EE-559 – Deep learning

11.2. LSTM and GRU

François Fleuret

https://fleuret.org/ee559/

December 14, 2018

The Long-Short Term Memory unit (LSTM) by Hochreiter and Schmidhuber (1997), is a recurrent network with a gating of the form

$$c_t = c_{t-1} + i_t \odot g_t$$

where c_t is a recurrent state, i_t is a gating function and g_t is a full update. This assures that the derivatives of the loss wrt c_t does not vanish.

It is noteworthy that this model implemented 20 years before the resnets of He et al. (2015) uses the exact same strategy to deal with depth.

This original architecture was improved with a forget gate (Gers et al., 2000), resulting in the standard LSTM in use.

In what follows we consider notation and variant from Jozefowicz et al. (2015).

The recurrent state is composed of a "cell state" c_t and an "output state" h_t. Gate f_t modulates if the cell state should be forgotten, i_t if the new update should be taken into account, and o_t if the output state should be reset.

$$
\begin{aligned}
f_t &= \text{sigm}\left(W_{(x\ f)}x_t + W_{(h\ f)}h_{t-1} + b_{(f)}\right) && \text{(forget gate)} \\
i_t &= \text{sigm}\left(W_{(x\ i)}x_t + W_{(h\ i)}h_{t-1} + b_{(i)}\right) && \text{(input gate)} \\
g_t &= \tanh\left(W_{(x\ c)}x_t + W_{(h\ c)}h_{t-1} + b_{(c)}\right) && \text{(full cell state update)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t && \text{(cell state)} \\
o_t &= \text{sigm}\left(W_{(x\ o)}x_t + W_{(h\ o)}h_{t-1} + b_{(o)}\right) && \text{(output gate)} \\
h_t &= o_t \odot \tanh(c_t) && \text{(output state)}
\end{aligned}
$$

As pointed out by Gers et al. (2000), the forget bias $b_{(f)}$ should be initialized with large values so that initially $f_t \simeq 1$ and the gating has no effect.

This model was extended by Gers et al. (2003) with "peephole connections" that allow gates to depend on c_{t-1}.
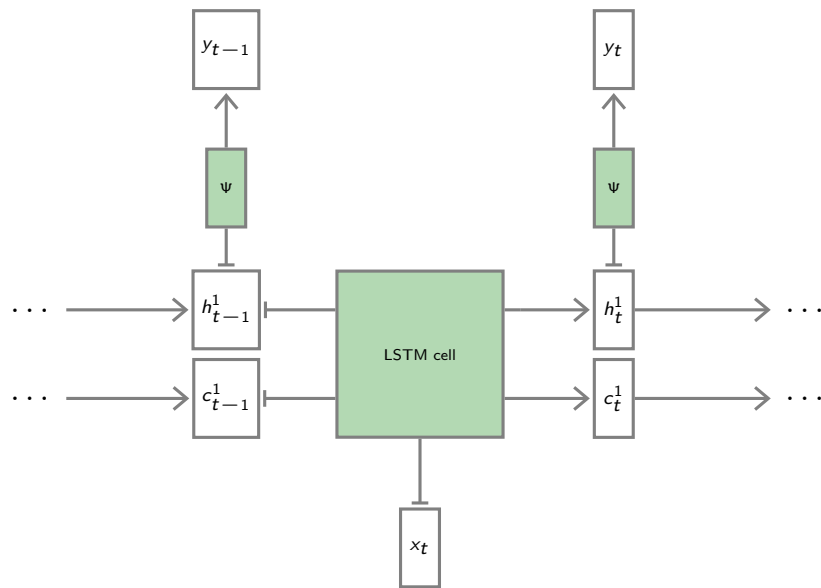
⚠ Prediction is done from the h_t state, hence called the **output** state.

