Extensions to Recurrent Neural Network

Eilif Solberg

September 20, 2018

Contents

1 Introduction

We will in this note look at some extensions to the RNN model. We will see how we can make more complex models by combining several RNNs (Section 2) and how we can extend the memory capabilities of RNNs (Section

3). Lastly we will brifely look at Recursive Neural Networks, which do not adhere to the strict serial processing model of RNNs but allows for more general structured procecessing of its input.

2 Composing RNNs

Like most neural network models, recurrent neural networks can be used as building blocks in larger models. We shall here look at some models that are build out of several RNNs. Although stacked RNNs are also a composition of RNNs, they may unlike the examples presented here, naturally be viewed as RNNs themselves (see problem of those lecture notes).

2.1 Bidirectional RNNs

So far we have only been able to make decisions about our output Y^t based on the input values up to time t, i.e. $X^1, \ldots, X^t$. In a lot of scenarios where time is involved this is the *only* way, we can't possibly make decisions based on information that is not yet available to us. However no one has told us yet that we can't *delay* our output. With this we mean that we at time t may choose to take the action of *not* taking an action. We might expect the future to shed more light on the situation, and would like to wait with our response to incorporate this information. Speech-to-text is an example where this could come in handy. The speech may contain ambiguities, and we would like to gather more context to narrow down our choice. Taking this to its extreme we could actually delay our output until we have finished processing the *entire* input sequence. Thus we would basically split our tasks into two phases, an *information gathering phase* followed by a *response phase*. In this way we are able to condition all our outputs on the *entire* input sequence.

An alternative approach was proposed in [1]. Imagine that the entire sequence is presented to us at once, or that we have a buffering mechanism that allows us to store the incoming sequence. The idea they then proposed was to have *two* RNNs, one processing the sequence from start to end, and an additional RNN processing the sequence from end to start. After both RNNs have finished processing the sequence we may at each time step take the output to be the concatenated output of the RNNs. We may then e.g. feed this output as input to a new *bidirectional layer* in a stacked architecture. In this way we can extract a features for each element in the sequence, that also depends on the rest of the sequence. Depending on the application we may at the end apply an output function which now can make decision based on both information from the past and that of the "future".

2.2 Encoder-decoder framework

Sequence-to-sequence problems comes in many forms. In some cases, like e.g. speech-to-text there is a very tight coupling between the sequences. Context can be quite useful, but a lot of the information is still very local. Only considering a small time window at the time it would still be possible to get meaningul, though suboptimal, results. Machine translation is another sequence-to-sequence problem. Here the coupling is much more loose. Looking at local time-windows only and translating them separately would give very odd results. Taking it to an extreme, word-for-word translations from one language to another usually results in absurd, but often funny, sentences that are hard to understand. This will of course depend on the particular languages at hand though. For machine translation then there is little connection between the order of the first sequence and the order of the second. The meaning of the first and second sequence are supposed to *represent the same information*, but there is not usually a direct relation between the third element in the first sequence and the third element in the second sequence. Thus we can look at machine translation as two steps. First we *encode* the meaning of the source language into an intermediate representation, then we *decode* this into a sentence which represents this meaning in the target language.

It is possible to use a single RNN to do both tasks. First it encodes the sentence producing no output, then it starts the decoding process for which no new input is presented, but the output is fed back in. This is illustrated in Figure [fig:create-figure]. The encoder and decoder though have two quite different tasks, and it may be more appropriate to have a separate RNN for the encoder and decoder. This also has the benefit of a very flexible system where we can potentially use the same encoder when translating from norwegian to english, that we would use to translate norwegian to german. Similarily we could share decoder when we want to translate different source languages to the same target language. This of course depends on training the different encoders and decoders jointly, so that they share the same intermediate representation. Assume we have N languages and want to be able to translate between any one of them to any other. Having a separate RNN between each of the languages we would need $N(N-1) = O(N^2)$ RNNs, while we only need $2N = O(N)$ RNNs in the encoder-decoder framework with shared encoders and decoders.

