Deep Learning

Lecture 6: Recurrent neural networks

Prof. Gilles Louppe
g.louppe@uliege.be

LIÈGE université

Today

How to make sense of sequential data?

- Temporal convolutions
- Recurrent neural networks
- Applications
- Beyond sequences

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 $S(\mathcal{X})$ denotes the set of sequences of elements from $\mathcal{X}$,

$$S(\mathcal{X}) = \cup_{t=1}^{\infty} \mathcal{X}^t,$$

then we formally define:

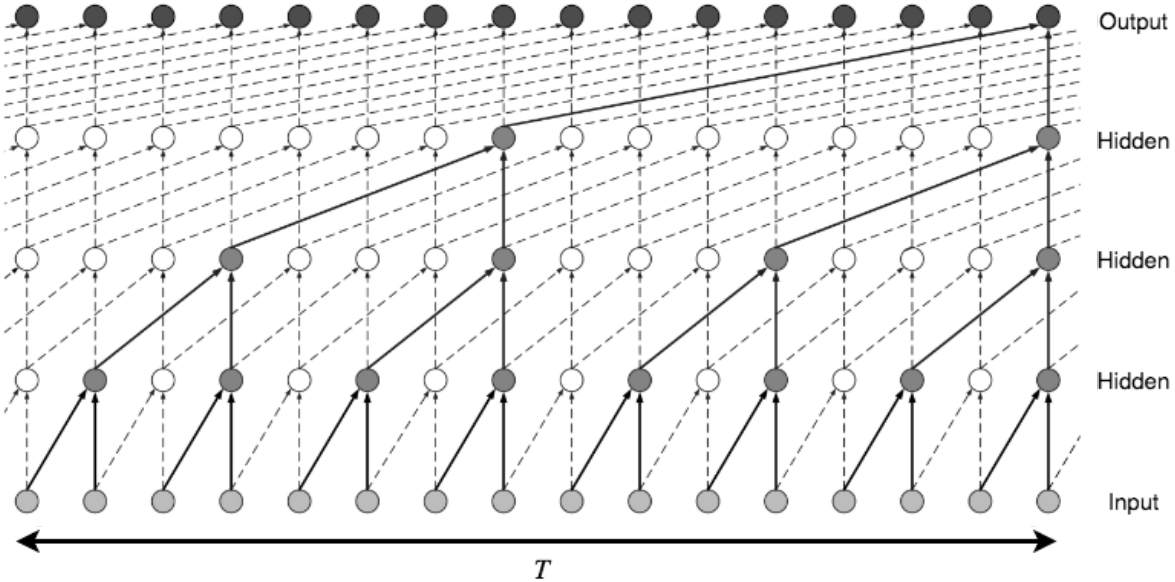
Sequence classification	$f : S(\mathcal{X}) \to \{1, ..., C\}$
Sequence synthesis	$f : \mathbb{R}^d \to S(\mathcal{X})$
Sequence-to-sequence translation	$f : S(\mathcal{X}) \to S(\mathcal{Y})$

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 (TCNs).

TCNs correspond to standard 1D convolutional networks. They process input sequences as fixed-size vectors of the maximum possible length.



Increasing exponentially the kernel sizes makes the required number of layers grow as $O(\log T)$ of the time window T taken into account.

Dilated convolutions make the model size grow as $O(\log T)$, while the memory footprint and computation are $O(T \log T)$.

Credits: Philippe Remy, keras-tcn, 2018; Francois Fleuret, EE559 Deep Learning, EPFL.

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	88.68
Word-level Wiki-103 (perplexity)	-	48.4	-	-	**45.19**
Word-level LAMBADA (perplexity)	-	4186	-	14725	**1279**
Char-level PTB (bpc$^{\ell}$)	3M	1.36	1.37	1.48	**1.31**
Char-level text8 (bpc)	5M	1.50	1.53	1.69	**1.45**

Credits: Bai et al, An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling, 2018.

Recurrent neural networks

When the input is a sequence $\mathbf{x} \in S(\mathbb{R}^p)$ of variable length $T(\mathbf{x})$, a standard approach is to use a recurrent model which maintains a recurrent state $\mathbf{h}_t \in \mathbb{R}^q$ updated at each time step t.

Formally, for $t = 1, \ldots, T(\mathbf{x})$,

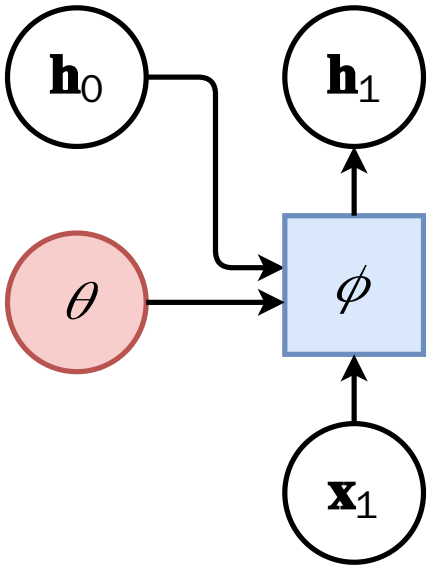
$$\mathbf{h}_t = \phi(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta),$$

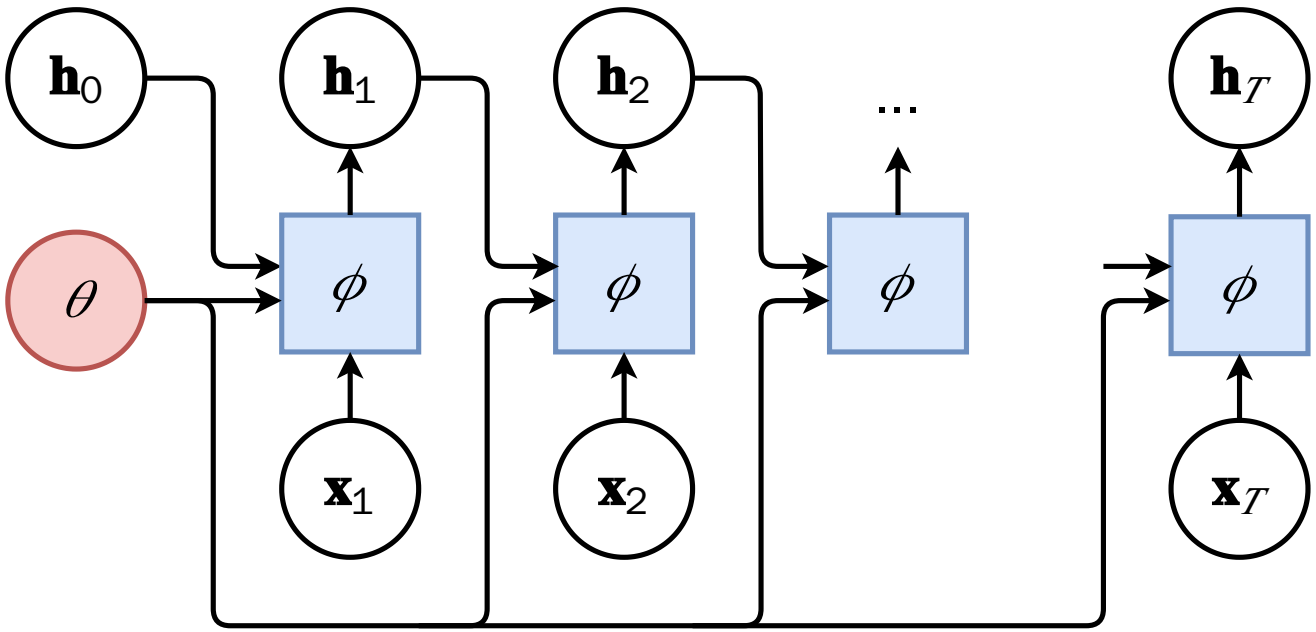
where $\phi : \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}^q$ and $\mathbf{h}_0 \in \mathbb{R}^q$.

