{h}_0 = \boldsymbol{0}$ (a vector full of zeroes), to ensure that we can properly calculate $\boldsymbol{h}_1$ for the first time step in our sequence. This single time step of a recurrent neural network is shown visually in the computation graph shown in Figure 15(a).

When performing this visual display of RNNs, it is also common to "unroll" the neural network in time as shown in Figure 15(b), which makes it possible to explicitly see the information flow between multiple time steps. From unrolling the network, we can see that we are still dealing with a standard computation graph in the same form as our feed-forward networks, on which we can still do forward computation and backward propagation, making it possible to learn our parameters. It also makes clear that the recurrent network has to start somewhere with an initial hidden state $\boldsymbol{h}_0$. This initial state is often set to be a vector full of zeros, treated as a parameter $\boldsymbol{h}_{\text{init}}$ and learned, or initialized according to some other information (more on this in Section 7).

Finally, for simplicity, it is common to abbreviate the whole recurrent neural network step with a single block "RNN" as shown in Figure 15. In this example, the boxes corresponding to RNN function applications are gray, to show that they are internally parameterized with W_{xh},

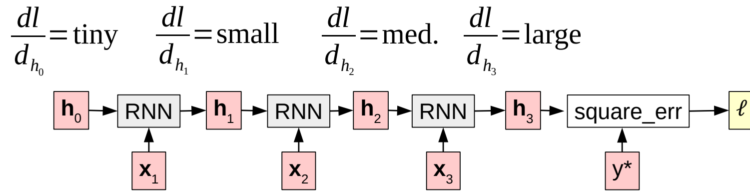

Figure 16: An example of the vanishing gradient problem.

W_{hh}, and $\boldsymbol{b}_h$. I will use this convention in the future to represent parameterized functions.

RNNs make it possible to model long distance dependencies because they have the ability to pass information between timesteps. For example, if some of the nodes in $\boldsymbol{h}_{t-1}$ encode the information that "the subject of the sentence is male", it is possible to pass on this information to $\boldsymbol{h}_t$, which can in turn pass it on to $\boldsymbol{h}_{t+1}$ and on to the end of the sentence. Thus, we can see that recurrent neural networks have the ability to preserve information across an arbitrary number of time steps.

Once we have the basics of RNNs, applying them to language modeling is (largely) straightforward [12]. We simply take the feed-forward language model of Equation 42 and enhance it with a recurrent connection as follows:

$$\begin{aligned}
\boldsymbol{m}_t &= M_{\cdot,e_{t-1}} \\
\boldsymbol{h}_t &= \begin{cases} \tanh(W_{mh}\boldsymbol{m}_t + W_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h) & t \geq 1, \\ \mathbf{0} & \text{otherwise.} \end{cases} \\
\boldsymbol{p}_t &= \text{softmax}(W_{hs}\boldsymbol{h}_t + b_s).
\end{aligned} \tag{45}$$

One thing that should be noted is that compared to the feed-forward language model, we are only feeding in the previous word instead of the two previous words. The reason for this is because (if things go well) we can expect that information about e_{t-2} and all previous words are already included in $\boldsymbol{h}_{t-1}$, making it unnecessary to feed in this information directly.

Also, for simplicity of notation, it is common to abbreviate the equation for $\boldsymbol{h}_t$ with a function $\text{RNN}(\cdot)$, following the simplified view of drawing RNNs in Figure 15(c):

$$\begin{aligned}
\boldsymbol{m}_t &= M_{\cdot,e_{t-1}} \\
\boldsymbol{h}_t &= \text{RNN}(\boldsymbol{m}_t, \boldsymbol{h}_{t-1}) \\
\boldsymbol{p}_t &= \text{softmax}(W_{hs}\boldsymbol{h}_t + b_s).
\end{aligned} \tag{46}$$

6.3 The Vanishing Gradient and Long Short-term Memory

However, while the RNNs in the previous section are conceptually simple, they also have problems: the **vanishing gradient** problem and the closely related cousin, the **exploding gradient** problem.

