Deep learning

12.1. Recurrent Neural Networks

François Fleuret

https://fleuret.org/dlc/

Inference from sequences

Many real-world problems require to process a signal with a sequence structure of variable size. E.g.

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.

Notes

The main motivation for using recurrent networks is to deal with signals of variable length.
Sequence classification:

- sentiment analysis: given a snippet of text such as the review of a movie, predict for instance if it is positive or negative,
- activity/action recognition: given a video, predict what is happening,
- DNA sequence classification: given gene expression measures, predict phenotypical properties of the cell or tissue,
- action selection: given the previous states of an agent, predict the optimal action to perform next.

Sequence synthesis:

- text or music synthesis: the same as for fixed-size synthesis, but here the output is of variable length,
- motion synthesis: the goal is to generate the degrees of freedom and the control values of an agent, for instance for CGI.

Sequence-to-sequence translation:

- speech recognition: going from sound to text (e.g. for video auto-captioning),
- text translation: generate a snippet of text in a target language with the same meaning as a snippet of text in a source language,
- part-of-speech tagging: tag each word in a sentence with its role in the text (noun, verb, etc.)

Given a set $\mathcal{X}$, if $S(\mathcal{X})$ is 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 signals, although all techniques generalize to arbitrary sequences.

Notes

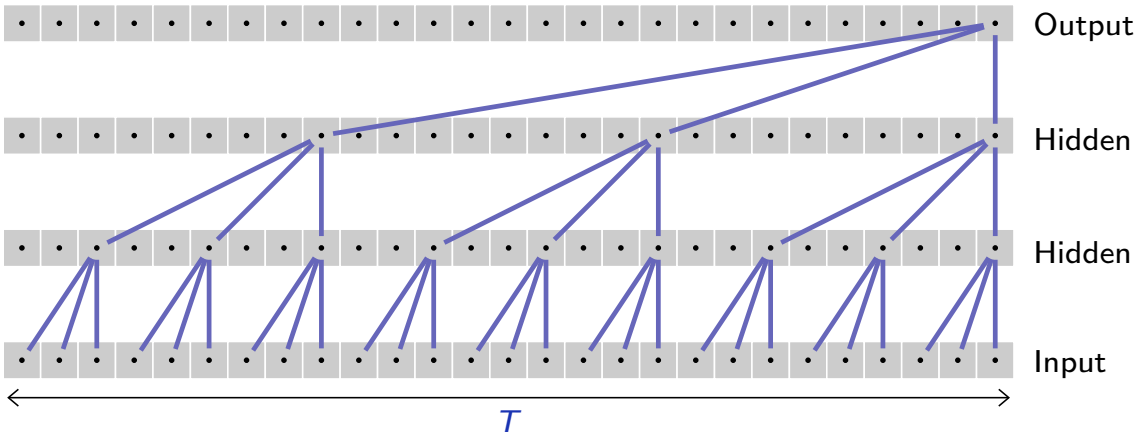
$\mathcal{X}$ is the set of symbols which can appear in the sequence. $\mathcal{X}^t$ denotes the set of sequences made of t elements. $S(\mathcal{X})$ is therefore the set of all the possible sequences which can be made from symbols from $\mathcal{X}$.
In NLP for instance, $\mathcal{X}$ is usually a set of parts of words ("subwords").

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.

There has been a renewal of interest since 2018 for such methods for computational reasons, since they are more amenable to batch processing.



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)

Notes

As we will see, the three main types of recurrent layers are "vanilla" RNN, GRU, and LSTM. Temporal convolutions networks (TCN) are competitive with recurrent networks on standard tasks and datasets.

RNN and backprop through time

The historical approach to processing sequences of variable size relies on a recurrent model which maintains a **recurrent state** updated at each time step.