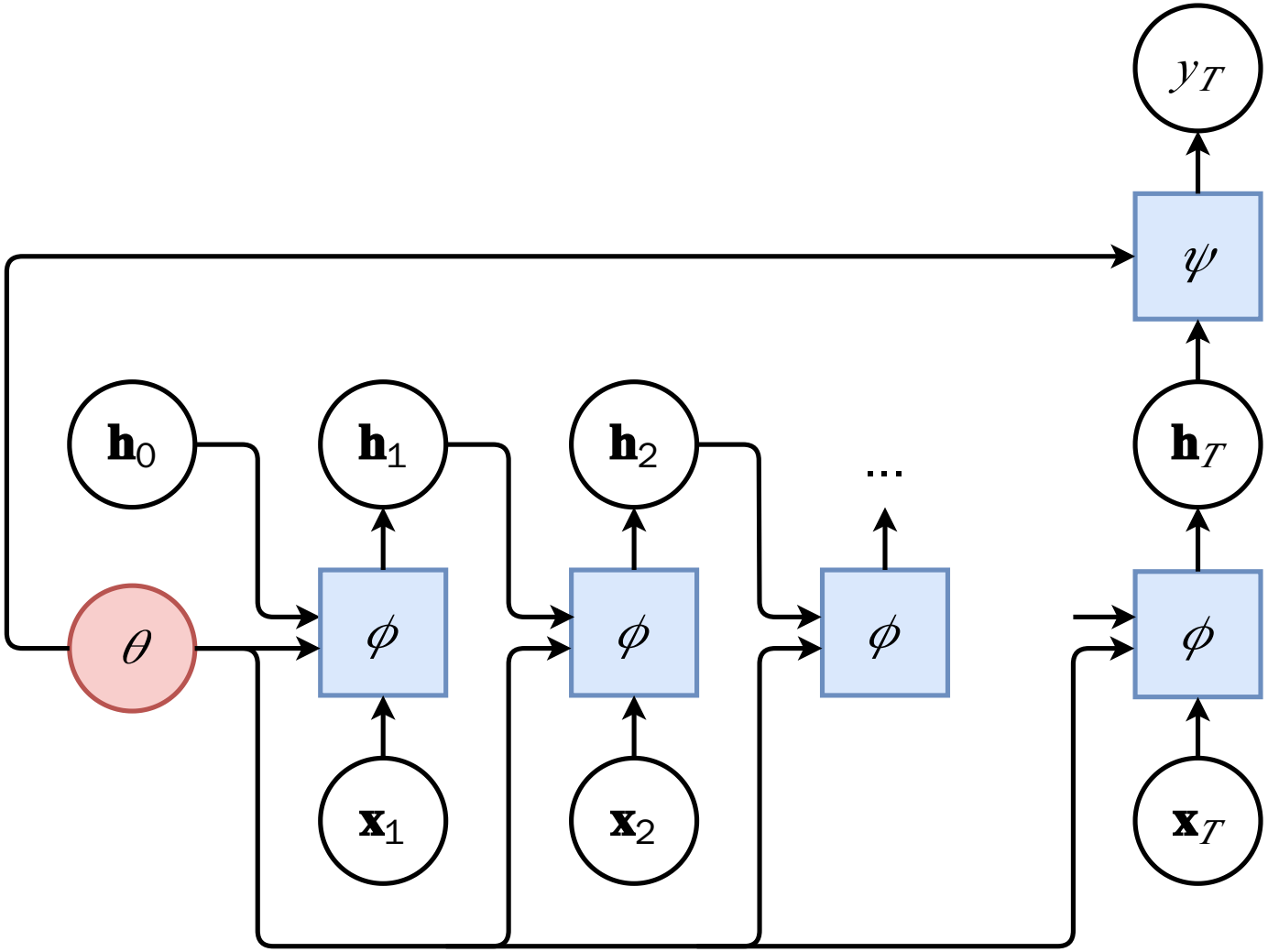
Predictions can be computed at any time step t from the recurrent state,

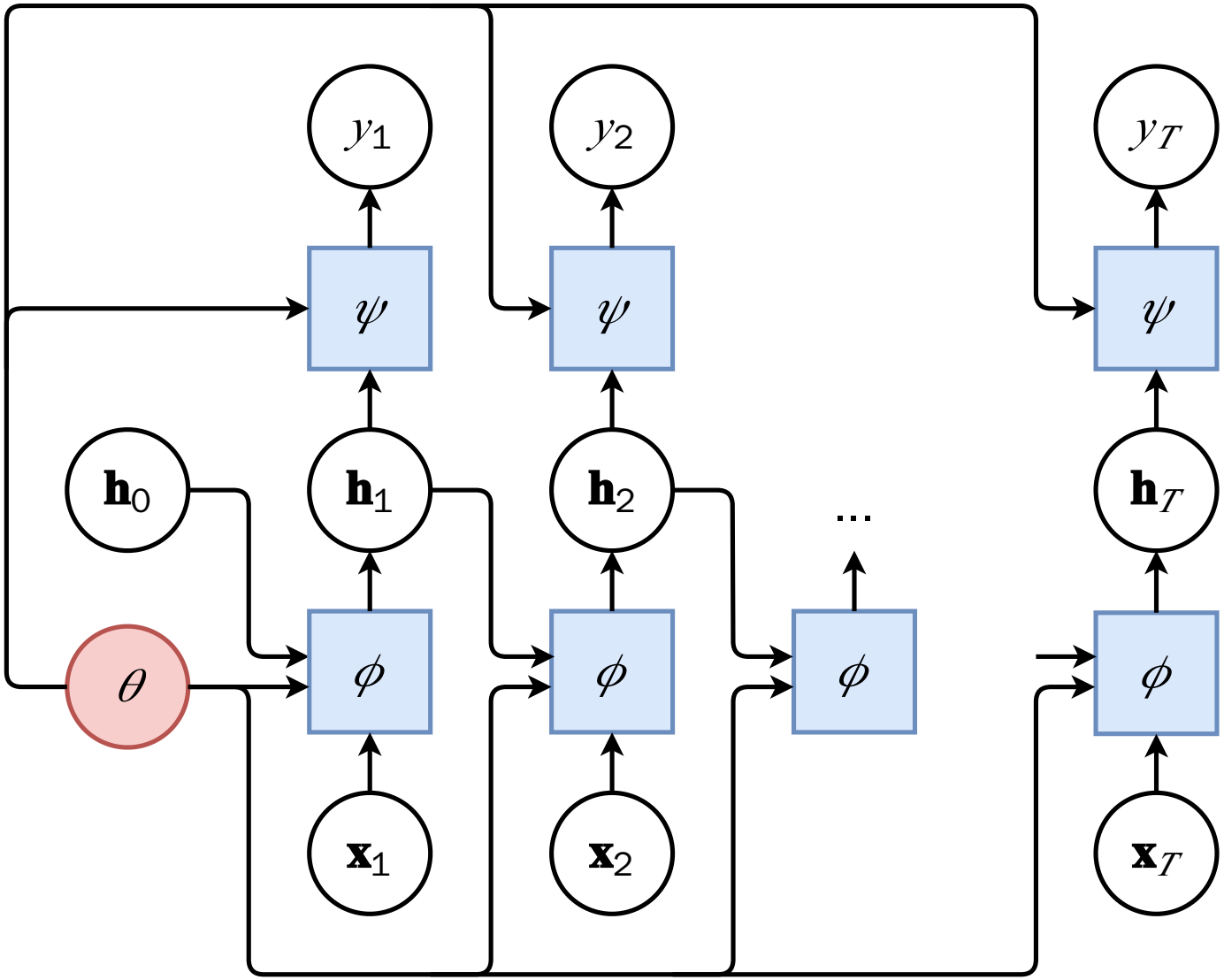
$$y_t = \psi(\mathbf{h}_t; \theta),$$

with $\psi : \mathbb{R}^q \to \mathbb{R}^C$.