There are two common variants of the encoder-decoder framework. One approach is two use the first RNN to create a fixed-sized vector to use as input to the second RNN. This has the potential disadvantage that the rep-

resentation of the meaning of the sentence is of fixed length and could potentially be a bottleneck, especially for long sentences. The second approach avoids this problem by producing a variable-length representation, typically of the same length as the input-sequence. This can e.g. be taken to be the sequence of states of the first RNN. The second RNN then typically uses some attention mechanism to attend to different parts of the intermediate representation during the generation of the sentence in the target language. (Note that first processing the whole sequence, as is how RNNs commonly work, and then create the appropriate representation in the target language would have the same issue as what we were trying to get around.)

3 Memory extensions

– want to change the order of next to...

3.1 External memory

We can think of the state vector of the neural network as the *memory* of the model. When describing the LSTM model we introduced gates to control both read and write access to the memory. There is however a huge limitation of this memory model, in that it is inherently tied to the computational model.

We will illustrate the limitation of this through an example. Assume we have an RNN capable of the task of reversing input sequences of finte length. The RNN needs to store all the elements in the input sequences before it can start outputting the sequence in reverse[1]. Any particular RNN has finite memory, and will thus not be able to reverse all sequences of arbitrary length.

It would be nice though if we could extend the RNN to handle longer sequences if more more memory were available to us. Within the basic RNN framework however this is not easily accomplished. We can't simply just add neurons to the state vector; for the neurons to be able to do anything useful we also need to set all the parameters associated with a state neuron. The problem is thus that if we want to introduce more memory, we also have to specify how to do computations with them. Is it possible to *decouple* memory from the computational model?

[1] The RNN could try to *compress* the input sequences. However it will not be able to compress *all* sequcences while still being able to reconstruct them without errors (see e.g. `https://en.wikipedia.org/wiki/Lossless_compression#Limitations`). As long as we don't restrict ourselves to a finite number of possible sequences it will not be possible to give an upper bound for the length of the encoding.

We will have to look no further than the digital computer for inspiration. On a computer we can install more memory without the need for altering or upgrading our CPU[2]. This separation is accomplished by making the memory cells simpler, not having specialized logic associated with them. While RNN memory cells may specialize themselves to store particular type of information, the memory cells on the computer is treated as a just an array of bytes. It is up to the *program* to decide where and what to store in the memory cells.

In this spirit we will now extend our RNN model with *external memory*. This memory will not be an integral part of our computational model, but a separate module which we may interact with through some interface[3]. We shall assume that the connection to the memory is of *limited bandwidth*. This means that, unlike for the RNN state cells, we will not be able to read from, or write to, the whole memory in one operation. Not to stray away from our main thread, a discussion of the resonableness of this model can be found later in Section 3.1.4.

3.1.1 Addressing

Given that we move on with a bandwidth-limited connection, the question to ask is then: how do we choose *what* part of memory to read from our write to? And even more fundamentally, how do we even refer to part a specific part of memory? We will use the term *addressing* to refer to this process.

In a computer program we have to indicate which parts of the memory we want to operate on by specifying the *location* of the memory cell. It is up to the program to keep track of where information is stored, though it could use the memory to store this information as well, creating a network of links. We call this *location-based* addresing. We shall interchangeably use the term *direct* addressing, as we are directly specifying the location of where to get the data. Another property that distinguishes one memory cell from another is it's *content*. With pure *content-based* addressing we don't care about the location of the memory cell, only what it contains. With content-based addressing the information in each memory cell needs to be *self-contained*. Assume that we would like to retrieve from our memory the

[2]Usually a particular CPU can only handle up to a certain amount of memory. This is not a fundamental constraint, however, just a consequence of the fact that adding more circuits on a chip adds to the complexity and cost of the chip.

[3]Taking this a step further we can imagine attaching all sorts of external devices to our program, just as for a computer. We refer the interested reader to [2] for further discussions and explorations of this idea.

year the french revolution started. Having a memory cell storing the number "1789" would in most cases not be very helpful. What if we had a different memory cell storing the number "1799", how would we know which, if any, of these years were the correct one? Thus we see that with content-based addressing we will need to store contextual information so that we are able to interpret the information in a way that reveals its meaning: e.g. "The french revolution started in 1789" and "The french revolution ended in 1799". Notice that this is not needed for location-based addressing. With location-based addressing we could in our program (our somewhere else in memory) keep the information that a particular memory cell refers to the year the french revolution started. Retrieving this information could then simply be done by looking up the information in that memory cell. We discuss a few other differences between location-based and content-based addressing in Section 3.1.3.

