EE-559 – Deep learning

11.1. Recurrent Neural Networks

François Fleuret

https://fleuret.org/ee559/

Mon Feb 18 13:33:17 UTC 2019

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Inference from sequences

Many real-world problems require to process a signal with a sequence structure.

Many real-world problems require to process a signal with a sequence structure.

Sequence classification:

- sentiment analysis,
- activity/action recognition,
- DNA sequence classification,
- action selection.

Sequence synthesis:

- text synthesis,
- music synthesis,
- motion synthesis.

Sequence-to-sequence translation:

- speech recognition,
- text translation,
- part-of-speech tagging.

Given a set $\mathcal{X}$, if $S(\mathcal{X})$ be the set of sequences of elements from $\mathcal{X}$:

$$S(\mathcal{X}) = \bigcup_{t=1}^{\infty} \mathcal{X}^t.$$

We can define formally:

Sequence classification: $f : S(\mathcal{X}) \to \{1, \ldots, C\}$

Sequence synthesis: $f : \mathbb{R}^D \to S(\mathcal{X})$

Sequence-to-sequence translation: $f : S(\mathcal{X}) \to S(\mathcal{Y})$

Given a set $\mathcal{X}$, if $S(\mathcal{X})$ be the set of sequences of elements from $\mathcal{X}$:

$$S(\mathcal{X}) = \bigcup_{t=1}^{\infty} \mathcal{X}^t.$$

We can define formally:

Sequence classification: $f : S(\mathcal{X}) \to \{1, \ldots, C\}$

Sequence synthesis: $f : \mathbb{R}^D \to S(\mathcal{X})$

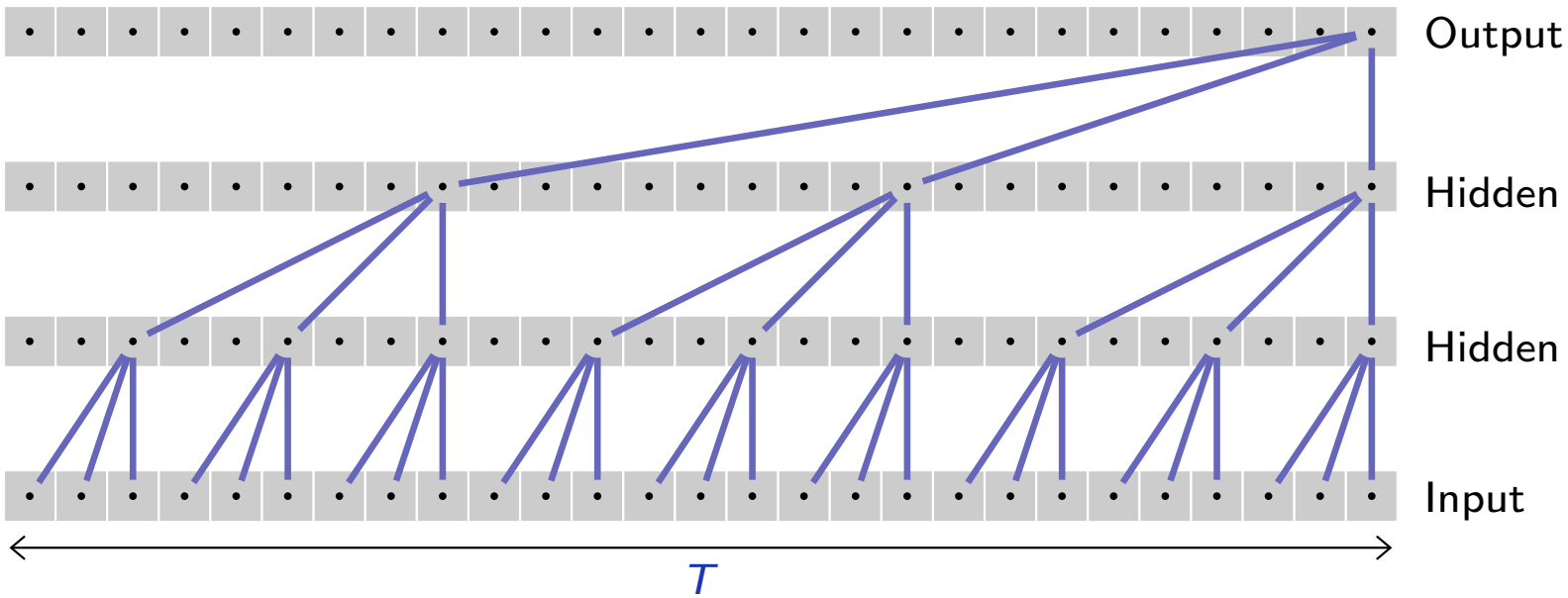
Sequence-to-sequence translation: $f : S(\mathcal{X}) \to S(\mathcal{Y})$

In the rest of the slides we consider only time-indexed signal, although it generalizes to arbitrary sequences.

Temporal Convolutions

The simplest approach to sequence processing is to use **Temporal Convolutional Networks** (Waibel et al., 1989; Bai et al., 2018).

Such a model is a standard 1d convolutional network, that processes an input of the maximum possible length.



Increasing exponentially the filter sizes makes the required number of layers grow in log of the time window T taken into account.

Thanks to dilated convolutions, the model size is $O(\log T)$. The memory footprint and computation are $O(T \log T)$.

Table 1. Evaluation of TCNs and recurrent architectures on synthetic stress tests, polyphonic music modeling, character-level language modeling, and word-level language modeling. The generic TCN architecture outperforms canonical recurrent networks across a comprehensive suite of tasks and datasets. Current state-of-the-art results are listed in the supplement. ^{h} means that higher is better. $^{\ell}$ means that lower is better.

Sequence Modeling Task	Model Size ($\approx$)	Models			
		LSTM	GRU	RNN	**TCN**
Seq. MNIST (accuracy^{h})	70K	87.2	96.2	21.5	**99.0**
Permuted MNIST (accuracy)	70K	85.7	87.3	25.3	**97.2**
Adding problem T=600 (loss$^{\ell}$)	70K	0.164	**5.3e-5**	0.177	**5.8e-5**
Copy memory T=1000 (loss)	16K	0.0204	0.0197	0.0202	**3.5e-5**
Music JSB Chorales (loss)	300K	8.45	8.43	8.91	**8.10**
Music Nottingham (loss)	1M	3.29	3.46	4.05	**3.07**
Word-level PTB (perplexity$^{\ell}$)	13M	**78.93**	92.48	114.50	89.21
Word-level Wiki-103 (perplexity)	-	48.4	-	-	**45.19**
Word-level LAMBADA (perplexity)	-	4186	-	14725	**1279**
Char-level PTB (bpc$^{\ell}$)	3M	1.41	1.42	1.52	**1.35**
Char-level text8 (bpc)	5M	1.52	1.56	1.69	**1.45**

(Bai et al., 2018)

RNN and backprop through time

The most classical approach to processing sequences of variable size is to use a recurrent model which maintains a **recurrent state** updated at each time step.

With $\mathscr{X} = \mathbb{R}^D$, given an input sequence $x \in S\left(\mathbb{R}^D\right)$, and an initial **recurrent state** $h_0 \in \mathbb{R}^Q$, the model computes the sequence of recurrent states iteratively

$$\forall t = 1, \ldots, T(x), \;\; h_t = \Phi(x_t, h_{t-1}),$$

where

$$\Phi_w : \mathbb{R}^D \times \mathbb{R}^Q \to \mathbb{R}^Q.$$

The most classical approach to processing sequences of variable size is to use a recurrent model which maintains a **recurrent state** updated at each time step.

With $\mathscr{X} = \mathbb{R}^D$, given an input sequence $x \in S\left(\mathbb{R}^D\right)$, and an initial **recurrent state** $h_0 \in \mathbb{R}^Q$, the model computes the sequence of recurrent states iteratively

$$\forall t = 1, \ldots, T(x), \;\; h_t = \Phi(x_t, h_{t-1}),$$

where

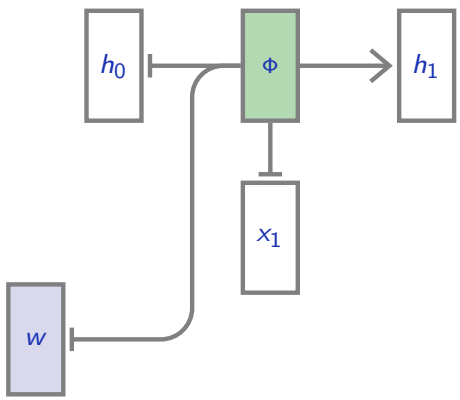
$$\Phi_w : \mathbb{R}^D \times \mathbb{R}^Q \to \mathbb{R}^Q.$$

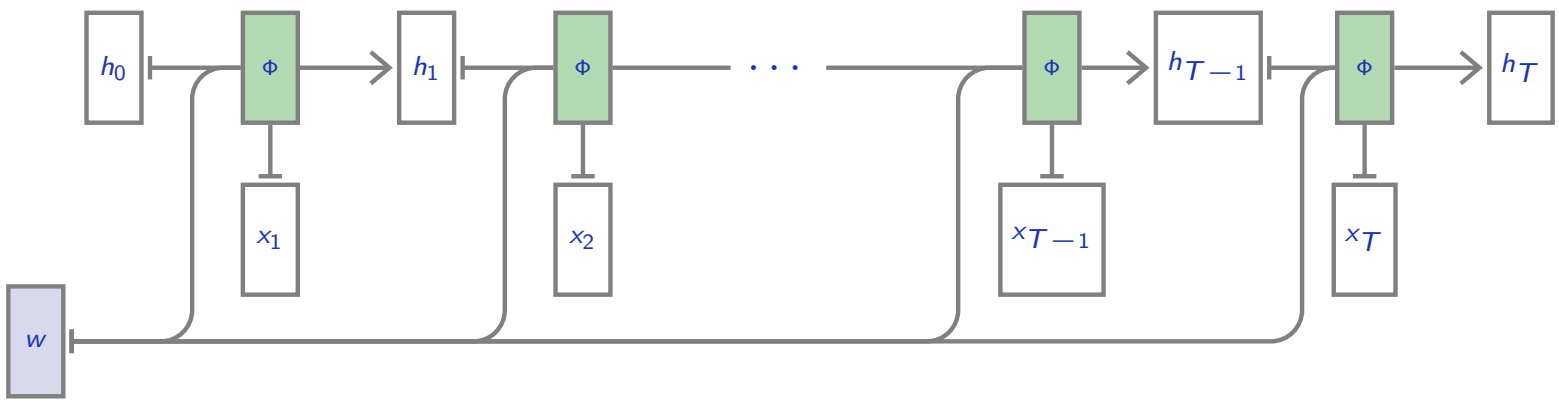
A prediction can be computed at any time step from the recurrent state