$\mathbf{h}_0$
$\mathbf{h}_1$
θ
ϕ
$\mathbf{x}_1$

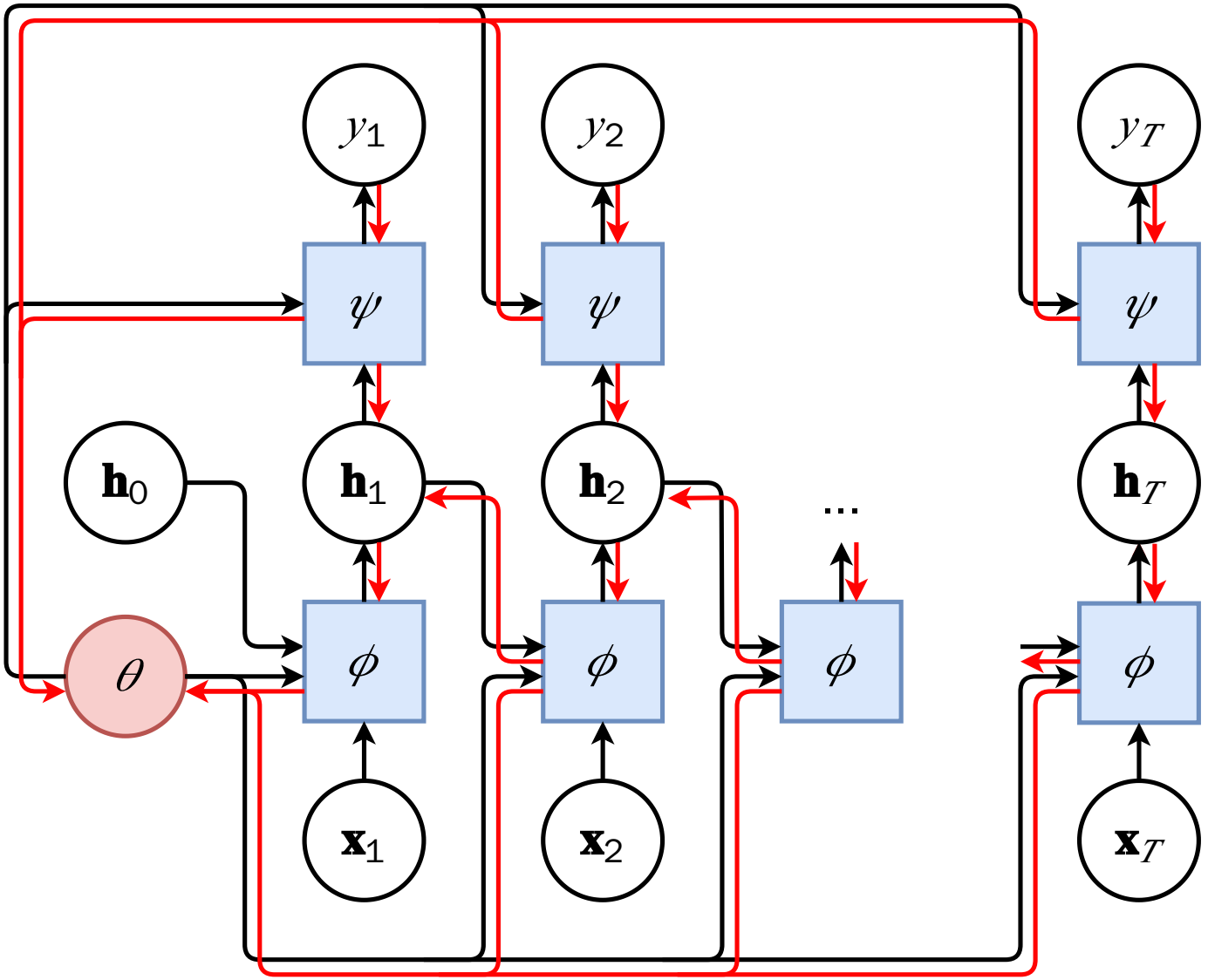

$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_2$
...
$\mathbf{h}_T$
θ
ϕ
ϕ
ϕ
ϕ
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_T$


y_T
ψ
$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_2$
...
$\mathbf{h}_T$
θ
ϕ
ϕ
ϕ
ϕ
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_T$


y_1
y_2
y_T
ψ
ψ
ψ
$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_2$
...
$\mathbf{h}_T$
θ
ϕ
ϕ
ϕ
ϕ
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_T$

Even though the number of steps T depends on $\mathbf{x}$, this is a standard computational graph, and automatic differentiation can deal with it as usual.

In the case of recurrent neural networks, this is referred to as backpropagation through time.


y_1
y_2
y_T
ψ
ψ
ψ
$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_2$
...
$\mathbf{h}_T$
θ
ϕ
ϕ
ϕ
ϕ
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_T$

Elman networks

Elman networks consist of ϕ and ψ defined as primitive neuron units, such as logistic regression units:

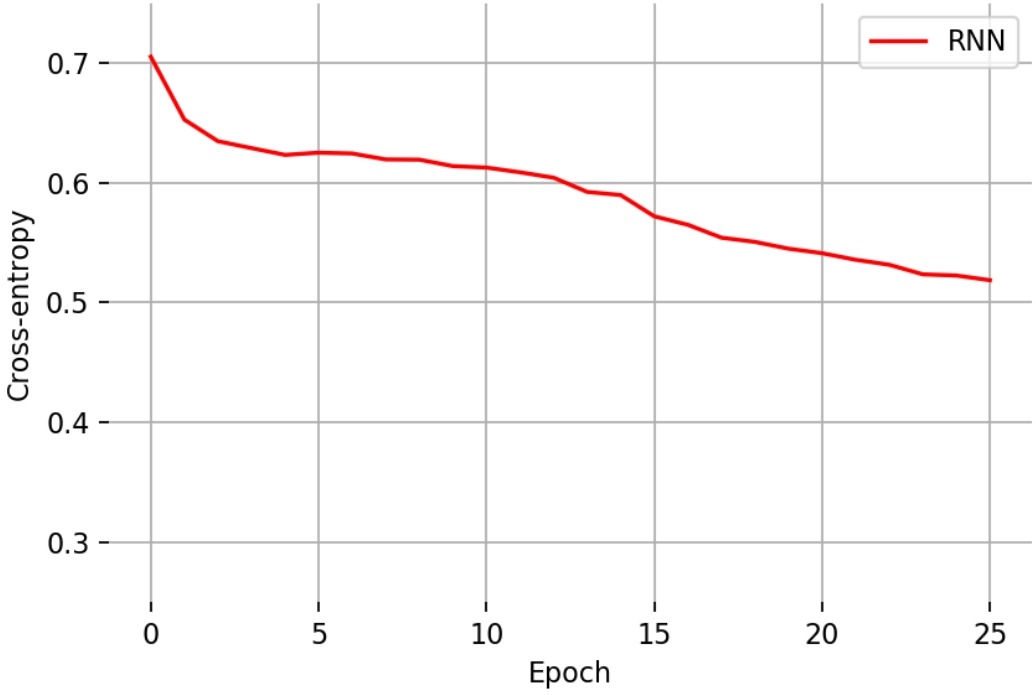
$$\mathbf{h}_t = \sigma_h \left(\mathbf{W}_{xh}^T \mathbf{x}_t + \mathbf{W}_{hh}^T \mathbf{h}_{t-1} + \mathbf{b}_h \right)$$
$$y_t = \sigma_y \left(\mathbf{W}_y^T \mathbf{h}_t + b_y \right)$$
$$\mathbf{W}_{xh}^T \in \mathbb{R}^{p \times q}, \mathbf{W}_{hh}^T \in \mathbb{R}^{q \times q}, \mathbf{b}_h \in \mathbb{R}^q, b_y \in \mathbb{R}, \mathbf{h}_0 = 0$$

where σ_h and σ_y are non-linear activation functions, such as the sigmoid function, $\tanh$ or ReLU.

Benchmark example

Learn to recognize variable-length sequences that are palindromes. For training, we will use sequences of random sizes, from 1 to 10.

