Lesson 8: Recurrent Neural Networks

Notes by Jvinniec

- Introduction
    - RNN vs. LSTM
- Basics of LSTMs
    - LSTM Architecture
        - Learn Gate
        - Forget Gate
        - Remember Gate
        - Use Gate
    - Alternative Architectures
        - Gated Recurrent Unit (GRU)
        - Peephole Connections
- RNNs in PyTorch
    - nn.RNN:
        - Construction:
    - nn.LSTM
    - Sequence Batching

Introduction

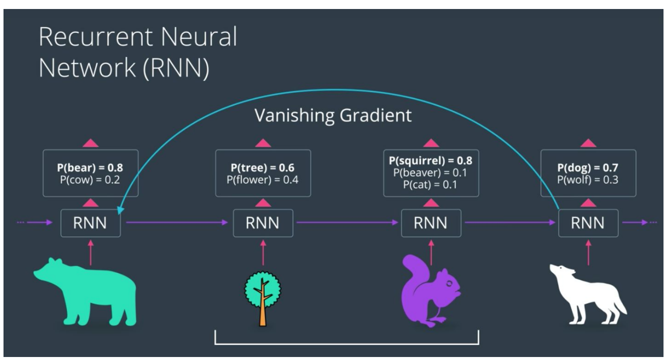
This lesson covers **recurrent neural networks** (RNNs) and **long short-term memory** (LSTM). Some helpful links on the subject:

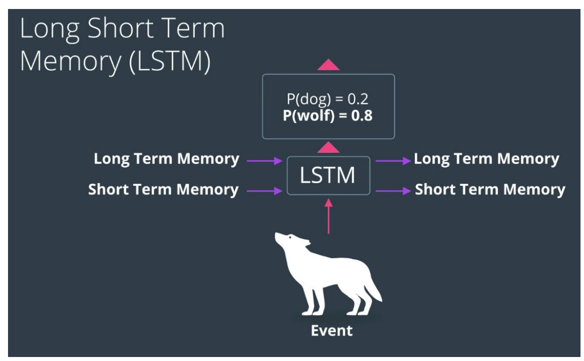
- Understanding LSTM Networks (Chris Olah's blog)
- Exploring LSTMs (Edwin Chen's blog)
- The Unreasonable Effectiveness of Recurrent Neural Networks (Andrej Karpathy blog)
- CS231n Lecture 10: RNNs, Image Captioning, LSTM (Andrej Karpathy video lecture)

RNN vs. LSTM

RNNs work by using the output from previous passes in the network as an additional input to the network. For example, if we have an image of a wolf, the network might think it is a dog. However, if it knows that the previous inputs were all wild animals (such as a fox and a bear) then it could use that information to conclude that the image is more likely to be a wolf.

One problem with RNNs is that with many inputs, the network tends to "forget" information after a few passes (i.e. there's a strong vanishing gradient problem). An input will have more impact on the input immediately following it than say 5 inputs later. This causes RNNs to have a "short-term" memory. This is where LSTMs are important.





Basics of LSTMs

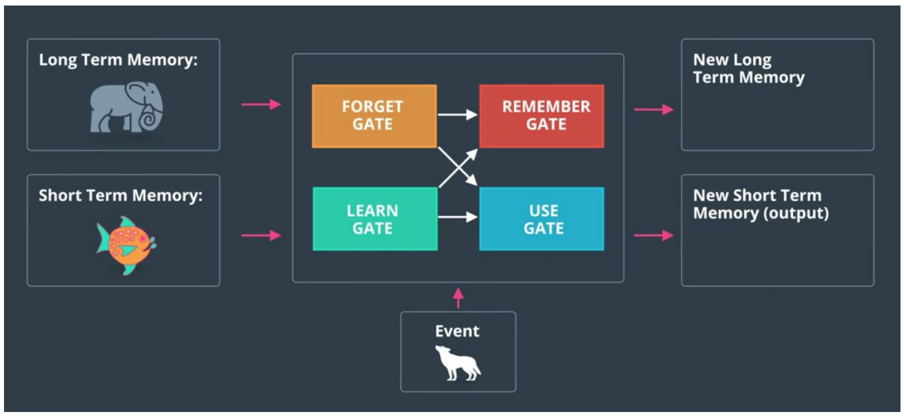
LSTMs consist of 3 types of inputs:

- **Long-term memory**: This is the information that is maintained over a long time
- **Short-term memory**: Relevant information about what our network has recently seen
- Event: The event or input that we are trying to classify

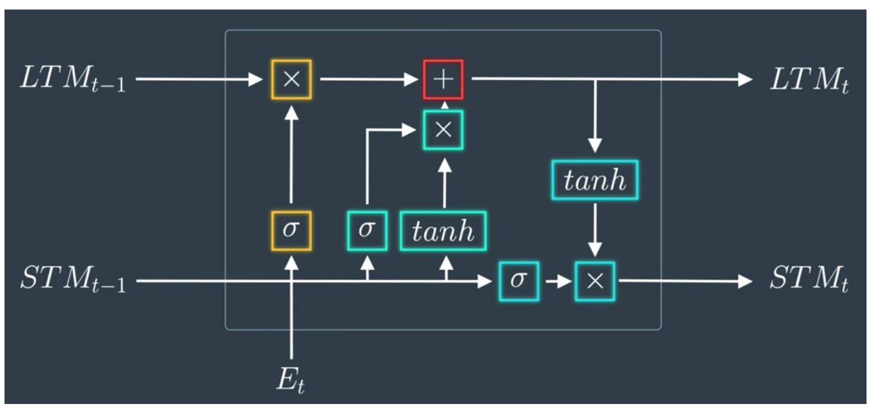
LSTM Architecture

The network we build also has a series of "gates":

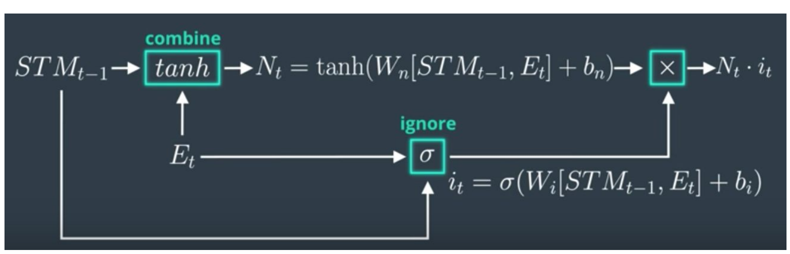
- Learn Gate: Processes the short-term memory and the input to combine what we've recently learned and remove unnecessary information
- Forget Gate: Processes the long-term memory to remove information deemed unimportant
- Remember Gate: Combine outputs from "forget gate" and "learn gate" to generate new long-term memory
- Use Gate: Combine outputs from "forget gate" and "learn gate" to generate the output for the most recent input as well as the new short-term memory



The actual architecture of an LSTM looks like the following (each gate will be covered in more detail later):



Learn Gate



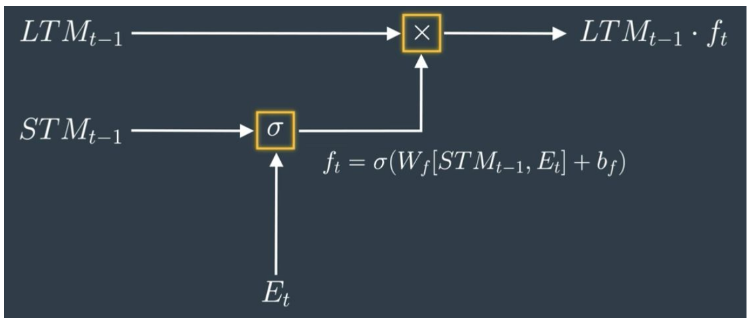
The "**learn gate**" does the following:

- Computes a "combine" matrix by:

    - Taking the short-term memory (output from previous layer) and the new input and joins them together
    - Multiplies the resulting vector by a weight and adds a bias
    - Takes the ‘tanh’ of the combined vector
- Computes an “ignore” matrix by:
    - Join the short-term memory and the new input vectors
    - Multiply by a weight and add a bias
    - Take the “sigmoid” of the combined vector
- Multiply the “combine” and “ignore” matrices

Forget Gate

Takes the long-term memory and decides what to keep and what to forget.