A conceptual example of the vanishing gradient problem is shown in Figure 16. In this example, we have a recurrent neural network that makes a prediction after several times steps, a model that could be used to classify documents or perform any kind of prediction over a sequence of text. After it makes its prediction, it gets a loss that it is expected to back-propagate over all time steps in the neural network. However, at each time step, when we run

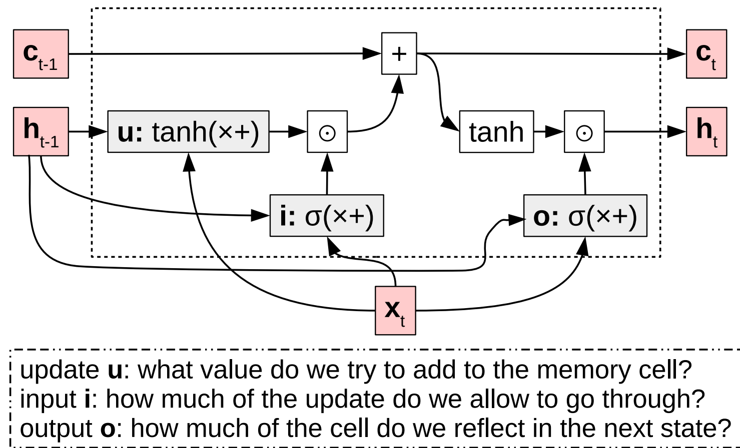

Figure 17: A single time step of long short-term memory (LSTM). The information flow between the $\boldsymbol{h}$ and cell $\boldsymbol{c}$ is modulated using parameterized input and output gates.

the back propagation algorithm, the gradient gets smaller and smaller, and by the time we get back to the beginning of the sentence, we have a gradient so small that it effectively has no ability to have a significant effect on the parameters that need to be updated. The reason why this effect happens is because unless $\frac{d\boldsymbol{h}_{t-1}}{d\boldsymbol{h}_t}$ is exactly one, it will tend to either diminish or amplify the gradient $\frac{d\ell}{d\boldsymbol{h}_t}$, and when this diminishment or amplification is done repeatedly, it will have an exponential effect on the gradient of the loss.[22]

One method to solve this problem, in the case of diminishing gradients, is the use of a neural network architecture that is specifically designed to ensure that the derivative of the recurrent function is exactly one. A neural network architecture designed for this very purpose, which has enjoyed quite a bit of success and popularity in a wide variety of sequential processing tasks, is the **long short-term memory** (LSTM; [7]) neural network architecture. The most fundamental idea behind the LSTM is that in addition to the standard hidden state $\boldsymbol{h}$ used by most neural networks, it also has a **memory cell** $\boldsymbol{c}$, for which the gradient $\frac{d\boldsymbol{c}_t}{d\boldsymbol{c}_{t-1}}$ is exactly one. Because this gradient is exactly one, information stored in the memory cell does not suffer from vanishing gradients, and thus LSTMs can capture long-distance dependencies more effectively than standard recurrent neural networks.

So how do LSTMs do this? To understand this, let's take a look at the LSTM architecture

[22] This is particularly detrimental in the case where we receive a loss only once at the end of the sentence, like the example above. One real-life example of such a scenario is document classification, and because of this, RNNs have been less successful in this task than other methods such as convolutional neural networks, which do not suffer from the vanishing gradient problem [9, 10]. It has been shown that pre-training an RNN as a language model before attempting to perform classification can help alleviate this problem to some extent [2].

in detail, as shown in Figure 17 and the following equations:

$$\boldsymbol{u}_t = \tanh(W_{xu}\boldsymbol{x}_t + W_{hu}h_{t-1} + \boldsymbol{b}_u) \quad (47)$$
$$\boldsymbol{i}_t = \sigma(W_{xi}\boldsymbol{x}_t + W_{hi}h_{t-1} + \boldsymbol{b}_i) \quad (48)$$
$$\boldsymbol{o}_t = \sigma(W_{xo}\boldsymbol{x}_t + W_{ho}h_{t-1} + \boldsymbol{b}_o) \quad (49)$$
$$\boldsymbol{c}_t = \boldsymbol{i}_t \odot \boldsymbol{u}_t + \boldsymbol{c}_{t-1} \quad (50)$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t). \quad (51)$$

Taking the equations one at a time: Equation 47 is the update, which is basically the same as the RNN update in Equation 44; it takes in the input and hidden state, performs an affine transform and runs it through the tanh non-linearity.