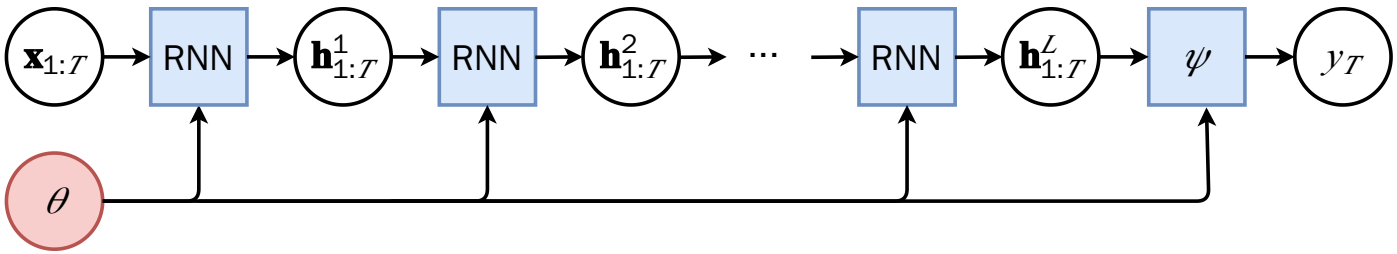
$\mathbf{x}$	y
$(1, 2, 3, 2, 1)$	1
$(2, 1, 2)$	1
$(3, 4, 1, 2)$	0
(0)	1
$(1, 4)$	0

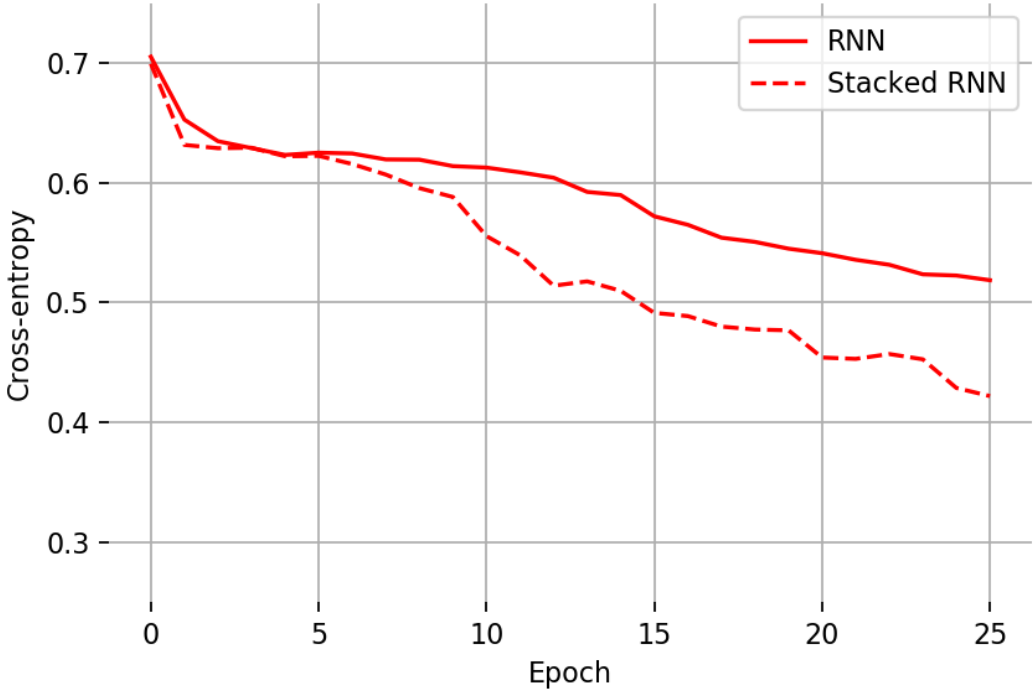

RNN
0.7
0.6
0.5
0.4
0.3
Cross-entropy
0
5
10
15
20
25
Epoch

Stacked RNNs

Recurrent networks can be viewed as layers producing sequences $\mathbf{h}^l_{1:T}$ of activations.

As for dense layers, recurrent layers can be composed in series to form a **stack** of recurrent networks.

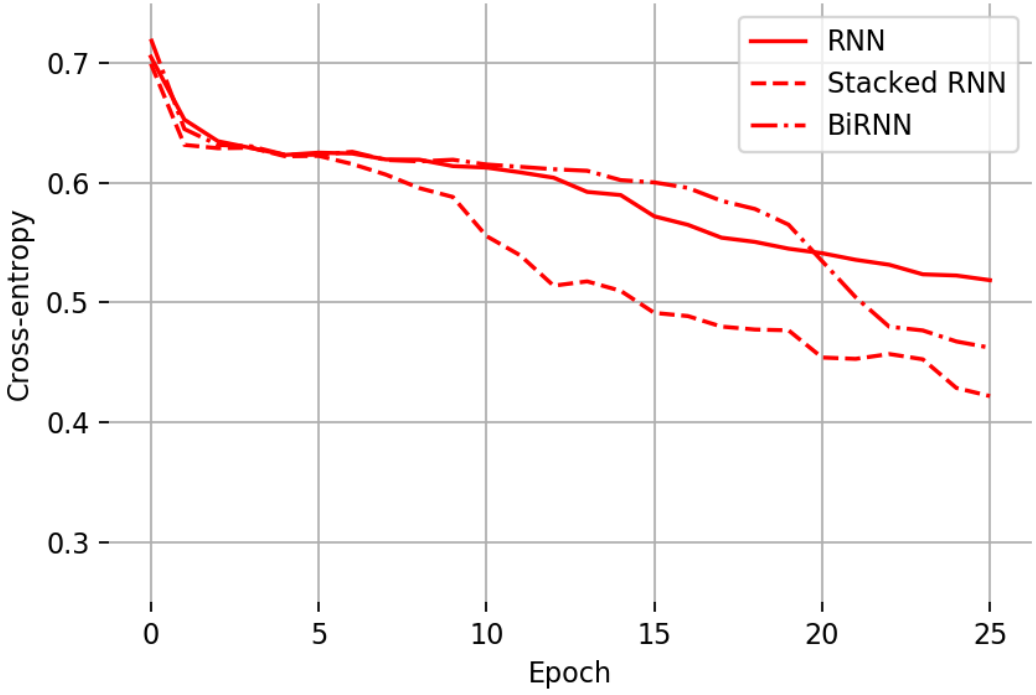



RNN
Stacked RNN
0.7
0.6
0.5
0.4
0.3
Cross-entropy
0
5
10
15
20
25
Epoch

Bidirectional RNNs

Computing the recurrent states forward in time does not make use of future input values $\mathbf{x}_{t+1:T}$, even though there are known.

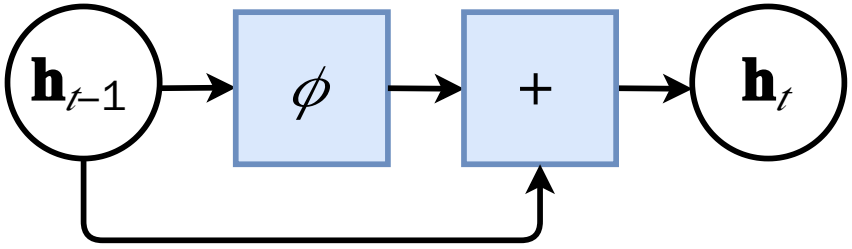
- RNNs can be made bidirectional by consuming the sequence in both directions.
- Effectively, this amounts to run the same (single direction) RNN twice:
  - once over the original sequence $\mathbf{x}_{1:T}$,
  - once over the reversed sequence $\mathbf{x}_{T:1}$.
- The resulting recurrent states of the bidirectional RNN is the concatenation of two resulting sequences of recurrent states.


RNN
Stacked RNN
BiRNN
0.7
0.6
0.5
0.4
0.3
Cross-entropy
0
5
10
15
20
25
Epoch

Gating

When unfolded through time, the graph of computation of a recurrent network can grow very deep, and training involves dealing with vanishing gradients.