Content-based addressing. Let's look into content-based addressing in more detail. The goal is to design a mechanism so that we can retrieve the content of a memory cell which contains some particular information of interest. At first it might seem like this kind of addressing is logically flawed. If we don't know the desired content, how can we know which memory cell to address? On the other hand, if we *knew* exactly what content we wanted to retrieve, there would be no need to retrieve it.

We will however see that we can resolve this apparent contradiction. Instead of trying to address a memory cell by its exact content, we shall introduce the concepts *query*, *key* and *matching function*. The *query* shall pose a question we would like answered. A natural language query could e.g. be "When did the french revolution start?". In most cases, however, we will encode the query into a vector of real-valued numbers.

For each memory cell we will associate with it a *key*, which we for now may think of as a concise description of the memory cell content. We will allow for the special case where the key function is just the identity function. Though not a requirement, the key function will usually map the memory cells into the same space as the key function is in (i.e. a vector space of the same dimension). Based on the query and a particular key, we now need to figure out how relevant the memory cell is in answering the query. This is implemented with a *matching function* g, such that for a given a query q and a key k, g returns a score indicating the match between the query and key. In the simplest case g can be taken to be the inner product function (that is one reason we often like the query and key to be of same dimensions). After we have applied the matching function to all query, key pairs we may

use the obtained scores to either return a particular memory cell, e.g. the one with the highest matching score, or a weighted average over the memory cells, where the weigth for a memory cell is based on its associated score.

We will now formulate the above in equations. Let M be our memory and M_j denote memory cell number j, and let J denote the total number of memory cells. Let K be a function that for a given memory cell returns a key for that cell. For a given query we then calculate the matching scores memory cell M_j by

$$\alpha_j = g(q, K(M_j)) \tag{1}$$

After this has been done for all memory cells we may then apply e.g. a softmax function to the scores to get a probability distribution over the memory cells. If $p_1, \ldots, p_J$ is the obtained probabilities, let π denote the probability distribution they induce over $\{M_1, \ldots, M_J\}$, i.e. if $X \sim \pi$ then $P(X = M_j) = p_j$. We shall now look at two ways we can use this distribution to get obtain our query result $v(q, M)$. With *hard addressing* the returned value v is *sampled* from the distribution

$$v(q, M) \sim \pi \tag{2}$$

With *soft addressing* we instead take the *expectation* over the distribution. This corresponds to a weighted average over the memory cells, i.e.

$$v(q, M) = \sum_{j=1}^{J} p_j M_j \tag{3}$$

It might seem strange to take a weighted average over the memory cells, and is unclear how the network should be able to decode such a signal. One benefit of the soft addressing however is that our memory system then becomes fully differentiable (as long as K and g are differentiable, and the queries are produced in a differentiable manner). This allows for end-to-end training of the system using stochastic gradient descent. With hard attention we will have to borrow optimization techniques from e.g. reinforcement learning, which may make training more challenging.

The quality of our addressing system will of course depend on all parts of our system. How good we are at asking questions q, how precise our function K for generating descriptions is and the ability of g to judge the relevance of a key in answering a query. In general we shall try to *learn* this system. In most cases we don't really have targets for our memory model, i.e. we know

what the most relevant information is[4]. In many cases we may still be able to train it with indirect supervision from the error signals of a *downstream task*.

We shall use the term *indirect* addressing interchangeably with *content-based* addressing. This comes from the fact that our addressing mechanism doesn't really specify where to retrieve the data from, it does it only indirectly by saying that we want to "get the data which best matches the query".

3.1.2 Example: External memory with content-based addressing

So far we have only discussed external memory and content-based addressing in general terms. We will now present an example of one possible way of extending an RNN with external memory using content-based addressing. We will present a very simplified version of the model introduced in [4] which uses a much more sophisticated hybrid content-location based addressing scheme. There are two operations which may be performed with respect to the memory, a *read* operation to extract information from memory and a *write* operation to alter the memory content.

