LSTMs and Named Entity Recognition

RNNs and Vanishing Gradients

RNN advantages

+ Captures dependencies within a short range

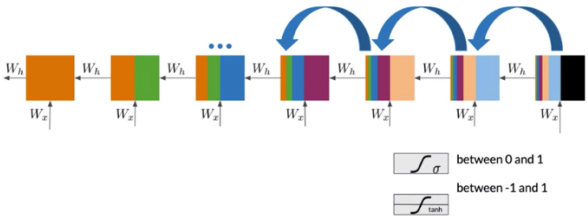
+ Takes up less RAM with other n-gram models

RNN disadvantages

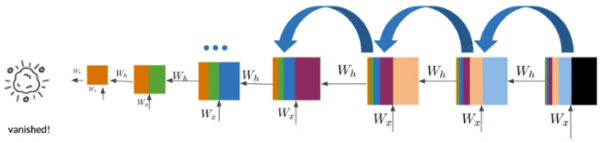
- Struggles with longer sequences

- Prone to vanishing or exploding gradients

Backpropagation through time

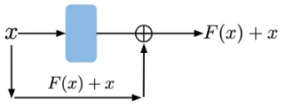


The vanishing gradient problem



Solving for vanishing or exploding gradients

- Identity RNN with ReLU activation $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ -1 ⟶ 0
- Gradient clipping 32 ⟶ 25
- Skip connections



LSTMs

* Solves vanishing gradient problem

LSTMs: a memorable solution

- Learns when to remember and when to forget
- Basic anatomy:
  - A cell state
  - A hidden state with three gates
  - Loops back again at the end of each time step
- Gates allow gradients to flow unchanged

Cell state – before conversation

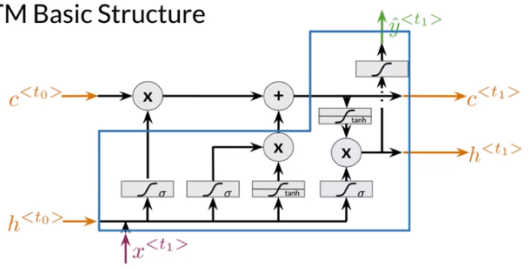
Forget gate – beginning of conversation

Input gate – thinking of a response

Output gate – responding

Updated cell state = after conversation.

LSTM Basic Structure



Application of LSTMs

* Next char prediction

* Chatbots

* Music comp

* Image captioning

* Speech recognition

LSTM Architecture

* LSTM uses a series of gates to decide which info to keep

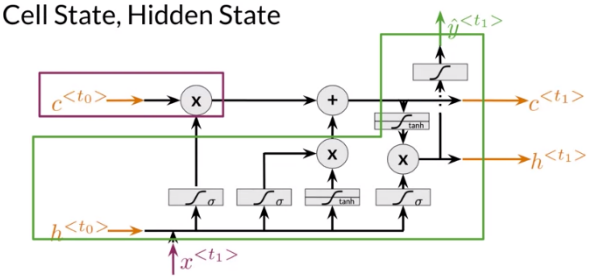
* Forget gate decides what to keep

* Input gate decides what to add

* Output gate decides what the next hidden state will be

* One time step is completed after updating the states

Cell State, Hidden State



a typical LSTM consists of a cell state and a hidden state, which holds the outputs from the cell. You can think of the cell as the memory of your network carrying all the relevant information down the sequence. As the cell travels, each gate adds or removes information from the cell state.

The gates make up the hidden states of your LSTM. They contain activation functions and element-wise operations. LSTMs typically have three gates: the forget gate, the input gate, and the output gate.
The first gate is the forget gate, which as you may have guessed, decides which information from the previous cell state and current input should be kept or tossed out. It does this with a sigmoid function, which squeezes each value from the cell states between zero and one. In this case, a value closer to zero indicates it should be thrown away, and a value closer to one indicates it should be kept. Now that you have the values indicating what to keep and what to throw out, you need to update the cell states.

This is where the input gate comes in. The input gate is actually two layers, a sigmoid layer and a tanh layer. The sigmoid takes the previous hidden states and current inputs and chooses which values to update by assigning zero or one to each value. The closer to one, the higher its importance. The tanh layer also takes the hidden states and current inputs and squeezes the values between negative one and one. This helps to regulate the flow of information in your network. Then the outputs of this sigmoid and tanh layers are multiplied. Now your model has what it needs to calculate a new cell state. Now, you can recalculate the values in the cell states. For this step, take the values provided by the forget gate and multiply them element-wise by the values in the cell state. Then take that result and do an element-wise addition with the values provided by the input gate. That's it. You've updated your cell states. Onward to the third and final gate.

Now that you've updated your cell states, the output gate will decide what your next hidden state should be. Remember that your cell state acts as the memory, and the hidden state is what makes predictions. To arrive at a new hidden state, the output gate takes the previous hidden state and the current input and passes them through a sigmoid layer. Then your freshly updated cell state is passed through a tanh function. Similarly to the input gate, the outputs from the sigmoid layer are multiplied by the tanh layer's output, and this makes the final decision on what will be included in the output. Now you have an updated cell state, an updated hidden state, and one completed timestep. Your model can move on to the next timestep.

