Forget hidden states?

Kacper Gibas, Sam Kierans and Billy Yan

2020

Contents

1 Why hide? **1**

2 "fastlane" **4**

3 Taming "faslane" **7**

4 citations/references **9**

Abstract

Reinventing the wheel looks like a fruitless endeavour, but it could lead to small discoveries or even uncover something profound; or you know, it could be just be painful... At least pain is the greatest teacher.

Chapter 1

Why hide?

A basic RNN [1] achieves memory by receiving 2 sets of inputs;new information input and previous hidden state(context units) and returning 2 sets of outputs the output and the new hidden state.
LSTM [2] perhaps the most successful type of RNN, uses hidden states and gates to retain information for arbitrary timesteps. This allows LSTMs ,when combined with other techniques such as transformers [3] become state of the art in practically all areas of a.i. research where it is applicable. Is implementing a hidden state essential for retaining information for many timesteps?

What we aim to do is retain information for many timesteps without relying on us implementing hidden states.

Let's use the simplest method for getting the information from past timesteps: don't use a blank version of the neural network for a new timestep. Add the value of the neuron to the input to the activation fucntion.

i.e. ignoring external input for a particular neuron

$n_{at} = f(n_{at-1} + \sum_{i=1}^{n} inputs_i \odot weights_{ai} + bias_a)$

$f(x)$ is the activation function,
n_{at-1} is the value of the neuron n_a from the previous timestep, n_{at} is the new value of n_a for this timestep,

$\sum_{i=1}^{n} inputs_i \odot weights_{ai}$ is the sum of the elements of the Hadamard or element-wise product of input to n_a: $inputs_i$ and weights connected to n_a: $weights_{ai}$, and $bias_a$ is the bias of the neuron n_a.

The only relevant difference between this and a "plain vanilla" simple neural network is that we add a n_{at-1} to the input of the activation function. This shouldn't be too much extra work in practice, the value of n_{at-1} would be in the tensor of neurons if you don't clear, deconstruct or delete it.

Can a network that works this way hold information for arbitrary timesteps? let's assume e = π = 3 (insert other ridiculous and covenient approximation/as-

sumption of choice), and let the speculating/"gedankenexperiment" begin.

One way to try and prove that it is possible for this type of network to hold information for arbitrary timesteps is to show for a neuron n_a , n_{at} can be equal to n_{at-1} . If n_{at} is equal to n_{at-1} for some non conflicting conditions for **any** timestep; we can have infinite timesteps statisfy the conditions[1]. Therefore we can hold information for arbitrary timesteps.

First we ignore $inputs_i$, $weights_{ai}$ and $bias_a$ considering just the activation function,we need to at least show: $n_{at} = f(n_{at-1})$ is possible,
sadly we are forced to use non-linear activation functions in most neural networks, or else the entire network becomes a linear regression model; a useful tool but usually not what most recurrent neural networks are trying to model.

A non-linear activation function would in theory change the input at least bit by bit through every timestep becuase of its non-linearity until it becomes useless as a memory. So back to the drawing board?

No, we are saved by the saviour of deep learning: reLU [4] !!!, for $n_{at} = \text{reLU}(n_{at-1})$ to be true we just need $n_{at-1} \geq 0$ becuase reLU is linear and has a gradient of 1 as long as $n_{at-1} \geq 0$.

Next, we need the conditions for $\sum_{i=1}^{n} inputs_i \odot weights_{ai} + bias_a = 0$,
because $n_{at} = f(n_{at-1} + \sum_{i=1}^{n} inputs_i \odot weights_{ai} + bias_a)$, would be then equal to $n_{at} = f(n_{at-1})$ which we know can be true.

So lets assume $bias_a = 0$,
then we need the conditions for $\sum_{i=1}^{n} inputs_i \odot weights_{ai}$ to equal to zero, that can be achieved if either $inputs_i$ or $weights_{ai}$ are filled with 0, if all elements of the weight tensor are 0 then that neuron recieves no useful input.

If the input tensor is filled with zeros instead for a network made of fully-connected layers that would mean no new information gets introduced to any neurons from that layer onwards;

We can get around this by assuming only some input elements are 0, and the weight elements that correspond to the non 0 input elements equal to zero
e.g. if $inputs = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ and $weights_a = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$
the sum of the Hadamard product of the 2 tensors is 0 even though not all elements are 0, because the weights that correspond to the non-zero inputs are 0. In other words we need sparse connections.