Let's start with the read mechanism. For now we assume a single read operation per time step. We first define a query for the the read operation as

$$q^t = Q^{(r)}(s^t) \tag{4}$$

For each memory cell M_i^{t-1} we calculate a key

$$k_j^t = K^{(r)}(M_j^{t-1}) \tag{5}$$

and then we calculate the scores for each memory cell by

$$\alpha_j^t = g(q^t, k_j^t) \tag{6}$$

To be concrete, assume that the state-vector and memory-cells to be vectors of dimension m and n resepectively. We might take $Q^{(r)}$ to be a $d \times m$ matrix, $K^{(r)}$ and $d \times n$ matrix and g to be the inner-product function. The rest follows as described in the introduction for general content-based

[4]See [3] for an exception. Here they train a question answering system where they store unprocessed sentences in the memory, and during training they give the model supversion on which sentences are most relevant

addressing. We apply the softmax function over the the scores to obtain probabilities $p_1, \ldots, p_J$. If we use hard attention we draw a memory cell from the distribution implied by the probabilities, with soft attention we acquire a weighted average. In either case we obtain a vector r^t, which is the result of our query.

The addressing mechanism for the write operation can be taken to be the same as the one for the read operation, except that we may use different query and key functions $Q^{(w)}$ and $K^{(w)}$. For writing there is a couple of additional considerations as well. For one thing, we can't just decide *where* to write something, we also need to decide *what* to write. We define a write function W to be a function of the current state.

$$w^t = W(s^t) \tag{7}$$

Given w^t and given a distribution over memory cells defined by $p_1^w, \ldots, p_J^w$, there are several ways we could proceed to update the memory M. Let's first assume we are using hard attention, and have drawn a memory cell M_j. One possible update rule could simply be to *overwrite* the content of the memory cell, i.e.

$$M_j^t = w^t \tag{8}$$

Another possibility would be to make *updates* instead by adding the vector to the memory cell

$$M_j^t = M_j^{t-1} + w^t \tag{9}$$

With soft addressing the same choices apply. If we decide upon taking the *expected update* the choices then translates to the update rule

$$M_j^t = (1 - p_j^w)M_j^{t-1} + p_j w^t \tag{10}$$

or

$$M_j^t = M_j^{t-1} + p_j^w w^t \tag{11}$$

where this update rule is applied to **all** memory cells $M_1, \ldots M_J$.

We have now seen how we can read from and write to memory in such an RNN architecture. We haven't seen how to use the retrieved memory to something useful yet however! One way to take advantage of the memory is to include the read vector into our update equation, i.e.

$$s^t = h(x^t, s^{t-1}, y^{t-1}, r^{t-1}) \tag{12}$$

In this way we can write things we believe to be useful to memory, and then tap into this knowledge base later by making queries to it. We may also use the retrieved vector from memory as extra input to the output function

$$y^t = f(s^t, r^t) \tag{13}$$

Having only one read and write operation per time step might be a severe bottleneck for our models, and limit the usability of our external memory[5]. We will now briefly discuss how we can overcome this limitation. Extending the model to have several *read heads* is straightforward. Instead of having single query and key functions $Q^{(r)}$ and $K^{(r)}$ (see equations (4) and (5)), we define N query, key function pairs $(Q_1^{(r)}, K_1^{(r)})$, ...,$(Q_N^{(r)}, K_N^{(r)})$. Each pair give rise to a read vector r_i^t, we concatenate all of them into a vector r^t. We may use the same matching function for all pairs, though in theory one could also use several different ones. To extend to several *write heads* is not quite as straightforward, as we may have conflicts when different heads try to update the same memory cells. One possible solution to this is to average the updates that each write head makes. Another solution is presented in [4].

Although appealing, we would like to point out some limitations to the simple external memory module presented here. Even though we can scale to arbitrarily large memory sizes without having to retrain the model, we still have to specify the size for the memory pool. We would have preferred if the model were able to keep track of what memory is used, dynamically expand the memory when needed and free up memory that is no longer used. Another undesirable property is that for each query we also try to match it against *every* memory slot, which makes the memory lookup scale linearly with the memory size. This might be fine if our memory is not too large, but may become an issue as we try to scale up.

3.1.3 Location vs content-based addressing

Let's look at some special cases to get some further intuition about the differences between content and location-based addressing:

- With location-based addressing we could in principle have memory cells that stores a single bit, while this would be meaningless for content-based addressing.

[5]Of course we could increase the *size* of each memory cell. This is not very flexible, however, as we might want to retrieve pieces of information stored in different places.