Equation 48 and Equation 49 are the **input gate** and **output gate** of the LSTM respectively. The function of gates, as indicated by their name, is to either allow information to pass through or block it from passing. Both of these gates perform an affine transform followed by the **sigmoid function**, also called the **logistic function**[23]

$$\sigma(x) = \frac{1}{1 - \exp(-x)}, \quad (52)$$

which squishes the input between 0 (which $\sigma(x)$ will approach as x becomes more negative) and 1 (which $\sigma(x)$ will approach as x becomes more positive). The output of the sigmoid is then used to perform a componentwise multiplication

$$\boldsymbol{z} = \boldsymbol{x} \odot \boldsymbol{y}$$
$$z_i = x_i * y_i$$

with the output of another function. This results in the "gating" effect: if the result of the sigmoid is close to one for a particular vector position, it will have little effect on the input (the gate is "open"), and if the result of the sigmoid is close to zero, it will block the input, setting the resulting value to zero (the gate is "closed").

Equation 50 is the most important equation in the LSTM, as it is the equation that implements the intuition that $\frac{d\boldsymbol{c}_t}{d\boldsymbol{c}_{t-1}}$ must be equal to one, which allows us to conquer the vanishing gradient problem. This equation sets $\boldsymbol{c}_t$ to be equal to the update $\boldsymbol{u}_t$ modulated by the input gate $\boldsymbol{i}_t$ plus the cell value for the previous time step $\boldsymbol{c}_{t-1}$. Because we are directly adding $\boldsymbol{c}_{t-1}$ to $\boldsymbol{c}_t$, if we consider only this part of Equation 50, we can easily confirm that the gradient will indeed be one.[24]

Finally, Equation 51 calculates the next hidden state of the LSTM. This is calculated by using a tanh function to scale the cell value between -1 and 1, then modulating the output using the output gate value $\boldsymbol{o}_t$. This will be the value actually used in any downstream calculation, such as the calculation of language model probabilities.

$$\boldsymbol{p}_t = \text{softmax}(W_{hs}\boldsymbol{h}_t + b_s). \quad (53)$$

[23] To be more accurate, the sigmoid function is actually any mathematical function having an s-shaped curve, so the tanh function is also a type of sigmoid function. The logistic function is also a slightly broader class of functions $f(x) = \frac{L}{1+\exp(-k(x-x_0))}$. However, in the machine learning literature, the "sigmoid" is usually used to refer to the particular variety in Equation 52.

[24] In actuality $\boldsymbol{i}_t \odot \boldsymbol{u}_t$ is also affected by $\boldsymbol{c}_{t-1}$, and thus $\frac{d\boldsymbol{c}_t}{d\boldsymbol{c}_{t-1}}$ is not exactly one, but the effect is relatively indirect. Especially for vector elements with $\boldsymbol{i}_t$ close to zero, the effect will be minimal.

6.4 Other RNN Variants

Because of the importance of recurrent neural networks in a number of applications, many variants of these networks exist. One modification to the standard LSTM that is used widely (in fact so widely that most people who refer to "LSTM" are now referring to this variant) is the addition of a **forget gate** [4]. The equations for the LSTM with a forget gate are shown below:

$$
\begin{aligned}
\boldsymbol{u}_t &= \tanh(W_{xu}\boldsymbol{x}_t + W_{hu}h_{t-1} + \boldsymbol{b}_u) \\
\boldsymbol{i}_t &= \sigma(W_{xi}\boldsymbol{x}_t + W_{hi}h_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{f}_t &= \sigma(W_{xf}\boldsymbol{x}_t + W_{hf}h_{t-1} + \boldsymbol{b}_f) \qquad (54) \\
\boldsymbol{o}_t &= \sigma(W_{xo}\boldsymbol{x}_t + W_{ho}h_{t-1} + \boldsymbol{b}_o) \\
\boldsymbol{c}_t &= \boldsymbol{i}_t \odot \boldsymbol{u}_t + \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} \qquad (55) \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh \boldsymbol{c}_t.
\end{aligned}
$$