Going over the conditions needed, we need reLU, $n_{at-1} \geq 0$, $bias_a = 0$, sparse connections and elements of input tensor equal to 0, or probably more relaxed

[1] as long as the conditons the previous timestep has to satisfy do not interfere with the conditions the current timestep has to fulfil, we can then have an infinitely long chain of timesteps that all realise the conditions.

criteria of reLU or some variation of reLU, $n_{at-1} \geq 0$, $bias_a$ close to 0, sparse connections and some elements of input tensor close to 0 for senarios where the information isn't needed indefinitely.
Looking at these conditions non of them are conflicting so storing information for many timesteps appears possible.

Sadly this is still a dream model of "spherical cows in a vacuum". There is still much more to do.

Firstly, will the network learn to keep the inputs to a neuron at 0 if it needs to that information some time later? (probably?, but we don't know how to prove it with any level of rigour)

Secondly, will the network learn to keep bias close to 0 to preserve information? (probably?, but we don't know how to prove it with any level of rigour)

Thirdly, will the neural network learn to work the way we want it to in the first place? (absolutely no idea)

Chapter 2

"fastlane"

So we decided to try and design a new variant of RNN, that can fit the criteria and has a few other features. We are going to refer to these as "fastlane networks". We ditched a layered structure, instead we have an order for the neurons to "fire" in and every neuron can have connections with any other neuron(no connections to itself, no point having them).
It however has many disadvantages, e.g. parallelisation of both the forward and backward pass requires more work, in order to exploit as much parallelism as possible we have to "map" the sequence of neurons into layers.

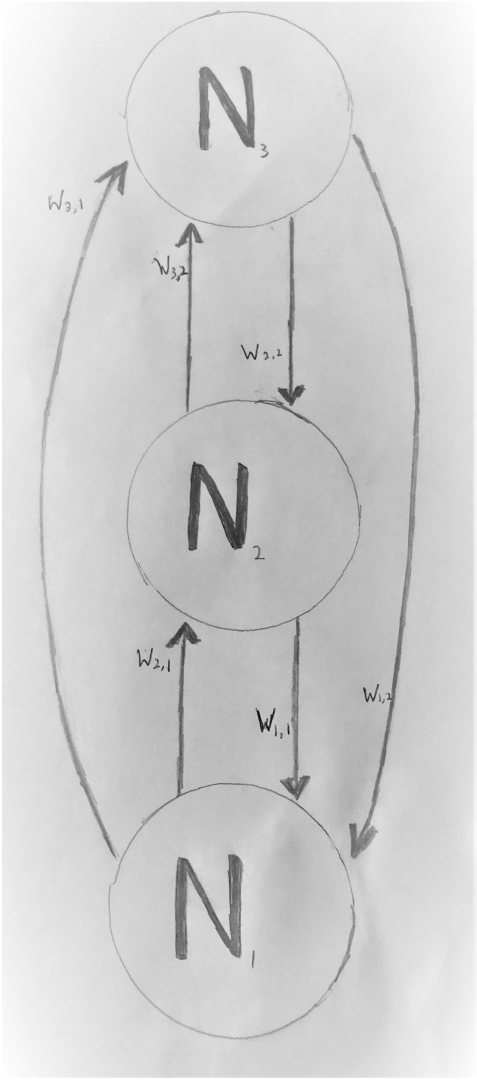
A 3 neuron fully connected example:

$$\text{neurons } N = \begin{bmatrix} N_1 \\ N_2 \\ N_3 \end{bmatrix}$$

$$\text{bias } B = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

$$\text{weights} W = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \\ W_{3,1} & W_{3,2} \end{bmatrix}$$

The connections would look like this



Ignoring external inputs a timestep t would look like this:

$N_{1,t} = reLU(N_{1,t-1} + W_{1,1} \times N_{2,t-1} + W_{1,2} \times N_{3,t-1} + B_1)$ then
$N_{2,t} = reLU(N_{2,t-1} + W_{2,1} \times N_{1,t} + W_{2,2} \times N_{3,t-1} + B_2)$ and finally
$N_{3,t} = reLU(N_{3,t-1} + W_{3,1} \times N_{1,t} + W_{3,2} \times N_{2,t} + B_3)$

"Fastlane" can have weights that connect across large sections of the neural network. We hope this can bring the abstracted information from the last neurons so that it can be made useful or maybe even abstracted again for better results[1],

[1]this doesn't count as using hidden states right?, all the information is still contained within the neural network, though in order to parallise the code without too much effort, we do make a copy of the neural network for the forward pass

we also think that it can help mitigate gradient vanishing to some degree by acting as shortcuts for information that doesn't need to be abstracted.

"Fastlane" should be sparse; the total number of possible connections scales quadractically with the total number of neuons: $n(n-1)$ where n is the total number of neurons

We think that it should be initialised with random connections and a preset "backbone" of connections; the network might start out hopeless if you rely completely on random connections, it is possible the there will input and output neurons with no path between them. A simple "backbone" could be a weight connecting every neuron with the one that fires before it, through training and new connections, if these weights become redundant whatever method used for pruning and regularisation will probably erase them.