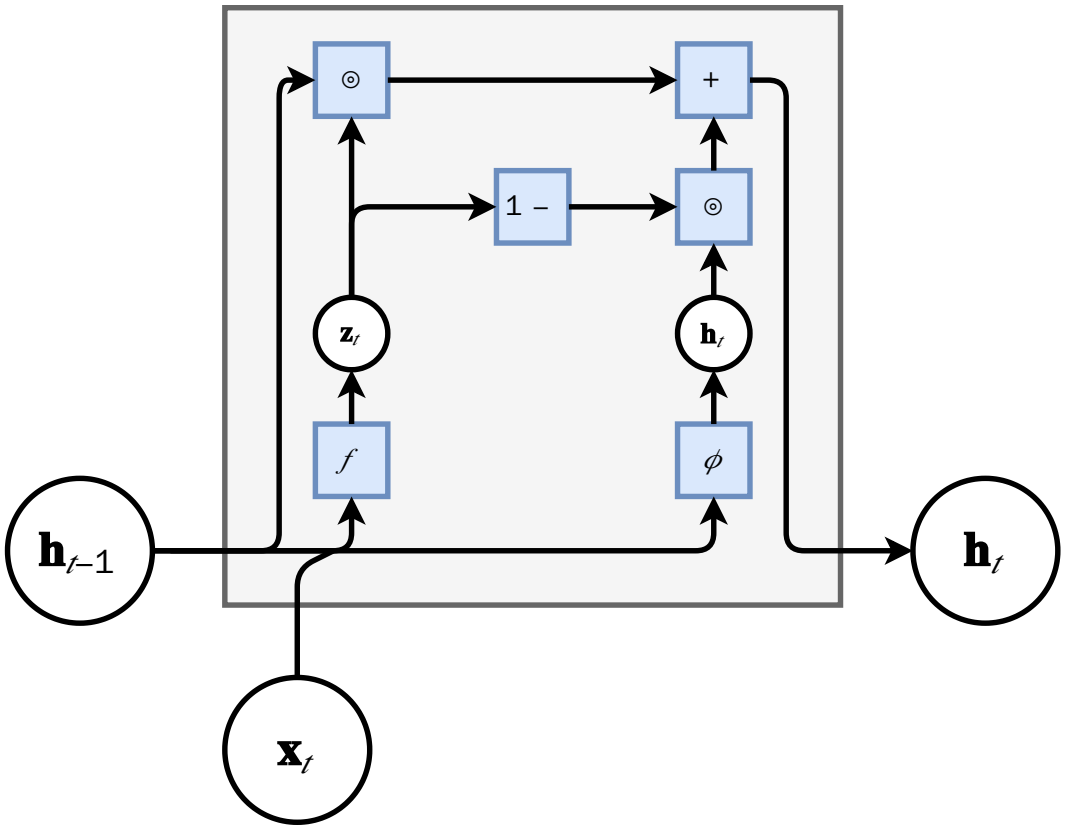
- RNN cells should include a pass-through, or additive paths, so that the recurrent state does not go repeatedly through a squashing non-linearity.
- This is identical to skip connections in ResNet.



For instance, the recurrent state update can be a per-component weighted average of its previous value $\mathbf{h}_{t-1}$ and a full update $\bar{\mathbf{h}}_t$, with the weighting $\mathbf{z}_t$ depending on the input and the recurrent state, hence acting as a forget gate.

Formally,

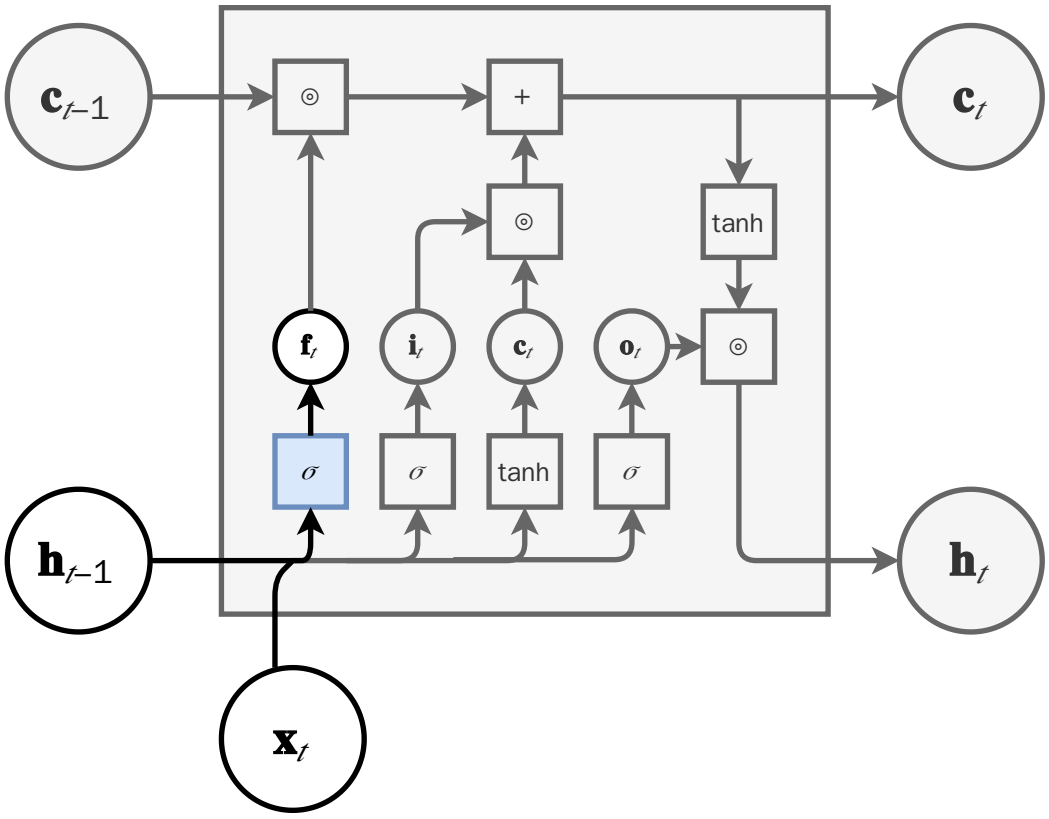
$$\begin{aligned}\bar{\mathbf{h}}_t &= \phi(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)\\ \mathbf{z}_t &= f(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)\\ \mathbf{h}_t &= \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \bar{\mathbf{h}}_t.\end{aligned}$$


⊙
+
1 −
⊙
$\mathbf{z}_t$
$\mathbf{h}_t$
f
ϕ
$\mathbf{h}_{t-1}$
$\mathbf{h}_t$
$\mathbf{x}_t$

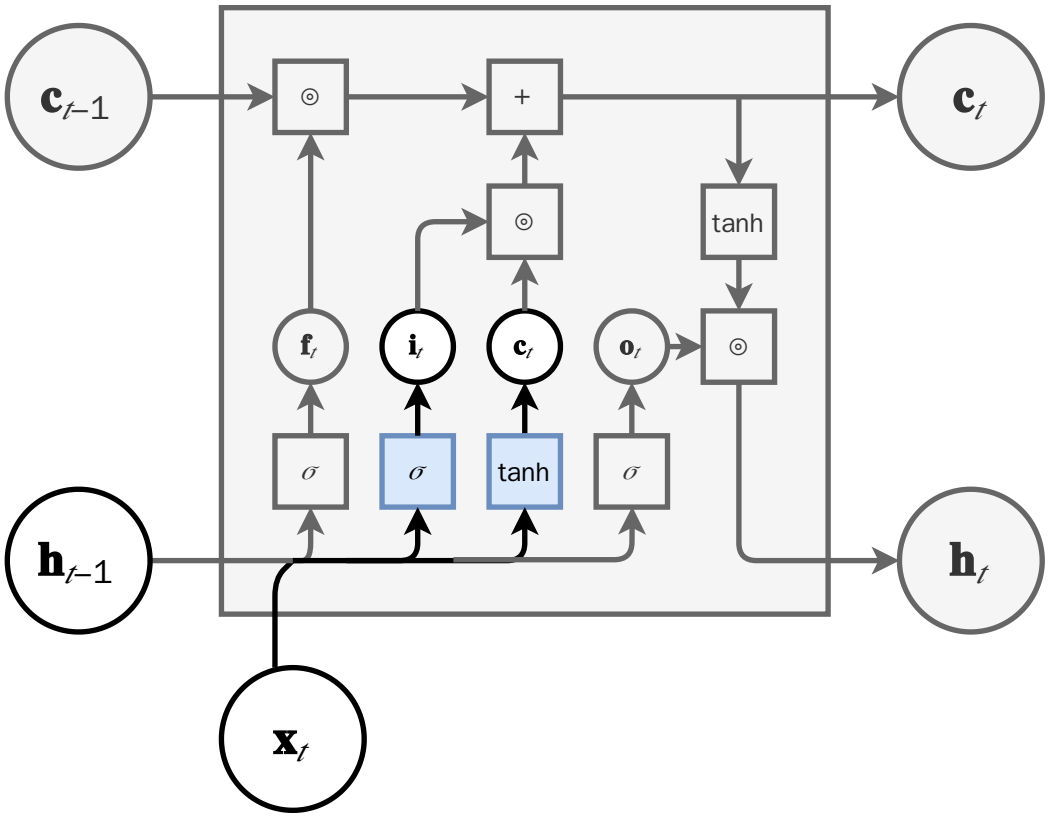
LSTM

The long short-term memory model (LSTM; Hochreiter and Schmidhuber, 1997) is an instance of the previous gated recurrent cell, with the following changes:

- The recurrent state is split into two parts $\mathbf{c}_t$ and $\mathbf{h}_t$, where
  - $\mathbf{c}_t$ is the cell state and
  - $\mathbf{h}_t$ is output state.
- A forget gate $\mathbf{f}_t$ selects the cell state information to erase.
- An input gate $\mathbf{i}_t$ selects the cell state information to update.
- An output gate $\mathbf{o}_t$ selects the cell state information to output.

