Outline

- RNNs and vanishing/exploding gradients
- Solutions

RNNs: Advantages

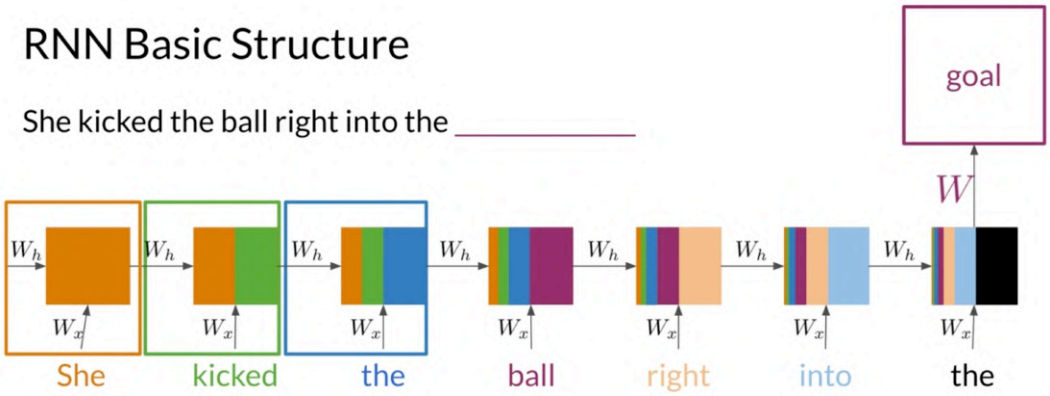
- ✚ Captures dependencies within a short range
- ✚ Takes up less RAM than other n-gram models

RNNs: Disadvantages

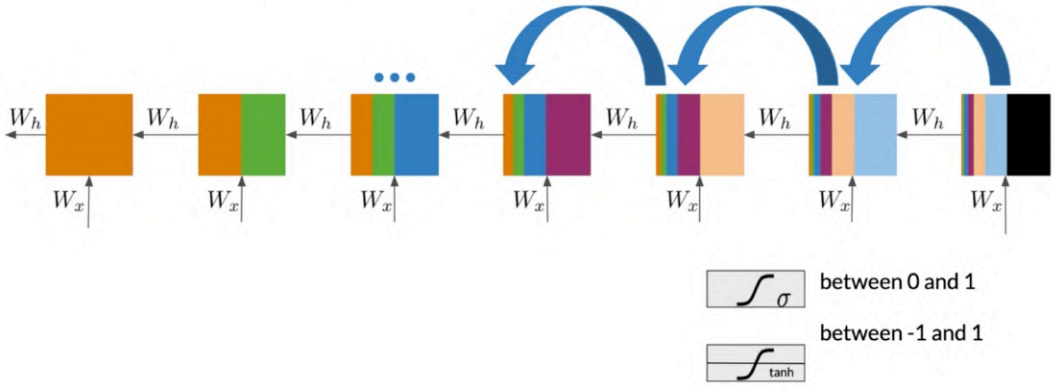
- ━ Struggles with longer sequences
- ━ Prone to vanishing or exploding gradients

RNN Basic Structure

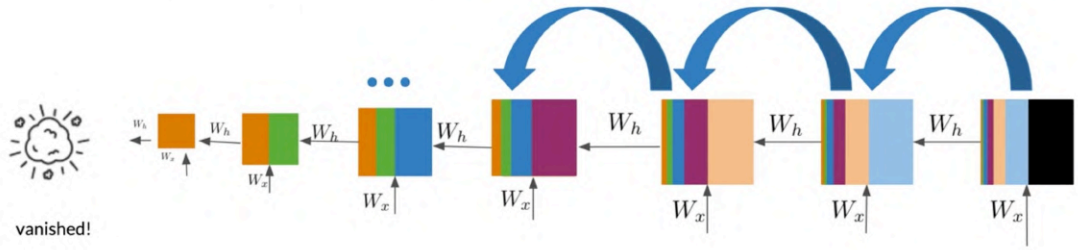
She kicked the ball right into the _____



