The Unreasonable Effectiveness of Recurrent Neural Networks

May 21, 2015

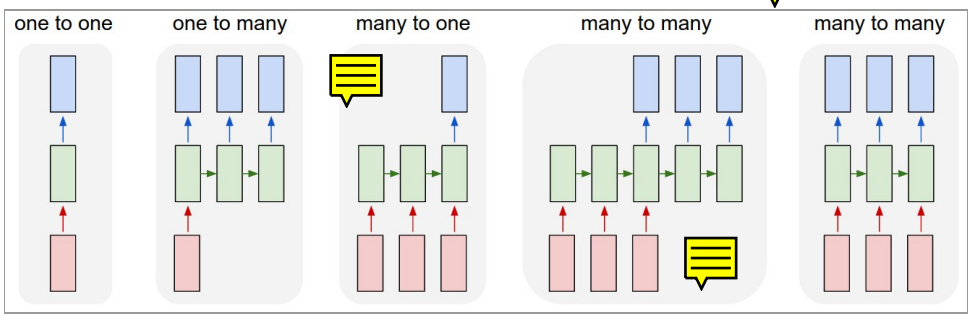
There's something magical about Recurrent Neural Networks (RNNs). I still remember when I trained my first recurrent network for Image Captioning. Within a few dozen minutes of training my first baby model (with rather arbitrarily-chosen hyperparameters) started to generate very nice looking descriptions of images that were on the edge of making sense. Sometimes the ratio of how simple your model is to the quality of the results you get out of it blows past your expectations, and this was one of those times. What made this result so shocking at the time was that the common wisdom was that RNNs were supposed to be difficult to train (with more experience I've in fact reached the opposite conclusion). Fast forward about a year: I'm training RNNs all the time and I've witnessed their power and robustness many times, and yet their magical outputs still find ways of amusing me. This post is about sharing some of that magic with you.

> *We'll train RNNs to generate text character by character and ponder the question "how is that even possible?"*

By the way, together with this post I am also releasing code on Github that allows you to train character-level language models based on multi-layer LSTMs. You give it a large chunk of text and it will learn to generate text like it one character at a time. You can also use it to reproduce my experiments below. But we're getting ahead of ourselves; What are RNNs anyway?

Recurrent Neural Networks

Sequences. Depending on your background you might be wondering: *What makes Recurrent Networks so special*? A glaring limitation of Vanilla Neural Networks (and also Convolutional Networks) is that their API is too constrained: they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes). Not only that: These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model). The core reason that recurrent nets are more exciting is that they allow us to operate over *sequences* of vectors: Sequences in the input, the output, or in the most general case both. A few examples may make this more concrete:

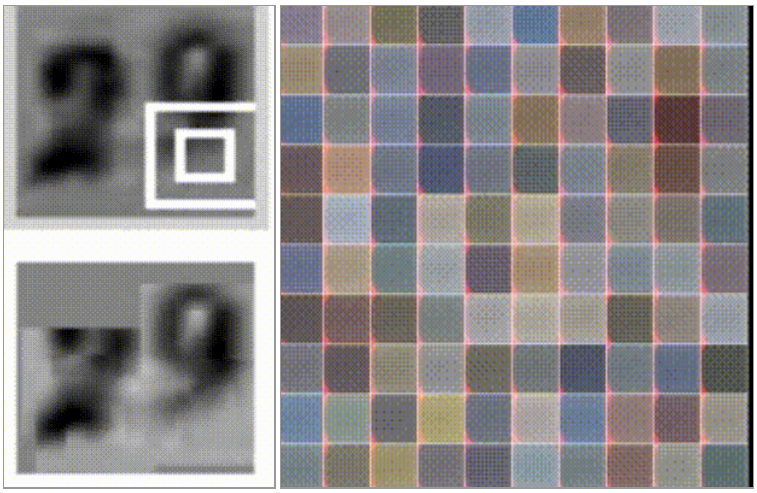


Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: (1) Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). (2) Sequence output (e.g. image captioning takes an image and outputs a sentence of words). (3) Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). (4) Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). (5) Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

As you might expect, the sequence regime of operation is much more powerful compared to fixed networks that are doomed from the get-go by a fixed number of computational steps, and hence also much more appealing for those of us who aspire to build more intelligent systems. Moreover, as we'll see in a bit, RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector. This can in programming terms be interpreted as running a fixed program with certain inputs and some internal variables. Viewed this way, RNNs essentially describe programs. In fact, it is known that RNNs are Turing-Complete in the sense that they can to simulate arbitrary programs (with proper weights). But similar to universal approximation theorems for neural nets you shouldn't read too much into this. In fact, forget I said anything.

> *If training vanilla neural nets is optimization over functions, training recurrent nets is optimization over programs.*

Sequential processing in absence of sequences. You might be thinking that having sequences as inputs or outputs could be relatively rare, but an important point to realize is that even if your inputs/outputs are fixed vectors, it is still possible to use this powerful formalism to *process* them in a sequential manner. For instance, the figure below shows results from two very nice papers from DeepMind. On the left, an algorithm learns a recurrent network policy that steers its attention around an image; In particular, it learns to read out house numbers from left to right (Ba et al.). On the right, a recurrent network *generates* images of digits by learning to sequentially add color to a canvas (Gregor et al.):



Left: RNN learns to read house numbers. Right: RNN learns to paint house numbers.

The takeaway is that even if your data is not in form of sequences, you can still formulate and train powerful models that learn to process it sequentially. You're learning stateful programs that process your fixed-sized data.

RNN computation. So how do these things work? At the core, RNNs have a deceptively simple API: They accept an input vector `x` and give you an output vector `y`. However, crucially this output vector's contents are influenced not only by the input you just fed in, but also on the entire history of inputs you've fed in in the past. Written as a class, the RNN's API consists of a single `step` function:

```
rnn = RNN()
y = rnn.step(x) # x is an input vector, y is the RNN's output vector
```

The RNN class has some internal state that it gets to update every time `step` is called. In the simplest case this state consists of a single *hidden* vector `h`. Here is an implementation of the step function in a Vanilla RNN:

```
class RNN:
  # ...
  def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

The above specifies the forward pass of a vanilla RNN. This RNN's parameters are the three matrices `W_hh, W_xh, W_hy`. The hidden state `self.h` is initialized with the zero vector. The `np.tanh` function implements a non-linearity that squashes the activations to the range `[-1, 1]`. Notice briefly how this works: There are two terms inside of the tanh: one is based on the previous hidden state and one is based on the current input. In numpy `np.dot` is matrix multiplication. The two intermediates interact with addition, and then get squashed by the tanh into the new state vector. If you're more comfortable with math notation, we can also write the hidden state update as $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$, where tanh is applied elementwise.

We initialize the matrices of the RNN with random numbers and the bulk of work during training goes into finding the matrices that give rise to desirable behavior, as measured with some loss function that expresses your preference to what kinds of outputs `y` you'd like to see in response to your input sequences `x`.

Going deep. RNNs are neural networks and everything works monotonically better (if done right) if you put on your deep learning hat and start stacking models up like pancakes. For instance, we can form a 2-layer recurrent network as follows:

```
y1 = rnn1.step(x)
y = rnn2.step(y1)
```

In other words we have two separate RNNs: One RNN is receiving the input vectors and the second RNN is receiving the output of the first RNN as its input. Except neither of these RNNs know or care - it's all just vectors coming in and going out, and some gradients flowing through each module during backpropagation.

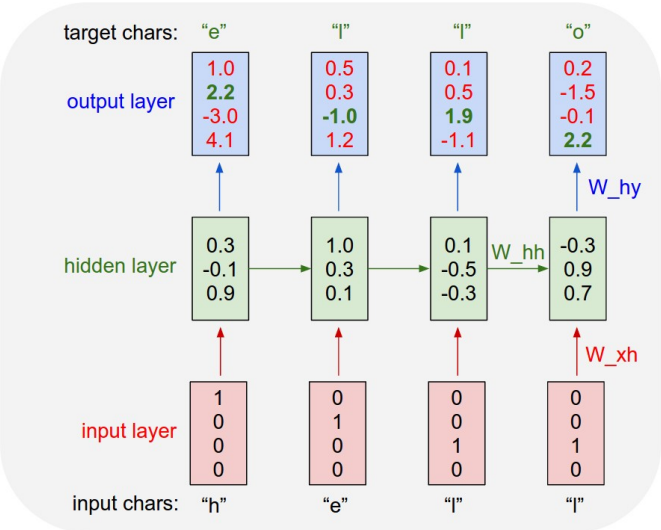
Getting fancy. I'd like to briefly mention that in practice most of us use a slightly different formulation than what I presented above called a *Long Short-Term Memory* (LSTM) network. The LSTM is a particular type of recurrent network that works slightly better in practice, owing to its more powerful update equation and some appealing backpropagation dynamics. I won't go into details, but everything I've said about RNNs stays exactly the same, except the mathematical form for computing the update (the line `self.h = ...`) gets a little more complicated. From here on I will use the terms "RNN/LSTM" interchangeably but all experiments in this post use an LSTM.

Character-Level Language Models

Okay, so we have an idea about what RNNs are, why they are super exciting, and how they work. We'll now ground this in a fun application: We'll train RNN character-level language models. That is, we'll give the RNN a huge chunk of text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters. This will then allow us to generate new text one character at a time.

As a working example, suppose we only had a vocabulary of four possible letters "helo", and wanted to train an RNN on the training sequence "hello". This training sequence is in fact a source of 4 separate training examples: 1. The probability of "e" should be likely given the context of "h", 2. "l" should be likely in the context of "he", 3. "l" should also be likely given the context of "hel", and finally 4. "o" should be likely given the context of "hell".

Concretely, we will encode each character into a vector using 1-of-k encoding (i.e. all zero except for a single one at the index of the character in the vocabulary), and feed them into the RNN one at a time with the `step` function. We will then observe a sequence of 4-dimensional output vectors (one dimension per character), which we interpret as the confidence the RNN currently assigns to each character coming next in the sequence. Here's a diagram:



An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.

For example, we see that in the first time step when the RNN saw the character "h" it assigned confidence of 1.0 to the next letter being "h", 2.2 to letter "e", -3.0 to "l", and 4.1 to "o". Since in our training data (the string "hello") the next correct character is "e", we would like to increase its confidence (green) and decrease the confidence of all other letters (red). Similarly, we have a desired target character at every one of the 4 time steps that we'd like the network to assign a greater confidence to. Since the RNN consists entirely of differentiable operations we can run the backpropagation algorithm (this is just a recursive application of the chain rule from calculus) to figure out in what direction we should adjust every one of its weights to increase the scores of the correct targets (green bold numbers). We can then perform a *parameter update*, which nudges every weight a tiny amount in this gradient direction. If we were to feed the same inputs to the RNN after the parameter update we would find that the scores of the correct characters (e.g. "e" in the first time step) would be slightly higher (e.g. 2.3 instead of 2.2), and the scores of incorrect characters would be slightly lower. We then repeat this process over and over many times until the network converges and its predictions are eventually consistent with the training data in that correct characters are always predicted next.

A more technical explanation is that we use the standard Softmax classifier (also commonly referred to as the cross-entropy loss) on every output vector simultaneously. The RNN is trained with mini-batch Stochastic

Gradient Descent and I like to use RMSProp or Adam (per-parameter adaptive learning rate methods) to stablilize the updates.

Notice also that the first time the character "l" is input, the target is "l", but the second time the target is "o". The RNN therefore cannot rely on the input alone and must use its recurrent connection to keep track of the context to achieve this task.

At test time, we feed a character into the RNN and get a distribution over what characters are likely to come next. We sample from this distribution, and feed it right back in to get the next letter. Repeat this process and you're sampling text! Lets now train an RNN on different datasets and see what happens.

To further clarify, for educational purposes I also wrote a minimal character-level RNN language model in Python/numpy. It is only about 100 lines long and hopefully it gives a concise, concrete and useful summary of the above if you're better at reading code than text. We'll now dive into example results, produced with the much more efficient Lua/Torch codebase.

Fun with RNNs

All 5 example character models below were trained with the code I'm releasing on Github. The input in each case is a single file with some text, and we're training an RNN to predict the next character in the sequence.

Paul Graham generator

Lets first try a small dataset of English as a sanity check. My favorite fun dataset is the concatenation of Paul Graham's essays. The basic idea is that there's a lot of wisdom in these essays, but unfortunately Paul Graham is a relatively slow generator. Wouldn't it be great if we could sample startup wisdom on demand? That's where an RNN comes in.

Concatenating all pg essays over the last ~5 years we get approximately 1MB text file, or about 1 million characters (this is considered a very small dataset by the way). *Technical:* Lets train a 2-layer LSTM with 512 hidden nodes (approx. 3.5 million parameters), and with dropout of 0.5 after each layer. We'll train with batches of 100 examples and truncated backpropagation through time of length 100 characters. With these settings one batch on a TITAN Z GPU takes about 0.46 seconds (this can be cut in half with 50 character BPTT at negligible cost in performance). Without further ado, lets see a sample from the RNN:

"The surprised in investors weren't going to raise money. I'm not the company with the time there are all interesting quickly, don't have to get off the same programmers. There's a super-angel round fundraising, why do you can do. If you have a different physical investment are become in people who reduced in a startup with the way to argument the acquirer could see them just that you're also the founders will part of users' affords that and an alternation to the idea. [2] Don't work at first member to see the way kids will seem in advance of a bad successful startup. And if you have to act the big company too."

Okay, clearly the above is unfortunately not going to replace Paul Graham anytime soon, but remember that the RNN had to learn English completely from scratch and with a small dataset (including where you put commas, apostrophes and spaces). I also like that it learns to support its own arguments (e.g. [2], above). Sometimes it says something that offers a glimmer of insight, such as *"a company is a meeting to think to investors"*. Here's a link to 50K character sample if you'd like to see more.

Temperature. We can also play with the temperature of the Softmax during sampling. Decreasing the temperature from 1 to some lower number (e.g. 0.5) makes the RNN more confident, but also more conservative in its samples. Conversely, higher temperatures will give more diversity but at cost of more mistakes (e.g. spelling mistakes, etc). In particular, setting temperature very near zero will give the most likely thing that Paul Graham might say:

"is that they were all the same thing that was a startup is that they were all the same thing that was a startup is that they were all the same thing that was a startup is that they were all the same"

looks like we've reached an infinite loop about startups.

Shakespeare

It looks like we can learn to spell English words. But how about if there is more structure and style in the data? To examine this I downloaded all the works of Shakespeare and concatenated them into a single (4.4MB) file. We can now afford to train a larger network, in this case lets try a 3-layer RNN with 512 hidden nodes on each layer. After we train the network for a few hours we obtain samples such as:

```
PANDARUS:
```

```
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

Remember, all the RNN knows are characters, so in particular it samples both speaker's names and the contents. Sometimes we also get relatively extented monologue passages, such as:

```
VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.
```

I can barely recognize these samples from actual Shakespeare :) If you like Shakespeare, you might appreciate this 100,000 character sample. Of course, you can also generate an infinite amount of your own samples at different temperatures with the provided code.

Wikipedia

We saw that the LSTM can learn to spell words and copy general syntactic structures. Lets further increase the difficulty and train on structured markdown. In particular, lets take the Hutter Prize 100MB dataset of raw Wikipedia and train an LSTM. Following Graves et al., I used the first 96MB for training, the rest for validation and ran a few models overnight. We can now sample Wikipedia articles! Below are a few fun excerpts. First, some basic markdown output:

```
Naturalism and decision for the majority of Arab countries' capitalide was grounded
by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated
with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal
in the [[Protestant Immineners]], which could be said to be directly in Cantonese
Communication, which followed a ceremony and set inspired prison, training. The
emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom
of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known
in western [[Scotland]], near Italy to the conquest of India with the conflict.
Copyright was the succession of independence in the slop of Syrian influence that
was a famous German movement based on a more popular servicious, non-doctrinal
and sexual power post. Many governments recognize the military housing of the
```

```
[[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]],
that is sympathetic to be to the [[Punjab Resolution]]
(PJS)[http://www.humah.yahoo.com/guardian.
cfm/7754800786d17551963s89.htm Official economics Adjoint for the Nazism, Montgomery
was swear to advance to the resources for those Socialism's rule,
was starting to signing a major tripad of aid exile.]]
```

In case you were wondering, the yahoo url above doesn't actually exist, the model just hallucinated it. Also, note that the model learns to open and close the parenthesis correctly. There's also quite a lot of structured markdown that the model learns, for example sometimes it creates headings, lists, etc.:

```
{ { cite journal | id=Cerling Nonforest Department|format=Newlymeslated|none } }
''www.e-complete''.

'''See also''': [[List of ethical consent processing]]

== See also ==
*[[Iender dome of the ED]]
*[[Anti-autism]]

===[[Religion|Religion]]===
*[[French Writings]]
*[[Maria]]
*[[Revelation]]
*[[Mount Agamul]]

== External links==
* [http://www.biblegateway.nih.gov/entrepre/ Website of the World Festival. The labour

==External links==
* [http://www.romanology.com/ Constitution of the Netherlands and Hispanic Competition
```

Sometimes the model snaps into a mode of generating random but valid XML:

```
<page>
  <title>Antichrist</title>
  <id>865</id>
  <revision>
    <id>15900676</id>
    <timestamp>2002-08-03T18:14:12Z</timestamp>
    <contributor>
      <username>Paris</username>
      <id>23</id>
    </contributor>
    <minor />
    <comment>Automated conversion</comment>
    <text xml:space="preserve">#REDIRECT [[Christianity]]</text>
  </revision>
</page>
```

The model completely makes up the timestamp, id, and so on. Also, note that it closes the correct tags appropriately and in the correct nested order. Here are 100,000 characters of sampled wikipedia if you're interested to see more.