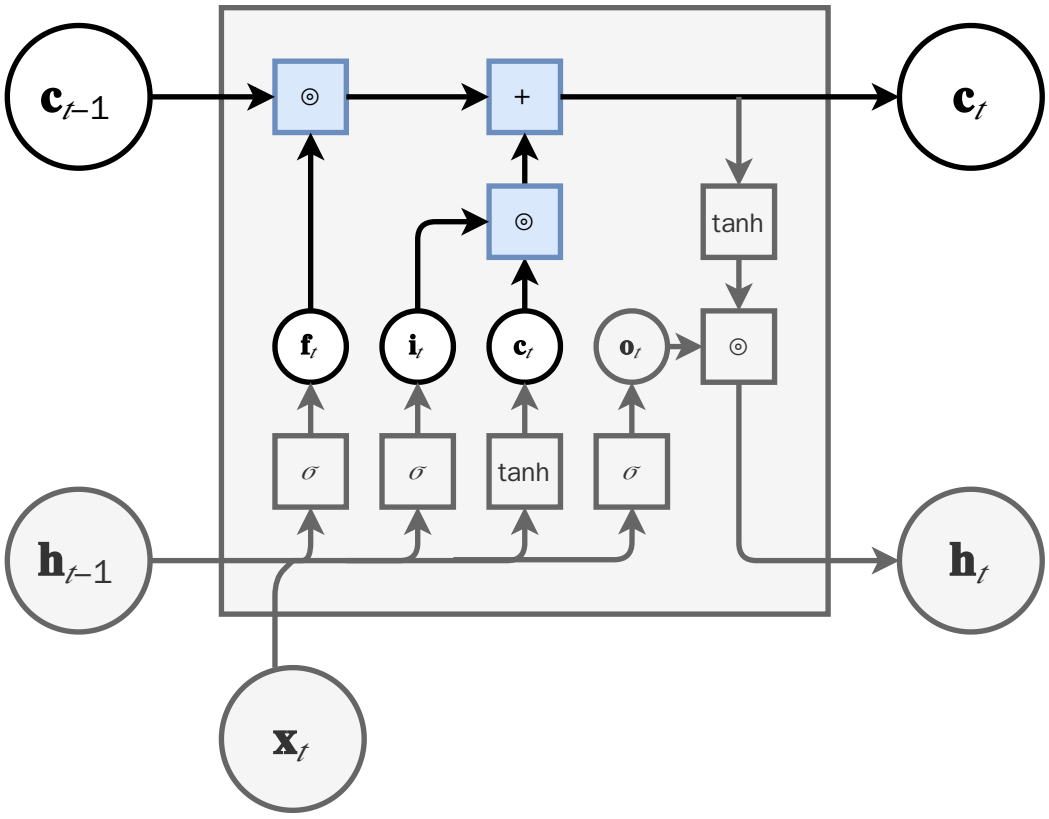


$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f^T[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f\right)$$

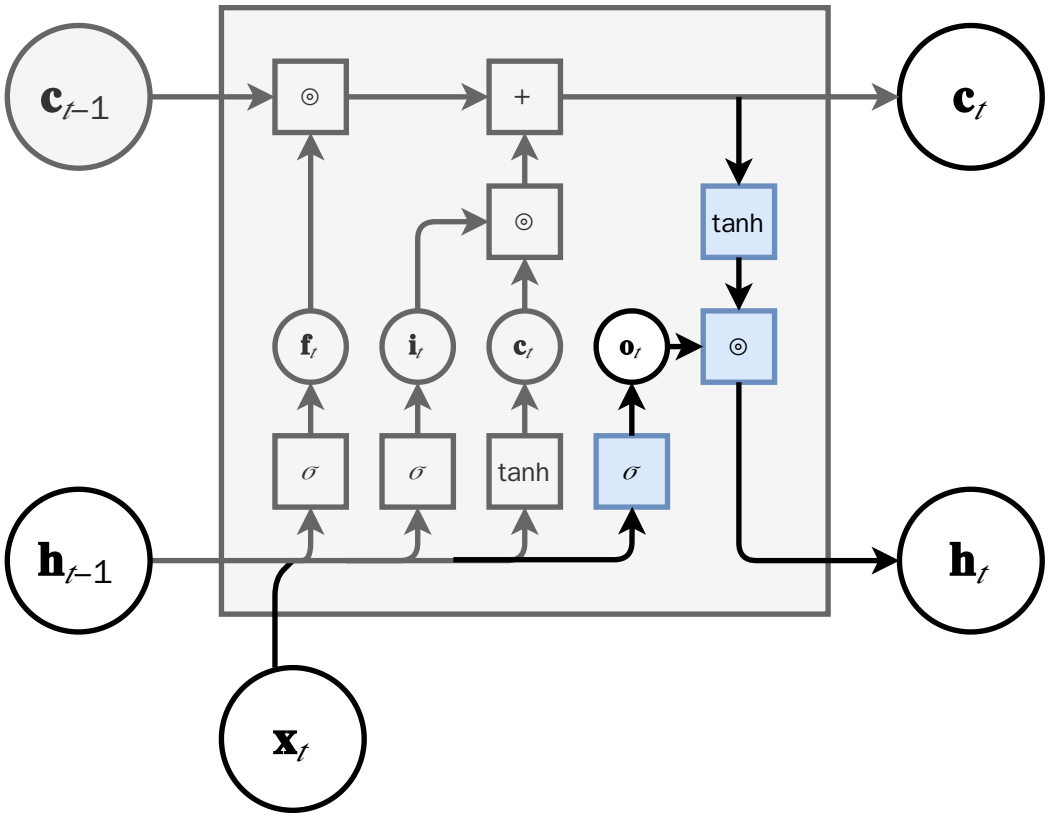


$$\mathbf{i}_t = \sigma\left(\mathbf{W}_i^T[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i\right)$$