Backpropagation through time



The vanishing gradient problem



Solving for vanishing or exploding gradients

- Identity RNN with ReLU activation

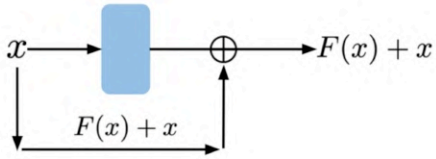
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

-1 ⟶ 0

- Gradient clipping

32 ⟶ 25

- Skip connections



You can deal with vanishing gradients by initializing your weights to the identity matrix, which carries values of 1 along the main diagonal and 0 everywhere else, and using a ReLU activation. What this essentially does is copy the previous hidden states, add information from the current inputs, and replace any negative values with 0. This has the effects of encouraging your network to stay close to the values in the identity matrix, which act like 1s during matrix multiplication. This method is referred to, unsurprisingly, as an identity RNN. The identity RNN approach only works for vanishing gradients through as the derivative of ReLU is equal to 1 for all values greater than 0.

To account for values growing exponentially, you can perform gradient clipping. To clip your gradients, simply choose a relevant value that you clip the gradients to, say 25. Using this technique, any value greater than 25 will be clipped to 25, this serves to limit the magnitude of the gradients.

Finally, skip connections provide a direct connection to the earlier layers. This effectively skips over the activation functions and adds the value from your initial inputs x to your outputs, or F(x) + x. This way, activations from early layers have more influence over the cost function.