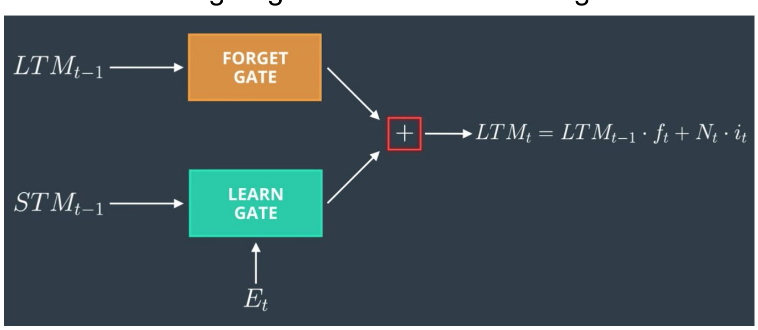
The “**forget gate**” is computed in the following way:

- Combines the short-term memory with the new input
    - Multiply by a weight and add a bias
    - Take sigmoid of resulting vector
- Multiply the resulting combined matrix by the long-term memory

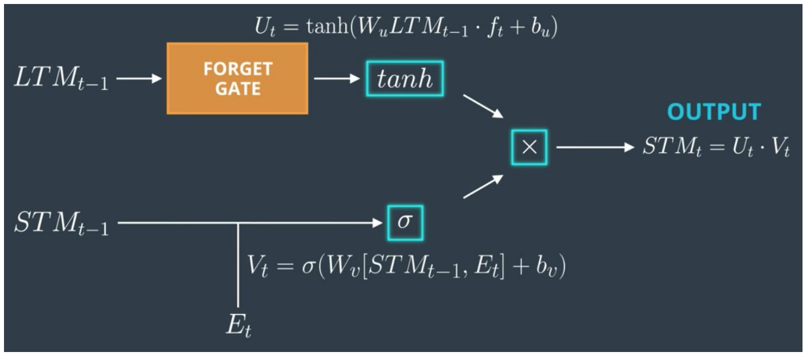
Remember Gate

The “**remember gate**” is the simplest of the four gates. It takes:

- Output from “learn” and “forget” gates and adds them together



Use Gate



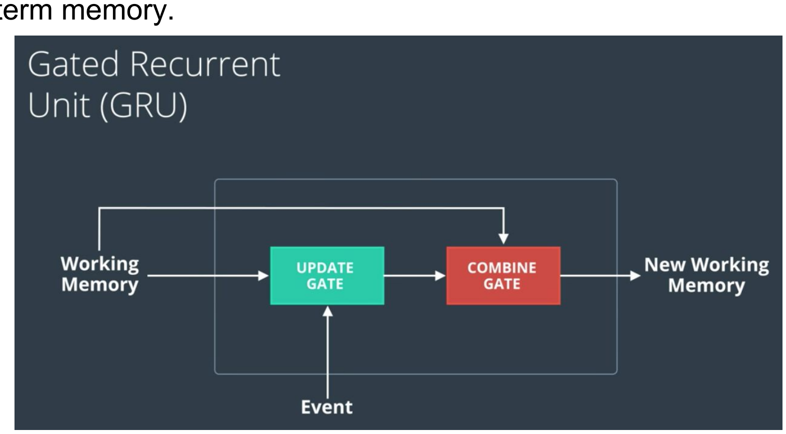
The "**use gate**" is the output for the latest input. It does the following:

- Apply a tanh function for the output from the "forget" gate
- Computes an additional matrix by a similar process to the "ignore" matrix (but with its own bias and weights):
    - Join the short-term memory and the new input vectors
    - Multiply by a weight and add a bias
    - Take the "sigmoid" of the combined vector
- Multiplies the above two matrices to form the output
    - This output also serves as the new short-term memory

Alternative Architectures

Gated Recurrent Unit (GRU)