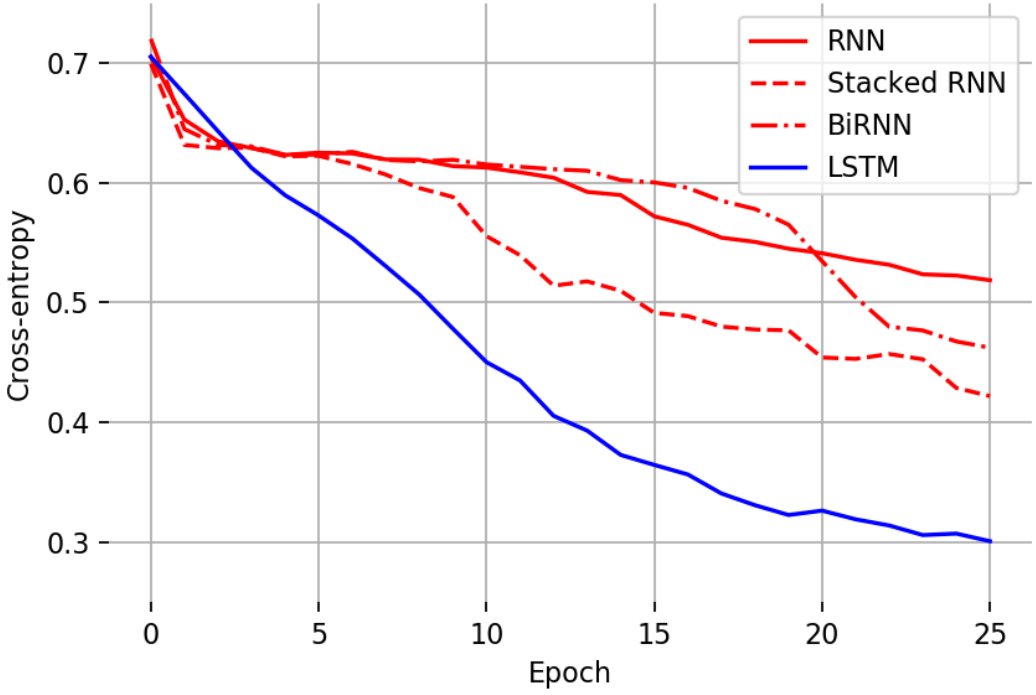
$$\bar{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c^T[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c\right)$$



$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \bar{\mathbf{c}}_t$$



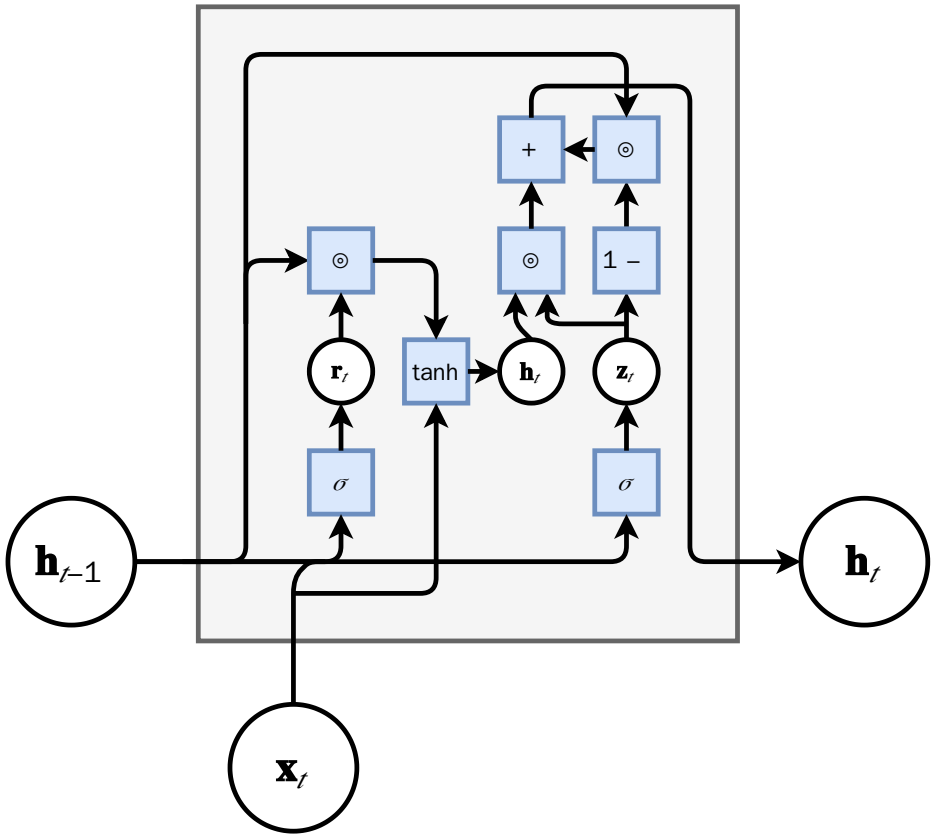
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o^T[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o\right)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$


RNN
Stacked RNN
BiRNN
LSTM
Cross-entropy
0.7
0.6
0.5
0.4
0.3
0
5
10
15
20
25
Epoch

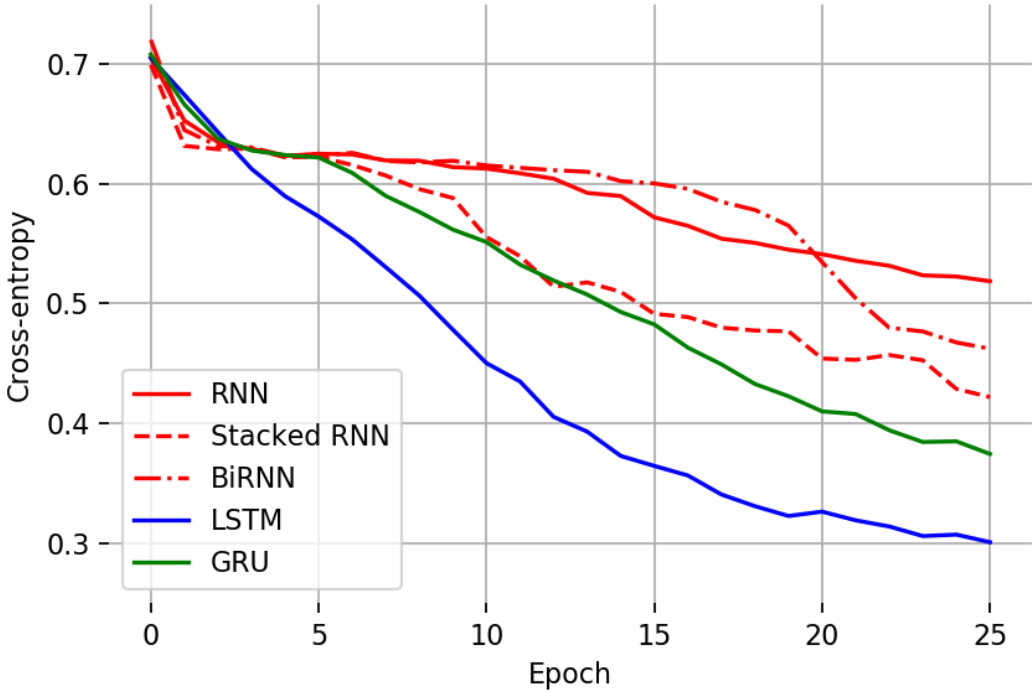
GRU

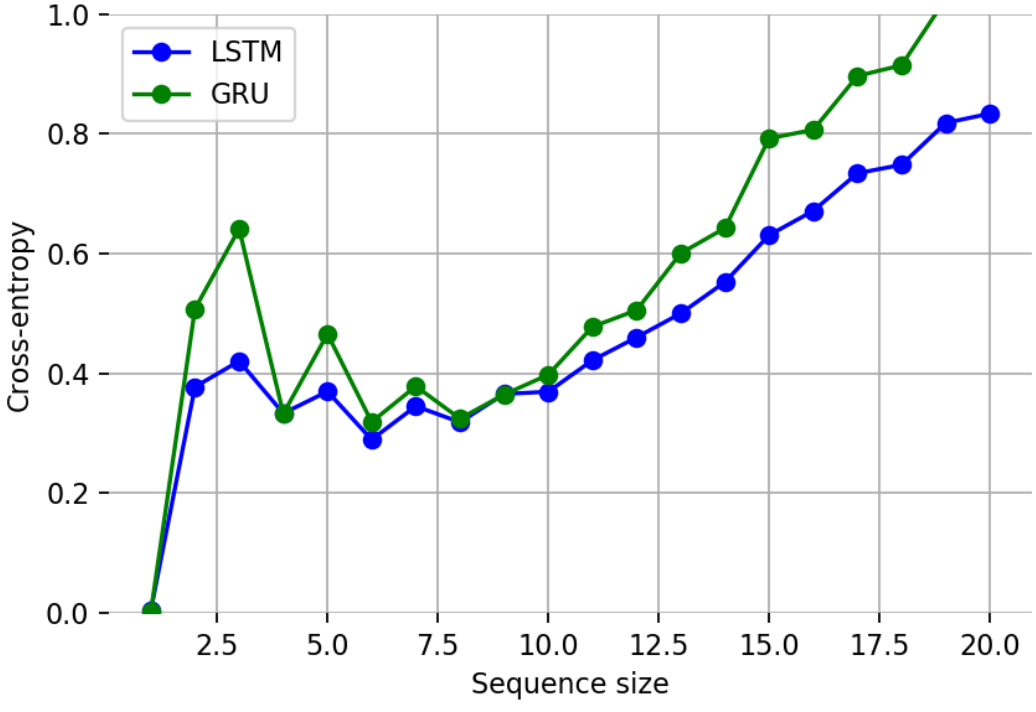
The gated recurrent unit (GRU; Cho et al, 2014) is another gated recurrent cell.