With $\mathcal{X} = \mathbb{R}^D$, given

$$\Phi(\cdot\,; w) : \mathbb{R}^D \times \mathbb{R}^Q \to \mathbb{R}^Q,$$

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

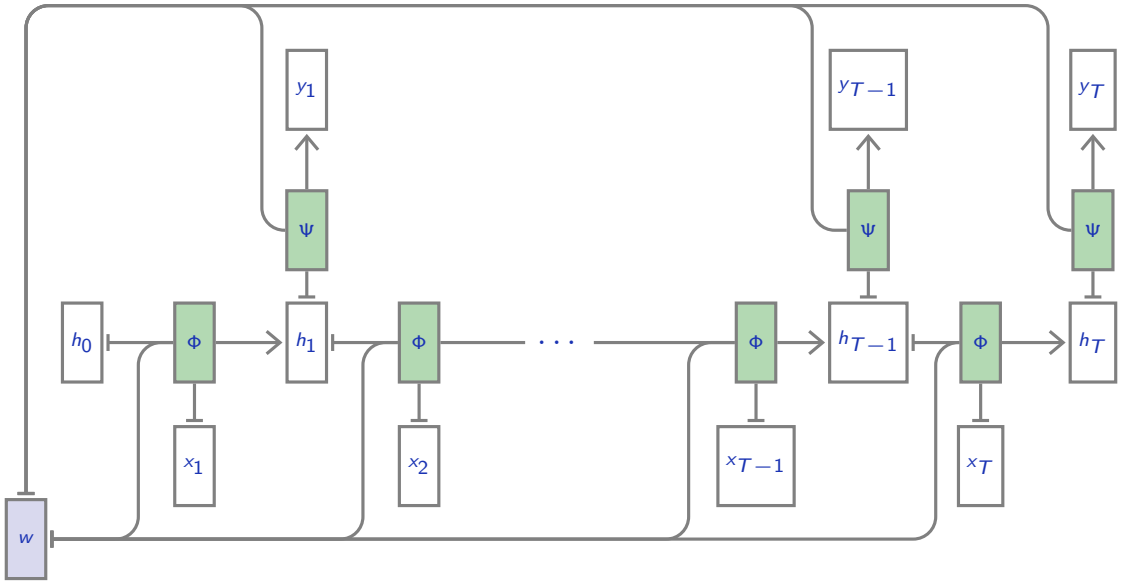
$$\forall t = 1, \ldots, T(x), \;\; h_t = \Phi(x_t, h_{t-1}; w).$$

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

$$y_t = \Psi(h_t; w)$$

with a "readout" function

$$\Psi(\cdot\,; w) : \mathbb{R}^Q \to \mathbb{R}^C.$$



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).

Notes

The operations described previously can be unfolded in time and result in a "directed acyclic graph" as seen in lecture 4.1. "DAG networks", with heavy weight sharing, since w is used at each time step.

We consider the following simple binary sequence classification problem:

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

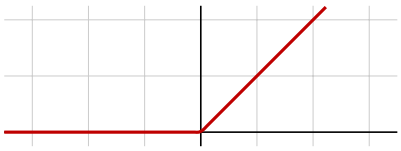
E.g.

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:

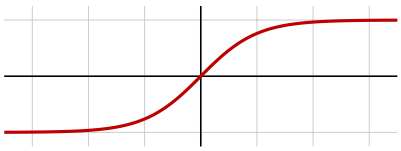
- The rectified linear unit:

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



- The hyperbolic tangent:

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



- The sigmoid:

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

And we encode the symbols as one-hot vectors (see lecture 5.1. "Cross-entropy loss"):

```
>>> nb_symbols = 6
>>> s = torch.tensor([0, 1, 2, 3, 2, 1, 0, 5, 0, 5, 0])
>>> x = F.one_hot(s, num_classes = nb_symbols)
>>> x
tensor([[1, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 1, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0]])
```

