Deep learning

12.1. Recurrent Neural Networks

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Inference from sequences

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

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.

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

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

We can define formally:

Sequence classification:	$f : S(\mathscr{X}) \rightarrow \{1, \dots, C\}$
Sequence synthesis:	$f : \mathbb{R}^D \rightarrow S(\mathscr{X})$
Sequence-to-sequence translation:	$f : S(\mathscr{X}) \rightarrow S(\mathscr{Y})$

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

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

We can define formally:

Sequence classification:	$f : S(\mathscr{X}) \rightarrow \{1, \dots, C\}$
Sequence synthesis:	$f : \mathbb{R}^D \rightarrow S(\mathscr{X})$
Sequence-to-sequence translation:	$f : S(\mathscr{X}) \rightarrow S(\mathscr{Y})$

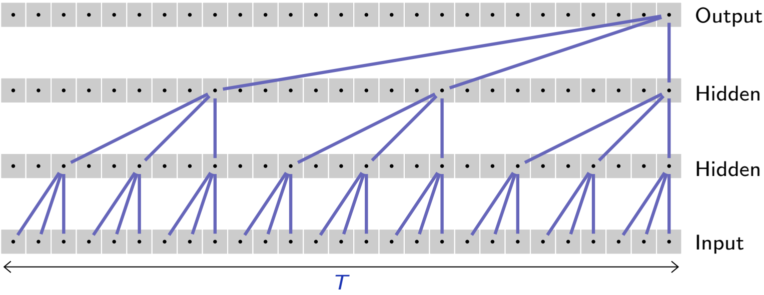
In the rest of the slides we consider only time-indexed signals, although all techniques generalize to arbitrary sequences.

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.

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 $O(T \log T)$.

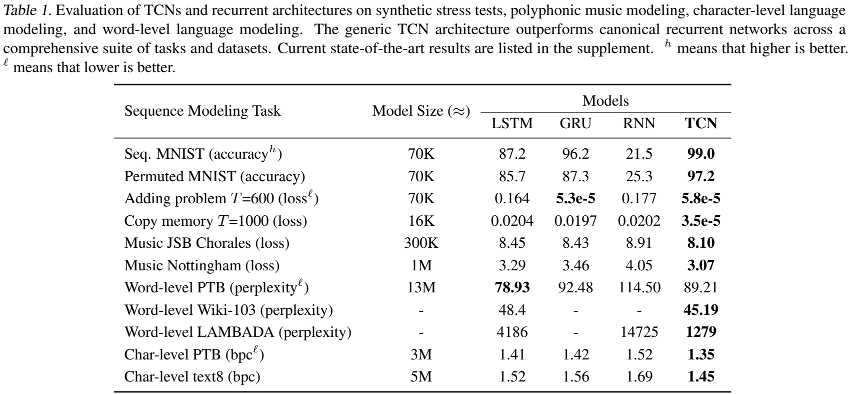


Table 1. Evaluation of TCNs and recurrent architectures on synthetic stress tests, polyphonic music modeling, character-level language modeling, and word-level language modeling. The generic TCN architecture outperforms canonical recurrent networks across a comprehensive suite of tasks and datasets. 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 (≈)	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 historical approach to processing sequences of variable size relies on a recurrent model which maintains a **recurrent state** updated at each time step.

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

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

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

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

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

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

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

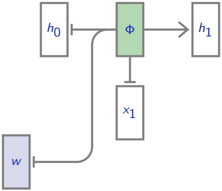
$$\forall t = 1, \dots, T(x),\;\; h_t = \Phi(x_t, h_{t-1}; w).$$

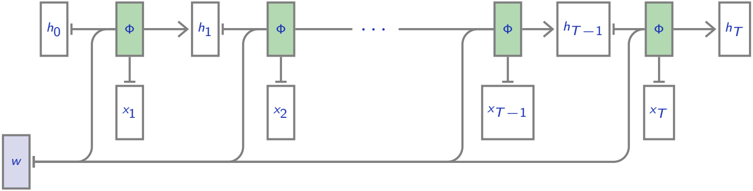
A prediction can be computed at any time step from the recurrent state