Algebraic Geometry (Latex)

The results above suggest that the model is actually quite good at learning complex syntactic structures. Impressed by these results, my labmate (Justin Johnson) and I decided to push even further into structured territories and got a hold of this book on algebraic stacks/geometry. We downloaded the raw Latex source file (a 16MB file) and trained a multilayer LSTM. Amazingly, the resulting sampled Latex *almost* compiles. We had to step in and fix a few issues manually but then you get plausible looking math, it's quite astonishing:

Understanding LSTM Networks

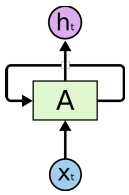
Posted on August 27, 2015

Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can't do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

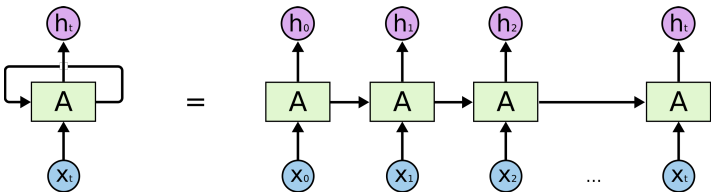
Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, A, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



An unrolled recurrent neural network.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.

And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning... The list goes on. I'll leave discussion of the amazing feats one can achieve with RNNs to Andrej Karpathy's excellent blog post, The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/). But they really are pretty amazing.

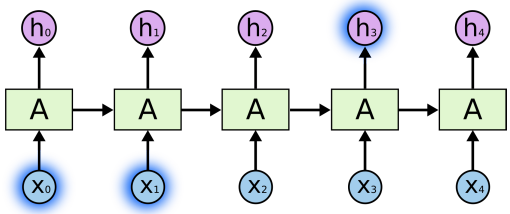
Essential to these successes is the use of "LSTMs," a very special kind of recurrent neural network which works, for many tasks, much much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It's these LSTMs that this essay will explore.

The Problem of Long-Term Dependencies

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If

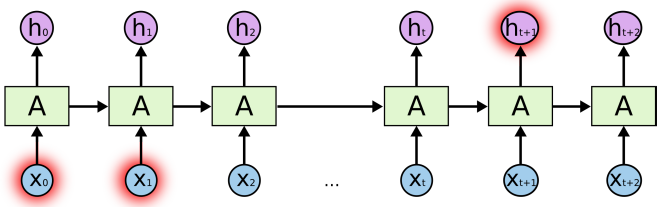
RNNs could do this, they'd be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the *sky*," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent *French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] (http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf) and Bengio, et al. (1994) (http://www-dsi.ing.unifi.it/~paolo/ps/tnn-94-gradient.pdf), who found some pretty fundamental reasons why it might be difficult.

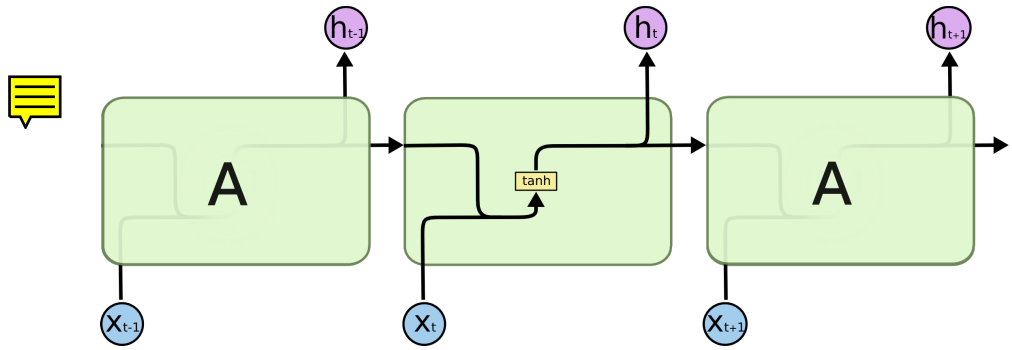
Thankfully, LSTMs don't have this problem!

LSTM Networks

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997) (http://www.bioinf.jku.at/publications/older/2604.pdf), and were refined and popularized by many people in following work.[1] They work tremendously well on a large variety of problems, and are now widely used.

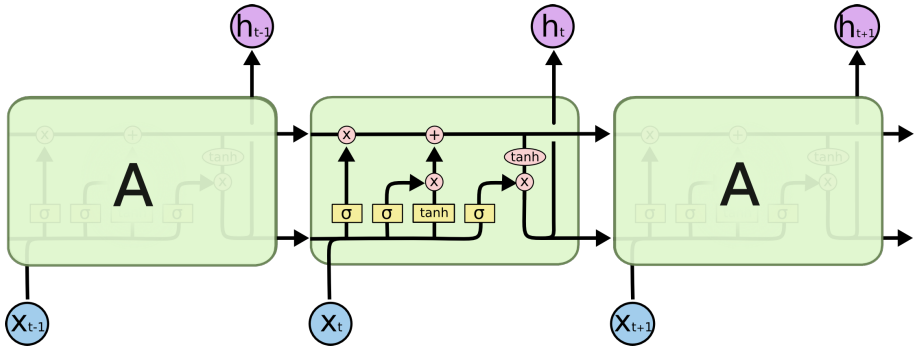
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



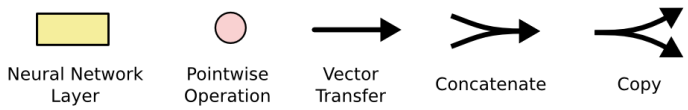
The repeating module in a standard RNN contains a single layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The repeating module in an LSTM contains four interacting layers.

Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by step later. For now, let's just try to get comfortable with the notation we'll be using.

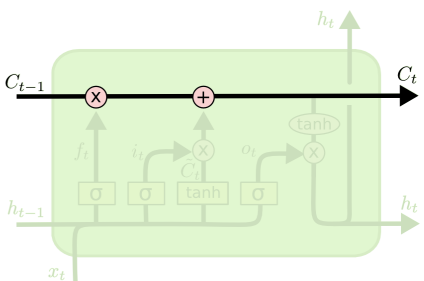


In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

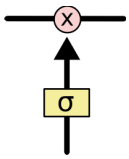
The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net

layer and a pointwise multiplication operation.



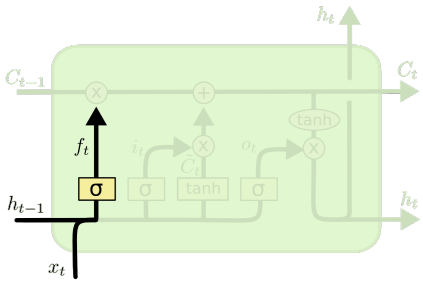
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"

An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM Walk Through

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."

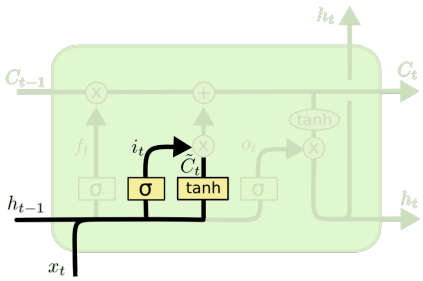
Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.

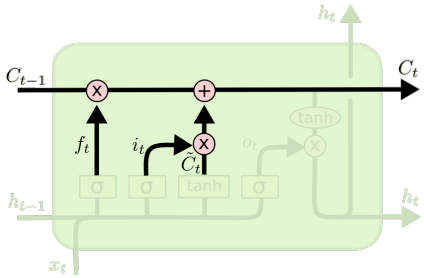


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.

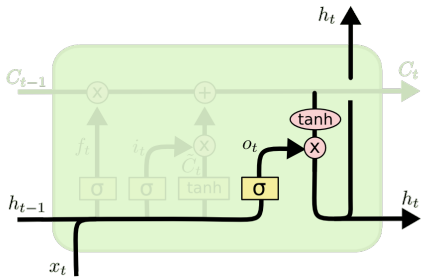
In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through $\tanh$ (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

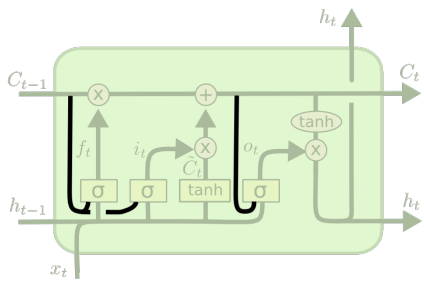


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Variants on Long Short Term Memory

What I've described so far is a pretty normal LSTM. But not all LSTMs are the same as the above. In fact, it seems like almost every paper involving LSTMs uses a slightly different version. The differences are minor, but it's worth mentioning some of them.