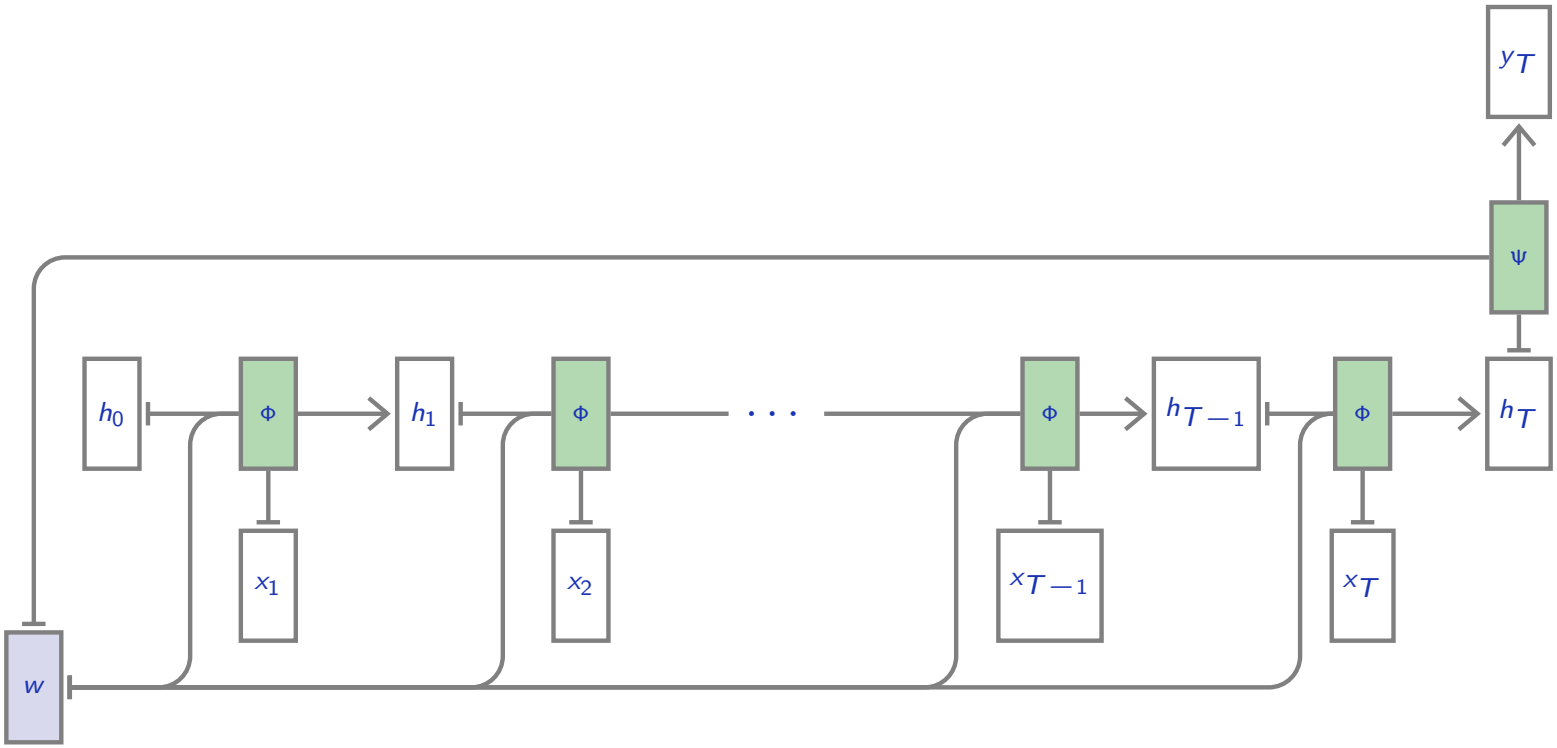
$$y_t = \Psi(h_t)$$

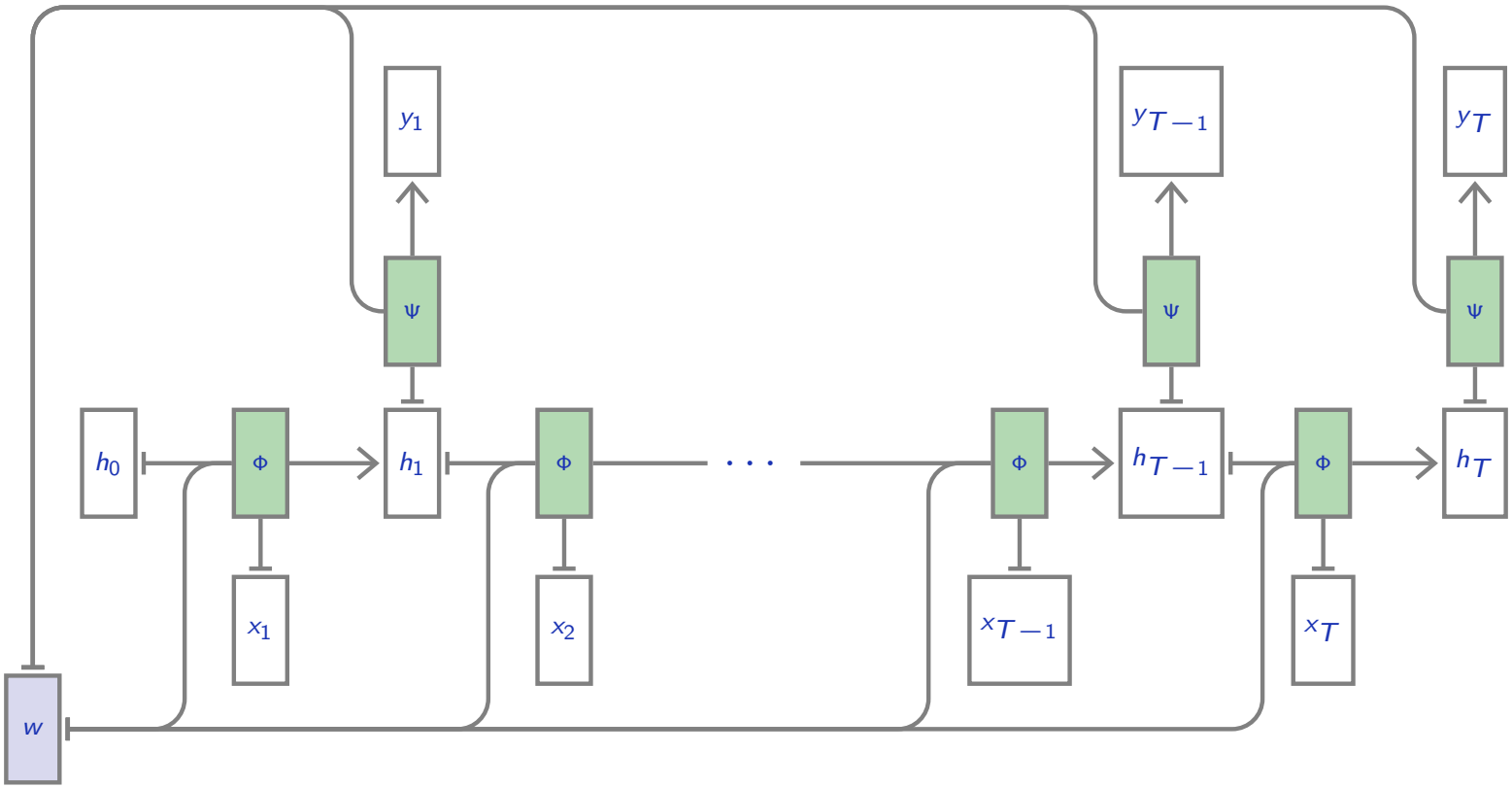
with

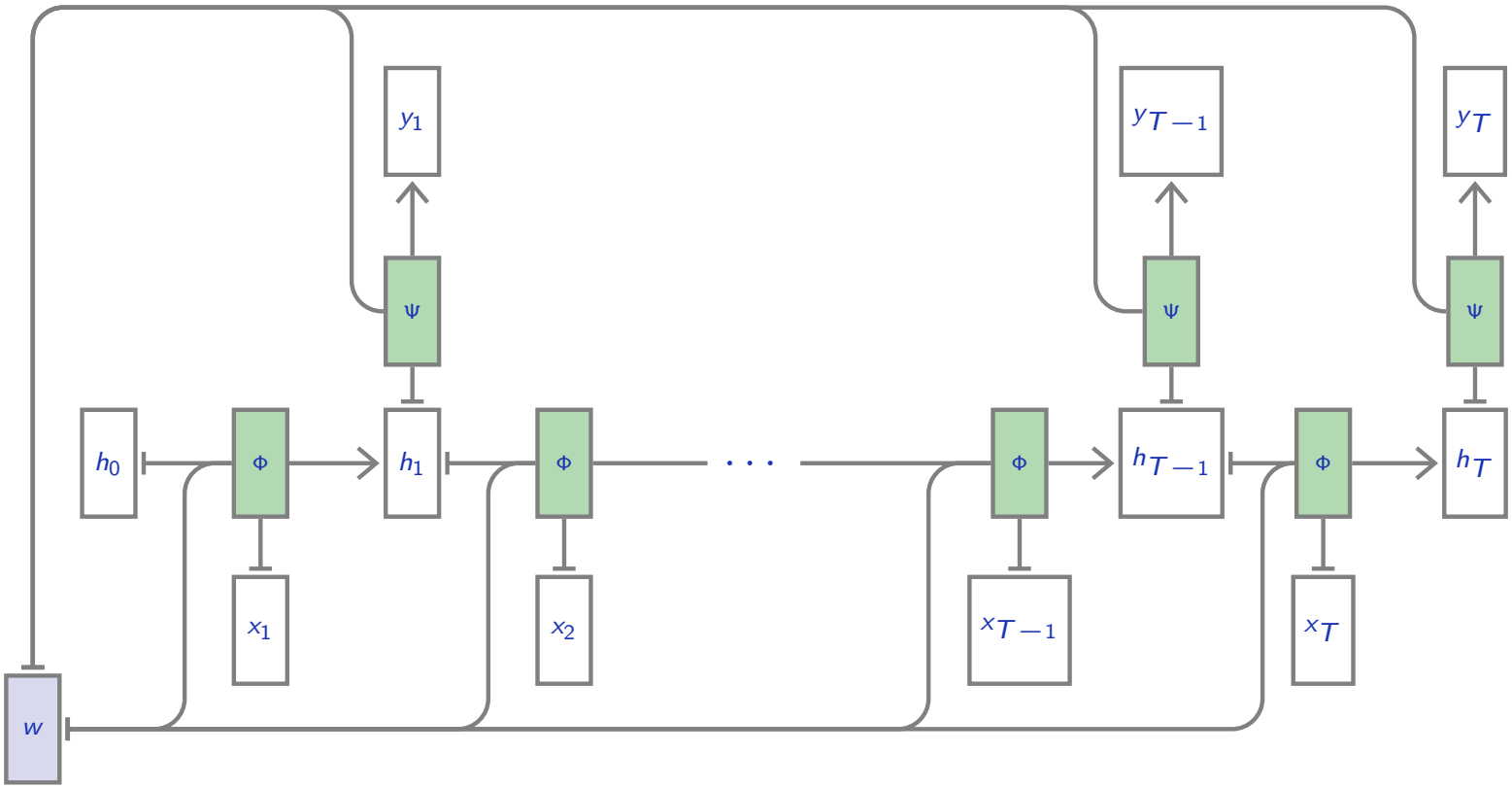
$$\Psi_w : \mathbb{R}^Q \to \mathbb{R}^C.$$


h_0
Φ
h_1
x_1
w

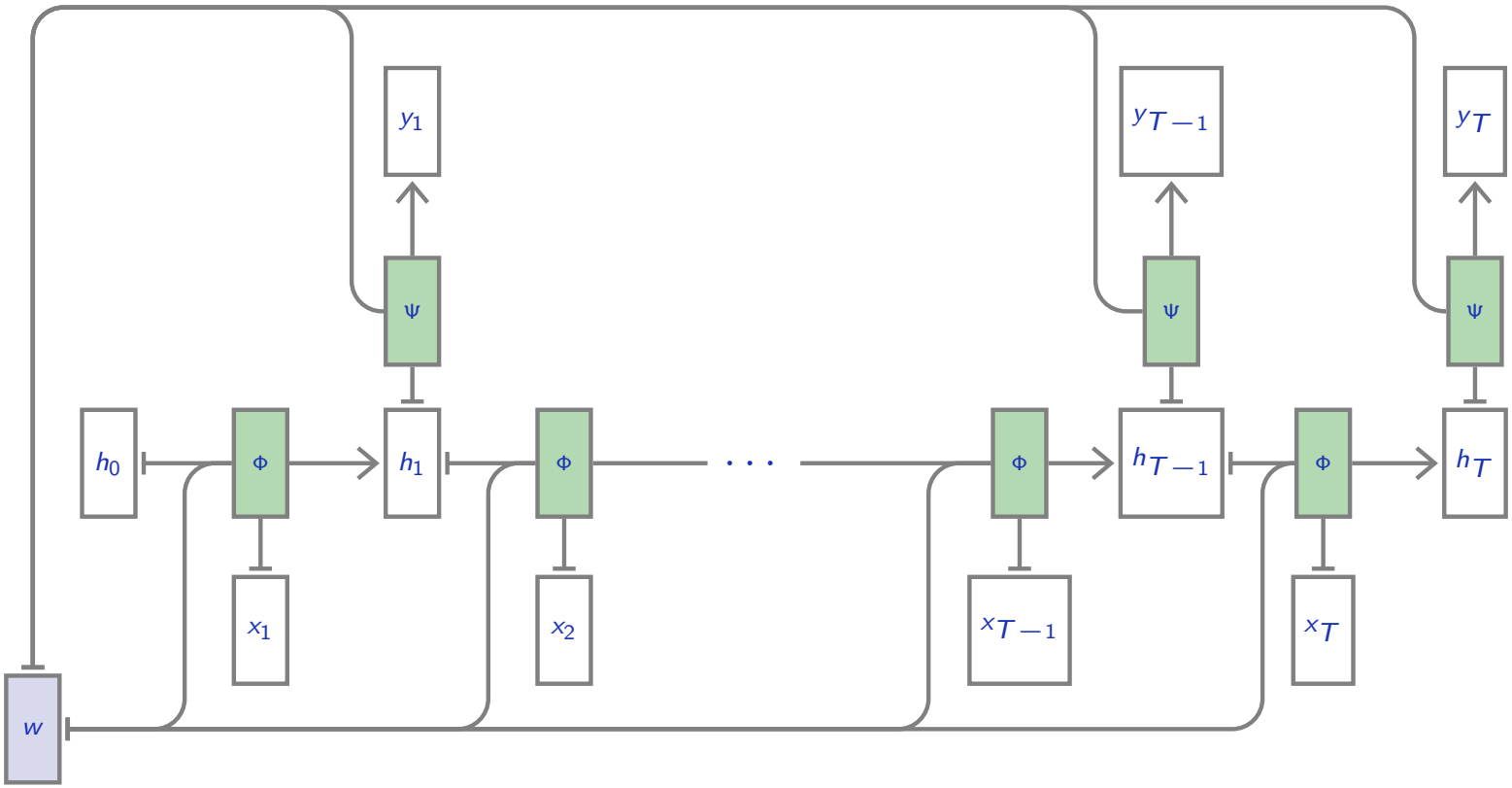

h_0
Φ
h_1
Φ
. . .
Φ
h_{T-1}
Φ
h_T
x_1
x_2
x_{T-1}
x_T
w


y_T
Ψ
h_0
Φ
h_1
Φ
$\cdots$
Φ
h_{T-1}
Φ
h_T
x_1
x_2
x_{T-1}
x_T
w


y_1
y_{T-1}
y_T
Ψ
Ψ
Ψ
h_0
Φ
h_1
Φ
· · ·
Φ
h_{T-1}
Φ
h_T
x_1
x_2
x_{T-1}
x_T
w



Even though the number of steps T depends on x, this is a standard graph of tensor operations, and autograd can deal with it as usual.



Even though the number of steps T depends on x, this is a standard graph of tensor operations, and autograd can deal with it as usual. This is referred to as "backpropagation through time" (Werbos, 1988).

We consider the following simple binary sequence classification problem:

- Class 1: the sequence is the concatenation of two identical halves,
- Class 0: otherwise.