Several such "cells" can be combined to create a multi-layer LSTM.
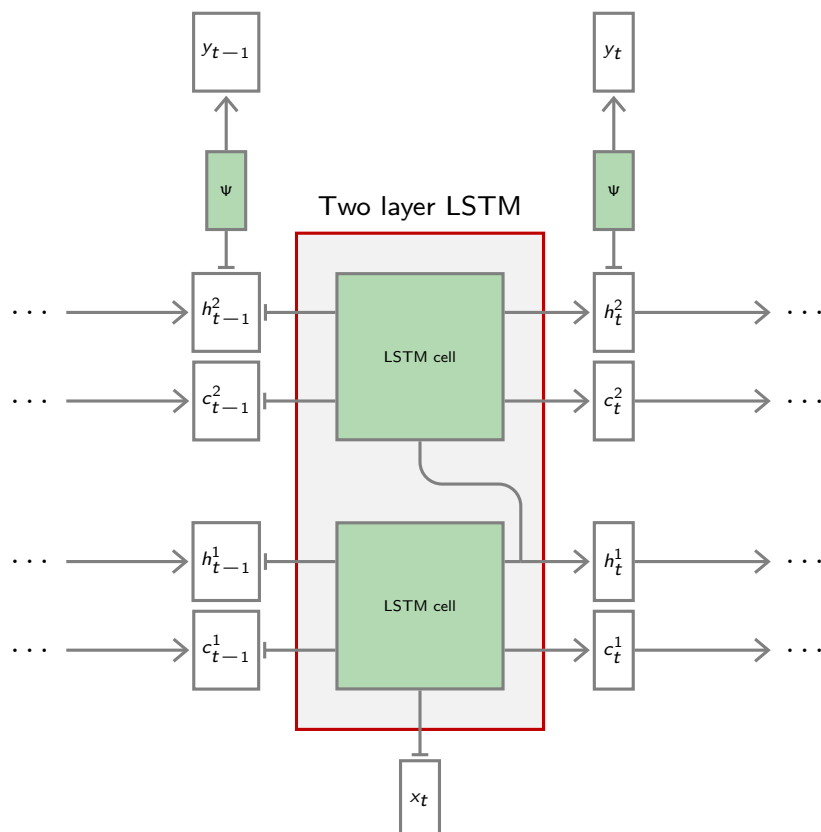
PyTorch's `torch.nn.LSTM` implements this model.

Its processes several sequences, and returns two tensors, with D the number of layers and T the sequence length:

- the outputs for all the layers at the last time step: h_T^1 and h_T^D, and
- the outputs of the last layer at each time step: $h_1^D, \ldots, h_T^D$.

The initial recurrent states $h_0^1, \ldots, h_0^D$ and $c_0^1, \ldots, c_0^D$ can also be specified.

PyTorch's RNNs can process batches of sequences of same length, that can be encoded in a regular tensor, or batches of sequences of various lengths using the type `nn.utils.rnn.PackedSequence`.

Such an object can be created with `nn.utils.rnn.pack_padded_sequence`:

```
>>> from torch.nn.utils.rnn import pack_padded_sequence
>>> pack_padded_sequence(torch.tensor([[[ 1. ], [ 2. ]],
...                                     [[ 3. ], [ 4. ]],
...                                     [[ 5. ], [ 0. ]]]),
...                      [3, 2])
PackedSequence(data=tensor([[ 1.],
        [ 2.],
        [ 3.],
        [ 4.],
        [ 5.]]), batch_sizes=tensor([ 2,  2,  1]))
```

⚠️   The sequences must be sorted by decreasing lengths.