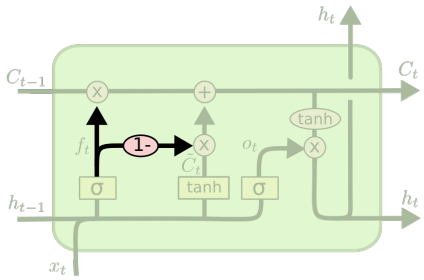
One popular LSTM variant, introduced by Gers & Schmidhuber (2000) (ftp://ftp.idsia.ch/pub/juergen/TimeCount-IJCNN2000.pdf), is adding "peephole connections." This means that we let the gate layers look at the cell state.



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

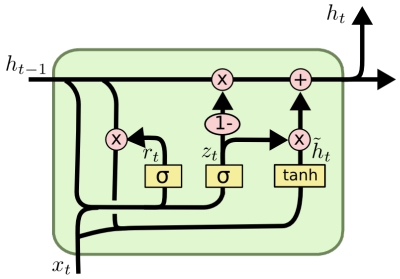
The above diagram adds peepholes to all the gates, but many papers will give some peepholes and not others.

Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014) (http://arxiv.org/pdf/1406.1078v3.pdf). It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

These are only a few of the most notable LSTM variants. There are lots of others, like Depth Gated RNNs by Yao, et al. (2015) (http://arxiv.org/pdf/1508.03790v2.pdf). There's also some completely different approach to tackling long-term dependencies, like Clockwork RNNs by Koutnik, et al. (2014) (http://arxiv.org/pdf/1402.3511v1.pdf).

Which of these variants is best? Do the differences matter? Greff, et al. (2015) (http://arxiv.org/pdf/1503.04069.pdf) do a nice comparison of popular variants, finding that they're all about the same. Jozefowicz, et al. (2015) (http://jmlr.org/proceedings/papers/v37/jozefowicz15.pdf) tested more than ten thousand RNN architectures, finding some that worked better than LSTMs on certain tasks.

Conclusion

Earlier, I mentioned the remarkable results people are achieving with RNNs. Essentially all of these are achieved using LSTMs. They really work a lot better for most tasks!

Written down as a set of equations, LSTMs look pretty intimidating. Hopefully, walking through them step by step in this essay has made them a bit more approachable.

LSTMs were a big step in what we can accomplish with RNNs. It's natural to wonder: is there another big step? A common opinion among researchers is: "Yes! There is a next step and it's attention!" The idea is to let every step of an RNN pick information to look at from some larger collection of information. For example, if you are using an RNN to create a caption describing an image, it might pick a part of the image to look at for every word it outputs. In fact, Xu, *et al.* (2015) (http://arxiv.org/pdf/1502.03044v2.pdf) do exactly this – it might be a fun starting point if you want to explore attention! There's been a number of really exciting results using attention, and it seems like a lot more are around the corner...

Attention isn't the only exciting thread in RNN research. For example, Grid LSTMs by Kalchbrenner, *et al.* (2015) (http://arxiv.org/pdf/1507.01526v1.pdf) seem extremely promising. Work using RNNs in generative models – such as Gregor, *et al.* (2015) (http://arxiv.org/pdf/1502.04623.pdf), Chung, *et al.* (2015) (http://arxiv.org/pdf/1506.02216v3.pdf), or Bayer & Osendorfer (2015) (http://arxiv.org/pdf/1411.7610v3.pdf) – also seems very interesting. The last few years have been an exciting time for recurrent neural networks, and the coming ones promise to only be more so!

Acknowledgments

I'm grateful to a number of people for helping me better understand LSTMs, commenting on the

CHAPTER 6

Contextualized Embeddings

This chapter provides an introduction to *contextualized word (CW) embeddings*. CW can be considered as the new generation of word (and sense) embeddings. The distinguishing factor here is the sensitiveness of a word's representation to the context: a target word's embedding can change depending on the context in which it appears. These *dynamic* embeddings alleviate many of the issues associated with *static* word embeddings and provide reliable means for capturing semantic and syntactic properties of natural language in context. Despite their young age, contextualized word embeddings have provided significant gains in almost any downstream NLP task to which they have been applied.

6.1 THE NEED FOR CONTEXTUALIZATION

Since their introduction, pre-trained word embeddings have dominated the field of semantic representation. They have been a key component in most neural natural language processing systems. Usually, an NLP system is provided with large pre-trained word embeddings for all the words in the vocabulary of the target language.[1] At the input layer, the system looks up the embedding for a given word and feeds the corresponding embedding to the subsequent layers (as opposed to a one-hot representation). Figure 6.1(a) depicts the general architecture for such a system. Moving from hard-coded one-hot representations to a continuous word embedding space usually results in improved generalisation power of the system, hence improved performance.

However, pre-trained word embeddings, such as Word2vec and GloVe, compute a single *static* representation for each word. The representation is fixed; it is independent from the context in which the word appears. In our example in Figure 6.1(a), the same embedding would be used at the input layer for *cell* even if the word was used in different contexts that would have triggered its other meanings, e.g., "the cells of a honeycomb", "mobile cell", and "prison cell".

[1] For instance, the widely-used Google News Word2vec embeddings has a vocabulary of three million words: https://code.google.com/archive/p/word2vec/

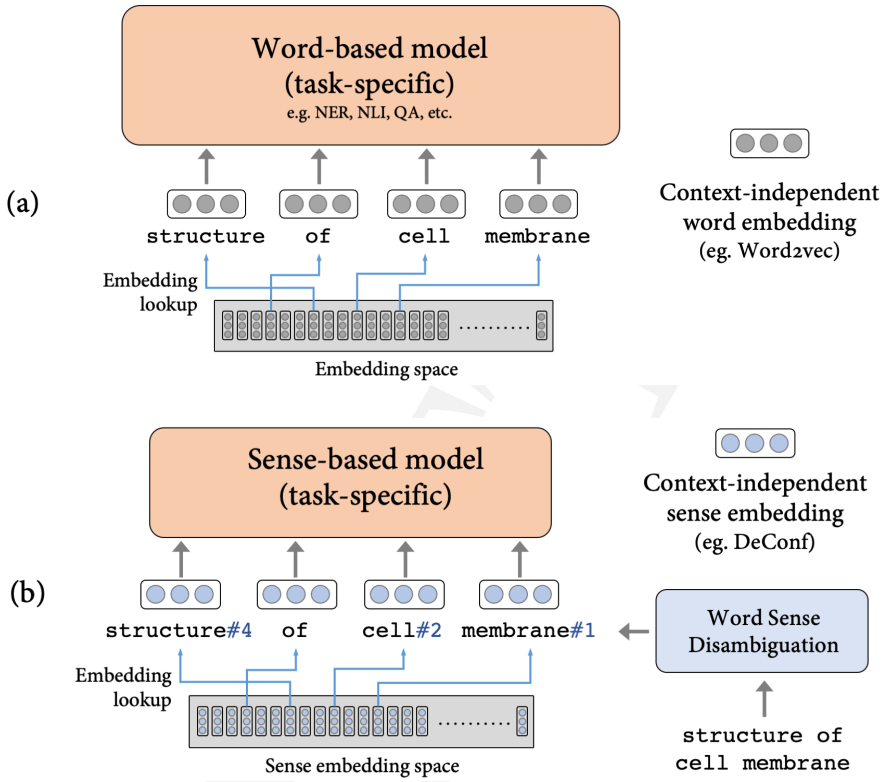


Figure 6.1: Context-independent (static) embeddings are fixed points in the semantic space: they do not change, irrespective of the context in which the target word appears. In the word-based model (a) For each input word, the static embedding is looked up from a pre-trained semantic space. Embeddings are introduced as features, usually in the input layer. In the sense based model (b), words are first disambiguated before being input to the system, and the corresponding sense embeddings are passed to the model.

Static semantic representations suffer from two important limitations: (1) ignoring the role of context in triggering specific meanings of words is certainly an oversimplification of the problem; this is not the way humans interprete meanings of words in texts; (2) due to restricting the semantic barriers to individual words, it is difficult for the model to capture higher order semantic phenomena, such as compositionality and long-term dependencies. Therefore, the static word-based representation of words can substantially hamper the ability of NLP systems in understanding the semantics of the input text. In this setting, all the load of deriving meaning from

a sequence of words is on the shoulders of the main system, which has to deal with ambiguity, syntactic nuances, agreement, negation, etc.

Knowledge-based sense representations (discussed in Chapter 5) can partly address the first issue. The distinct representations they provide for specific meanings of polysemous words enable the model to have a clear interpretation of the intended meaning of a word, pure from its other irrelevant meanings. Swapping word embeddings with sense embeddings requires the system to carry out an additional step: a word sense disambiguation module has to identify the intended meaning of ambiguous words (e.g., "cell"). Having identified the intended sense, the system can swap the word embedding with the corresponding sense embedding. This swapping coupled with the disambiguation stage can be regarded as a way of contextualizing each word's representation to the semantics of its surrounding words.