Outline

- Meet the Long short-term memory unit!
- LSTM architecture
- Applications

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

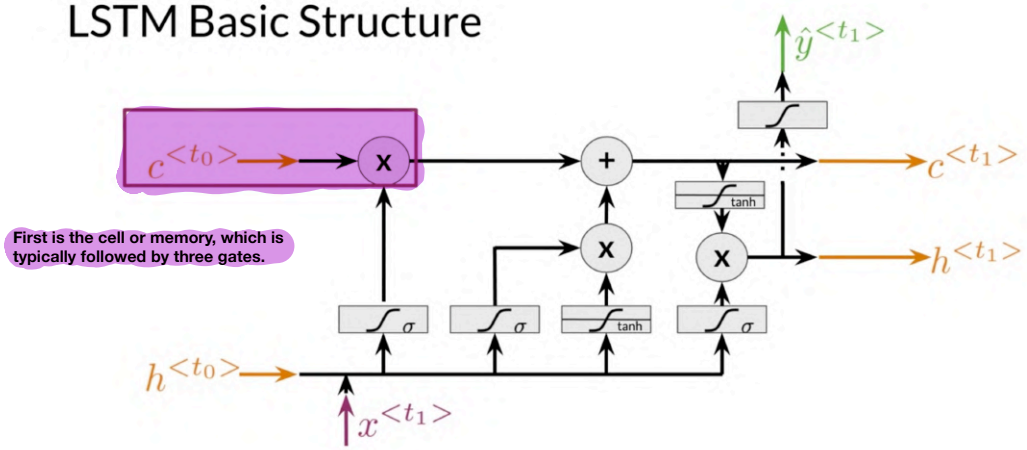
LSTMs: Based on previous understanding

Cell state = before conversation
Forget gate = beginning of conversation

Input gate = thinking of a response
Output gate = responding

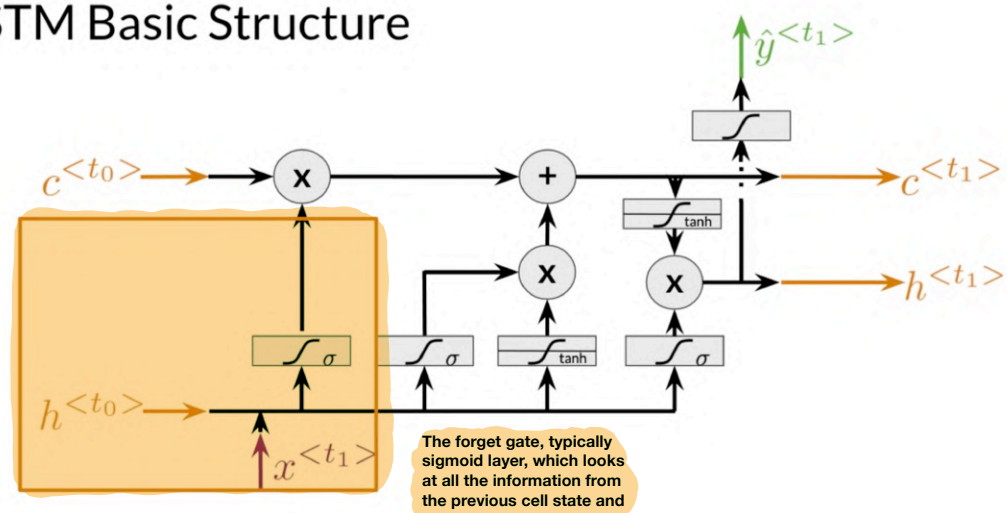
Updated cell state = after conversation

LSTM Basic Structure



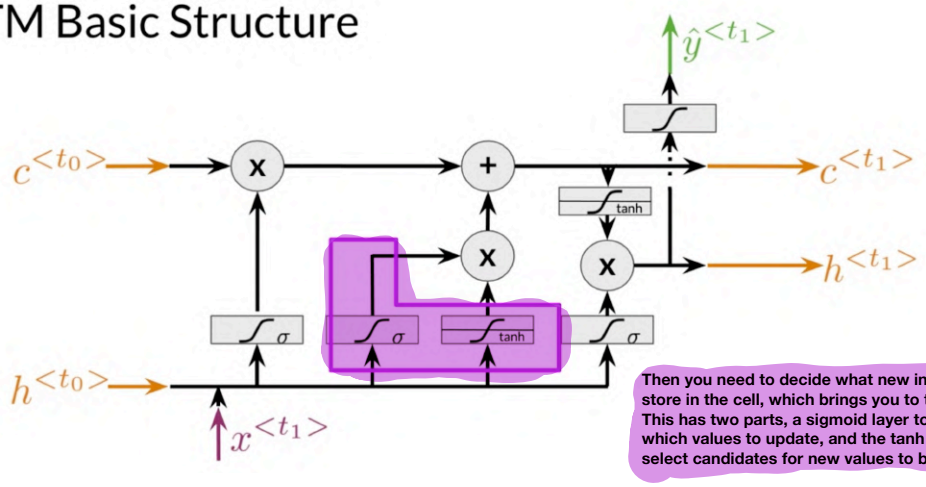
First is the cell or memory, which is typically followed by three gates.

LSTM Basic Structure



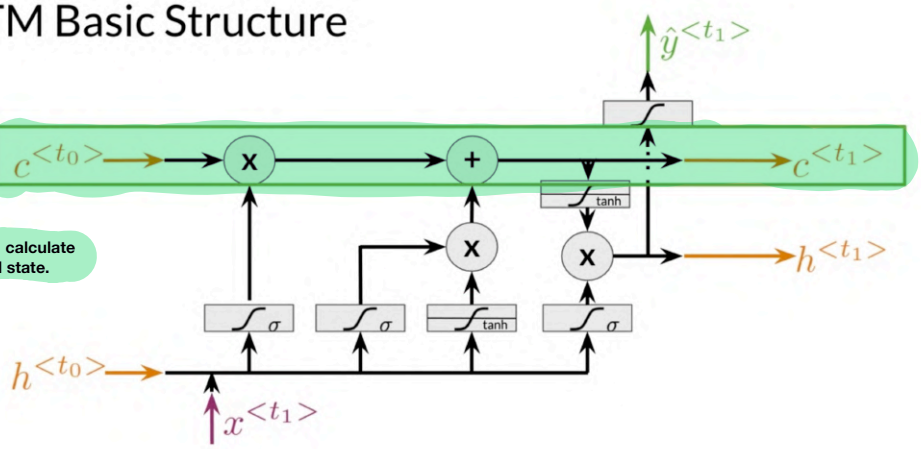
The forget gate, typically sigmoid layer, which looks at all the information from the previous cell state and decides what to throw away.