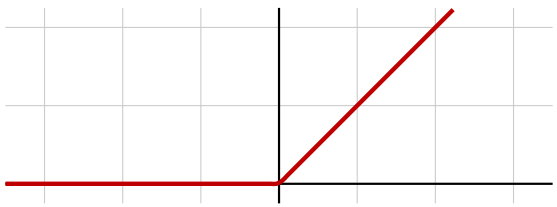
E.g.

x	y
$(1, 2, 3, 4, 5, 6)$	0
$(3, 9, 9, 3)$	0
$(7, 4, 5, 7, 5, 4)$	0
$(7, 7)$	1
$(1, 2, 3, 1, 2, 3)$	1
$(5, 1, 1, 2, 5, 1, 1, 2)$	1

In what follows we use the three standard activation functions:

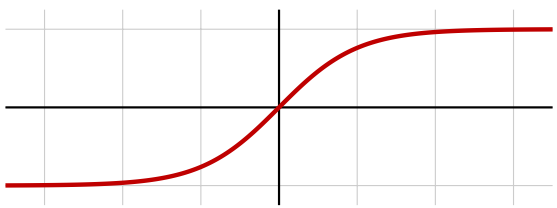
- The rectified linear unit:

$$\text{ReLU}(x) = \max(x, 0)$$



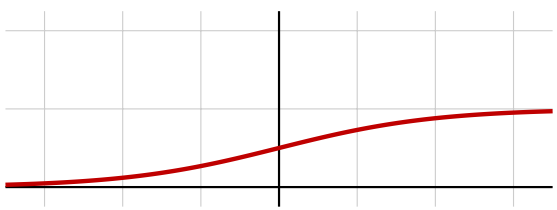
- The hyperbolic tangent:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



- The sigmoid:

$$\text{sigm}(x) = \frac{1}{1 + e^{-x}}$$



We can build an "Elman network" (Elman, 1990), with $h_0 = 0$, the update

$$h_t = \text{ReLU}\left(W_{(x\ h)} x_t + W_{(h\ h)} h_{t-1} + b_{(h)}\right) \qquad \text{(recurrent state)}$$

and the final prediction

$$y_T = W_{(h\ y)} h_T + b_{(y)}.$$

We can build an "Elman network" (Elman, 1990), with $h_0 = 0$, the update

$$h_t = \text{ReLU}\left(W_{(x\ h)} x_t + W_{(h\ h)} h_{t-1} + b_{(h)}\right) \qquad \text{(recurrent state)}$$

and the final prediction

$$y_T = W_{(h\ y)} h_T + b_{(y)}.$$

```
class RecNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, dim_output):
        super(RecNet, self).__init__()
        self.fc_x2h = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2h = nn.Linear(dim_recurrent, dim_recurrent, bias = False)
        self.fc_h2y = nn.Linear(dim_recurrent, dim_output)

    def forward(self, input):
        h = input.new_zeros(1, self.fc_h2y.weight.size(1))
        for t in range(input.size(0)):
            h = F.relu(self.fc_x2h(input[t:t+1]) + self.fc_h2h(h))
        return self.fc_h2y(h)
```

We can build an "Elman network" (Elman, 1990), with $h_0 = 0$, the update

$$h_t = \text{ReLU}\left(W_{(x\ h)} x_t + W_{(h\ h)} h_{t-1} + b_{(h)}\right) \qquad \text{(recurrent state)}$$

and the final prediction

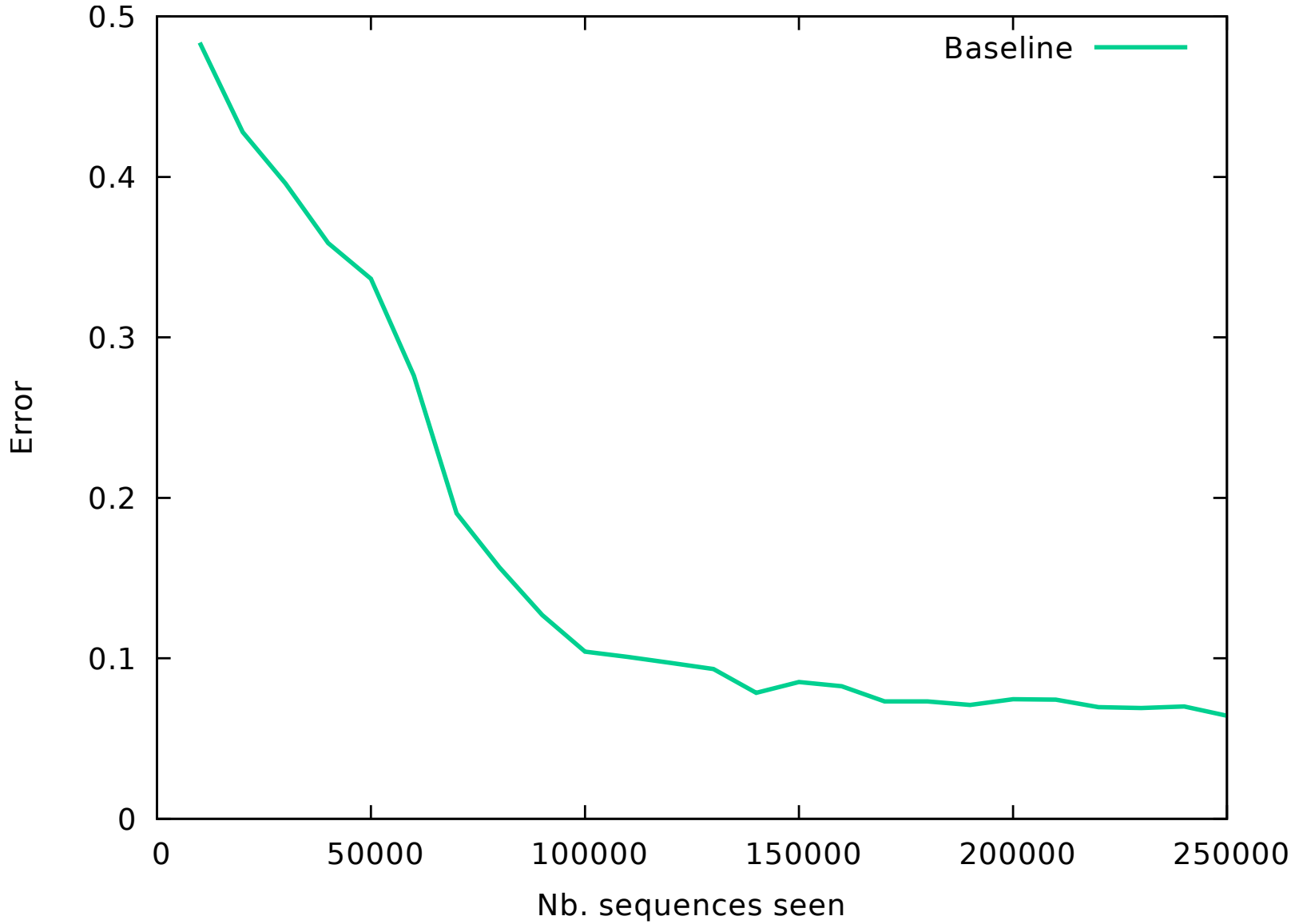
$$y_T = W_{(h\ y)} h_T + b_{(y)}.$$

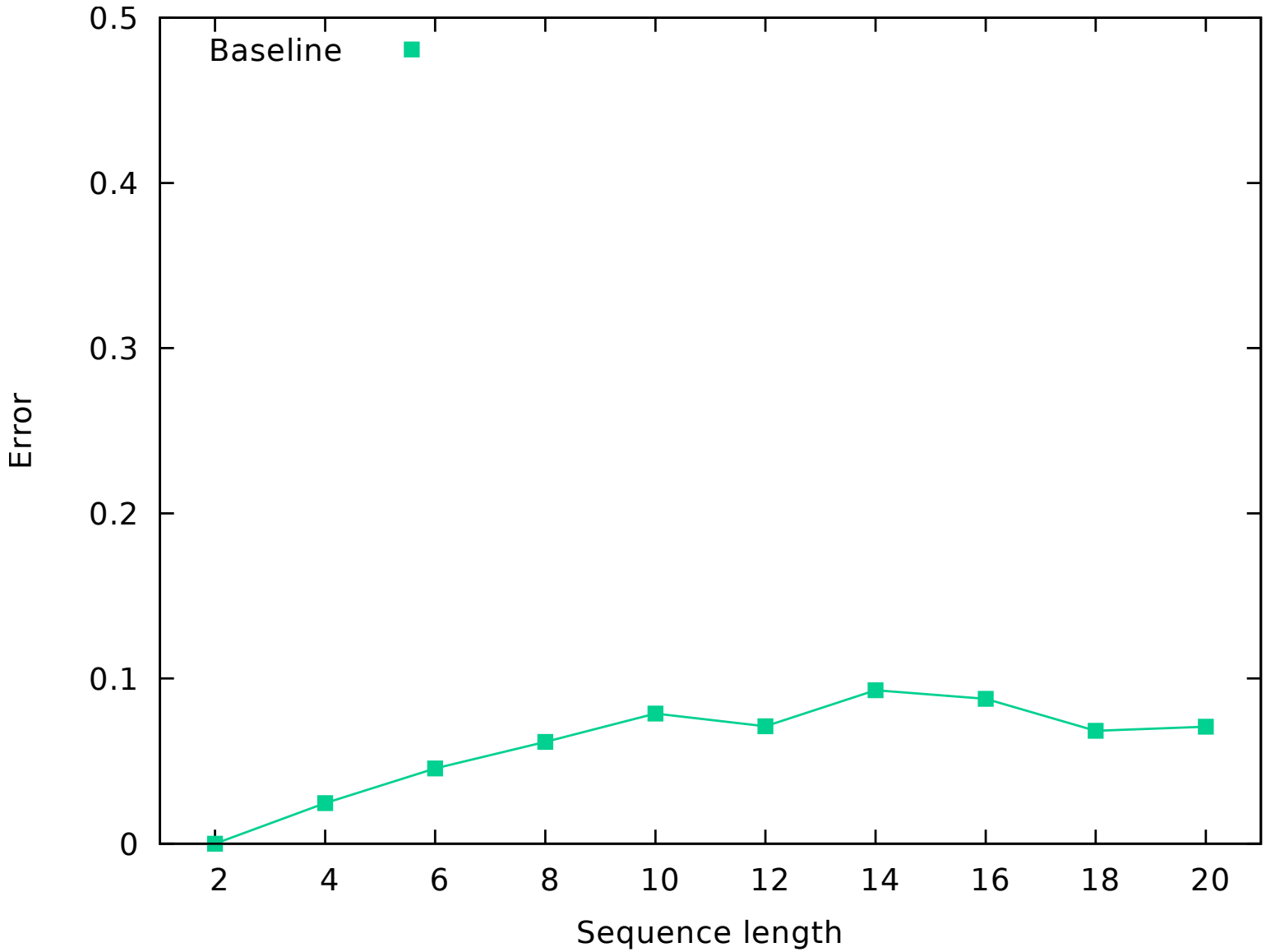
```
class RecNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, dim_output):
        super(RecNet, self).__init__()
        self.fc_x2h = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2h = nn.Linear(dim_recurrent, dim_recurrent, bias = False)
        self.fc_h2y = nn.Linear(dim_recurrent, dim_output)

    def forward(self, input):
        h = input.new_zeros(1, self.fc_h2y.weight.size(1))
        for t in range(input.size(0)):
            h = F.relu(self.fc_x2h(input[t:t+1]) + self.fc_h2h(h))
        return self.fc_h2y(h)
```

To simplify the processing of variable-length sequences, we are processing samples (sequences) one at a time here.