However, there are multiple factors that limit the efficacy of sense embeddings. Firstly, word sense disambiguation is far from being optimal; hence, the initial stage of mapping words to word senses introduces inevitable noise to the pipeline [Pilehvar et al., 2017]. Secondly, it is not straightforward to benefit from raw texts, which are available at scale, to directly improve these representations. Hence, their coverage and semantic depth is limited to the knowledge encoded in lexical resources, which can be too restrictive. Thirdly, these representations are still not fully contextualized. The intended sense of a word in a given context is assumed to fully align with that defined in the target inventory, which might not be always true. For instance, the closest meaning for the noun "tweet" in WordNet is "a weak chirping sound as of a small bird" which certainly will not fully align if the intended meaning refers to "a post on Twitter". Even worse, the intended meaning of the word, or the word itself, might not have been covered in the underlying sense inventory (for instance, the noun "embedding", as it is widely used in NLP, is not defined in WordNet).

Unsupervised sense representations can be adapted to specific text domains; hence, they might not suffer as much in terms of coverage. However, they still need a disambiguation stage which is not be as straightforward as that for knowledge-based counterparts. Given that these representations are often produced as a result of clustering, their semantic distinctions are unclear and their mapping to well-defined concepts is not simple. Hence, a more complicated word sense disambiguation stage would be required, one that can disambiguate the input words according to the inventory of induced senses. Given that such a technique cannot easily benefit from rich sense-specific information available in existing lexical resources, it is usually not that effective. In fact, one of the main limitations of unsupervised sense representation models lies in their difficult integration into downstream models [Li and Jurafsky, 2015].

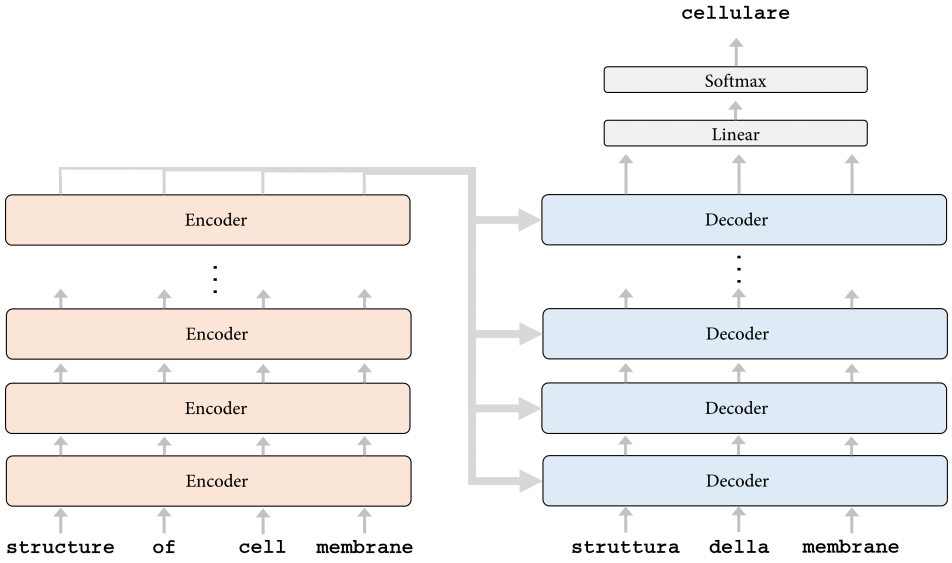


Figure 6.2: A high-level illustration of the Transformer model used for translation. The model is auto-regressive and has an encoder-decoder structure. The encoder and decoder have six identical encoders and decoders, respectively (only four shown here).

6.2 BACKGROUND: TRANSFORMER MODEL

Given that most of the recent literature on contextualized embeddings are based on a novel model called Transformer, in this section, we provide a brief overview of the Transformer architecture. Figure 6.2 provides a high-level illustration of the Transformer model. The model is an auto-regressive sequence transducer: the goal is to convert an input sequence to an output sequence, while the predictions are done one part at a time, consuming the previously generated parts as additional input. Similarly to most other sequence to sequence (Seq2Seq) models (cf. Section 2.2.2), the Transformer employs an encoder-decoder structure. However, unlike previous models which conventionally used a recurrent networ (e.g., LSTM) for their encoder and decoder, the Transformer model is based on self-attention only with no recurrence. The Transformer forgoes the recurrence of RNN's for a fully feedforward attention-based architecture.

The main idea behind the Transformer model is self-attention. Self-attention, also known as intra-attention, is a mechanism that enables the sequence encoder to "attend" to specific parts of the sequence while processing a specific word.

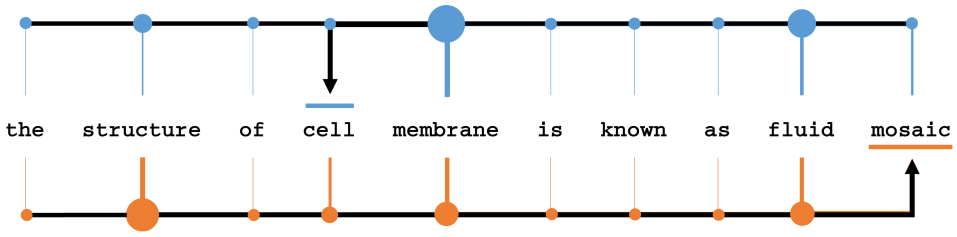


Figure 6.3: An illustration of self attention for the words *cell* (top) and *mosaic* (bottom). By attending to the context, particularly *membrane*, the interpretation of *cell* gets adapted to this specific usage and for the biological meaning. The same applies for *mosaic*, with a self attention mostly towards *structure*, *membrane*, and *fluid*. In the Transformer model, there are multiple spaces for self-attention (multi-head attention) that allows the model to have different interpretations in multiple representation sub-spaces.

6.2.1 SELF-ATTENTION

We saw in Section 2.2.2 the intuition behind the attention mechanism in sequence tranduction models. The basic idea was to focus the attention of the model to those source tokens for which the decoder is currently trying to generate the corresponding output (for instance, translation). The same idea can be applied to the process of reading and understanding of natural language text.

Figure 6.3 shows an example for self-attention for two semantically ambiguous words, *mosaic* and *cell*. Consider the word *cell*. Even for a human reading this sentence, it would be almost impossible to identify the intended meaning of the word unless the succeeding word (i.e., *membrane*) is taken into account. In fact, to be able to get a clear grasp of the meaning of a sentence, humans often require to scan the context or finish reading the sentence.

Self-attention, also known as intra-attention, is a special attention mechanism that tries to mimic this process. Instead of relating positions across two different sequences, self-attention looks for relations between positions in the same sequence. The goal of self-attention is to allow the model to consider the context while "reading" a word. For the case of our example, while "reading" the target word *cell* the self-attention mechanism focuses the attention to the word *membrane* in order to allow a better representation for the target word, adapted to the biological meaning. Note that, similarly to the Seq2Seq attention mechanism, self-attention is a *soft* measure: multiple words can be attended with varying degrees.

Consider the input sequence $x_1, ..., x_n$. The self-attention mechanism maps the input embeddings for this sequence to an adapted output sequence $z_1, ..., z_n$. For an

input word[2] x_t, the process of computing the self-attention vector z_t in the Transformer model can be summarized as follows:

1. For every input x_i, compute three different vectors: query q_i, key k_i, and value v_i. This is done by multiplying the input vector x_i with the corresponding matrices W^q, W^k, and W^v. The weights of these matrices are among the parameters that are learned during training.

2. Compute a score s_i for every input x_i. The score is computed as the dot product of the query vector q_t and the corresponding key vectors for every x_i (i.e., all k_i).

3. Normalize the scores by $\sqrt{d_k}$, where d_k is the dimensionality of the key (and query) vector.

4. Compute a weighted average of all value vectors (v_i), weighted by their corresponding scores s_i. The resulting vector is z_t.

   The above procedure can be written as the following equation in matrix form:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{6.1}$$

The Transformer model makes use of multiple attention *heads*. In other words, multiple sets of W matrices are considered to produce different query, key, and value vectors for the same word. This allows the model to have multiple representation sub-spaces to focus on different positions.

6.2.2 ENCODER

The original Transformer model makes use of six identical encoder layers. Figure 6.4 (left) shows the the stack of three of these encoders (for one, we are showing more details of the internal structure). Each encoder layer has two sub-layers: self-attention and feedforward. We saw in the previous section how the self-attention layer can help the model to look at the context words while "reading", in order to get a clearer understanding of the semantics of individual tokens, and in turn the meaning of the sentence. As was explained before, each attention layer is coupled with multiple "heads", with the hope of enabling the model to attend to different parts and two have multiple independent representation subspaces for capturing distinct patterns.

[2]In the actual model each word might be split into multiple tokens; for instance, *membrane* can be split into *mem*, *bra*, and *ne_*. The input to the model would be a sequence of tokens (rather than words).