LSTM Basic Structure



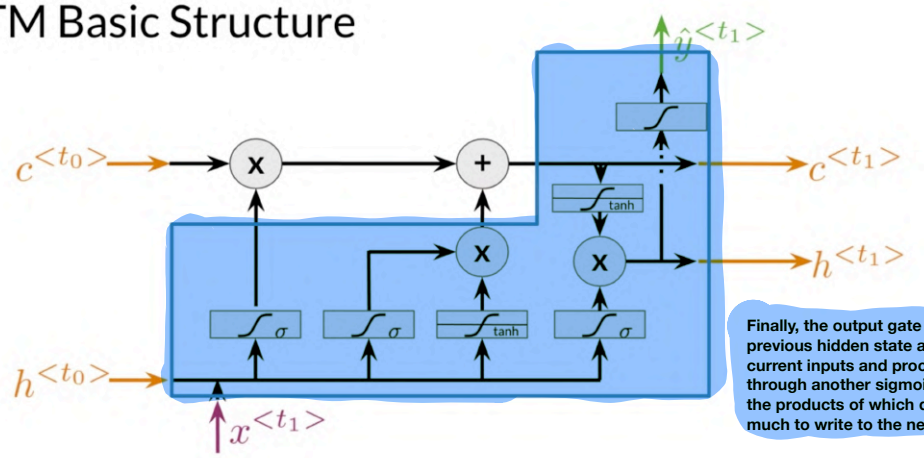
Then you need to decide what new information to store in the cell, which brings you to the input gate. This has two parts, a sigmoid layer to choose which values to update, and the tanh layer to select candidates for new values to be added

LSTM Basic Structure



Then you will calculate your new cell state.

LSTM Basic Structure



Finally, the output gate takes the previous hidden state along with the current inputs and processes them through another sigmoid and tanh layer, the products of which decides how much to write to the next hidden state.