We encode the symbol at time t as a one-hot vector x_t, and thanks to autograd, the training can be implemented as

```
generator = SequenceGenerator(nb_symbols = 10,
                              pattern_length_min = 1, pattern_length_max = 10,
                              one_hot = True)

model = RecNet(dim_input = 10,
               dim_recurrent = 50,
               dim_output = 2)

cross_entropy = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr = lr)

for k in range(args.nb_train_samples):
    input, target = generator.generate()
    output = model(input)
    loss = cross_entropy(output, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```


Baseline
Error
Nb. sequences seen
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000


Baseline
Error
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
12
14
16
18
20
Sequence length

Gating

When unfolded through time, the model is deep, and training it involves in particular dealing with vanishing gradients.

An important idea in the RNN models used in practice is to add in a form or another a **pass-through**, so that the recurrent state does not go repeatedly through a squashing non-linearity.

For instance, the recurrent state update can be a per-component weighted average of its previous value h_{t-1} and a full update $\bar{h}_t$, with the weighting z_t depending on the input and the recurrent state, and playing the role of a "forget gate".

For instance, the recurrent state update can be a per-component weighted average of its previous value h_{t-1} and a full update $\bar{h}_t$, with the weighting z_t depending on the input and the recurrent state, and playing the role of a "forget gate".

So the model has an additional "gating" output

$$f : \mathbb{R}^D \times \mathbb{R}^Q \to [0,1]^Q,$$

and the update rule takes the form

$$\begin{aligned} \bar{h}_t &= \Phi(x_t, h_{t-1}) \\ z_t &= f(x_t, h_{t-1}) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \bar{h}_t, \end{aligned}$$

where $\odot$ stands for the usual component-wise Hadamard product.

We can improve our minimal example with such a mechanism, from our simple

$$h_t = \text{ReLU}\left(W_{(x\ h)} x_t + W_{(h\ h)} h_{t-1} + b_{(h)}\right) \qquad \text{(recurrent state)}$$

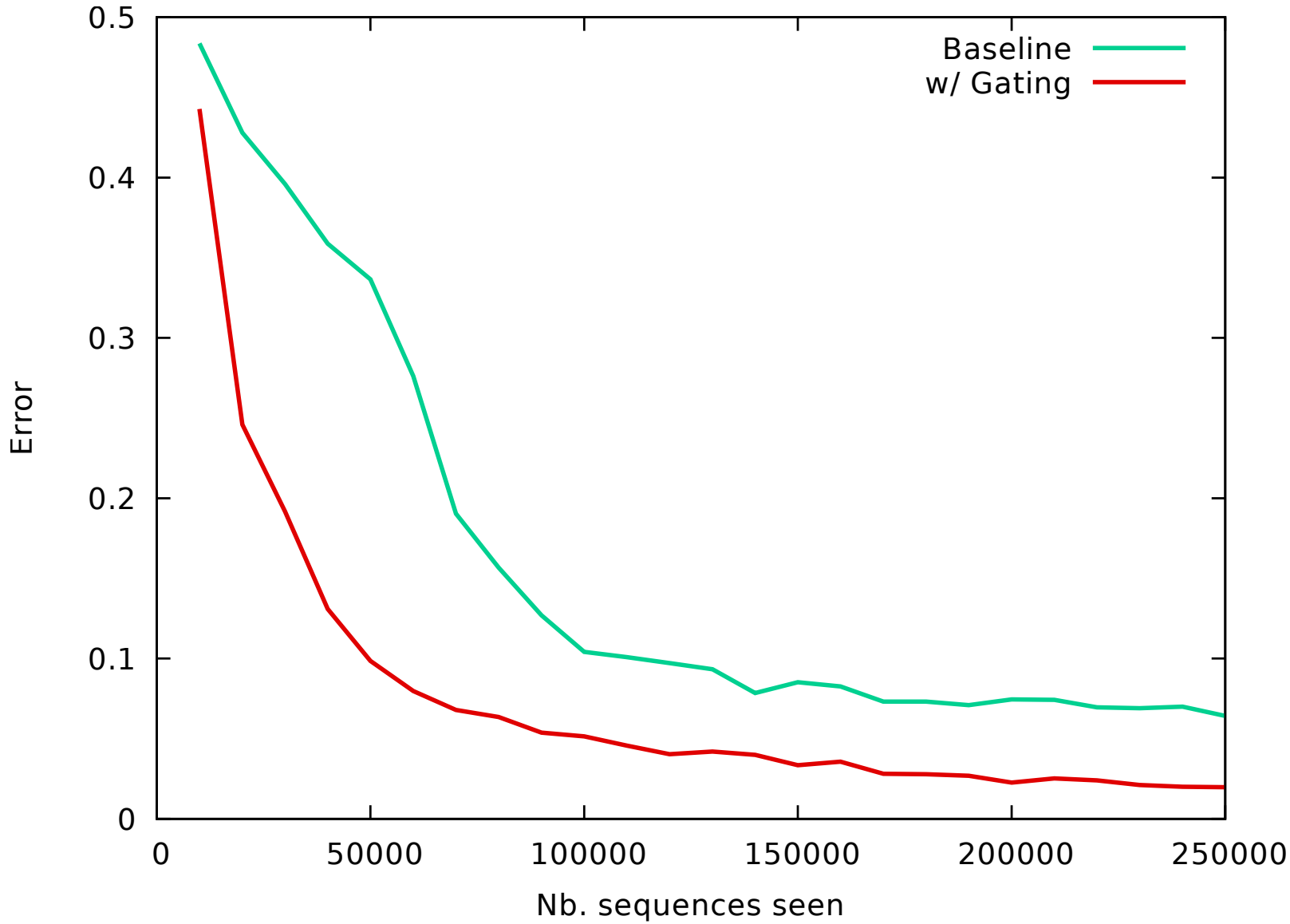
to

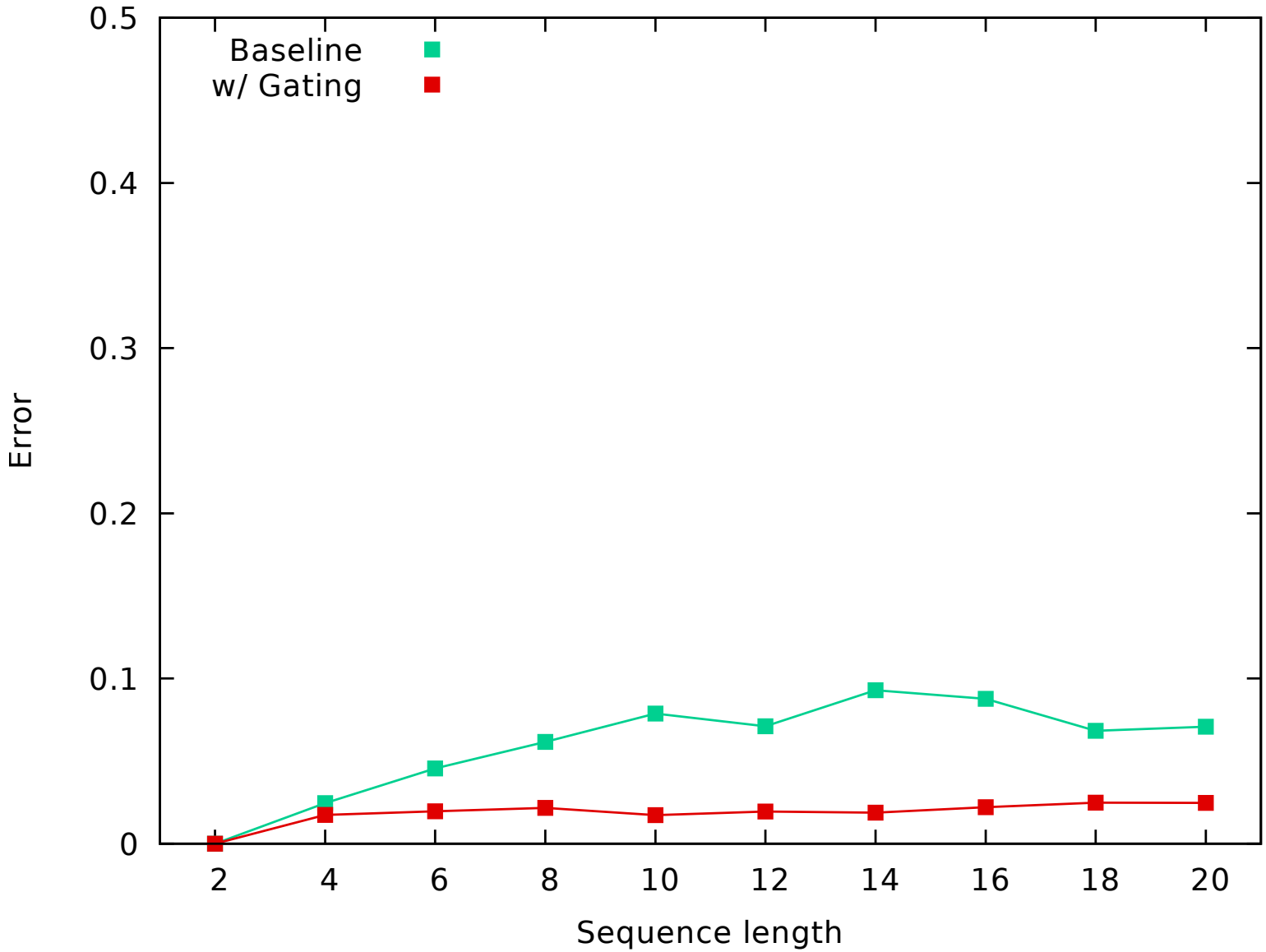
$$\bar{h}_t = \text{ReLU}\left(W_{(x\ h)} x_t + W_{(h\ h)} h_{t-1} + b_{(h)}\right) \qquad \text{(full update)}$$

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

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

```
class RecNetWithGating(nn.Module):
    def __init__(self, dim_input, dim_recurrent, dim_output):
        super(RecNetWithGating, self).__init__()

        self.fc_x2h = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2h = nn.Linear(dim_recurrent, dim_recurrent, bias = False)
        self.fc_x2z = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2z = nn.Linear(dim_recurrent, dim_recurrent, bias = False)

        self.fc_h2y = nn.Linear(dim_recurrent, dim_output)