Compared to the standard LSTM, there are two changes. First, in Equation 54, we add an additional gate, the forget gate. Second, in Equation 55, we use the gate to modulate the passing of the previous cell $\boldsymbol{c}_{t-1}$ to the current cell $\boldsymbol{c}_t$. This forget gate is useful in that it allows the cell to easily clear its memory when justified: for example, let's say that the model has remembered that it has seen a particular word strongly correlated with another word, such as "he" and "himself" or "she" and "herself" in the example above. In this case, we would probably like the model to remember "he" until it is used to predict "himself", then forget that information, as it is no longer relevant. Forget gates have the advantage of allowing this sort of fine-grained information flow control, but they also come with the risk that if $\boldsymbol{f}_t$ is set to zero all the time, the model will forget everything and lose its ability to handle long-distance dependencies. Thus, at the beginning of neural network training, it is common to initialize the bias $\boldsymbol{b}_f$ of the forget gate to be a somewhat large value (e.g. 1), which will make the neural net start training without using the forget gate, and only gradually start forgetting content after the net has been trained to some extent.

While the LSTM provides an effective solution to the vanishing gradient problem, it is also rather complicated (as many readers have undoubtedly been feeling). One simpler RNN variant that has nonetheless proven effective is the **gated recurrent unit** (GRU; [1]), expressed in the following equations:

$$\boldsymbol{r}_t = \sigma(W_{xr}\boldsymbol{x}_t + W_{hr}h_{t-1} + \boldsymbol{b}_r) \qquad (56)$$

$$\boldsymbol{z}_t = \sigma(W_{xz}\boldsymbol{x}_t + W_{hz}h_{t-1} + \boldsymbol{b}_z) \qquad (57)$$

$$\tilde{\boldsymbol{h}}_t = \tanh(W_{xh}\boldsymbol{x}_t + W_{hh}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \qquad (58)$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t)\boldsymbol{h}_{t-1} + \boldsymbol{z}_t\tilde{\boldsymbol{h}}_t. \qquad (59)$$

The most characteristic element of the GRU is Equation 59, which interpolates between a candidate for the updated hidden state $\tilde{\boldsymbol{h}}_t$ and the previous state $\tilde{\boldsymbol{h}}_{t-1}$. This interpolation is modulated by an **update gate** $\boldsymbol{z}_t$ (Equation 57), where if the update gate is close to one, the GRU will use the new candidate hidden value, and if the update is close to zero, it will use the previous value. The candidate hidden state is calculated by Equation 58, which is similar to a standard RNN update but includes an additional modulation of the hidden state

input by a **reset gate** $\boldsymbol{r}_t$ calculated in Equation 56. Compared to the LSTM, the GRU has slightly fewer parameters (it performs one less parameterized affine transform) and also does not have a separate concept of a "cell". Thus, GRUs have been used by some to conserve memory or computation time.

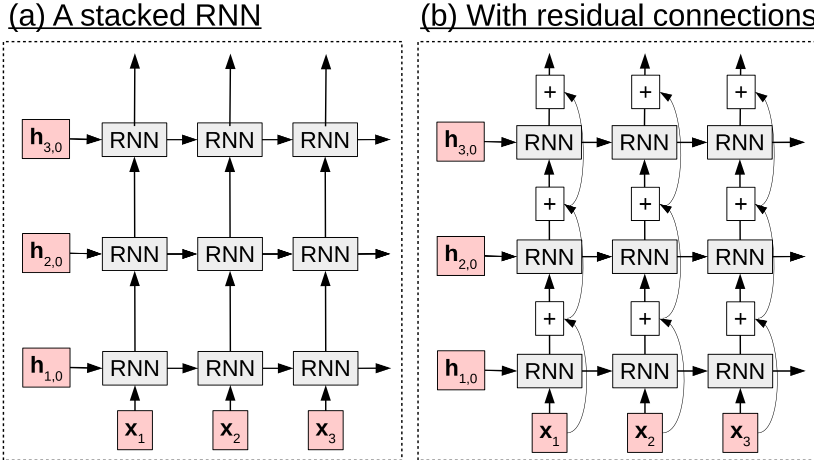

Figure 18: An example of (a) stacked RNNs and (b) stacked RNNs with residual connections.