Applications of LSTMs

Next-character prediction

Chatbots

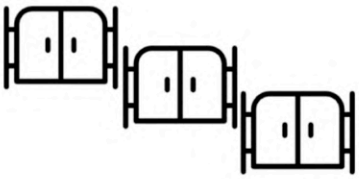
Music composition

Image captioning

Speech recognition

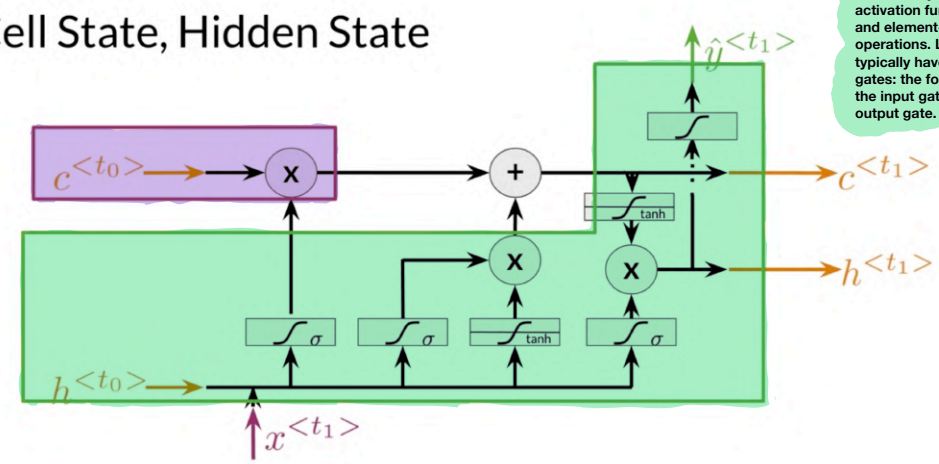
Summary

- LSTMs offer a solution to vanishing gradients
- Typical LSTMs have a cell and three gates:
  - Forget gate
  - Input gate
  - Output gate



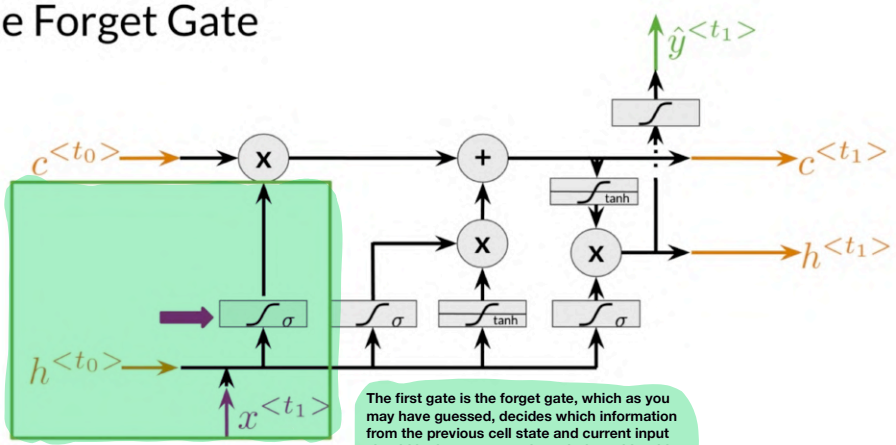
Cell State, Hidden State

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.

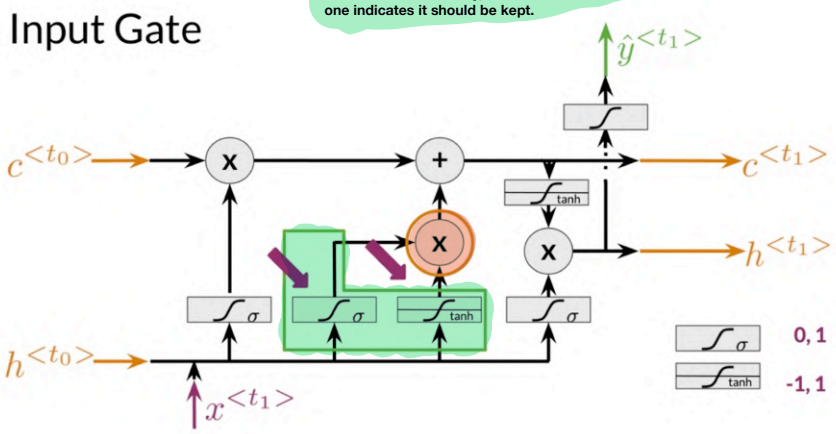
The Forget Gate



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.

The Input Gate

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.