- With content-based addressing, having two memory cells with the same content, the second memory cell does not add to the total information content. E.g. storing the information "The french revolution started in 1789" a second time does not provide us with new knowledge. In a location-based addressing system, having "1789" stored in two different cells can provide us with different information if the context surrounding them are different. The first cell could refer to the start of the french revolution, while the second referred to the year George Washington was elected the first President of the United States.

- Location-based addressing only makes sense in an iterative (serial) setting. We need to already have acquired knowledge of where different kinds of information are stored from previous iterations. With content-based addressing this is not necessary. We can blindly query information.

3.1.4 Limited-bandwidth assumption

Computers have a limited bandwidth between the CPU and memory. This often imposes a bottleneck on programs as they are not able to move data to the CPU for processing as fast as they would like. This bottleneck is of course something hardware manufacturers constantly try to reduce, but so far they have not been able to keep up with the increase processing power of the CPU. For our logical model, why should we voluntarily impose such a bottleneck?

Let's take a moment to reflect upon our own usage of long-term memory. What percentage of all the knowledge you have do you use at any moment? Even over the course of a day or week the estimate should probably be only a tiny fraction. That does not mean that most of what we know is not useful (though probably some things are!), but that most situations only calls for a tiny fraction of it. A similar, and probably even more convincing, argument applies to the writing to, or updating of, memory. We certainly learn new things and sometimes finds that some of the things we thought we knew were wrong. Most of our memories and knowledge are fairly stable however.

Note that the analogy to human long-term memory is far from perfect. Most machine-learning systems today are specialized to accomplish very narrow tasks. They only get limited input data over a short time period, and the designer of the system has probably carefully chosen to only feed the system with input that are at least somewhat relevant for the task.

Instead of thinking of it from the bottleneck perspective, we may simply

view it as a convenient way to get fixed size results back from our query. If the query where not completely satisfactory we could perform another query.

3.2 Attending to previous states

We will here discuss another approach for improving the memory capacity of RNNs. The standard RNN model assumes a strong Markov property, i.e. that given the state s^t we have that all future outputs are conditionally independent of $x^1, \ldots, x^t$, and thus also of $s^1, \ldots, s^{t-1}$. In other words, the model assumes that we are able to capture all relevant information of the past into the current state. This is a very strong assumption and may be hard to satisfy in practice, especially for long sequences. If we took ourselves the liberty to peek into previous states $s^1, \ldots, s^{t-1}$, we could take some pressure off the memory requirements of the state vectors and scale better with sequence length.

How do we take advantage of the previous states in practice? The previous states constitutes a sequence, so given what we have learned it seems natural to use an RNN to process this information! Introducing an RNN, with an initial state of 0 and processing $s^1, \ldots s^{t-1}$ sequentially however contains the very same problem which we are trying to get around (why?)! We want to get information *relevant to our current situation* and thus need to make the processing of the past states dependent on our current state. In the RNN setting we could do this by using s^t as the initial hidden state, or perhaps having s^t as an extra input at each time step. We will however not pursue this idea further here, we only note that the number of serial steps required then scales as T^2. Instead we will look into another approach of processing the past information in a way that depends on our current state.

[5] proposed an *attention* mechanism over previous states. At each state update we take all the previous states as input, i.e. we have an update equation of the form

$$s^t = h(x^t, (s^1, \ldots, s^{t-1}), y^{t-1}) \tag{14}$$

The attention mechanism however reduces the sequence $s^1, \ldots, s^{t-1}$ into a single vector through a convex combination, where we use the current state to influence the weighting of the past states. We let $\tilde{s}^{t-1}$ denote the convex combination of $s^1, \ldots, s^{t-1}$. With the language of mathematics we can write this as

$$p_i^t = f(x^t, (s^1, \ldots, s^{t-1}), y^{t-1}) \tag{15}$$

$$\tilde{s}^{t-1} = \sum_{i=1}^{t-1} p_i^t s^i \tag{16}$$

$$\sum_{i=1}^{t-1} p_i^t = 1 \tag{17}$$

In the particular implementation of [5] they determined the weights by

$$\alpha_i^t = \langle v, \tanh(U_\alpha x^t + V_\alpha s^{t-1} + W_\alpha s^i) \rangle \tag{18}$$