One other important modification we can do to RNNs, LSTMs, GRUs, or really any other neural network layer is simple but powerful: stack multiple layers on top of each other (**stacked RNNs** Figure 18(a)). For example, in a 3-layer stacked RNN, the calculation at time step t would look as follows:

$$\begin{aligned}\boldsymbol{h}_{1,t} &= \mathrm{RNN}_1(\boldsymbol{x}_t, \boldsymbol{h}_{1,t-1}) \\ \boldsymbol{h}_{2,t} &= \mathrm{RNN}_2(\boldsymbol{h}_{1,t}, \boldsymbol{h}_{2,t-1}) \\ \boldsymbol{h}_{3,t} &= \mathrm{RNN}_3(\boldsymbol{h}_{2,t}, \boldsymbol{h}_{3,t-1}),\end{aligned} \tag{60}$$

where $\boldsymbol{h}_{n,t}$ is the hidden state for the nth layer at time step t, and $\mathrm{RNN}(\cdot)$ is an abbreviation for the RNN equation in Equation 44. Similarly, we could substitute this function for $\mathrm{LSTM}(\cdot)$, $\mathrm{GRU}(\cdot)$, or any other recurrence step. The reason why stacking multiple layers on top of each other is useful is for the same reason that non-linearities proved useful in the standard neural networks introduced in Section 5: they are able to progressively extract more abstract features of the current words or sentences. For example, [14] find evidence that in a two-layer stacked LSTM, the first layer tends to learn granular features of words such as part of speech tags, while the second layer learns more abstract features of the sentence such as voice or tense.

While stacking RNNs has potential benefits, it also has the disadvantage that it suffers from the vanishing gradient problem in the vertical direction, just as the standard RNN did in the horizontal direction. That is to say, the gradient will be back-propagated from the layer close to the output (RNN_3) to the layer close to the input (RNN_1), and the gradient may vanish in the process, causing the earlier layers of the network to be under-trained. A simple solution to this problem, analagous to what the LSTM does for vanishing gradients over time,

is **residual networks** (Figure 18(b)) [6]. The idea behind these networks is simply to add the output of the previous layer directly to the result of the next layer as follows:

$$\begin{aligned}
\boldsymbol{h}_{1,t} &= \text{RNN}_1(\boldsymbol{x}_t, \boldsymbol{h}_{1,t-1}) + \boldsymbol{x}_t \\
\boldsymbol{h}_{2,t} &= \text{RNN}_2(\boldsymbol{h}_{1,t}, \boldsymbol{h}_{2,t-1}) + \boldsymbol{h}_{1,t} \\
\boldsymbol{h}_{3,t} &= \text{RNN}_3(\boldsymbol{h}_{2,t}, \boldsymbol{h}_{3,t-1}) + \boldsymbol{h}_{2,t}.
\end{aligned} \tag{61}$$

As a result, like the LSTM, there is no vanishing of gradients due to passing through the RNN(·) function, and even very deep networks can be learned effectively.

6.5 Online, Batch, and Minibatch Training

As the observant reader may have noticed, the previous sections have gradually introduced more and more complicated models; we started with a simple linear model, added a hidden layer, added recurrence, added LSTM, and added more layers of LSTMs. While these more expressive models have the ability to model with higher accuracy, they also come with a cost: largely expanded parameter space (causing more potential for overfitting) and more complicated operations (causing much greater potential computational cost). This section describes an effective technique to improve the stability and computational efficiency of training these more complicated networks, **minibatching**.

Up until this point, we have used the stochastic gradient descent learning algorithm introduced in Section 4.2 that performs updates according to the following iterative process. This type of learning, which performs updates a single example at a time is called **online learning**.

Algorithm 1 A fully online training algorithm

```
1: procedure Online
2:     for several epochs of training do
3:         for each training example in the data do
4:             Calculate gradients of the loss
5:             Update the parameters according to this gradient
6:         end for
7:     end for
8: end procedure
```

In contrast, we can also think of a **batch learning** algorithm, which treats the entire data set as a single unit, calculates the gradients for this unit, then only performs update after making a full pass through the data.

These two update strategies have tradeoffs.

- Online training algorithms usually find a relatively good solution more quickly, as they don't need to make a full pass through the data before performing an update.
- However, at the end of training batch, learning algorithms can be more stable, as they are not overly influenced by the most recently seen training examples.