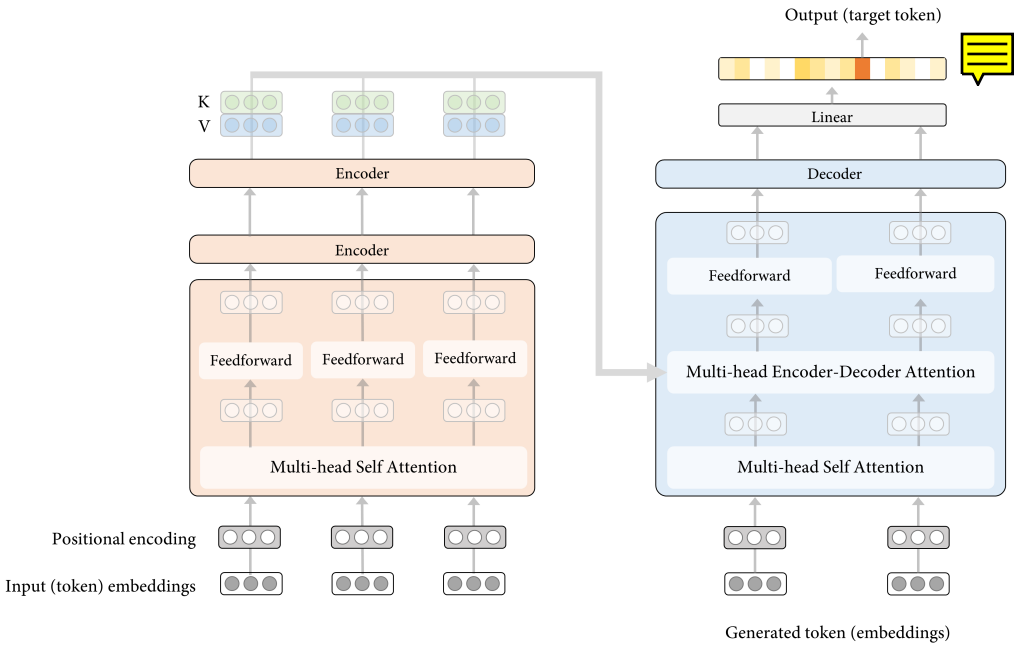


Figure 6.4: Encoders and decoders have similar internal structure, other than an encoder-decoder attention sub-layer added to the decoder. Input word embeddings are summed up with positional encodings and are fed to the bottom encoder. Decoders receive the outputs generated so far (as well as signal from the encoder) and predict the next token. Prediction is done via a fully-connected layer that generates the scores over the vocabulary (logits vector), followed by a softmax (to make the scores probability-like).

The z_i outputs of the self-attention sub-layer are fed to the feedforward sub-layer which is in fact a fully-connected network. The feedforward layer is *point-wise*, i.e., the same feedforward network is applied independently to individual z_i vectors.

6.2.3 DECODER

Similarly to the encoder, the decoder of the Transformer model also consists of a stack of six identical decoder layers. Each decoder is very similar to encoder in architecture with the slight difference that it has a third sub-layer which performs a cross-attention between encoder's output and decoder's state.[3] Also, it is necessary

[3]Both decoder and encoder involve other small architectural details, such as residual connections around sub-layers and normalization. We skip these for simplicity.

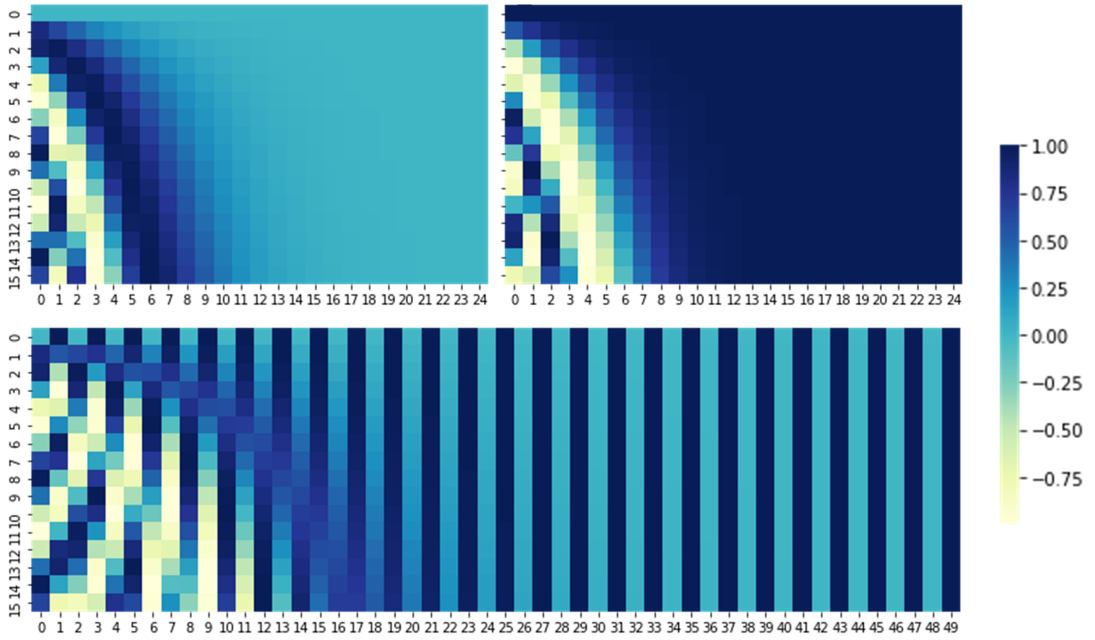


Figure 6.5: The positional encodings used to capture relative positions in the Transformer model. Sixteen encodings are visualised (rows), enough to encode a sequence of 16 tokens (10,000 in the original model). The model also use of 512-d encodings; but, for simplicity we show 50-d. Two sub-encodings are generated using $sin()$ (top, left) and $cos()$ (top, right) functions. The two are merged to generate the full encoding, shown at the bottom.

to modify the self-attention sub-layer in the decoder in order to prevent the model from attending to subsequent positions. Transformer achieves this by means of masking the subsequent positions. This masking is to ensure that the predictions at any position can depend only on the outputs generated so far (and not future outputs).

6.2.4 POSITIONAL ENCODING

As explained above, the Transformer model does not involve any recurrence (or convolution) to capture the relative positioning of tokens in the sequence. However, we know that word order is crucial to semantics; ignoring this would diminish the model to a simple bag-of-words. The authors of the Transformer model made use of a mechanism called positional encoding in order to inject information about token positions and hence making the model sensitive to word order.

To this end, each input embedding to the encoder and decoder is added with a positional embedding which denotes the position of each input word with respect to the sequence. To facilitate the summation, positional encodings are of the same size as the input token embeddings. There can be different ways of encoding the position; Transformer makes use of a sinosuidal function. Specifically, the positional encoding P for the t^{th} token (starting from 0) is computed as follows:

$$\begin{aligned} D_i &= \frac{1}{10000^{\frac{2i}{d}}} \\ P(t, 2i) &= sin(tD_i) \\ P(t, 2i+1) &= cos(tD_i) \end{aligned} \tag{6.2}$$

where $i \in \{0, .., d-1\}$ is the encoding index, and d is the dimensionality of the positional encodings which is the same as input token embedding size (512 in the original model). An example is shown in Figure 6.5. Note that the final encoding is a merger of the two sub-encodings from $sin()$ and $cos()$ functions, where the former fills the even positions and the latter the odd ones.

6.3 CONTEXTUALIZED WORD EMBEDDINGS

Unlike static word embeddings, contextualized embeddings are representations of words in context. They can circumvent many of the limitations associated with word and sense embeddings, bringing about multiple advantages, one of the most important of which is seamless integration into most neural language processing models. Unlike knowledge-based sense representations, these embeddings do not rely on annotated data or external lexical resources and can be learned in an unsupervised manner. More importantly, their introduction to neural models does not require extra efforts such as word sense disambiguation as they function at the level of words. Interestingly, contextualized embeddings not only can capture various semantic roles of a word, but also its syntactic properties [Hewitt and Manning, 2019, Goldberg, 2019].

In contrast to static word embeddings which are fixed, contextualized word embeddings are *dynamic* in that the same word can be assigned different embeddings if it appears in different contexts. Therefore, unlike static word embeddings, contextualized embeddings are assigned to tokens as opposed to types. Instead of receiving words as distinct units and providing independent word embeddings for each, contextualized models receive the whole text span (the target word along with its context) and provide specialized embeddings for individual words which are adjusted to their context. Figure 6.6 provides an illustration: to produce a dynamic embedding for the target word (i.e., *cell*) the contextualized model analyzes the whole context.