$$p^t = \text{softmax}(\alpha^t) \tag{19}$$

for $i = 1, \ldots, t-1$, where $\langle \cdot, \cdot \rangle$ denotes the inner product, $\alpha^t = (\alpha_i^t, \ldots, \alpha_{t-1}^t)$, $p^t = (p_i^t, \ldots, p_{t-1}^t)$ and v is a learnable vector. This is however only one of many possibilities. In fact the attention mechanism above is not really able to fully take into consideration interactions between the various inputs, which in many cases will be useful to extract the most relevant information from previous states. Although equation (18) can be viewed as an example of the content-based addressing of the previous section (the form is so general that basically everything can!) we might consider other attention mechanisms of the form

$$\alpha_i^t = f(Q(s^{t-1}, x^t), K(s^i)) \tag{20}$$

where an example could be

$$\alpha_i^t = \langle U_\alpha x^t + V_\alpha s^{t-1}, W_\alpha s^i \rangle \tag{21}$$

After we have applied the attention mechanism we can now proceed with a 'normal' RNN update function h that uses the alternative state $\tilde{s}^{t-1}$

$$s^t = h(x^t, \tilde{s}^{t-1}, y^{t-1}) \tag{22}$$

As a final remark we add that the added memory flexibility comes at a cost; our processing time is no longer constant for each time step, but increases over time. Thus our total processing time of the sequence does not scale linearly with sequence length. An alternative, increasing memory capacity by increasing the dimension of the state vector so that it is large enough to handle the most demanding cases, has the disadvantage of being unneccasarily large for less demanding situations.

4 Recursive neural networks

In the introduction to this chapter we motivated the RNN model by arguing that the *time* dimension naturally induces a sequential processing model. Both the stimuli from the outside world, and our actions upon it, are spread out in the linear order of time.

Since then, however, we have been happy to apply RNN models to all sorts of data that can be viewed as a sequence, were the motivating reasons for the RNN may no longer apply. It's like the cliche tells us, "If you have a hammer, everything looks like a nail". That is not to say we shall not apply RNNs in situations where we feel like we can't justify the processing model, but perhaps it shouldn't be the *only* model we try.

Assume our input is a sequence of real numbers X^1, ..., X^T, and we would like to learn the *min* function[6], i.e. $f(X^1, \ldots, X^T) = \min(X^1, \ldots, X^T)$. We could imagine an RNN solving this problem by storing the minimum value encountered so far, and for each iteration compare this with the new input value, updating the state if a lower value is encountered. Instead of this serial approach we could also split the sequence into pairs of two, applying a binary *min* function to each of these pairs. We may then put the resulting minimum values into a new sequence, which is of half the length. The same procedure may be applied again to this new sequence. Though both functions in this case solves the problem, the second may in many cases be preferred as it is more parallelizeable. The first approach requires $T - 1$ sequental steps, while the second can be solved with $\log_2 T$ sequential steps[7].

The example above illustrates a more general problem. Assume we have a sequence $X^1, X^2, \ldots, X^T$ as input. We would like to learn a function f that maps the sequence to a fized size vector[8] $Y \in R^d$, i.e. $f(X^1, X^2, \ldots, X^T) \in R^d$ for all sequences. In the case $X^1, X^2, \ldots, X^T$ is a sentence, where X^i is word number i, you may think of this as representing an *encoder* which task is to interpret the sentence and create a semantically meaningful representation of it. This representation can be used as input to e.g. a some simple classifier, or a decoder in the case of translation.

A recurrent neural network solves this prolem by processing the sequence

[6] This is for illustrative purposes only, of course we know how to implement a *min* function. Also, the algorithm does not really know that it is supposed to learn the min function, it only gets a sequence of numbers with (what we know happens to be) the min value of the sequence as input and needs to generalize from there.

[7] Note however that they both need a total amount of *computational operations* that is *linear* in the sequence length.