The forget gate decides what to keep.
The input gate decides what to add.
The output gate decides what the next hidden state will be.
After each of these gates performs its function and the states are updated, you will have completed one timestep.

Named Entity Recognition

Entities – Country, Org, geopolitical, time, artifact, person, Other

* Search Engine Efficiency

* Recommendation engines

* Customer service

* Automatic trading

Training NERs Data Processing

1. Assign each class a number. B-per = 1, B-geo = 2, B-tim = 3
2. Assign each word a number.

Processing data for NERs

- Assign each class a number
- Assign each word a number

Sharon flew to Miami last Friday.

[4282, 853, 187, 5388, 2894, 7]

B-per O O B-geo O B-tim

Token padding

For LSTMs, all sequences need to be the same size.

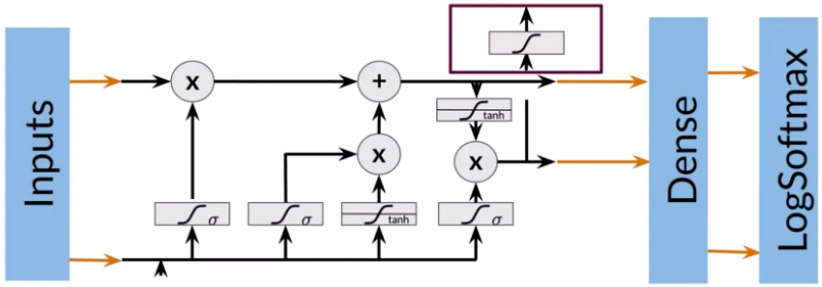
- Set sequence length to a certain number
- Use the <PAD> token to fill empty spaces

Training the NER

1. Create a tensor for each input and its corresponding number
2. Put them in a batch ⟶ 64, 128, 256, 512 ...
3. Feed it into an LSTM unit
4. Run the output through a dense layer
5. Predict using a log softmax over K classes

Log softmax gets better in numerical performance and gradients optimization. K corresponds to the number of possible output

Training the NER




```
model = tl.Serial(
    tl.Embedding(),
    tl.LSTM(),
    tl.Dense()
    tl.LogSoftmax()
  )
```

Computing Accuracy

Evaluating the model

* Pass test set through the model

* Get arg max across the prediction array

* Mask padded tokens

* Compare output against test labels

* If padding tokens remember to mask then when computing accuracy.

Evaluating the model in Python

```
def evaluate_model(test_sentences, test_labels, model):
    pred = model(test_sentences)
    outputs = np.argmax(pred, axis=2)
    mask = ...
    accuracy = np.sum(outputs==test_labels)/float(np.sum(mask))

    return accuracy
```

REFERENCE:

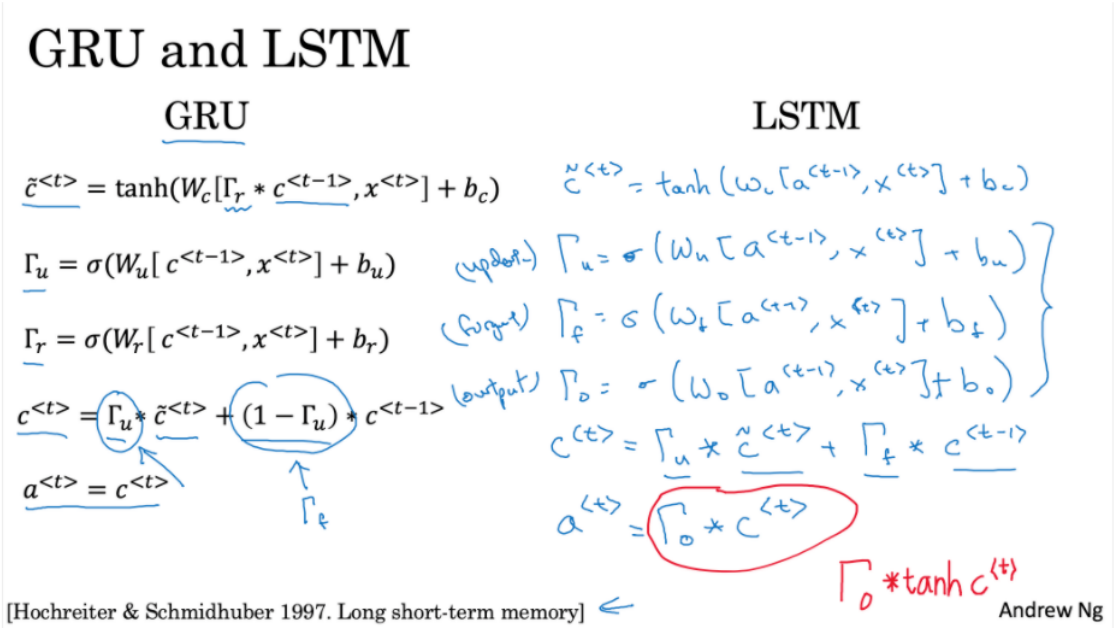
Intro to optimization in deep learning – Gradient descent

https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

$$a^{<t>} = \Gamma_o * \tanh(c^{<t>})$$

See also the correction in the screenshot below (in red ink).



$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

See the correction in red.