You can also easily add new neurons into "Fastlane", just append them onto the end of your list/vecter/array and you are done, no need to worry about where it should go, we think that through new connections and pruning it will be able to eventually integrate itself into a role that is needed. Being honest one of the big reasons that we wanted to implement our own type of neural network is because we didn't want to decide and test out many different configurations of layers and convolutions e.g. and an extra layer here or an extra neuron there etc. We just initialise this "Fastlane" neural network and through training it will hopefully create its own layered structure.

Chapter 3

Taming "faslane"

We decided to try and test this network by making it echo a value.
For this test the neural network receives a list of random integers between 0 and 9 inclusive as part of the input, it is trained to remember one number and then recall it as output some timesteps later.

input	output	
2,0	NaN	
8,0	NaN	
3,0	NaN	
...	...	
5,9	5	[1]
7,0	NaN	
0,0	NaN	
...	...	
0,1	5	[2]

This started off in python, we were making the terrible mistake of doing all the hard work and math in loops in python itself and not numpy, or we couldn't to do exactly what we needed in numpy to be exact. We were already stupid enough to try and implement simple layered neural networks by ourselves in python(we wrote a poor preforming neuron by neuron recursive backpropagation function,it probably had $O(n^n)$ complexity), python was too slow, assembly too hard, Fortran too exotic and C loves to segfault. So we tried and stuck with C++(we really shot ourselves in the foot there, but it is fast). Training this neural network comes with a myriad of problems, gradient explosion, floating point overflow, dying reLU , vanishing gradi

[1] the number that the neural network needs to remember, labelled by setting the second input neuron's value to 9

[2] we tell the neural network to recall the number by setting the first input neuron to 0 an the second one to 1

(segmentation fault core dumped)
C++ was lots of fun.

Gradient explosion was a serious problem with fastlane, because we constanly added to the values of the neurons we often ended up with floating point overflow and we weren't even using 8 or 16bit floats!!.
In order to takle this problem, we first used a capped reLU i.e. reLU6 [5] or "reLU9" in our case because didn't one hot encode the input and outputs. We also applied a logistic function to the accumulated partial derivatives/gradients to each weight and bias so larger and larger values only give diminishing returns. This stops the network from learning harmful large weights and biases, we also but a cap on the absolute value of weights. As for dying reLU we are using a technique we came across from Cross Validated Stack Exchange; "NecroReLU" [6].

> I'd even go as far as to say we should ban calling them "derivatives" and start calling them something else, for example, error activation functions to not close our minds to possibilities of tinkering with them. You can actually, for example, use ReLU activation, but provide a 0.1, or something like that instead of 0 as a derivative for x<0. In a way, you then have a plain ReLU, but with neurons not being able to "die out of adaptability". I call this NecroRelu, because it's a ReLU that can't die.

- Íhor Mé

Using leaky reLU probably wouldn't work as well as reLU for "fastlane" (we never tested this), since neurons are never 0 when using leaky reLU, using "Necro reLU" seems like a good substitute here. As for gradient vanishing, we are just hoping that far reaching connections can help mitigate the problem.

Our C++ code can be found here:
`https://github.com/2win9s/Wo-Long-Zhu-Ge-/tree/master/project\%20ba_zhen_tu/echo\%20test`

The parameters (weights, biases etc.) will be stored as XML files created with C++ Boost serialisation.

Chapter 4

citations/references

[1] Elman, Jeffrey L. (1990). "Finding Structure in Time". https://crl.ucsd.edu/~elman/Papers/fsit.pdf
[2] Hochreiter, Sepp; Schmidhuber, Jürgen (1997-11-01). "Long Short-Term Memory". Neural Computation. https://www.bioinf.jku.at/publications/older/2604.pdf
[3] Polosukhin, Illia; Kaiser, Lukasz; Gomez, Aidan N.; Jones, Llion; Uszkoreit, Jakob; Parmar, Niki; Shazeer, Noam; Vaswani, Ashish (2017-06-12). "Attention Is All You Need" https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf
[4] Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011). Deep sparse rectifier neural networks http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf [5] Alex Krizhevsky, Convolutional Deep Belief Networks on CIFAR-10 http://www.cs.utoronto.ca/~kriz/conv-cifar10-aug2010.pdf [6] Íhor Mé https://stats.stackexchange.com/users/128472/ÍhorMé How does rectilinear activation function solve the vanishing gradient problem in neural networks? Cross Validated - Stack Exchange https://stats.stackexchange.com/questions/176794/how-does-rectilinear-activation-function-