[8] This is not a requirement, but will suffice for our discussions here

sequentially from start to end. The RNN keeps the representation of the part of the sentence processed so far in an *internal* state vector s^t, which it updates for each word it processes, using the *same* update function at each time step. When we have finished processing the whole secuence, s^T is our desired vector. We will now try to generalize to allow for other *structured* processing models, beyond the strictly sequntial one. If our input was a sentence we could perhaps analyze different parts of the sentence separately, and then later merge the information. Each part of the sentence could of course be split again into even smaller pieces, analyzed separately and then merged. When we move away from the strictly sequential processing scheme, the concept of *state* will make less sense. Instead of having an *update function* that *consumes* one of the input data to update its internal state, we shall now introduce the *composite function*. The composite function takes two or more data points as input, and *combine* them into a vector of *the same form* as the input vectors. The output of the composite function replaces the input data in the data structure, and can be used as input in the next iteration. We will call models of this form *recursive neural networks*. We see that the computational structure will be that of a *tree*, though with a few modifications we could also have allowed for arbitrary *acyclic graph* structures.

Although the RNN model discussed earlier had a quite a few choices with respect to e.g. the update function, the way to implementation was pretty clear. The general definition of recursive neural networks on the other hand is so abstract that it's not easy to see how we can move forward. A question you should ask yourself, if you haven't already, is how we decide what order to perform our merges in? In our motivating example we had a *fixed* tree structure - a perfectly balanced binary tree. This worked well for the *min* function we wanted to learn there. Of course this was in some sense a very easy example, as *any* tree structure could solve the problem given that we were able to learn a binary *min* function as a composite function. In fact the problem did not even depend on the order of the input numbers. This is certainly not the case in general, and we should assume that some tree structures works better than others for processing the data. Unfortunately, the optimal tree structure will probably be different for different applications. The tree structure may thus be be treated as another hyperparameter for the model. This will often require some engineering to define a suitable search space over tree structures, a completely random search may be to inefficient for most applications.

The situation is even more complicated. Instead of having a fixed tree structure we may also have an *input-dependent* tree structure. If our input

data is e.g. a sentence, the best way of processing the words might be different for different sentences. For natural language processing we might take advantage of an already existing *parser* which creates a binary tree from a sentence. There are however many different such parsers that uses different *grammars* as a foundation. One may have some intution on which ones that may be appropritate, but in general one would have to test this empirically. Though some other applications also may have such *oracles* that can be used to create input-dependent tree structures for processing, this will certainly not always be the case. Even in cases where we do have it, the trees they produce need not correspond to the optimal *processing tree* for your application. One thing that we have learned from *deep learning* is the advantages of *end-to-end* learning. In this spirit we could try to e.g. learn a policy that at each iteration decides which nodes to merge next. For a binary tree structure our policy π could given a set of n nodes return a probability vector over all possible merge pairs. In general the number of such pairs are $n(n-1)$, but through e.g. defining a *neighbouring* function which restricts the number of pairs that are allowed to merge, this number may be severely reduced. For the sentence example we could define the *neighbour* of a word to be the words *preceding* and *succeeding* it. When two nodes are merged, the neighbour of the new node is the union of the neighbours of the merged nodes, and all other nodes that had either of the two merged nodes as neighbour will have the merged node as a new neighbour. As we will rarely have a direct supervisory signal, we will in general have to learn such a policy by using e.g. reinforcement learning techniques. Although input-dependent tree structures allowd for great flexibility, it comes at a cost: difficulty of parallelization. If we for each iteration needs to decide what nodes to merge next, the number of serial steps will be *linear* in the number of input elements, the same as for RNNs. Even if the tree is given to us by an oracle, parallelization is hindered as individual tree structures does not fit well with one of the most useful tricks for high throughput, *minibatching*.

As we have seen there are quite a bit of design choices to make when implementing a recursive neural network model. Some of the design choices may lead to models that are just as serial as RNNs. The learning process could also be more challenging if we decide to learn the tree structure. All of the above may be reasons why recursive neural networks haven't yet had the same traction as recurrent neural networks. In addition it is hard to see how similiar ideas to those we had for composing and extending RNNs in Sections 2 and 3 could be used for generic recursive neural networks. This is not to dismiss the idea of recursive neural networks, rather a realization

that more research may be needed for their general practical applicability.

5 Problems

5.1 Content-based addressing for computers

5.1.1 Problem

We discussed two different types of addressing in Section 3.1, namely location-based and content-based addressing. Modern digital computers use location-based addressing. Could they just as well have used content-based addressing or can you think of any potential issues with this?

6 Bibliography

References

[1] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[2] Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines-revised. *arXiv preprint arXiv:1505.00521*, 2015.

[3] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014.

[4] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[5] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.