`nn.utils.rnn.pad_packed_sequence` converts back to a padded tensor.

```
class LSTMNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, num_layers, dim_output):
        super(LSTMNet, self).__init__()
        self.lstm = nn.LSTM(input_size = dim_input,
                            hidden_size = dim_recurrent,
                            num_layers = num_layers)
        self.fc_o2y = nn.Linear(dim_recurrent, dim_output)

    def forward(self, input):
        # Makes this a batch of size 1
        input = input.unsqueeze(1)
        # Get the activations of all layers at the last time step
        output, _ = self.lstm(input)
        # Drop the batch index
        output = output.squeeze(1)
        output = output[output.size(0) - 1:output.size(0)]
        return self.fc_o2y(F.relu(output))
```
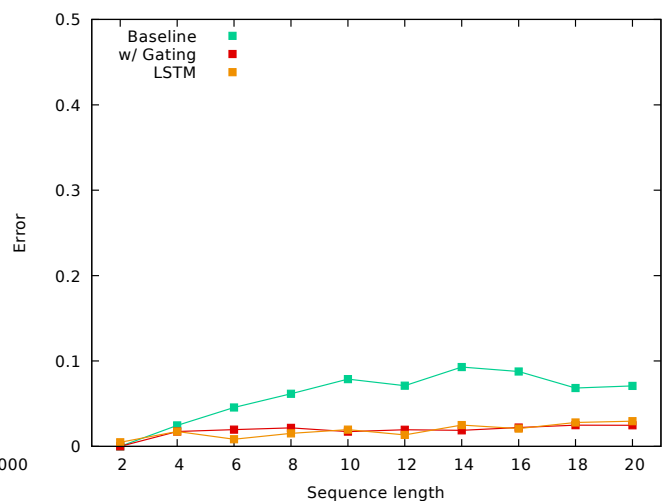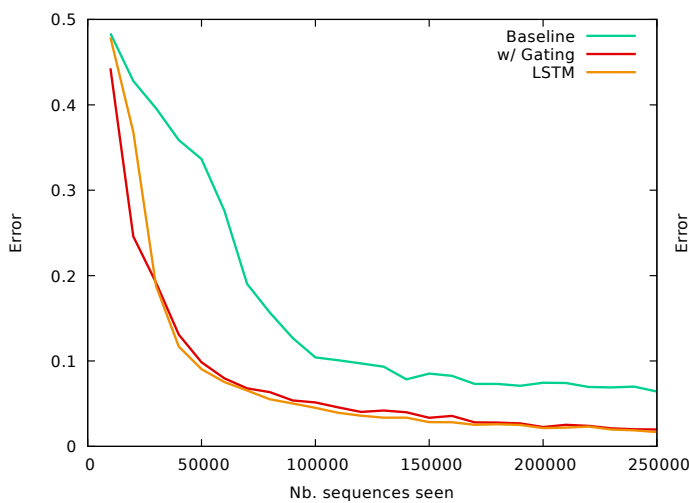
The LSTM were simplified into the Gated Recurrent Unit (GRU) by Cho et al. (2014), with a gating for the recurrent state, and a reset gate.

$$r_t = \text{sigm}\left(W_{(x\ r)}x_t + W_{(h\ r)}h_{t-1} + b_{(r)}\right) \qquad \text{(reset gate)}$$

$$z_t = \text{sigm}\left(W_{(x\ z)}x_t + W_{(h\ z)}h_{t-1} + b_{(z)}\right) \qquad \text{(forget gate)}$$