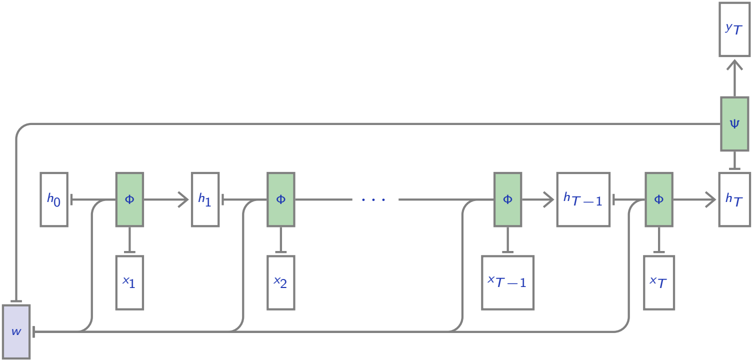
$$y_t = \Psi(h_t; w)$$

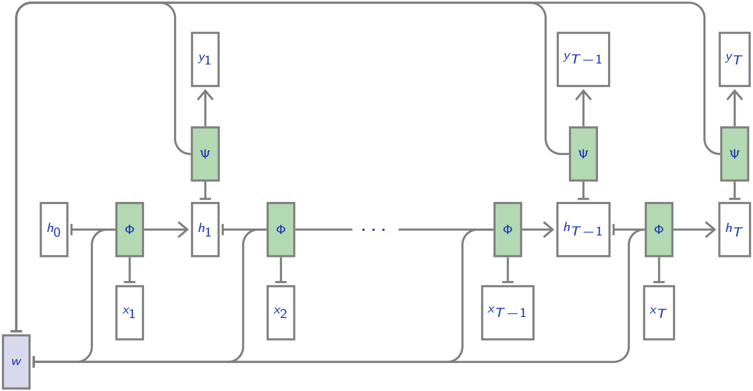
with a "readout" function

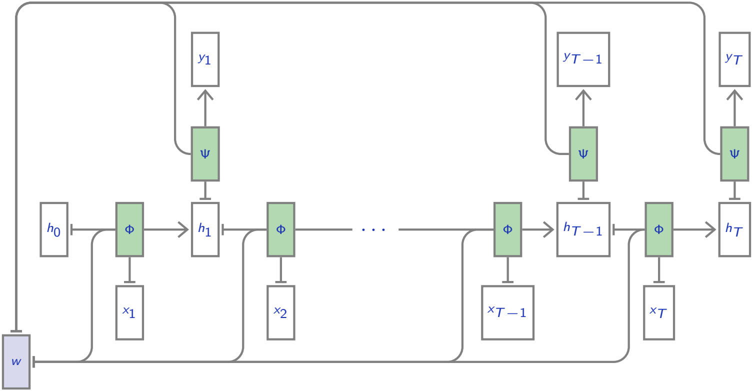
$$\Psi(\,\cdot\,; w) : \mathbb{R}^Q \to \mathbb{R}^C.$$


h_0
Φ
h_1
x_1
w

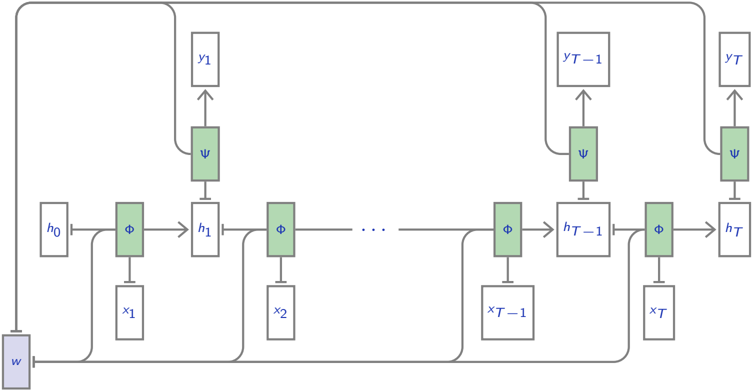

h_0
Φ
h_1
Φ
$\cdots$
Φ
h_{T-1}
Φ
h_T
x_1
x_2
x_{T-1}
x_T
w


y_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
$\cdots$
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.

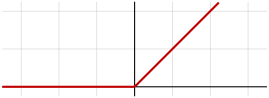
E.g.

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:

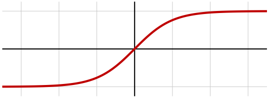
- The rectified linear unit:

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



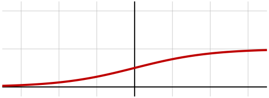
- The hyperbolic tangent:

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



- The sigmoid:

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



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

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

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

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

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)
```

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

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

and the final prediction

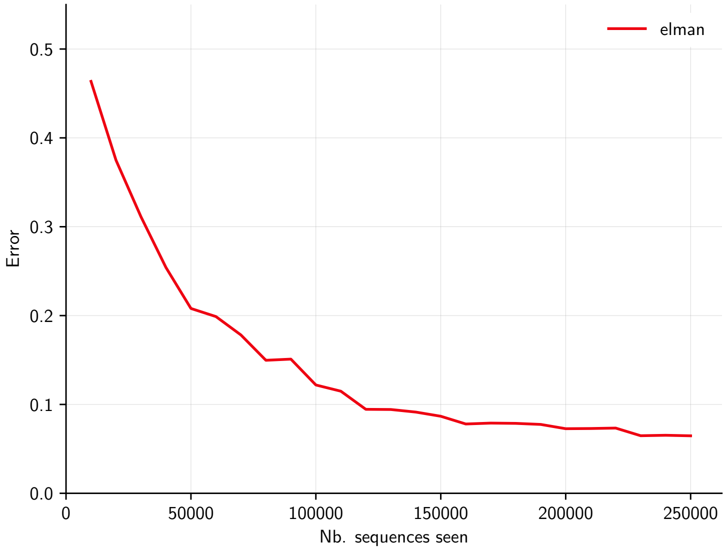
$$y_T = W_{(h\ \ y)} h_T + b_{(y)}.$$

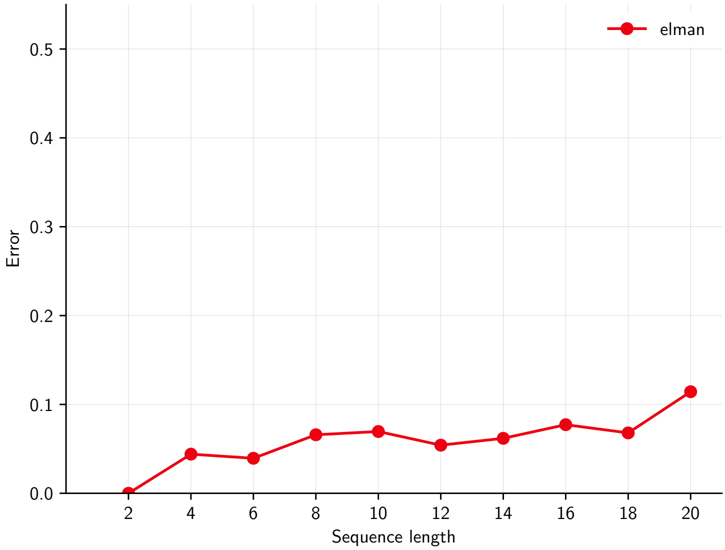
```
class RecNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, dim_output):
        super().__init__()
        self.fc_x2h = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2h = nn.Linear(dim_recurrent, dim_recurrent, bias = False)
        self.fc_h2y = nn.Linear(dim_recurrent, dim_output)

    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()
```


elman
Error
0.0
0.1
0.2
0.3
0.4
0.5
0
50000
100000
150000
200000
250000
Nb. sequences seen


elman
Error
0.0
0.1
0.2
0.3
0.4
0.5
2
4
6
8
10
12
14
16
18
20
Sequence length

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”.

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.

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

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

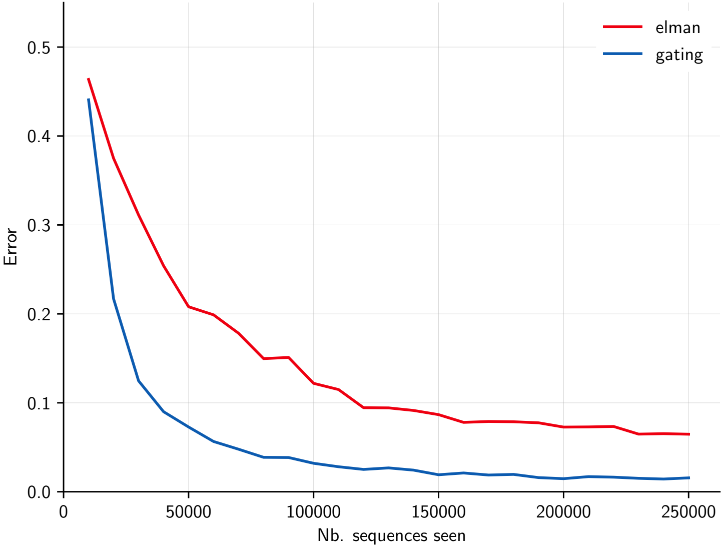
with

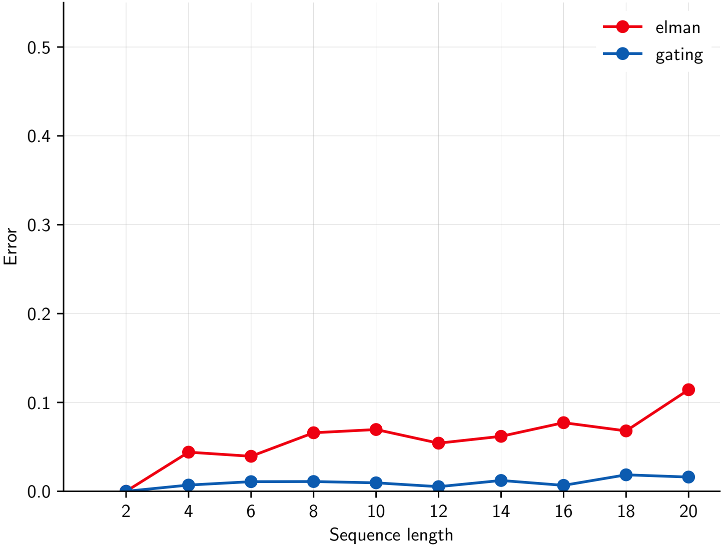
$$\bar{h}_t = \mathrm{ReLU}\left(W_{(x\ h)} x_t + W_{(h\ h)} h_{t-1} + b_{(h)}\right) \qquad \text{(full update)}$$

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

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

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


elman
gating
Error
0.0
0.1
0.2
0.3
0.4
0.5
0
50000
100000
150000
200000
250000
Nb. sequences seen


0.5
0.4
0.3
0.2
0.1
0.0
Error
2
4
6
8
10
12
14
16
18
20
Sequence length
elman
gating

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.