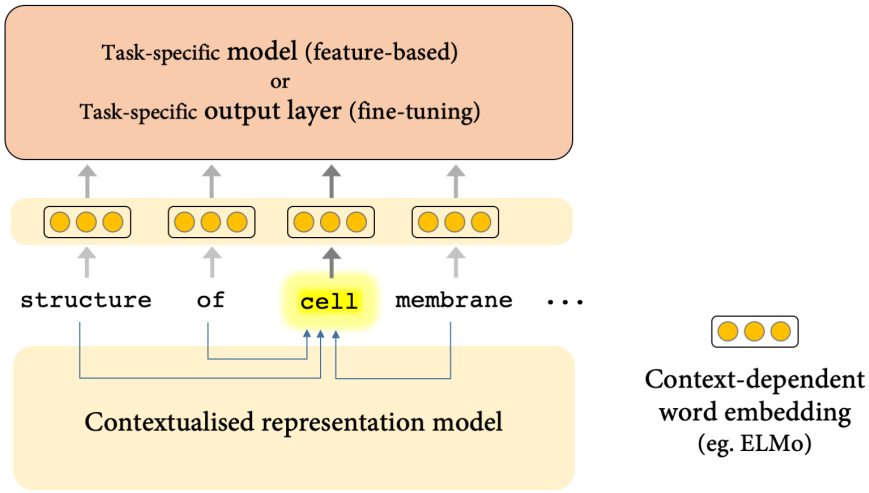


Figure 6.6: Unlike static (context-independent) word embeddings, contextualized (dynamic) embeddings are not fixed: they adapt to their representation to the context. The contextualized representation model processes the context of the target word (*cell* in the figure) and generates its dynamic embedding.

The following sections will provide more information on the specifics of the model in the figure.

6.3.1 EARLIER METHODS

The sequence tagger of Li and McCallum [2005] is one of the pioneering works that employ contextualized representations. The model infers context sensitive latent variables for each word based on a soft word clustering and integrates them, as additional features, to a CRF sequence tagger. The clustering technique enabled them to associate the same word with different features in different contexts.

With the introduction of word embeddings [Collobert et al., 2011, Mikolov et al., 2013c] and the efficacy of neural networks, and in the light of meaning conflation deficiency of word embeddings, context-sensitive models have once again garnered research attention. **Context2vec** [Melamud et al., 2016] is one of the first proposals in the new branch of contextualized representations. Context2vec's initial goal was to compute a better representations for the context of a given target word. The widely-practised and usually competitive baseline to compute representation for multiple words (a piece of text) is to simply average the embedding of the words. This baseline is unable to capture important properties of natural language, such as word order or semantic prominence. Instead, Context2vec makes use of a

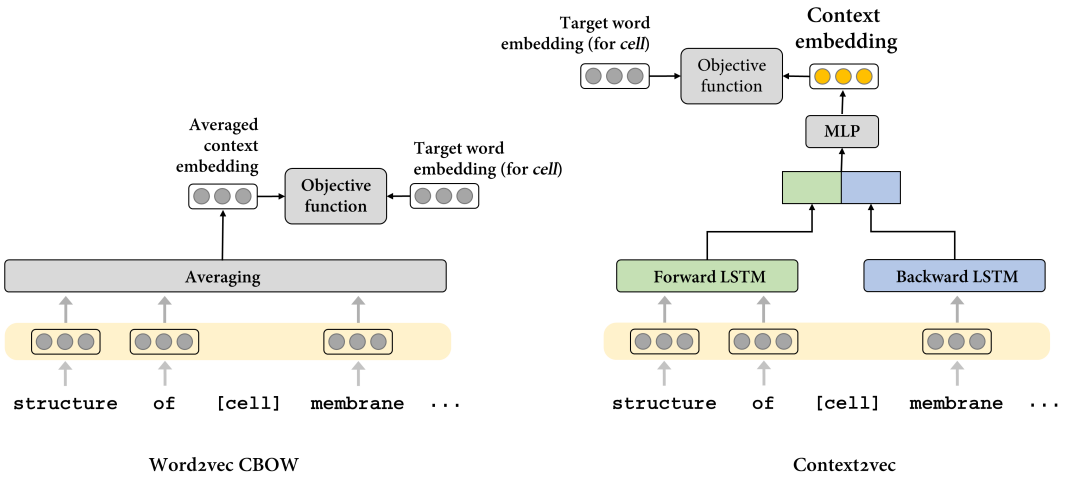


Figure 6.7: Architecture of Context2vec and how it differs from Word2vec CBOW: instead of modeling the context by naively averaging embeddings of words in the context window (as in CBOW), context2vec models the context using a bidirectional LSTM.

bidirectional LSTM language model to better encode these properties. Figure 6.7 shows the architecture of Context2vec and illustrates the different context modeling of this technique in comparison with Word2vec CBOW. Context2vec embeds sentential contexts and target words in the same semantic space.

The encoded representation for the context of a target word can be taken as the embedding of that word which is *contextualized* to its context. Hence, though the authors of Context2vec did not explicitly view the approach as a means of computing dynamic word embeddings, it is highly similar to subsequent works in contextualized word embeddings and constitutes one of the bases for this field of research. The most important distinguishing factor to subsequent techniques is that Context2vec ignores the target word while computing the contextualized representation, which turns out to be crucial.

6.3.2 LANGUAGE MODELS FOR WORD REPRESENTATION

As was discussed in Chapter 2, Language Models (LM) aim at predicting the next word in a sentence given the preceding words. To be able to accurately predict a word in a sequence, LMs need to encode both the semantic and syntactic roles of words in context. This makes them suitable candidates for word representation. In fact, nowadays, LMs are key components not only in natural language generation, but also

in natural language understanding. Additionally, knowledge acquisition bottleneck is not an issue for LMs, since they can essentially be trained on multitude of raw texts in an unsupervised manner. In fact, extensive models can be trained with LM objectives and then transferred to specific tasks. Though still at early stages, this technique has been shown to be a promising direction [Radford et al., 2018], reminiscent of the pre-training procedure in Computer Vision which involves training an initial model on ImageNet or other large image datasets and then transferring the knowledge to new tasks.

Figure 6.6 provides a high-level illustration of the integration of contextualized word embeddings into an NLP model. At the training time, for each word (e.g., *cell* in the figure) in a given input text, the language model unit is responsible for analyzing the context (usually using sequence-based neural networks) and adjusting the target word's representation by contextualising (adapting) it to the context. These context-sensitive embeddings are in fact the internal states of a deep neural network which is trained with language modeling objectives either in an *unsupervised* manner [Peters et al., 2017, Peters et al., 2018] or on a *supervised* task, such as bilingual translation configuration [McCann et al., 2017]. The training of contextualized embeddings is carried out as a pre-training stage, independently from the main task on a large unlabeled or differently-labeled text corpus. Depending on the sequence encoder used in language modeling, these models can be put into two broad categories: **RNN** (mostly LSTM) and **Transformer**.

6.3.3 RNN-BASED MODELS

For this branch of techniques, the "Contextualized representation model" in Figure 6.6 is on the shoulders of an LSTM-based encoder, usually a multi-layer bidirectional LSTM (BiLSTM). LSTMs are known to be able to capture word order to some good extend. Also, unlike the word embedding averaging baseline, LSTMs are capable of combining word representations in a more reasonable manner, giving higher weights to those words that are semantically more central in the context. The TagLM model of Peters et al. [2017] is a an example of this branch which trains a BiLSTM sequence encoder on monolingual texts. The outputs of the sequence encoder are concatenated and fed to a neural CRF sequence tagger as additional features.

The Context Vectors (CoVe) model of McCann et al. [2017] similarly computes contextualized representations using a two-layer biLSTM network, but in the machine translation setting. CoVe vectors are pre-trained using an LSTM encoder from an attentional sequence-to-sequence machine translation model.[4]

[4]In general, the pre-training property of contextualized embeddings makes them closely related to transfer learning [Pratt, 1993], which is out of the scope of this book. For more detailed information on transfer learning for NLP, we would refer to [Ruder, 2019].

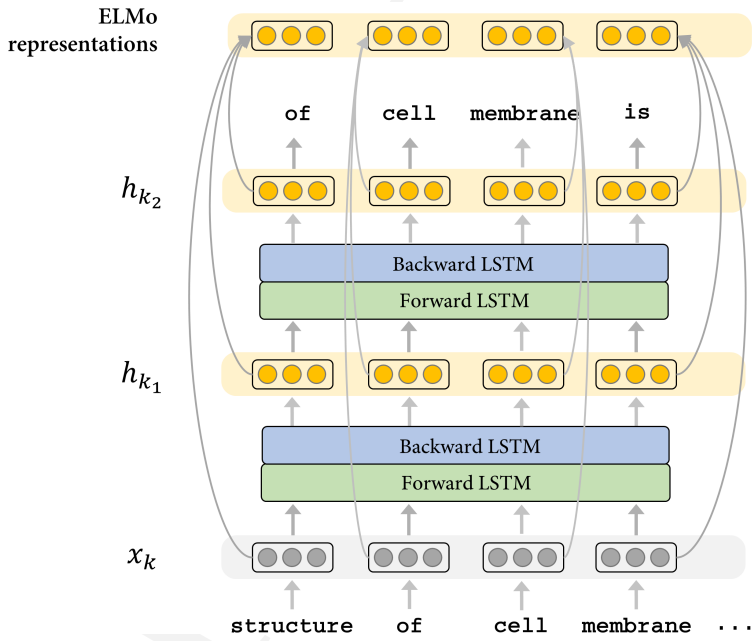


Figure 6.8: ELMo makes use of a 2-layer bidirectional LSTM to encode words in the context word. The ELMo representation for the target word is a combination of the hidden states of the two BiLSTM layers, i.e., h_{k_1} and h_{k_2}, which encode the context-sensitive representation of each word, and the static representation of the word, i.e., x_k, which is character-based. ELMo uses some residual connections across LSTMs which are not shown in the figure for simplicity.