$$\bar{h}_t = \tanh\left(W_{(x\ h)}x_t + W_{(h\ h)}(r_t \odot h_{t-1}) + b_{(h)}\right) \qquad \text{(full update)}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \bar{h}_t \qquad \text{(hidden update)}$$

```
class GRUNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, num_layers, dim_output):
        super(GRUNet, self).__init__()
        self.gru = nn.GRU(input_size = dim_input,
                          hidden_size = dim_recurrent,
                          num_layers = num_layers)
        self.fc_y = nn.Linear(dim_recurrent, dim_output)

    def forward(self, input):
        # Make this a batch of size 1
        input = input.unsqueeze(1)
        # Get the activations of all layers at the last time step
        _, output = self.gru(input)
        # Drop the batch index
        output = output.squeeze(1)
        output = output.narrow[output.size(0) - 1:output.size(0)]
        return self.fc_y(F.relu(output))
```
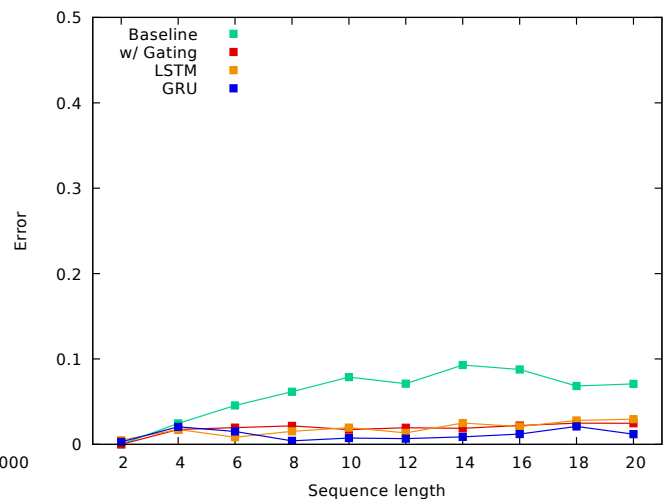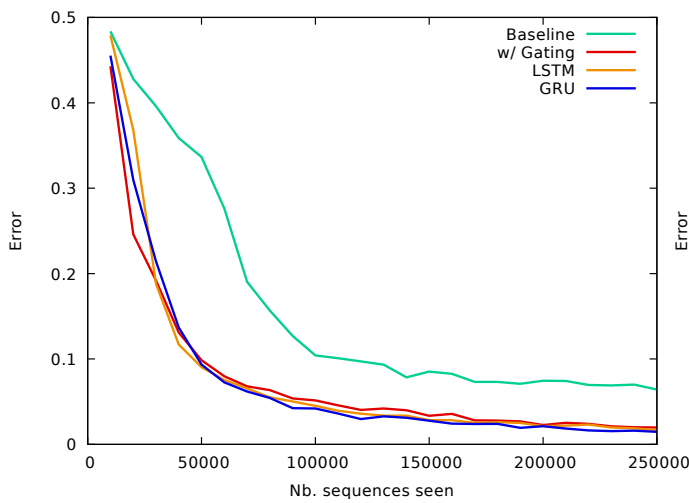
The specific form of these units prevent the gradient from vanishing, but it may still be excessively large on certain mini-batch.

The standard strategy to solve this issue is **gradient norm clipping** (Pascanu et al., 2013), which consists of re-scaling the [norm of the] gradient to a fixed threshold δ when if it was above:

$$\widetilde{\nabla f} = \frac{\nabla f}{\|\nabla f\|} \min\left(\|\nabla f\|, \delta\right).$$

The function `torch.nn.utils.clip_grad_norm` applies this operation to the gradient of a model, as defined by an iterator through its parameters:

```
>>> x = torch.empty(10)
>>> x.grad = x.new(x.size()).normal_()
>>> y = torch.empty(5)
>>> y.grad = y.new(y.size()).normal_()
>>> torch.cat((x.grad, y.grad)).norm()
tensor(4.0303)
>>> torch.nn.utils.clip_grad_norm_((x, y), 5.0)
tensor(4.0303)
>>> torch.cat((x.grad, y.grad)).norm()
tensor(4.0303)
>>> torch.nn.utils.clip_grad_norm_((x, y), 1.25)
tensor(4.0303)
>>> torch.cat((x.grad, y.grad)).norm()
tensor(1.2500)
```

Jozefowicz et al. (2015) conducted an extensive exploration of different recurrent architectures through meta-optimization, and even though some units simpler than LSTM or GRU perform well, they wrote:

"We have evaluated a variety of recurrent neural network architectures in order to find an architecture that reliably out-performs the LSTM. Though there were architectures that outperformed the LSTM on some problems, we were unable to find an architecture that consistently beat the LSTM and the GRU in all experimental conditions."

(Jozefowicz et al., 2015)

References

K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

F. A. Gers, J. A. Schmidhuber, and F. A. Cummins. Learning to forget: Continual prediction with lstm. *Neural Computation*, 12(10):2451–2471, 2000.

F. A. Gers, N. N. Schraudolph, and J. Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of Machine Learning Research (JMLR)*, 3:115–143, 2003.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning (ICML)*, pages 2342–2350, 2015.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2013.