    def forward(self, input):
        h = input.new_zeros(1, self.fc_h2y.weight.size(1))
        for t in range(input.size(0)):
            z = torch.sigmoid(self.fc_x2z(input[t:t+1]) + self.fc_h2z(h))
            hb = F.relu(self.fc_x2h(input[t:t+1]) + self.fc_h2h(h))
            h = z * h + (1 - z) * hb
        return self.fc_h2y(h)
```


Baseline
w/ Gating
Error
Nb. sequences seen
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000


Baseline
w/ Gating
Error
Sequence length
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
12
14
16
18
20

The end

References

S. Bai, J. Kolter, and V. Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *CoRR*, abs/1803.01271, 2018.

J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179 – 211, 1990.

A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328–339, 1989.

P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.

EE-559 – Deep learning

11.2. LSTM and GRU

François Fleuret

https://fleuret.org/ee559/

Mon Feb 18 13:33:24 UTC 2019

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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.

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.

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.

$$f_t = \text{sigm}\left(W_{(\text{x f})} x_t + W_{(\text{h f})} h_{t-1} + b_{(\text{f})}\right) \quad \text{(forget gate)}$$

$$i_t = \text{sigm}\left(W_{(\text{x i})} x_t + W_{(\text{h i})} h_{t-1} + b_{(\text{i})}\right) \quad \text{(input gate)}$$

$$g_t = \tanh\left(W_{(\text{x c})} x_t + W_{(\text{h c})} h_{t-1} + b_{(\text{c})}\right) \quad \text{(full cell state update)}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad \text{(cell state)}$$

$$o_t = \text{sigm}\left(W_{(\text{x o})} x_t + W_{(\text{h o})} h_{t-1} + b_{(\text{o})}\right) \quad \text{(output gate)}$$

$$h_t = o_t \odot \tanh(c_t) \quad \text{(output state)}$$

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.

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

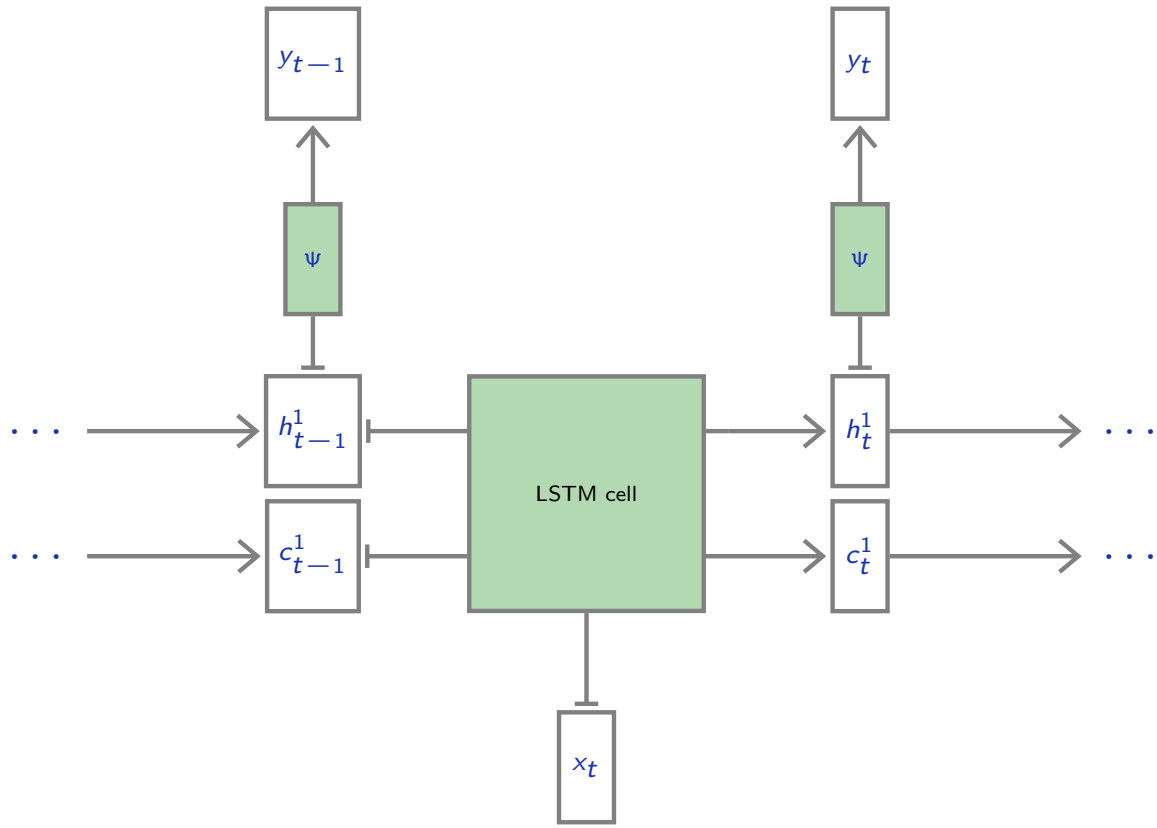
As pointed out by Gers et al. (2000), the forget bias $b_{(\text{f})}$ should be initialized with large values so that initially $f_t \simeq 1$ and the gating has no effect.

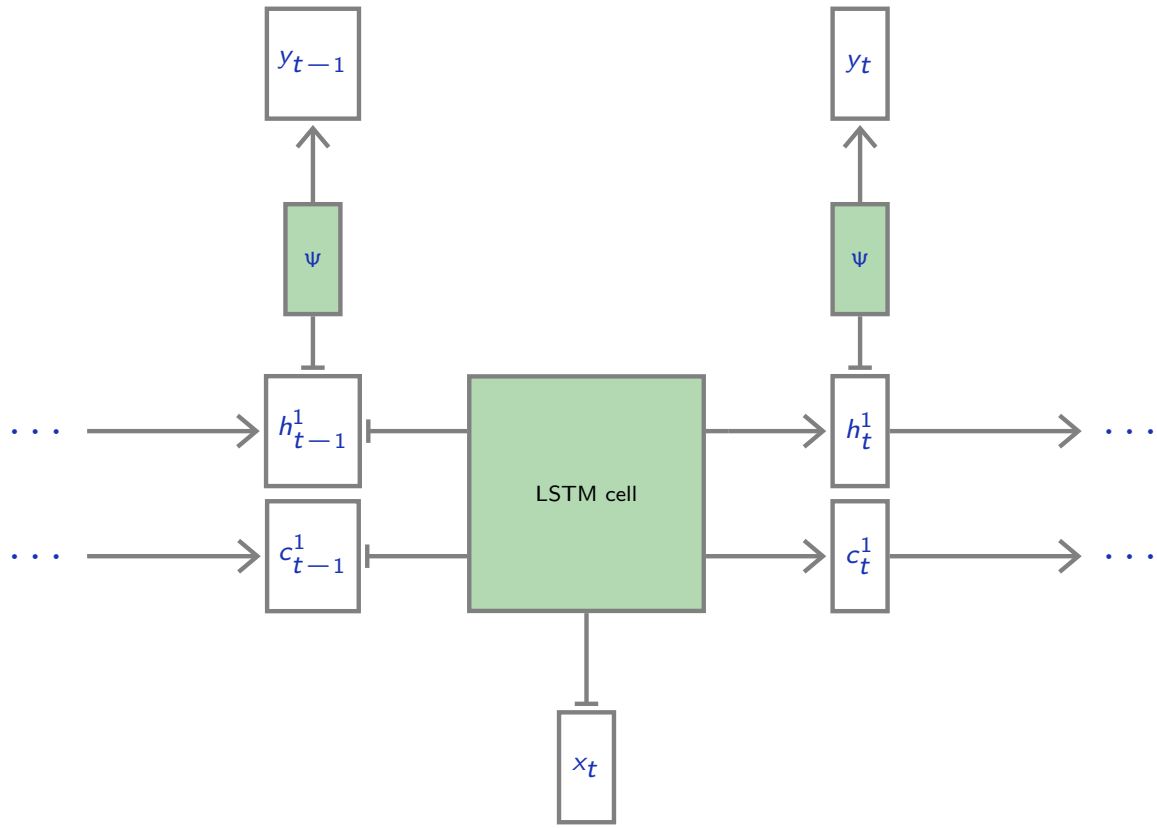
The recurrent state is composed of a "cell state" c_t and an "output state" h_t. Gate f_t modulates if the cell state should be forgotten, i_t if the new update should be taken into account, and o_t if the output state should be reset.

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

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

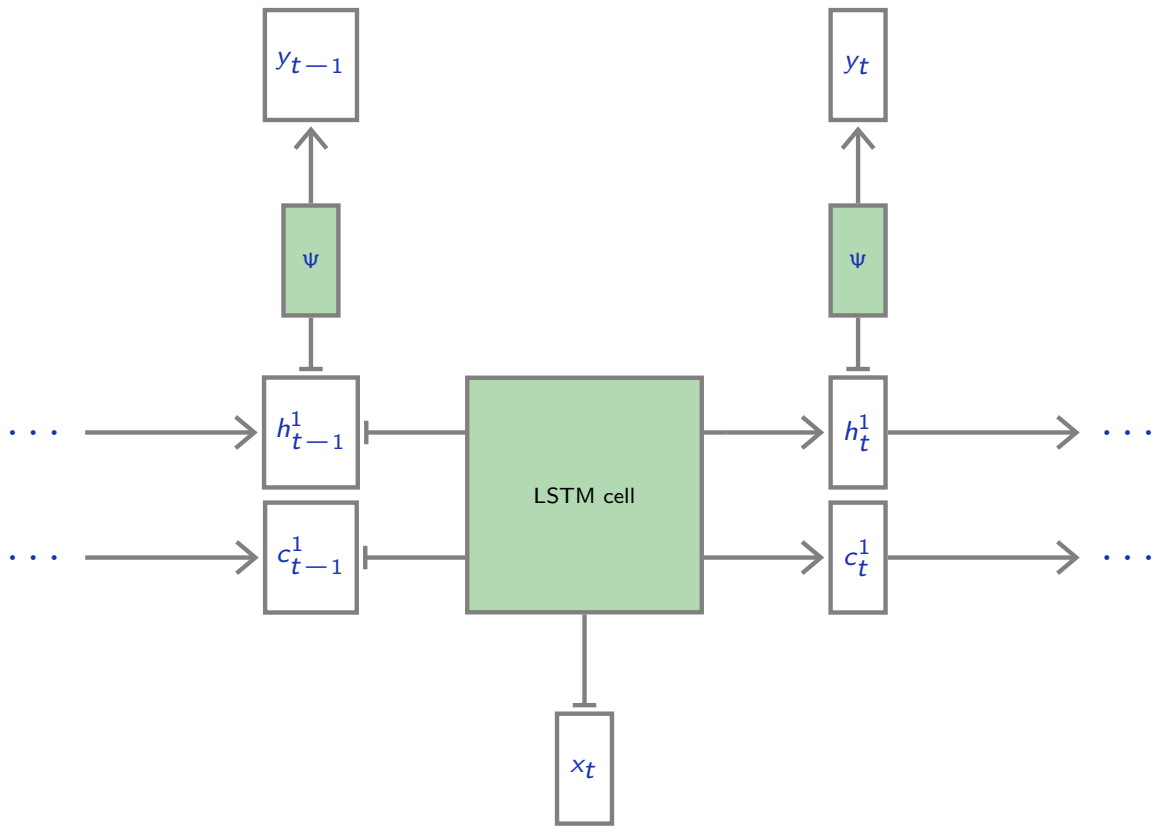
This model was extended by Gers et al. (2003) with "peephole connections" that allow gates to depend on c_{t-1}.


y_{t-1}
y_t
ψ
ψ
h^1_{t-1}
LSTM cell
h^1_t
c^1_{t-1}
c^1_t
x_t

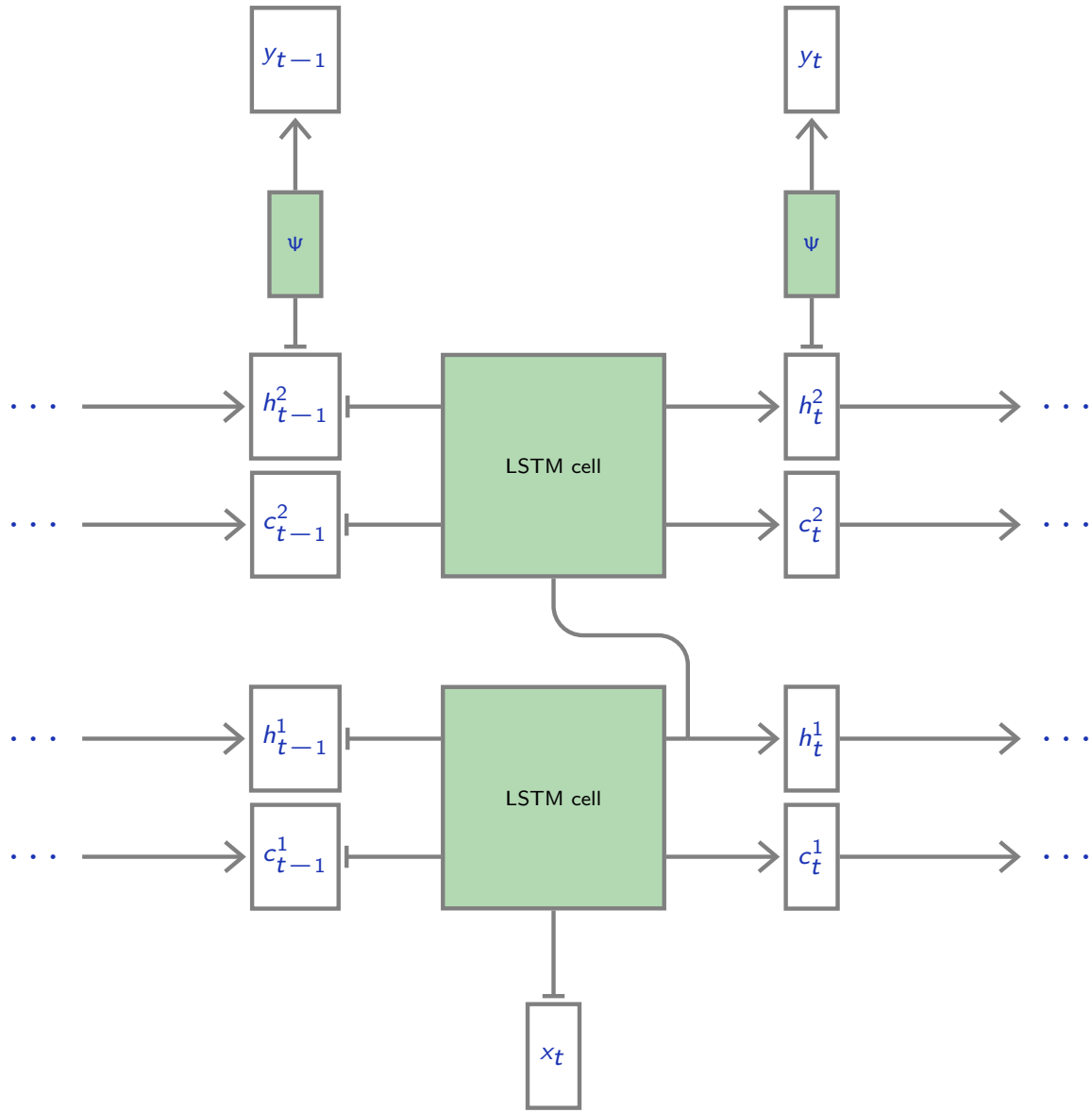


Prediction is done from the h_t state, hence called the **output** state.

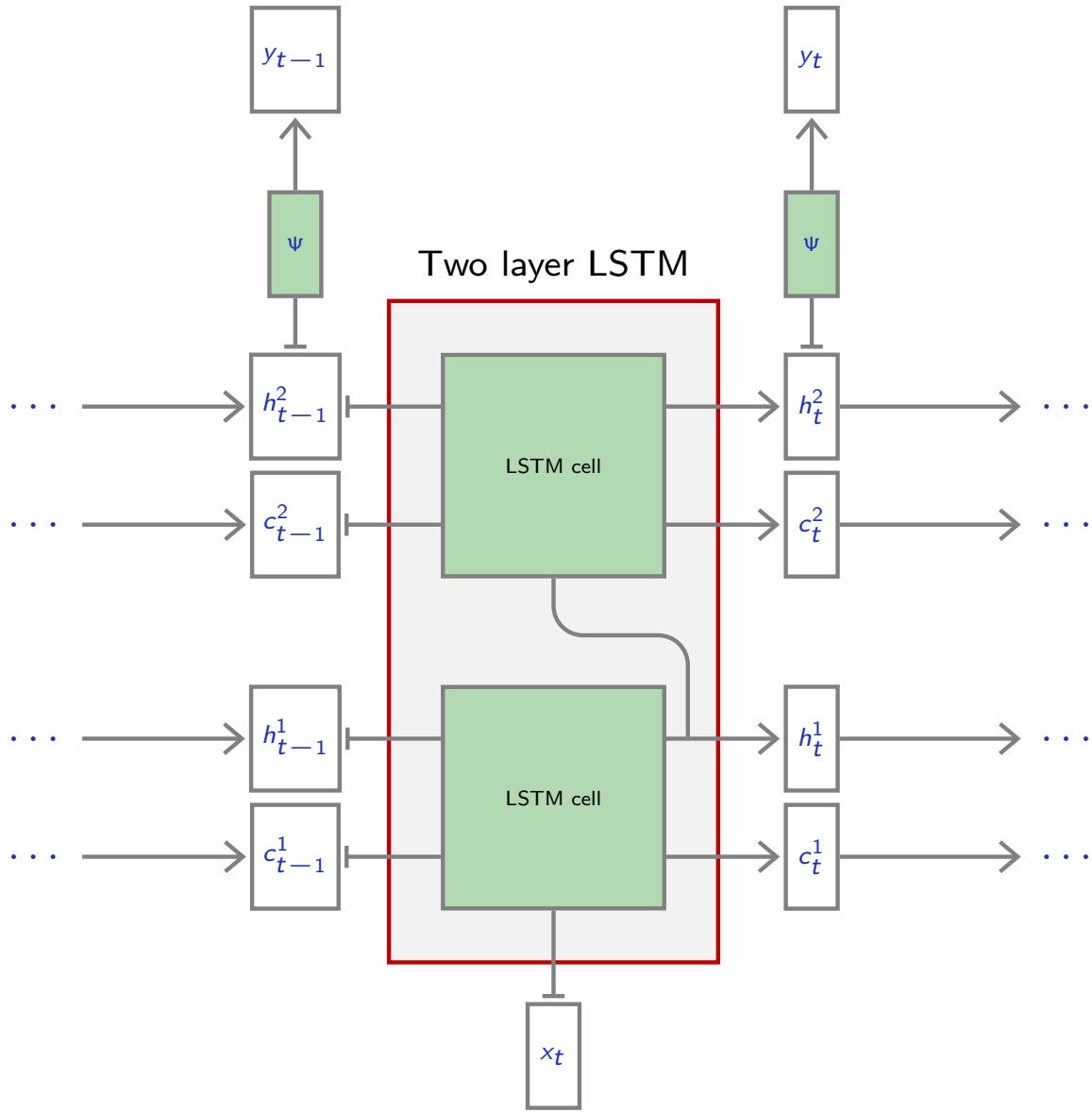
Several such "cells" can be combined to create a multi-layer LSTM.



Several such "cells" can be combined to create a multi-layer LSTM.



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`:

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]))
```

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.

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`.