Algorithm 2 A batch learning algorithm

```
1: procedure Batch
2:     for several epochs of training do
3:         for each training example in the data do
4:             Calculate and accumulate gradients of the loss
5:         end for
6:         Update the parameters according to the accumulated gradient
7:     end for
8: end procedure
```

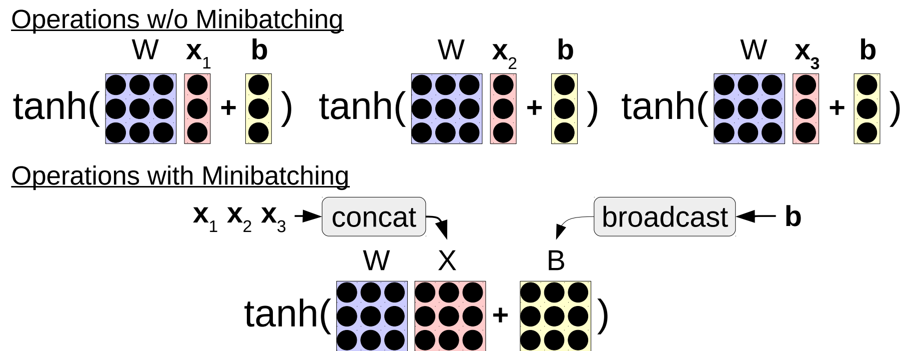

Figure 19: An example of combining multiple operations together when minibatching.

- Batch training algorithms are also more prone to falling into local optima; the randomness in online training algorithms often allows them to bounce out of local minima and find a better global solution.

Minibatching is a happy medium between these two strategies. Basically, minibatched training is similar to online training, but instead of processing a single training example at a time, we calculate the gradient for n training examples at a time. In the extreme case of $n = 1$, this is equivalent to standard online training, and in the other extreme where n equals the size of the corpus, this is equivalent to fully batched training. In the case of training language models, it is common to choose minibatches of $n = 1$ to $n = 128$ sentences to process at a single time. As we increase the number of training examples, each parameter update becomes more informative and stable, but the amount of time to perform one update increases, so it is common to choose an n that allows for a good balance between the two.

One other major advantage of minibatching is that by using a few tricks, it is actually possible to make the simultaneous processing of n training examples significantly faster than processing n different examples separately. Specifically, by taking multiple training examples and grouping similar operations together to be processed simultaneously, we can realize large gains in computational efficiency due to the fact that modern hardware (particularly GPUs, but also CPUs) have very efficient vector processing instructions that can be exploited with appropriately structured inputs. As shown in Figure 19, common examples of this in neural networks include grouping together matrix-vector multiplies from multiple examples into a single matrix-matrix multiply or performing an element-wise operation (such as tanh) over

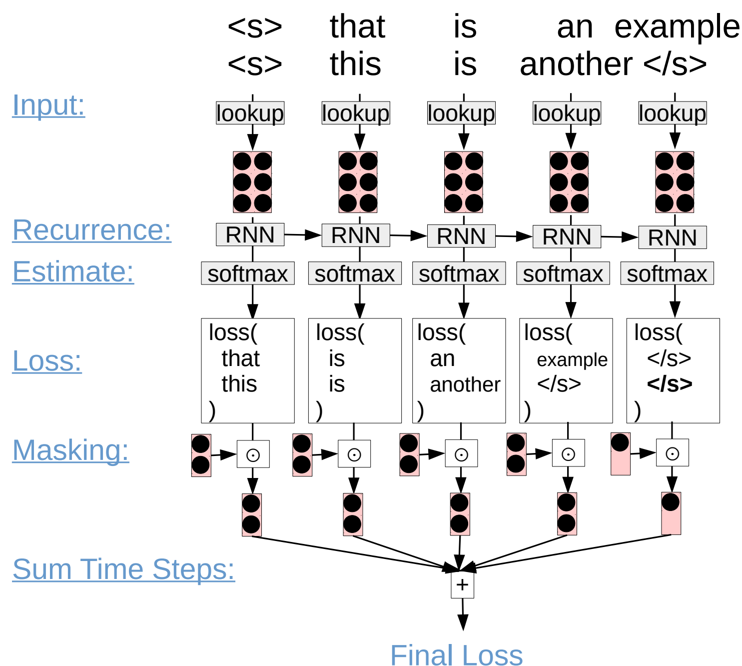

Figure 20: An example of minibatching in an RNN language model.

multiple vectors at the same time as opposed to processing single vectors individually. Luckily, in DyNet, the library we are using, this is relatively easy to do, as much of the machinery for each elementary operation is handled automatically. We'll give an example of the changes that we need to make when implementing an RNN language model below.

