12.2. LSTM and GRU
The Long-Short Term Memory unit (LSTM) by Hochreiter and Schmidhuber (1997), is a recurrent network that originally had 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 w.r.t. \( c_t \) does not vanish.

Notes

The three main ideas behind recurrent models seen so far are:

- a hidden state which is updated each time a new entry from the sequence is provided,
- the network can be trained with autograd as usual to do “backprop trough time”, with the recurrent network being unfolded as a directed acyclic graph,
- a gating mechanism is very beneficial.

In LSTM, the hidden state is called a “cell state”, and we note it \( c_t \).
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 used today.

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.

$$
\begin{align*}
    f_t &= \text{sigm} \left( W(x_r)x_t + W(h_r)h_{t-1} + b_f \right) & \text{(forget gate)} \\
    i_t &= \text{sigm} \left( W(x_i)x_t + W(h_i)h_{t-1} + b_i \right) & \text{(input gate)} \\
    g_t &= \tanh \left( W(x_c)x_t + W(h_c)h_{t-1} + b_c \right) & \text{(full cell state update)} \\
    c_t &= f_t \odot c_{t-1} + i_t \odot g_t & \text{(cell state)} \\
    o_t &= \text{sigm} \left( W(x_o)x_t + W(h_o)h_{t-1} + b_o \right) & \text{(output state)} \\
    h_t &= o_t \odot \tanh(c_t) & \text{(output gate)}
\end{align*}
$$

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

This model was extended by Gers et al. (2003) with “peephole connections” that allow gates to depend on $c_{t-1}$.

---

**Notes**

The main difference with the gating mechanism we saw in lecture 12.1. “Recurrent Neural Networks” is that the weight $f_t$ of the previous cell state, and the weight $i_t$ of the full update are independent of each other. In particular, they can both be zero, resulting in a reset of the state.
Prediction is done from the $h_t$ state, hence called the output state.

Notes
This picture shows the inputs, states and outputs involved in a LSTM cell. The gates $i_t$, $f_t$, and $o_t$ are not displayed here, but are “inside” the cell.
Several such “cells” can be combined to create a multi-layer LSTM.

Notes

When several layers of LSTM are combined, the first layer takes as input the sequence $x_t$ itself, while the next layer take as input the output state of the previous layer, the $h_t$. 
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: $h^1_T$ and $h^D_T$, and
- the outputs of the last layer at each time step: $h^D_1, \ldots, h^D_T$.

The initial recurrent states $h^1_0, \ldots, h^D_0$ and $c^1_0, \ldots, c^D_0$ 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`, which expects as argument a first tensor of $x_{t,n} T \times N \times \ldots$ padded with zeros, and a second tensor of $T_n$.

```python
>>> from torch.nn.utils.rnn import pack_padded_sequence
>>> pack_padded_sequence(torch.tensor([[ 1.], [ 2.]],
... [[ 3.], [ 4.]],
... [[ 5.], [ 0.]]),
... torch.tensor([3, 2]))
PackedSequence(data=tensor([[1.],
  [2.],
  [3.],
  [4.],
  [5.]]), batch_sizes=tensor([2, 2, 1]),
  sorted_indices=None, unsorted_indices=None)
```

⚠️ The sequences must be sorted by decreasing lengths.

`nn.utils.rnn.pad_packed_sequence` converts back to a padded tensor.
We implement a small model to test it on the toy task of lecture 12.1. “Recurrent Neural Networks”.

```python
class LSTMNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, num_layers, dim_output):
        super().__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):
        # Get the last layer's last time step activation
        output, _ = self.lstm(input.permute(1, 0, 2))
        output = output[-1]
        return self.fc_o2y(F.relu(output))
```

**permute** makes the tensor \( T \times N \times \ldots \) as expected by `LSTM.forward`, and for simplicity, we consider all sequences to be of same length when picking the last step.

---

**Notes**

Contrary to other PyTorch modules which expect a mini-batch of size \( N \times \ldots \), where \( N \) is the number of samples in the mini-batch, `nn.LSTM` expects a mini-batch to be of size \( T \times N \times \ldots \), where \( T \) is the sequence length.
Notes

The graph on the left shows the test error as a function of the number of sequences seen during training.

The graph on the right shows the classification error of the final trained model as a function of the number of elements in the input sequence. As expected the longer the sequence, the higher the error.

The performance of gating and LSTM are the same, which is not surprising because the task is easy.
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.

\[
\begin{align*}
    r_t &= \text{sigm} \left( W(x, r)x_t + W(h, r)h_{t-1} + b(r) \right) & \text{(reset gate)} \\
    z_t &= \text{sigm} \left( W(x, z)x_t + W(h, z)h_{t-1} + b(z) \right) & \text{(forget gate)} \\
    \tilde{h}_t &= \text{tanh} \left( W(x, h)x_t + W(h, h)(r_t \odot h_{t-1}) + b(h) \right) & \text{(full update)} \\
    h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t & \text{(hidden update)}
\end{align*}
\]
class GRUNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, num_layers, dim_output):
        super().__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):
        # Get the last layer's last time step activation
        output, _ = self.gru(input.permute(1, 0, 2))
        output = output[-1]
        return self.fc_y(F.relu(output))

permute makes the tensor $T \times N \times \ldots$ as expected by GRU.forward, and for simplicity, we consider all sequences to be of same length when picking the last step.
Notes

The graph on the left shows the test error as a function of the number of sequences seen during training.

The graph on the right shows the classification error of the final trained model as a function of the number of elements in the input sequence. As expected the longer the sequence, the higher the error.
The specific form of these units prevents 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 $\delta$ when it is above:

$$\tilde{\nabla}f = \frac{\nabla f}{\|\nabla f\|} \min (\|\nabla f\|, \delta).$$
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:

```python
>>> 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)

Notes

The conclusion of this extensive experiments is that LSTM is generally a good choice of recurrent architecture.
References