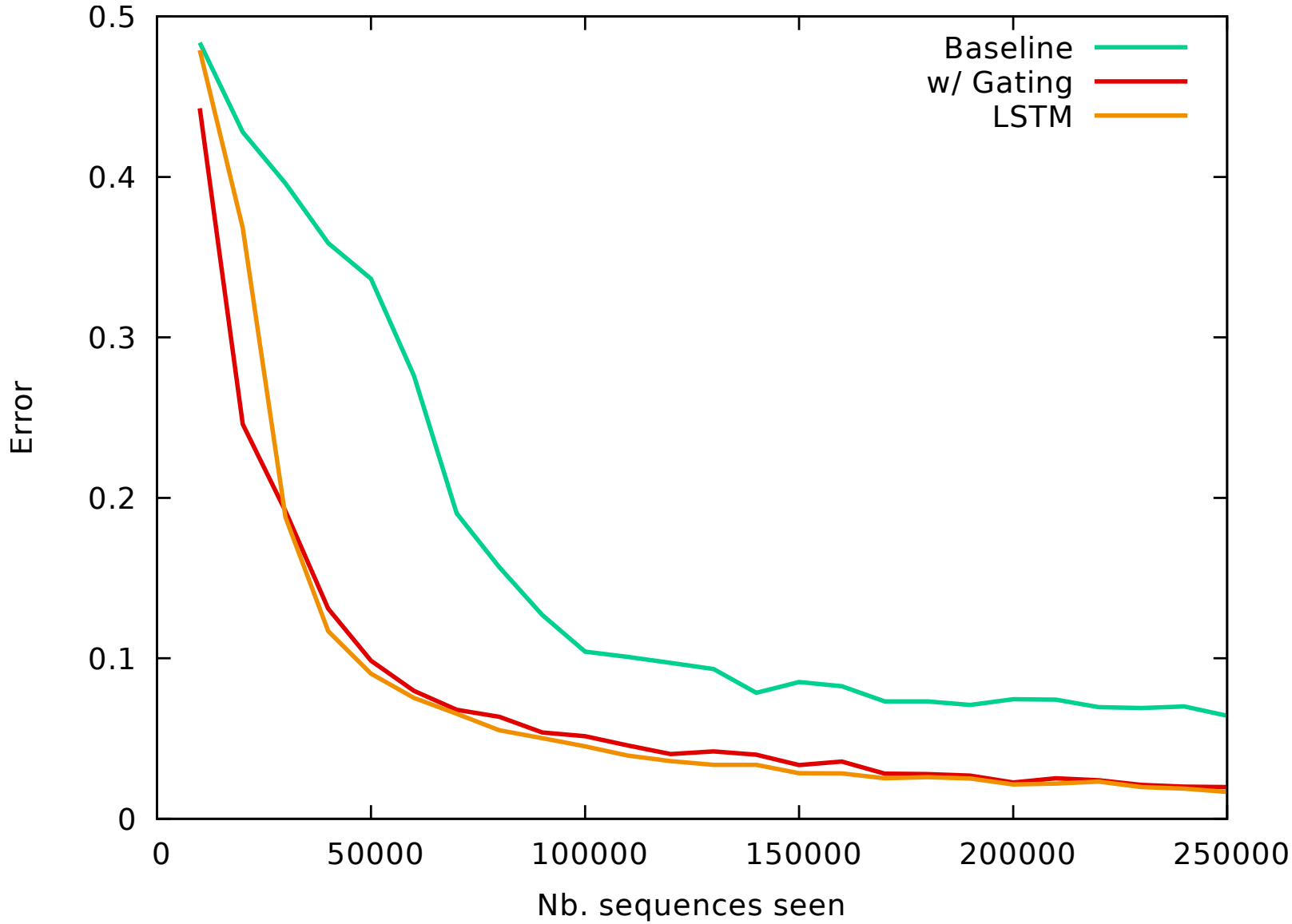
Such an object can be created with `nn.utils.rnn.pack_padded_sequence`:

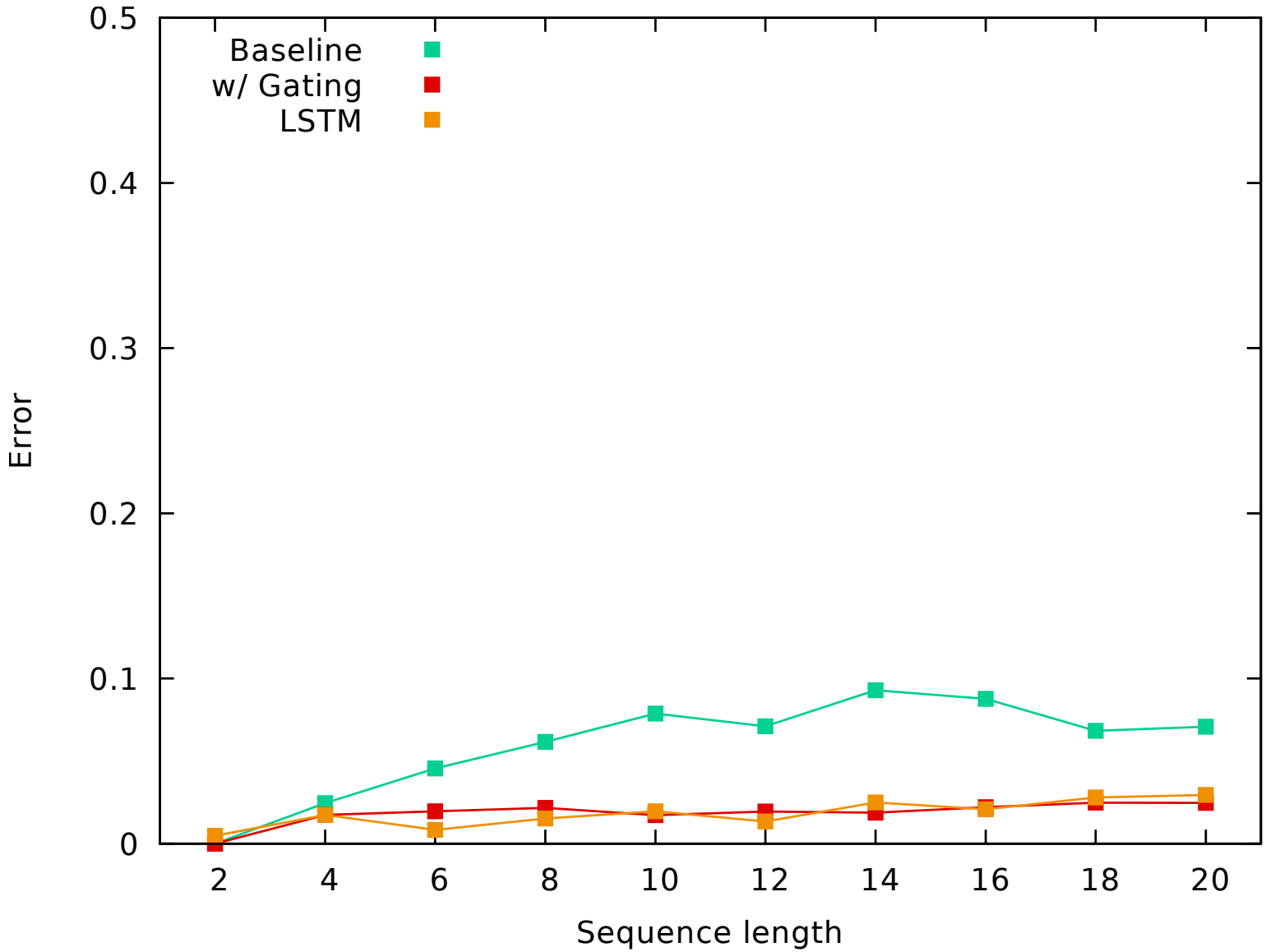
```
>>> from torch.nn.utils.rnn import pack_padded_sequence
>>> pack_padded_sequence(torch.tensor([[[ 1. ], [ 2. ]],
...                                    [[ 3. ], [ 4. ]],
...                                    [[ 5. ], [ 0. ]]]),
...                      [3, 2])
PackedSequence(data=tensor([[ 1.],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


Baseline
w/ Gating
LSTM
Error
Nb. sequences seen
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000


Baseline
w/ Gating
LSTM
Error
Sequence length
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
12
14
16
18
20

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.

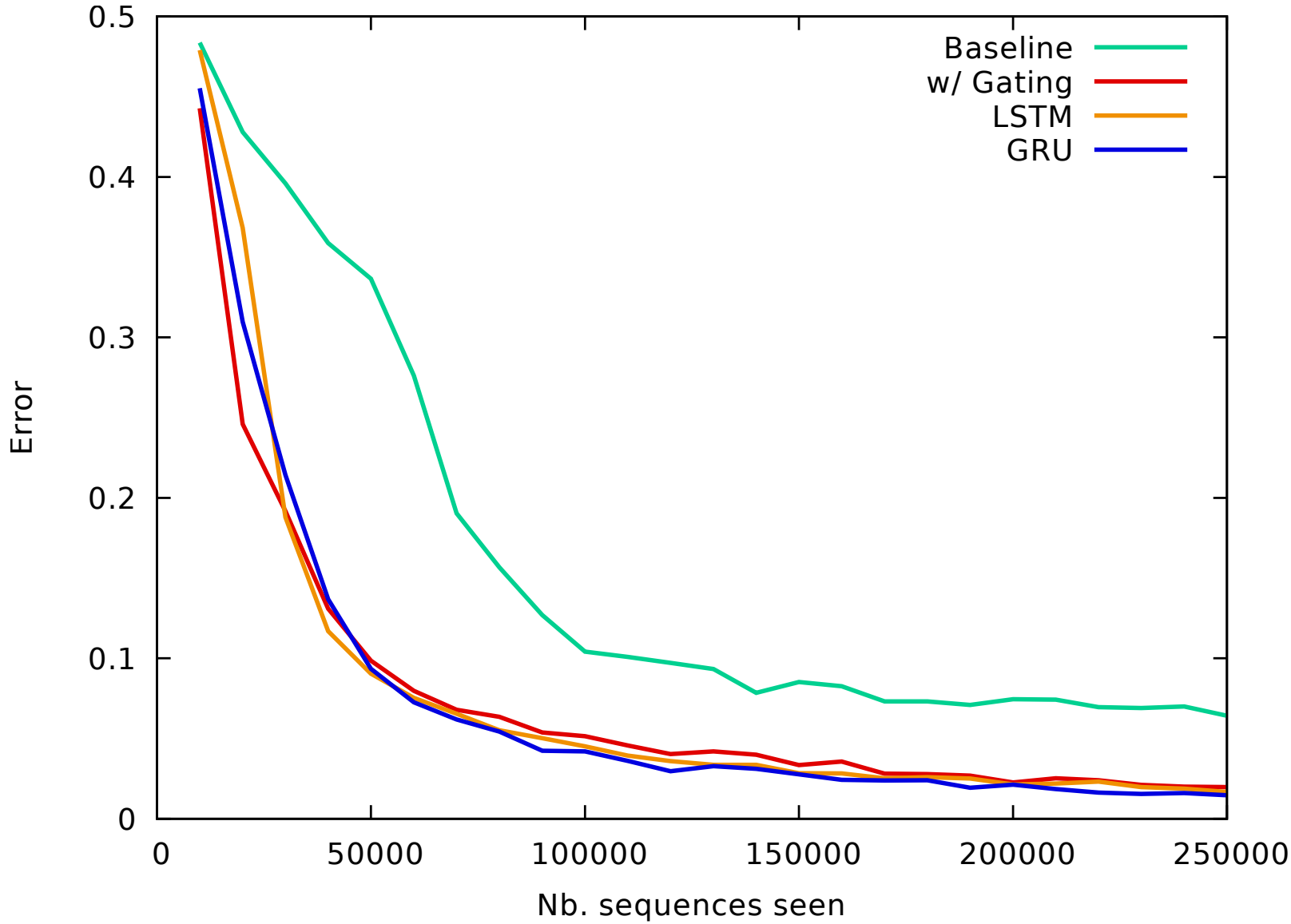
$$r_t = \text{sigm}\left(W_{(\text{x r})} x_t + W_{(\text{h r})} h_{t-1} + b_{(\text{r})}\right) \quad \text{(reset gate)}$$

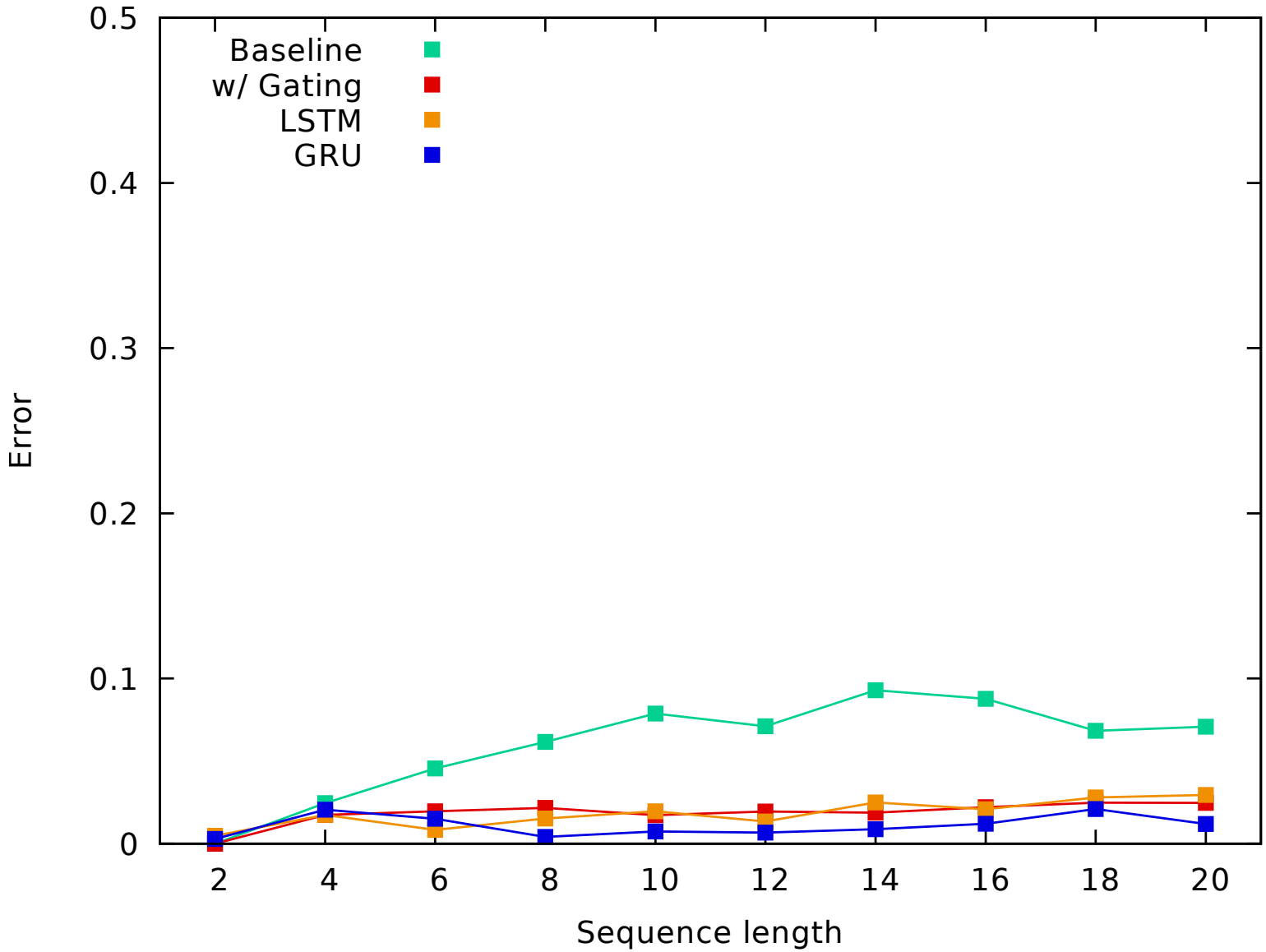
$$z_t = \text{sigm}\left(W_{(\text{x z})} x_t + W_{(\text{h z})} h_{t-1} + b_{(\text{z})}\right) \quad \text{(forget gate)}$$

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

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