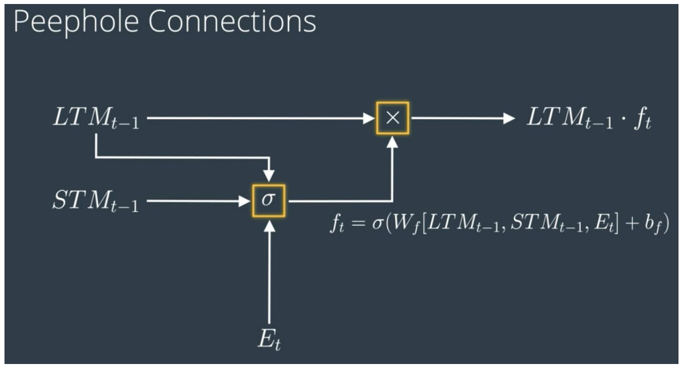
A GRU uses fewer gates by combining the learn and forget gate into an "**update**" gate and runs this through a "**combine**" gate. This is then translated into a single working memory, instead of a long and short-term memory.



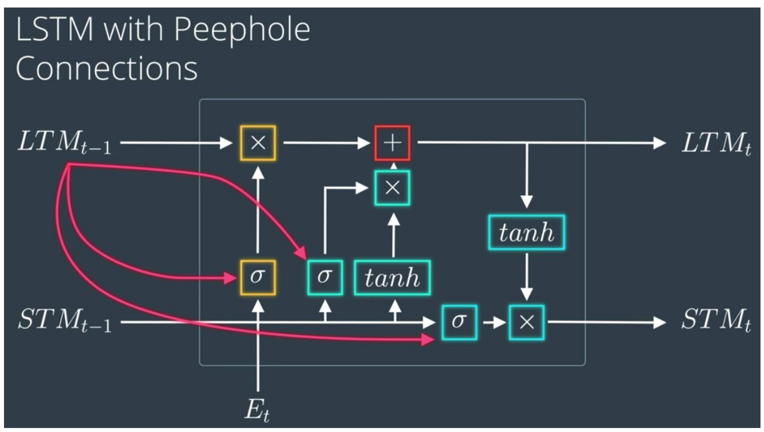
More details are available on [this blog post](#).

Peephole Connections

A question might be asked “Why doesn’t the long-term memory have a say in what information is remembered?” This can be done by adding a connection between the LSTM and the sigmoid input.



In fact, we can apply this to all parts of our network in a similar way:



RNNs in PyTorch

An RNN layer in PyTorch is constructed using the “`nn.RNN()`” function:

nn.RNN:

[Constructor parameters](#):

- **input_size**: number of expected features in x
- **hidden_size**: number of features in hidden state h
- **num_layers**: number of recurrent layers (default=1)
- **nonlinearity**: non-linearity to use (‘tanh’ or ‘relu’, default=’tanh’)
- **bias**: if False, layer does not use bias weights (default=True)
- **batch_first**: If True, input and output tensors are provided as [batch, seq, feature], (default=False)

- **dropout**: If non-zero introduces a Dropout layer (with probability=dropout value) on the outputs of each RNN layer except last layer (default=0)
- **bidirectional**: If True, becomes a bidirectional RNN (default=False)

Network inputs:

- **input**: tensor containing the features of the input sequence (has shape [seq_len, batch, input_size], unless *batch_first=True*)
- **h_0**: tensor containing initial hidden state for each element in the batch (has shape [num_layers*num_directions, batch, hidden_size])

Network outputs:

- **output**: tensor containing output features from last layer of RNN for each step (has shape [seq_len, batch, num_directions*hidden_size]).
- **h_n**: tensor containing the hidden state for t=seq_len (has shape [num_layers*num_directions, batch, hidden_size]).

Construction:

A very simple class that uses an RNN to predict a single value would be as follows:

```
class RNN(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(RNN, self).__init__()

        self.hidden_dim=hidden_dim

        # define an RNN with specified parameters
        # batch_first means that the first dim of the
        # input and output will be the batch_size
        self.rnn = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)

        # last, fully-connected layer to actually make prediction
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x, hidden):
        # x (batch_size, seq_length, input_size)
        # hidden (n_layers, batch_size, hidden_dim)
        # r_out (batch_size, time_step, hidden_size)
        batch_size = x.size(0)

        # get RNN outputs
        r_out, hidden = self.rnn(x, hidden)
        # shape output to be (batch_size*seq_length, hidden_dim)
```

```
        r_out = r_out.view(-1, self.hidden_dim)

        # get final output
        output = self.fc(r_out)

        return output, hidden
```

nn.LSTM

Construction parameters:

- **input_size**: Number of expected features in input *x*
- **hidden_size**: Number of features in the hidden state *h*
- **num_layers**: Number of LSTM layers to use (default=1)
- **bias**: If 'False', then layer does not use bias weights (default=True)
- **batch_first**: If 'True', then the input and output tensors are provided as (batch, seq, feature). (default='False')
- **dropout**: If non-zero, introduces a Dropout layer on the outputs of each LSTM layer (except the last layer) with probability=*dropout*. (default=0)
- **bidirectional**: If 'True', becomes a bidirectional LSTM (default=False)

Network inputs (of the form `model(input, (h_0, c_0))`):

- **input** (seq_len, batch, input_size): Tensor containing features of input sequence
- **h_0** (num_layers*num_directions, batch, hidden_size): tensor containing initial hidden state for each element in the batch.
- **c_0** (num_layers*num_directions, batch, hidden_size): Tensor containing initial cell state for each element in the batch.

Model outputs (of the form `output, (h_n, c_n)`)

- **output** (seq_len, batch, num_directions*hidden_size): Tensor containing the output features (h_t) from the last layer of the LSTM, for each t.
- **h_n** (num_layers*num_directions, batch, hidden_size): tensor containing the hidden state for t=seq_len.
- **c_n** (num_layers*num_directions, batch, hidden_size): Tensor containing the cell state for t=seq_len.

Sequence Batching

For RNNs we are going to take the data and split it into batches. To visualize this, assume we have our input data in the following format, a sequence of numbers:

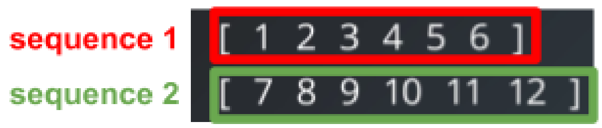
[1 2 3 4 5 6 7 8 9 10 11 12]

We can split this data into two sequences in order to speed up our training:

```
[ 1 2 3 4 5 6 ]
[ 7 8 9 10 11 12 ]
```

We have a few properties for the data:

- **batch_size**: Number of individual training samples in a batch. This will be the number of simultaneous sequences that are fed into our RNN model during training. In the above example, we split our data into two sequences, i.e. batch_size=2

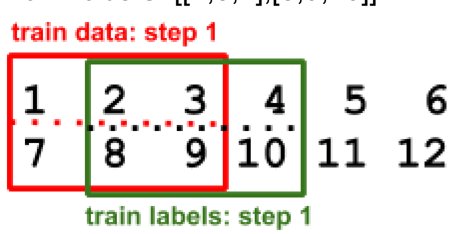


- **seq_length**: Number of entries we are feeding into our RNN, for example if we feed our values in 3 at a time we have seq_length=3. In this case the first sequence in the above example would be split in the following way:

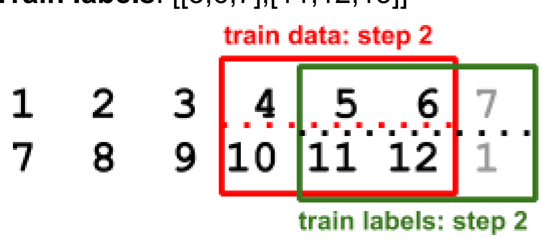
- **n_batches**: Number of individual batches we have split our data into for training purposes. This is equivalent to the number of entries in a full sequence divided by "seq_length".

Now, when we pass the data to the RNN, a training step will pass the first "seq_length" entries from each batch. Continuing our example from above:

- Step 1:
  - **Train data**: [[1,2,3],[7,8,9]]
  - **Train labels**: [[2,3,4],[8,9,10]]



- **Step 2**
  - **Train data**: [[4,5,6],[10,11,12]]
  - **Train labels**: [[5,6,7],[11,12,13]]



Note that we assume our data is cyclic, that is, after the last entry the data wraps around back to the first entry in the array.