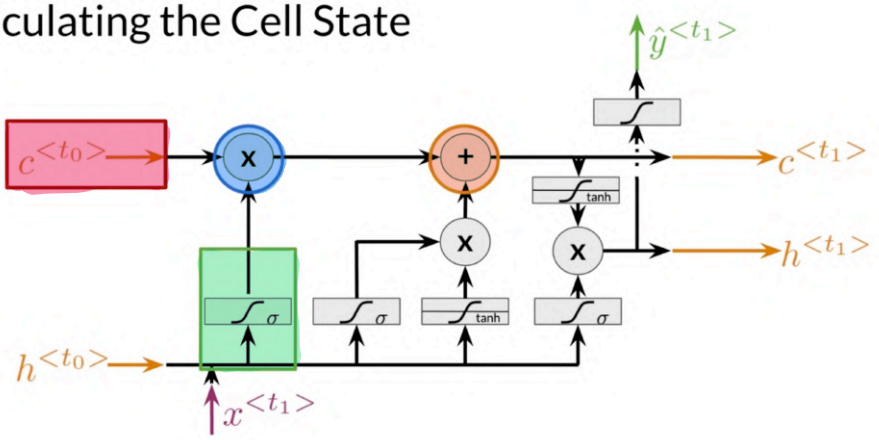


Then the outputs of this sigmoid and tanh layers are multiplied. Now your model has what it needs to calculate a new cell state.

Calculating the Cell State

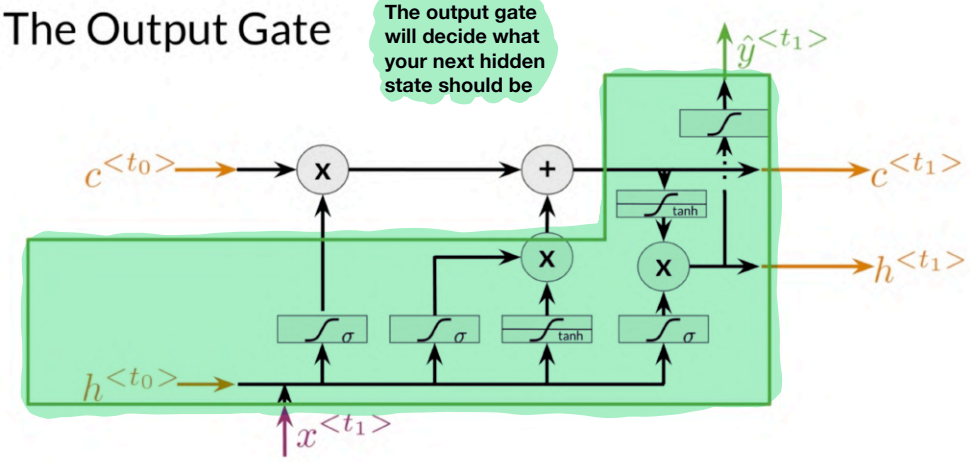
Now, you can recalculate the values in the cell states.

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.

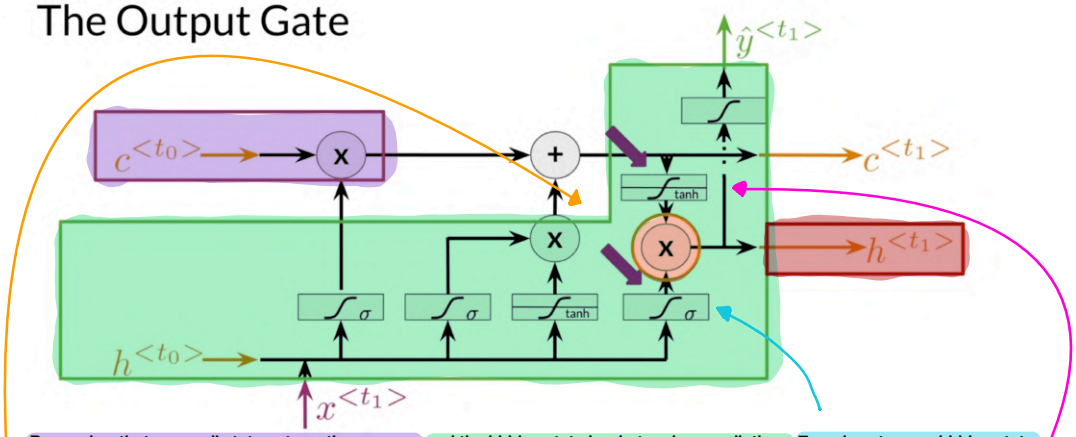


The Output Gate

The output gate will decide what your next hidden state should be



The Output Gate



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.