```
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))
```


Baseline
w/ Gating
LSTM
GRU
Error
0.5
0.4
0.3
0.2
0.1
0
0
50000
100000
150000
200000
250000
Nb. sequences seen


Baseline
w/ Gating
LSTM
GRU
Error
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
12
14
16
18
20
Sequence length

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)

The end

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.

EE-559 – Deep learning

11.3. Word embeddings and translation

François Fleuret

https://fleuret.org/ee559/

Mon Feb 18 13:33:29 UTC 2019

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Word embeddings and CBOW

An important application domain for machine intelligence is Natural Language Processing (NLP).

- Speech and (hand)writing recognition,
- auto-captioning,
- part-of-speech tagging,
- sentiment prediction,
- translation,
- question answering.

An important application domain for machine intelligence is Natural Language Processing (NLP).

- Speech and (hand)writing recognition,
- auto-captioning,
- part-of-speech tagging,
- sentiment prediction,
- translation,
- question answering.

While language modeling was historically addressed with formal methods, in particular generative grammars, state-of-the-art and deployed methods are now heavily based on statistical learning and deep learning.

A core difficulty of Natural Language Processing is to devise a proper density model for sequences of words.

However, since a vocabulary is usually of the order of $10^4 - 10^6$ words, empirical distributions can not be estimated for more than triplets of words.

The standard strategy to mitigate this problem is to embed words into a geometrical space to take advantage of data regularities for further [statistical] modeling.

The standard strategy to mitigate this problem is to embed words into a geometrical space to take advantage of data regularities for further [statistical] modeling.

The geometry after embedding should account for synonymy, but also for identical word classes, etc. *E.g.* we would like such an embedding to make "cat" and "tiger" close, but also "red" and "blue", or "eat" and "work", etc.

The standard strategy to mitigate this problem is to embed words into a geometrical space to take advantage of data regularities for further [statistical] modeling.

The geometry after embedding should account for synonymy, but also for identical word classes, etc. *E.g.* we would like such an embedding to make "cat" and "tiger" close, but also "red" and "blue", or "eat" and "work", etc.

Even though they are not "deep", classical word embedding models are key elements of NLP with deep-learning.

Let

$$k_t \in \{1, \ldots, W\}, \ t = 1, \ldots, T$$

be a training sequence of T words, encoded as IDs through a vocabulary of W words.

Let

$$k_t \in \{1, \ldots, W\}, \ t = 1, \ldots, T$$

be a training sequence of T words, encoded as IDs through a vocabulary of W words.

Given an embedding dimension D, the objective is to learn vectors

$$E_k \in \mathbb{R}^D, \ k \in \{1, \ldots, W\}$$

so that "similar" words are embedded with "similar" vectors.

A common word embedding is the Continuous Bag of Words (CBOW) version of word2vec (Mikolov et al., 2013a).

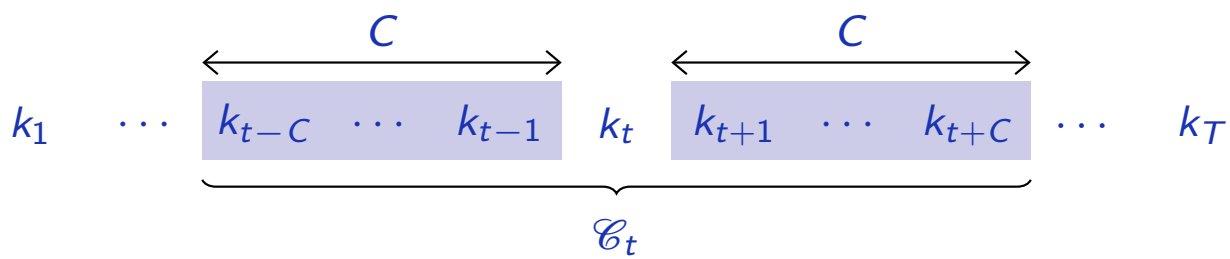
A common word embedding is the Continuous Bag of Words (CBOW) version of word2vec (Mikolov et al., 2013a).

In this model, they embedding vectors are chosen so that a word can be predicted from [a linear function of] the sum of the embeddings of words around it.

More formally, let $C \in \mathbb{N}^*$ be a "context size", and

$$\mathscr{C}_t = (k_{t-C}, \ldots, k_{t-1}, k_{t+1}, \ldots, k_{t+C})$$

be the "context" around k_t, that is the indexes of words around it.



The embeddings vectors

$$E_k \in \mathbb{R}^D, \ k = 1, \ldots, W,$$

are optimized jointly with an array

$$M \in \mathbb{R}^{W \times D}$$

so that the predicted vector of W scores

$$\psi(t) = M \sum_{k \in \mathscr{C}_t} E_k$$

is a good predictor of the value of k_t.

Ideally we would minimize the cross-entropy between the vector of scores $\psi(t) \in \mathbb{R}^W$ and the class k_t

$$\sum_t -\log \left(\frac{\exp \psi(t)_{k_t}}{\sum_{k=1}^W \exp \psi(t)_k} \right).$$

However, given the vocabulary size, doing so is numerically unstable and computationally demanding.

The "negative sampling" approach uses a loss estimated on the prediction for the correct class k_t and only $Q \ll W$ incorrect classes $\kappa_{t,1}, \ldots, \kappa_{t,Q}$ sampled at random.

In our implementation we take the later uniformly in $\{1, \ldots, W\}$ and use the same loss as Mikolov et al. (2013b):

$$\sum_t \log\left(1 + e^{-\psi(t)_{k_t}}\right) + \sum_{q=1}^{Q} \log\left(1 + e^{\psi(t)_{\kappa_{t,q}}}\right).$$

We want $\psi(t)_{k_t}$ to be large and all the $\psi(t)_{\kappa_{t,q}}$ to be small.

Although the operation

$$x \mapsto E_x$$

could be implemented as the product between a one-hot vector and a matrix, it is far more efficient to use an actual lookup table.

The PyTorch module `nn.Embedding` does precisely that. It is parametrized with a number N of words to embed, and an embedding dimension D.

It gets as input an integer tensor of arbitrary dimension $A_1 \times \cdots \times A_U$, containing values in $\{0, \ldots, N-1\}$ and it returns a float tensor of dimension $A_1 \times \cdots \times A_U \times D$.

If w are the embedding vectors, x the input tensor, y the result, we have

$$y[a_1, \ldots, a_U, d] = w[x[a_1, \ldots, a_U]][d].$$

```
>>> e = nn.Embedding(10, 3)
>>> x = torch.tensor([[1, 1, 2, 2], [0, 1, 9, 9]], dtype = torch.int64)
>>> e(x)
tensor([[[ 0.0386, -0.5513, -0.7518],
         [ 0.0386, -0.5513, -0.7518],
         [-0.4033,  0.6810,  0.1060],
         [-0.4033,  0.6810,  0.1060]],

        [[-0.5543, -1.6952,  1.2366],
         [ 0.0386, -0.5513, -0.7518],
         [ 0.2793, -0.9632,  1.6280],
         [ 0.2793, -0.9632,  1.6280]]])
```

Our CBOW model has as parameters two embeddings

$$E \in \mathbb{R}^{W \times D} \quad \text{and} \quad M \in \mathbb{R}^{W \times D}.$$

Its `forward` gets as input a pair of integer tensors corresponding to a batch of size B:

- c of size $B \times 2C$ contains the IDs of the words in a context, and
- d of size $B \times R$ contains the IDs, for each of the B contexts, of the R words for which we want the prediction score (that will be the correct one and Q negative ones).

it returns a tensor y of size $B \times R$ containing the dot products.

$$y[n,j] = \frac{1}{D} M_{d[n,j]} \cdot \left(\sum_i E_{c[n,i]} \right).$$

```
class CBOW(nn.Module):
    def __init__(self, voc_size = 0, embed_dim = 0):
        super(CBOW, self).__init__()
        self.embed_dim = embed_dim
        self.embed_E = nn.Embedding(voc_size, embed_dim)
        self.embed_M = nn.Embedding(voc_size, embed_dim)

    def forward(self, c, d):
        sum_w_E = self.embed_E(c).sum(1).unsqueeze(1).transpose(1, 2)
        w_M = self.embed_M(d)
        return w_M.matmul(sum_w_E).squeeze(2) / self.embed_dim
```

Regarding the loss, we can use `nn.BCEWithLogitsLoss` which implements

$$\sum_t y_t \log(1 + \exp(-x_t)) + (1 - y_t) \log(1 + \exp(x_t)).$$

It takes care in particular of the numerical problem that may arise for large values of x_t if implemented "naively".

Before training a model, we need to prepare data tensors of word IDs from a text file. We will use a 100Mb text file taken from Wikipedia and

- make it lower-cap,
- remove all non-letter characters,
- replace all words that appear less than 100 times with '*',
- associate to each word a unique id.

From the resulting sequence of length T stored in a integer tensor, and the context size C, we will generate mini-batches, each of two tensors

- a 'context' integer tensor c of dimension $B \times 2C$, and
- a 'word' integer tensor w of dimension B.

If the corpus is "The black cat plays with the black ball.", we will get the following word IDs:

`the`: 0, `black`: 1, `cat` : 2, `plays`: 3, `with`: 4, `ball`: 5.

The corpus will be encoded as

`the`	`black`	`cat`	`plays`	`with`	`the`	`black`	`ball`
0	1	2	3	4	0	1	5

If the corpus is "The black cat plays with the black ball.", we will get the following word IDs:

`the`: 0, `black`: 1, `cat` : 2, `plays`: 3, `with`: 4, `ball`: 5.

The corpus will be encoded as

`the`	`black`	`cat`	`plays`	`with`	`the`	`black`	`ball`
0	1	2	3	4	0	1	5

and the data and label tensors will be

Words					IDs					c	w
the	black	cat	plays	with	0	1	2	3	4	$0, 1, 3, 4$	2
black	cat	plays	with	the	1	2	3	4	0	$1, 2, 4, 0$	3
cat	plays	with	the	black	2	3	4	0	1	$2, 3, 0, 1$	4
plays	with	the	black	ball	3	4	0	1	5	$3, 4, 1, 5$	0

We can train the model for an epoch with:

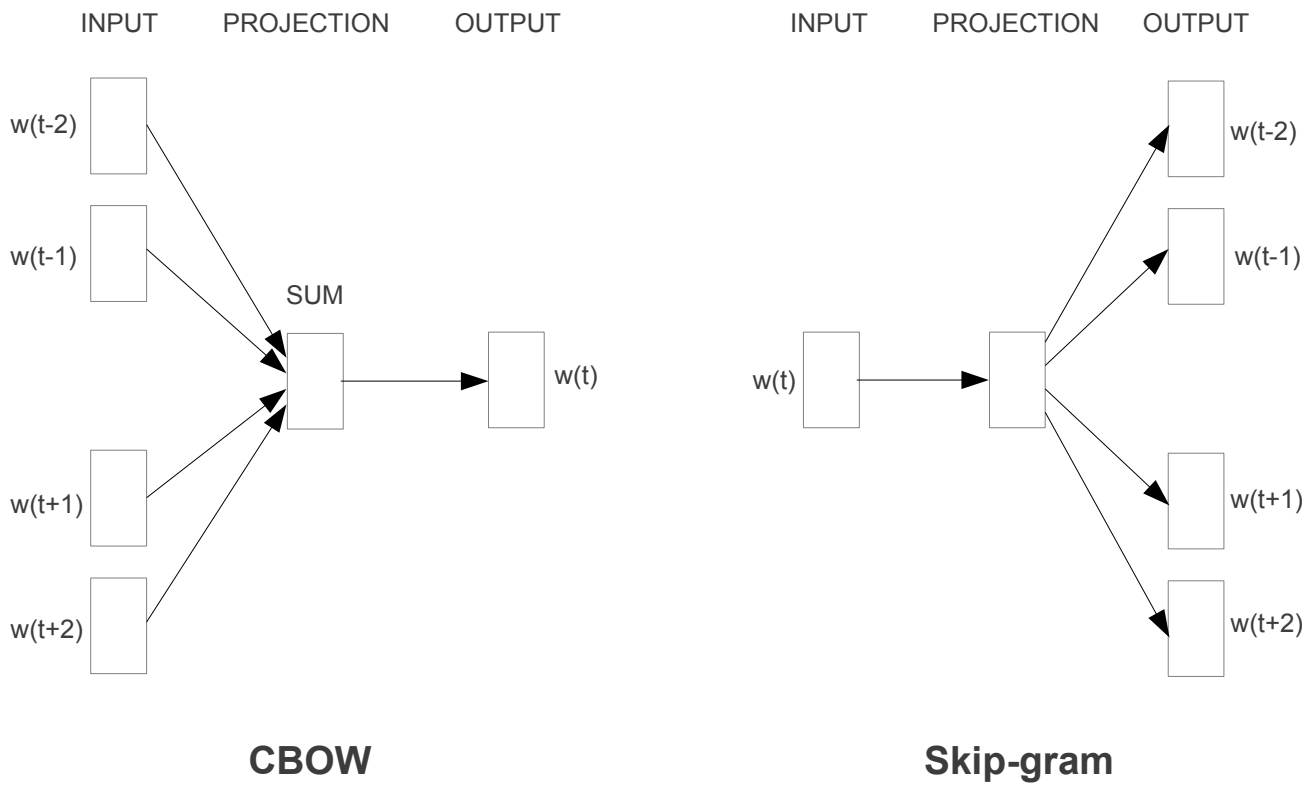
```
for k in range(0, id_seq.size(0) - 2 * context_size - batch_size, batch_size):
    c, w = extract_batch(id_seq, k, batch_size, context_size)