The basic idea in the batched RNN language model (Figure 20) is that instead of processing a single sentence, we process multiple sentences at the same time. So, instead of looking up a single word embedding, we look up multiple word embeddings (in DyNet, this is done by replacing the `lookup` function with the `lookup_batch` function, where we pass in an array of word IDs instead of a single word ID). We then run these batched word embeddings through the RNN and softmax as normal, resulting in two separate probability distributions over words in the first and second sentences. We then calculate the loss for each word (again in DyNet, replacing the `pickneglogsoftmax` function with the `pickneglogsoftmax_batch` function and pass word IDs). We then sum together the losses and use this as the loss for our entire sentence.

One sticking point, however, is that we may need to create batches with sentences of different sizes, also shown in the figure. In this case, it is common to perform **sentence padding** and **masking** to make sure that sentences of different lengths are treated properly. Padding works by simply adding the "end-of-sentence" symbol to the shorter sentences until they are of the same length as the longest sentence in the batch. Masking works by multiplying all loss functions calculated over these padded symbols by zero, ensuring that the losses for sentence end symbols don't get counted twice for the shorter sentences.

By taking these two measures, it becomes possible to process sentences of different lengths, but there is still a problem: if we perform lots of padding on sentences of vastly different

lengths, we'll end up wasting a lot of computation on these padded symbols. To fix this problem, it is also common to sort the sentences in the corpus by length before creating mini-batches to ensure that sentences in the same mini-batch are approximately the same size.

6.6 Further Reading

Because of the prevalence of RNNs in a number of tasks both on natural language and other data, there is significant interest in extensions to them. The following lists just a few other research topics that people are handling:

What can recurrent neural networks learn?: RNNs are surprisingly powerful tools for language, and thus many people have been interested in what exactly is going on inside them. [8] demonstrate ways to visualize the internal states of LSTM networks, and find that some nodes are in charge of keeping track of length of sentences, whether a parenthesis has been opened, and other salietn features of sentences. [11] show ways to analyze and visualize which parts of the input are contributing to particular decisions made by an RNN-based model, by back-propagating information through the network.

Other RNN architectures: There are also quite a few other recurrent network architectures. [5] perform an interesting study where they ablate various parts of the LSTM and attempt to find the best architecture for particular tasks. [15] take it a step further, explicitly training the model to find the best neural network architecture.

6.7 Exercise

In the exercise for this chapter, we will construct a recurrent neural network language model using LSTMs.

Writing the program will entail:

- Writing a function such as `lstm_step` or `gru_step` that takes the input of the previous time step and updates it according to the appropriate equations. For reference, in DyNet, the componentwise multiply and sigmoid functions are `dy.cmult` and `dy.logistic` respectively.
- Adding this function to the previous neural network language model and measuring the effect on the held-out set.
- Ideally, implement mini-batch training by using the `lookup_batch` and `pickneglogsoftmax_batch` functionality implemented in DyNet.

Language modeling accuracy should be measured in the same way as previous exercises and compared with the previous models.

Potential improvements to the model include: Implementing mini-batching and seeing speed/stability improvements. Implementing dropout or regularization and measuring the accuracy improvements.

References

[1] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[2] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3079–3087, 2015.

[3] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[4] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[5] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 2016.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[8] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[9] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, 2014.

[10] Tao Lei, Regina Barzilay, and Tommi Jaakkola. Molding cnns for text: non-linear, non-consecutive convolutions. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1565–1575, 2015.

[11] Jiwei Li, Xinlei Chen, Eduard Hovy, and Dan Jurafsky. Visualizing and understanding neural models in nlp. *arXiv preprint arXiv:1506.01066*, 2015.

[12] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 1045–1048, 2010.

[13] Philip Resnik. Selectional preference and sense disambiguation. In *Proceedings of the ACL SIGLEX Workshop on Tagging Text with Lexical Semantics: Why, What, and How*, pages 52–57. Washington, DC, 1997.

[14] Xing Shi, Inkit Padhi, and Kevin Knight. Does string-based neural mt learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1526–1534, 2016.

[15] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.