The prominent **ELMo** (Embeddings from Language Models) technique [Peters et al., 2018] is similar in principle. A multi-layer (two in the original model) BiLSTM sequence encoder is responsible for capturing the semantics of the context. The main difference between TagLM and ELMo lies in the fact that in the latter some weights are shared between the two directions of the language modeling unit. Figure 6.8 provides a high-level illustration of how ELMo embeddings are constructed. A residual connection between the LSTM layers allows the deeper layer(s) to have a better look at the original input and to allow the gradients to better backpropagate to the initial layers. The model is trained on large amounts of texts with the language modeling objective: given a sequence of tokens, predict the next token. The trained model is then used to derive contextualized embeddings that can be used as input into various NLP systems.

There can be multiple ways for combining the outputs of ELMo model, i.e., the hidden states of the two BiLSTM layers, h_{k_1} and h_{k_2}, and the context-independent representation x_k. One may take only the top layer output or concatenate the three layers to have long vectors for each token, to be fed as inputs to an NLP system. One can also learn a weighted combination of the three layers, based on the target task, or concatenate other static word embeddings with ELMo embeddings. ELMo makes use of character-based technique (based on Convolution Neural Networks) for computing x_k embeddings. Therefore, it benefits from all the characteristics of character-based representations (cf. Section 3.3), such as robustness to unseen words.

6.4 TRANSFORMER-BASED MODELS: BERT

The introduction of **Transformers** [Vaswani et al., 2017] and their immense potential in encoding text sequences resulted in another boost in the already fast-moving field of LM-based contextualized representations. Transformers come with multiple advantages over recurrent neural networks (which were previously the dominant role players): (1) compared to RNNs which process the input sequentially, Transformers are parallel which makes them suitable for GPUs and TPUs which excel at massive parallel computation; (2) Unlike RNNs which have memory limitation and tend to process the input in one direction, thanks to the self-attention mechanism (cf. Section 6.2.1), Transformers can attend to contexts about a word from distant parts of a sentence, both earlier and later than the word appears, in order to enable a better understanding of the target word without any locality bias. For instance, the word "cell" in Figure 6.8 can be disambiguated by looking at the next word in the context, "membrane".

The impressive initial results obtained by Transformers on sequence to sequence tasks, such as Machine Translation and syntactic parsing [Kitaev and Klein, 2018], suggested a potential replacement for LSTMs in sequence encoding tasks. As

of now, Transformers are dominantly exceeding the performance levels of conventional recurrent models on most NLP tasks that involve sequence encoding.

The OpenAI's **GPT** model (Generative Pre-trained Transformer) [Radford, 2018] was one of the first attempts at representation learning using Transformers. Moving from LSTMs to Transformers resulted in a significant performance improvement and enabled a more effective way of fine-tuning the pre-trained models to specific tasks.

The architecture of the Transformer model was discussed in Section 6.2. The GPT model is based on a modified version of Transformer, called the Transformer Decoder [Li et al., 2018], which discards the encoder part. Therefore, instead of having a source and a target sentence for the sequence transduction model, a single sentence is given to the decoder. Instead of generating a target sequence, the objective is set as a standard language modeling in which the goal is to predict the next word given a sequence of words. GPT was also one of the first works to popularize the fine-tuning procedure (to be discussed in Section 6.6).

However, like ELMo, GPT was based on unidirectional language modeling. While reading a token, GPT can only attend to previously seen tokens in the self-attention layers. This can be very limiting for encoding sentences, since understanding a word might require processing future words in the sentence. This is despite the fact that Transformers are characterized by their self-attention layer and the capability of receiving the input sequence in parallel. What hindered a bidirectional Transformers was that bidirectional conditioning would result in a word to indirectly "see" itself in a multi-layered context.

BERT. BERT, short for Bidirectional Encoder Representations from Transformers [Devlin et al., 2019] revolutionized the NLP field in 2018/2019. Similarly to GPT, BERT is based on the Transformer architecture; however, BERT makes use of the full encoder-decoder architecture (see 6.2 for more details).

The essential improvement over GPT is that BERT provides a solution for making Transformers bidirectional. This addition enables BERT to perform a joint conditioning on both left and right context in all layers. This is achieved by changing the conventional next-word prediction objective of language modeling to a modified version, called Masked Language Modeling.

6.4.1 MASKED LANGUAGE MODELING

Before BERT, the commonly-practised language modeling objective was to predict the next token (given a sequence of tokens). Inspired by the cloze test [Taylor, 1953], BERT introduced an alternative language modeling objective to be used during the training of the model. According to this objective, instead of predicting the next to-

ken, the model is expected to guess a "maseked" token; hence, the name Masked Language Modeling (MLM). MLM randomly masks some of the tokens from the input sequence (15% for example), by replacing them with a special token, e.g., "[MASK]".

For instance, the sequence "the structure of cell membrane is known as fluid mosaic" is changed to "the structure of cell [MASK] is known [MASK] fluid mosaic". The goal would be to predict the masked (missing) tokens based on the information available from unmasked tokens in the sequence. This allows the model to have conditioning not only on the right (next token prediction) or left side (previous token prediction), but on context from both sides of the token to be predicted.

To be more precise, given that the [MASK] token only appears during the training phase, BERT employs a more comprehensive masking strategy. Instead of always replacing the token with the special [MASK] token (that has 80% chance), BERT sometimes replaces the word with a random word (10% chance) or with the same word (10%).

It is important to note that the model is not provided with the information on missing words (or words that have been replaced). The only information is the proportion of this change (e.g., 15% of the input size). It is on the model to guess these words and suggest predictions/replacements. The objective enabled BERT to capture both left and the right contexts, and to alleviate the unidirectional limitation of earlier models.

6.4.2 NEXT SENTENCE PREDICTION

In addition to the MLM objective, BERT also uses a Next Sentence Prediction (NSP) task in which the model has to identify if a given sentence can be considered as the subsequent sentence to the current sentence or not. This is motivated by the fact that to perform good in some tasks the model needs to encode relationships between sentences or to resort to information that are beyond the boundary of the sentence.

The task is a binary classification. For each sentence A the mode is provided with a second sentence B and is asked if B is the next sentence for A? To make a balanced self-training dataset, the actual next sentence is replaced with a random sentence 50% of the time. This objective helps the model in learning the relationships between sentences and was shown to be beneficial in tasks such as Natural Language Inference and Question Answering [Devlin et al., 2019].

6.4.3 TRAINING

The training objective of BERT is to minimize a combined loss function of MLM and NSP. Note that the training of BERT is carried out on pairs of sentences (given the NSP objective). In order to distinguish the two input sentences, BERT makes use of two special tokens: [CLS] and [SEP]. The [CLS] token is inserted at the beginning

and the [SEP] token in between the two sentences. The entire sequence is then fed to the encoder. The output of the [CLS] token encodes the information about the NSP objective and is used in a softmax layer for this classification.

The original BERT is trained in two settings: Base and Large. The two versions differ in their number of encoder layers, representation size and number of attention heads. BERT has given rise to several subsequent models, many of which are in fact variations of the original BERT in terms of the training objective or the number of parameters.

Subword tokenization. Unlike conventional word embedding techniques, such as Word2vec and GloVe, that take whole words as individual tokens and generate an embedding for each token, usually resulting in hundreds of thousand or millions of token embeddings, more recent models, such as BERT and GPT, segment words into subword tokens and aim at embedding these units. In practise, different tokenization algorithms are used in order to split words into subword units.

Segmenting words into subword units can bring about multiple advantages: (1) It drastically reduces the vocabulary size, from millions of tokens to dozens of thousands; (2) It provides a solution for handling out-of-vocabulary words as any unseen word can theoretically be re-constructed based on its subwords (for which embeddings are available); (3) It allows the model to share knowledge among words that have similar structures (look similar) with the hope that they share semantics, for instance, cognates across different languages or lexically-related terms in the same language.

The most commonly used tokenizers are Byte-Pair Encoding (BPE) and WordPiece tokenizer. Both tokenizers leverage a similar iterative algorithm: the vocabulary is initialized with all the characters (symbols) in a given text corpus. Then in each iteration, the vocabulary is updated with the most likely pairs of existing symbols in the vocabulary. BPE [Witten et al., 1994] takes the most frequent pair as the most "likely" one whereas WordPiece [Schuster and Nakajima, 2012] considers likelihood on the training data.

6.5 EXTENSIONS

BERT is undoubtedly a revolutionary proposal that has changed the landscape of NLP. Therefore, it is natural to expect massive waves of research on improving the model or on applying the ideas from the model or the model itself to various other tasks in NLP.