Summary

- LSTMs use a series of gates to decide which information 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

What is Named Entity Recognition?

- Locates and extracts predefined entities from text
- Places, organizations, names, time and dates

Types of Entities

Thailand:
Geographical

Google:
Organization

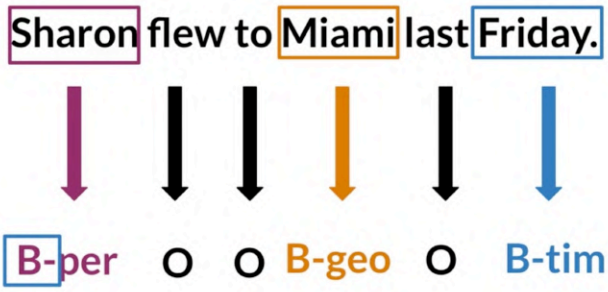
More Types of Entities

December:
Time Indicator

Egyptian statue:
Artifact

Barack Obama:
Person

Example of a labeled sentence



Applications of NER systems

- Search engine efficiency
- Recommendation engines
- Customer service
- Automatic trading

Outline

- Convert words and entity classes into arrays
- Token padding
- Create a data generator

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.

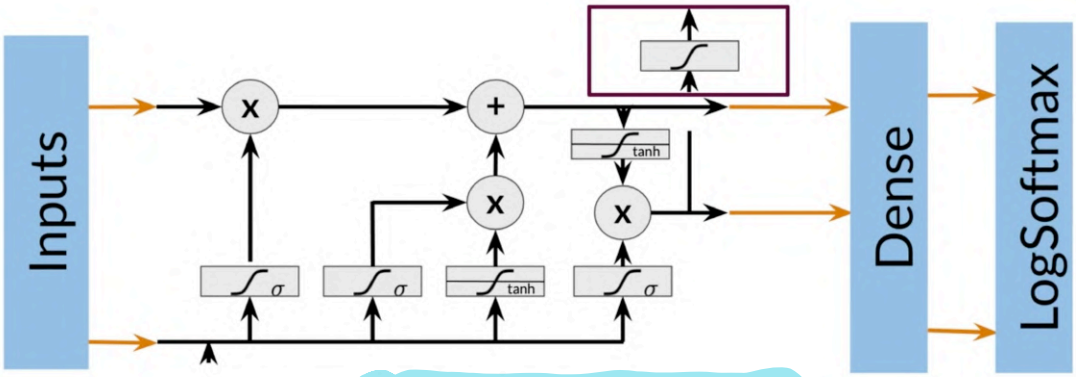
- Set sequence length to a certain number
- Use the <PAD> token to fill empty spaces

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

Using log softmax instead of softmax is important here, because log softmax gets better in numerical performance and gradients optimization.

Training the NER



Here are the inputs, which include the arrays you created when processing the data. You feed them through an LSTM layer with its series of activations and element wise operations, run the outputs of that through a final dense layer, which is a linear operation on the input vectors. And then you compute a log softmax to get the corresponding outputs.

Layers in Trax

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

Summary

- Convert words and entities into same-length numerical arrays
- Train in batches for faster processing
- Run the output through a final layer and activation

Evaluating the model

1. Pass test set through the model
2. Get arg max across the prediction array
3. Mask padded tokens
4. Compare outputs against test labels

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
```

Note that the arg max function takes an access parameter. To indicate this accurately, you'll need to consider the dimensions of your array. Here it's set to 2. The mask variable is where you identify any token IDs you need to skip over during evaluation. One token you might want to skip is your PAD token.

Finally, the accuracy metric is computed by taking the sum of all your test labels, and dividing it by the sum of all your mask tokens.

Summary

- If padding tokens, remember to mask them when computing accuracy
- Coding assignment!