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

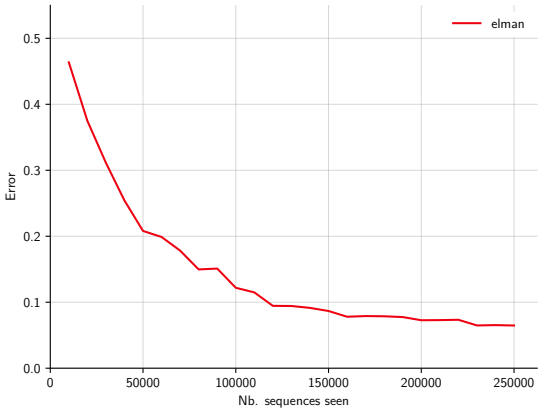
$$y_T = W_{(h\ y)} h_T + b_{(y)}.$$

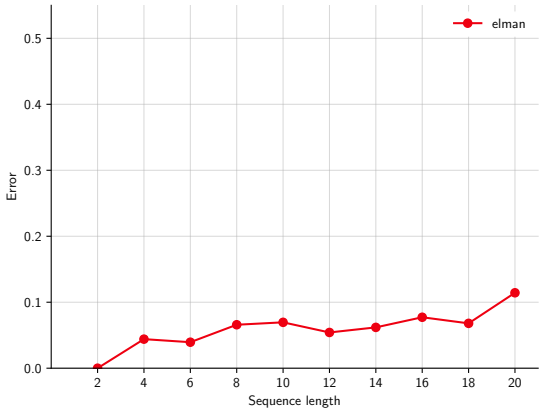
```
class RecNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, dim_output):
        super().__init__()
        self.fc_x2h = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2h = nn.Linear(dim_recurrent, dim_recurrent, bias = False)
        self.fc_h2y = nn.Linear(dim_recurrent, dim_output)

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

For simplicity, we process a batch of sequences of same length.

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.batch_of_one()
    output = model(input)
    loss = cross_entropy(output, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```





Notes

The graph on the left shows the test error as a function of the number of sequences seen during training. The error reaches 7% after 250,000 examples.
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 reasonably good performance of the model shows that it was able to encode [some relevant information about] the first half of the sequence, to do a comparison with the second half.
Note that the task is not that hard since predicting "class 1" if the first element of both halves are identical yields a test error of 5%, since it happens for only 10% of samples of class 0, and for 100% of samples of class 1.

Gating

When unfolded through time, the model depth is proportional to the input length, 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.

This is highly related to the residual connections that were introduced later in computer vision (see lecture 6.5. "Residual networks").

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, acting as 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.

Notes

The gating output z_t indicates component-wise how much of the hidden state should be kept:

- if $z_{t,i} = 1$, the previous state component $h_{t-1,i}$ is used for $h_{t,i}$,
- if $z_{t,i} = 0$, the new update state component $\bar{h}_{t,i}$ is used for $h_{t,i}$.

We can improve our minimal example with such a mechanism, replacing

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

with

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

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

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

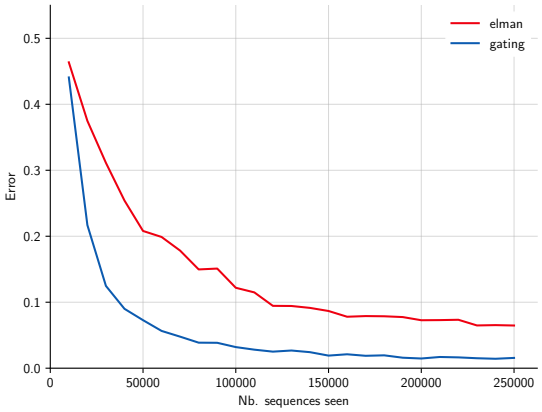
Notes

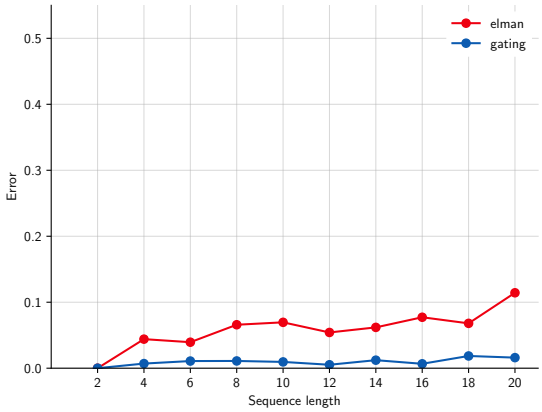
The non-linearity for the forget gate is the sigmoid function so that it takes value in $[0, 1]$.

```
class RecNetWithGating(nn.Module):
    def __init__(self, dim_input, dim_recurrent, dim_output):
        super().__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(input.size(0), self.fc_h2y.weight.size(1))
        for t in range(input.size(1)):
            z = torch.sigmoid(self.fc_x2z(input[:, t]) + self.fc_h2z(h))
            hb = F.relu(self.fc_x2h(input[:, t]) + self.fc_h2h(h))
            h = z * h + (1 - z) * hb
        return self.fc_h2y(h)
```


Notes

The implementation is quite straight forward and very similar to the vanilla one. Once again, for simplicity, we process a single sample as input.





Notes

The left graph shows the test error as a function of the number of sequences seen during training. The graph on the right shows the error as a function of the number of elements in the sequence. The longer the sequence, and the higher the error.

By updating the model with a gating mechanism, which simply computes a per-component weighted average by the previous recurrent state and the full update improves massively the performance of the model.

An intuitive explanation of this improvement is that if the gradient vanishes at one of the time steps, some information can still flow back through time thanks to the pass-through.

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.