    d = torch.empty(w.size(0), 1 + nb_neg_samples, dtype = torch.int64)
    d.random_(voc_size)
    d[:, 0] = w

    target = torch.empty(d.size())
    target.narrow(1, 0, 1).fill_(1)
    target.narrow(1, 1, nb_neg_samples).fill_(0)

    output = model(c, d)
    loss = bce_loss(output, target)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Some nearest-neighbors for the cosine distance between the embeddings

$$d(w, w') = \frac{E_w \cdot E_{w'}}{\|E_w\| \|E_{w'}\|}.$$

	paris		bike		cat		fortress		powerful
0.61	parisian	0.61	bicycle	0.55	cats	0.61	fortresses	0.47	formidable
0.59	france	0.51	bicycles	0.54	dog	0.55	citadel	0.44	power
0.55	brussels	0.51	bikes	0.49	kitten	0.55	castle	0.44	potent
0.53	bordeaux	0.49	biking	0.44	feline	0.52	fortifications	0.40	fearsome
0.51	toulouse	0.47	motorcycle	0.42	pet	0.51	forts	0.40	destroy
0.51	vienna	0.43	cyclists	0.40	dogs	0.50	siege	0.39	wielded
0.51	strasbourg	0.42	riders	0.40	kittens	0.49	stronghold	0.38	versatile
0.49	munich	0.41	sled	0.39	hound	0.49	castles	0.38	capable
0.49	marseille	0.41	triathlon	0.39	squirrel	0.48	monastery	0.38	strongest
0.48	rouen	0.41	car	0.38	mouse	0.48	besieged	0.37	able

An alternative algorithm is the skip-gram model, which optimizes the embedding so that a word can be predicted by any individual word in its context (Mikolov et al., 2013a).



(Mikolov et al., 2013a)

Trained on large corpora, such models reflect semantic relations in the linear structure of the embedding space. *E.g.*

$$E_{[paris]} - E_{[france]} + E_{[italy]} \simeq E_{[rome]}$$

Trained on large corpora, such models reflect semantic relations in the linear structure of the embedding space. *E.g.*

$$E_{[paris]} - E_{[france]} + E_{[italy]} \simeq E_{[rome]}$$

Table 8: *Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).*

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

(Mikolov et al., 2013a)

The main benefit of word embeddings is that they are trained with unsupervised corpora, hence possibly extremely large.

This modeling can then be leveraged for small-corpora tasks such as

- sentiment analysis,
- question answering,
- topic classification,
- etc.

Sequence-to-sequence translation

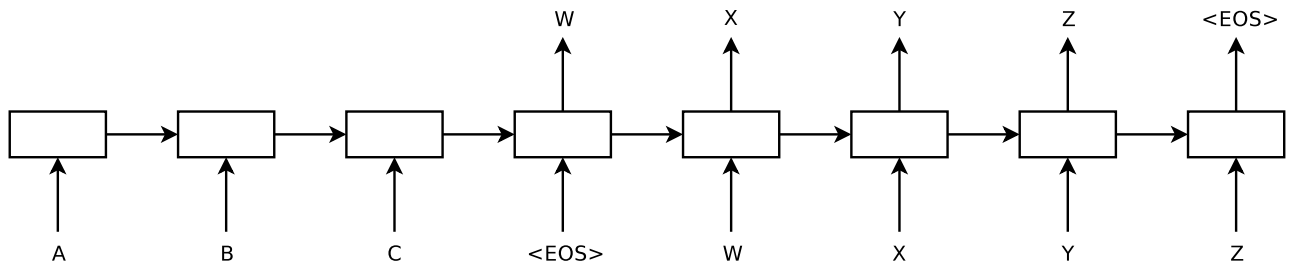


Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

(Sutskever et al., 2014)

English to French translation.

Training:

- corpus 12M sentences, 348M French words, 30M English words,
- LSTM with 4 layers, one for encoding, one for decoding,
- $160,000$ words input vocabulary, $80,000$ output vocabulary,
- $1,000$ dimensions word embedding, 384M parameters total,
- input sentence is reversed,
- gradient clipping.

The hidden state that contains the information to generate the translation is of dimension $8,000$.

Inference is done with a "beam search", that consists of greedily increasing the size of the predicted sequence while keeping a bag of K best ones.

Comparing a produced sentence to a reference one is complex, since it is related to their semantic content.

A widely used measure is the BLEU score, that counts the fraction of groups of one, two, three and four words (aka "n-grams") from the generated sentence that appear in the reference translations (Papineni et al., 2002).

The exact definition is complex, and the validity of this score is disputable since it poorly accounts for semantic.

Method	test BLEU score (ntst14)
Bahdanau et al. [2]	28.45
Baseline System [29]	33.30
Single forward LSTM, beam size 12	26.17
Single reversed LSTM, beam size 12	30.59
Ensemble of 5 reversed LSTMs, beam size 1	33.00
Ensemble of 2 reversed LSTMs, beam size 12	33.27
Ensemble of 5 reversed LSTMs, beam size 2	34.50
Ensemble of 5 reversed LSTMs, beam size 12	**34.81**

Table 1: The performance of the LSTM on WMT'14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than of a single LSTM with a beam of size 12.

(Sutskever et al., 2014)

Type	Sentence
Our model	Ulrich UNK , membre du conseil d' administration du constructeur automobile Audi , affirme qu' il s' agit d' une pratique courante depuis des années pour que les téléphones portables puissent être collectés avant les réunions du conseil d' administration afin qu' ils ne soient pas utilisés comme appareils d' écoute à distance .
Truth	Ulrich Hackenberg , membre du conseil d' administration du constructeur automobile Audi , déclare que la collecte des téléphones portables avant les réunions du conseil , afin qu' ils ne puissent pas être utilisés comme appareils d' écoute à distance , est une pratique courante depuis des années .
Our model	" Les téléphones cellulaires , qui sont vraiment une question , non seulement parce qu' ils pourraient potentiellement causer des interférences avec les appareils de navigation , mais nous savons , selon la FCC , qu' ils pourraient interférer avec les tours de téléphone cellulaire lorsqu' ils sont dans l' air " , dit UNK .
Truth	" Les téléphones portables sont véritablement un problème , non seulement parce qu' ils pourraient éventuellement créer des interférences avec les instruments de navigation , mais parce que nous savons , d' après la FCC , qu' ils pourraient perturber les antennes-relais de téléphonie mobile s' ils sont utilisés à bord " , a déclaré Rosenker .
Our model	Avec la crémation , il y a un " sentiment de violence contre le corps d' un être cher " , qui sera " réduit à une pile de cendres " en très peu de temps au lieu d' un processus de décomposition " qui accompagnera les étapes du deuil " .
Truth	Il y a , avec la crémation , " une violence faite au corps aimé " , qui va être " réduit à un tas de cendres " en très peu de temps , et non après un processus de décomposition , qui " accompagnerait les phases du deuil " .

Table 3: A few examples of long translations produced by the LSTM alongside the ground truth translations. The reader can verify that the translations are sensible using Google translate.

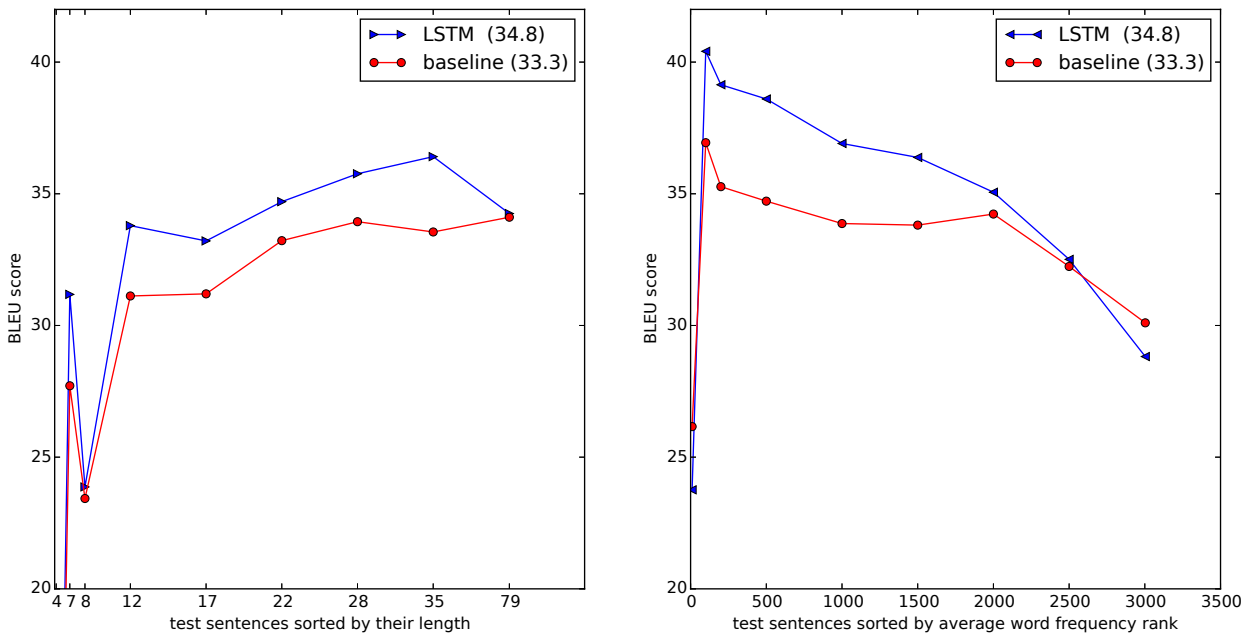


Figure 3: The left plot shows the performance of our system as a function of sentence length, where the x-axis corresponds to the test sentences sorted by their length and is marked by the actual sequence lengths. There is no degradation on sentences with less than 35 words, there is only a minor degradation on the longest sentences. The right plot shows the LSTM's performance on sentences with progressively more rare words, where the x-axis corresponds to the test sentences sorted by their "average word frequency rank".

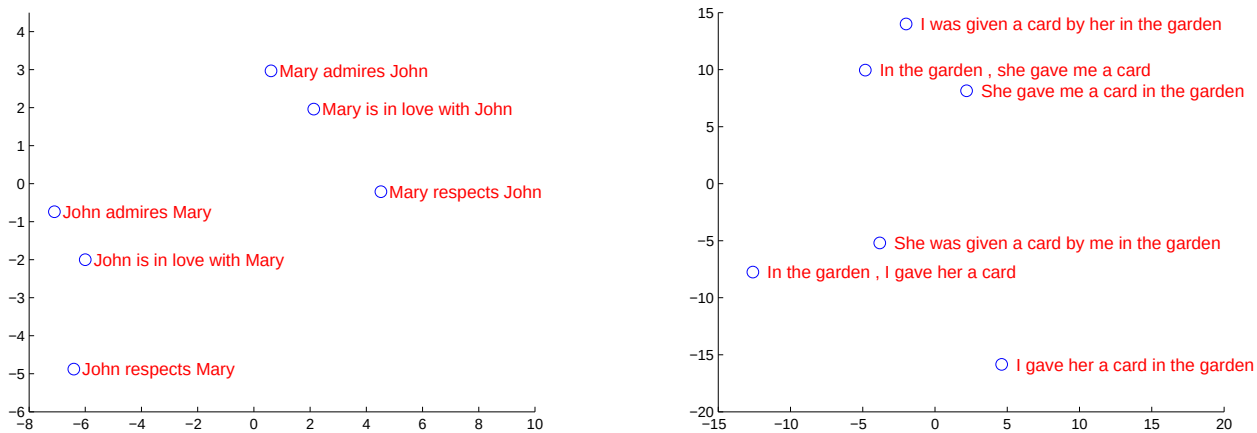


Figure 2: The figure shows a 2-dimensional PCA projection of the LSTM hidden states that are obtained after processing the phrases in the figures. The phrases are clustered by meaning, which in these examples is primarily a function of word order, which would be difficult to capture with a bag-of-words model. Notice that both clusters have similar internal structure.

(Sutskever et al., 2014)

The end

References

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013a.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Neural Information Processing Systems (NIPS)*, 2013b.

K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318. Association for Computational Linguistics, 2002.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014.