- It uses two gates instead of three: an update gate $\mathbf{z}_t$ and a reset gate $\mathbf{r}_t$.
- GRUs perform similarly as LSTMs for language or speech modeling sequences, but with fewer parameters.
- However, LSTMs remain strictly stronger than GRUs.



$$\mathbf{z}_t = \sigma\left(\mathbf{W}_z^T[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z\right)$$
$$\mathbf{r}_t = \sigma\left(\mathbf{W}_r^T[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r\right)$$
$$\bar{\mathbf{h}}_t = \tanh\left(\mathbf{W}_h^T[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h\right)$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{h-1} + \mathbf{z}_t \odot \bar{\mathbf{h}}_t$$

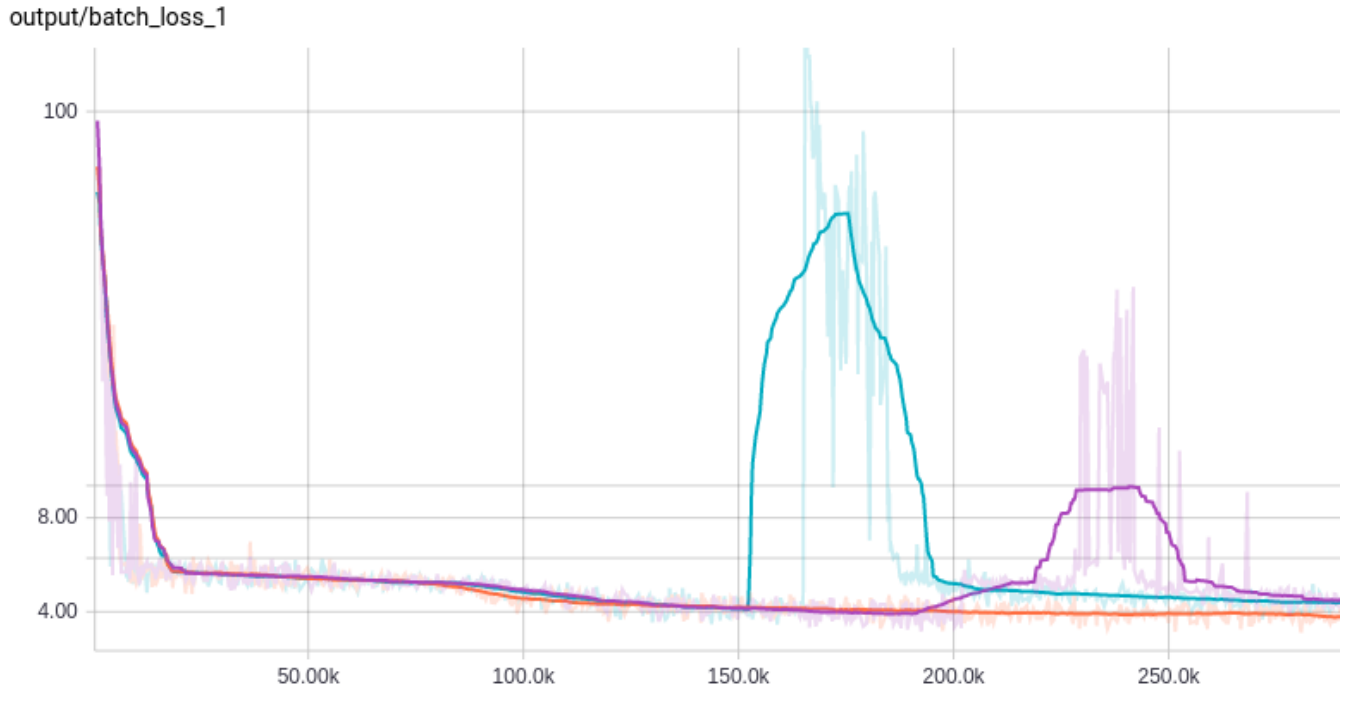

0.7
0.6
0.5
0.4
0.3
Cross-entropy
0
5
10
15
20
25
Epoch
RNN
Stacked RNN
BiRNN
LSTM
GRU



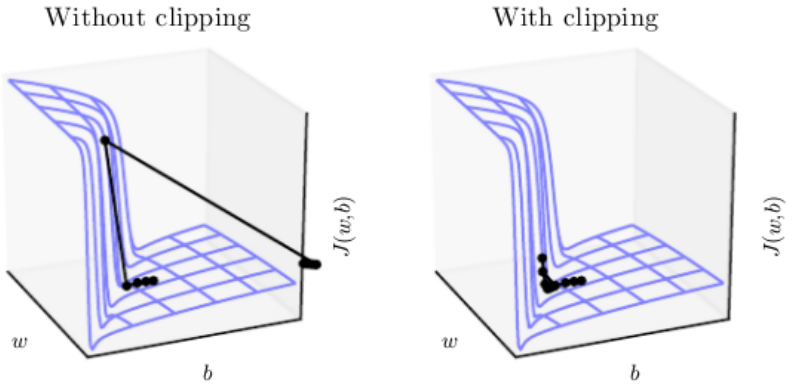
The models do not generalize to sequences longer than those in the training set!

Exploding gradients

Gated units prevent gradients from vanishing, but not from exploding.



Credits: pat-coady.



The standard strategy to solve this issue is gradient norm clipping, which rescales the norm of the gradient to a fixed threshold δ when it is above:

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

Orthogonal initialization

Let us consider a simplified RNN, with no inputs, no bias, an identity activation function σ (as in the positive part of a ReLU) and the initial recurrent state $\mathbf{h}_0$ set to the identity matrix.

We have,

$$\begin{aligned}\mathbf{h}_t &= \sigma\left(\mathbf{W}_{xh}^T\mathbf{x}_t + \mathbf{W}_{hh}^T\mathbf{h}_{t-1} + \mathbf{b}_h\right)\\ &= \mathbf{W}_{hh}^T\mathbf{h}_{t-1}\\ &= \mathbf{W}^T\mathbf{h}_{t-1}.\end{aligned}$$

For a sequence of size n, it comes

$$\mathbf{h}_n = \mathbf{W}(\mathbf{W}(\mathbf{W}(\ldots(\mathbf{W}\mathbf{h}_0)\ldots))) = \mathbf{W}^n\mathbf{h}_0 = \mathbf{W}^n I = \mathbf{W}^n.$$

Ideally, we would like $\mathbf{W}^n$ to neither vanish nor explode as n increases.

Fibonacci digression

The Fibonacci sequence is

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

It grows fast! But how fast?

In matrix form, the Fibonacci sequence is equivalently expressed as

$$\begin{pmatrix} f_{k+2} \\ f_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{k+1} \\ f_k \end{pmatrix}.$$

With $\mathbf{f}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, we have

$$\mathbf{f}_{k+1} = \mathbf{A}\mathbf{f}_k = \mathbf{A}^{k+1}\mathbf{f}_0.$$

The matrix $\mathbf{A}$ can be diagonalized as

$$\mathbf{A} = \mathbf{S}\Lambda\mathbf{S}^{-1},$$

where

$$\Lambda = \begin{pmatrix} \varphi & 0 \\ 0 & -\varphi^{-1} \end{pmatrix}$$
$$\mathbf{S} = \begin{pmatrix} \varphi & -\varphi^{-1} \\ 1 & 1 \end{pmatrix}.$$

In particular,

$$\mathbf{A}^n = \mathbf{S}\Lambda^n\mathbf{S}^{-1}.$$

Therefore, the Fibonacci sequence grows exponentially fast with the golden ratio φ.

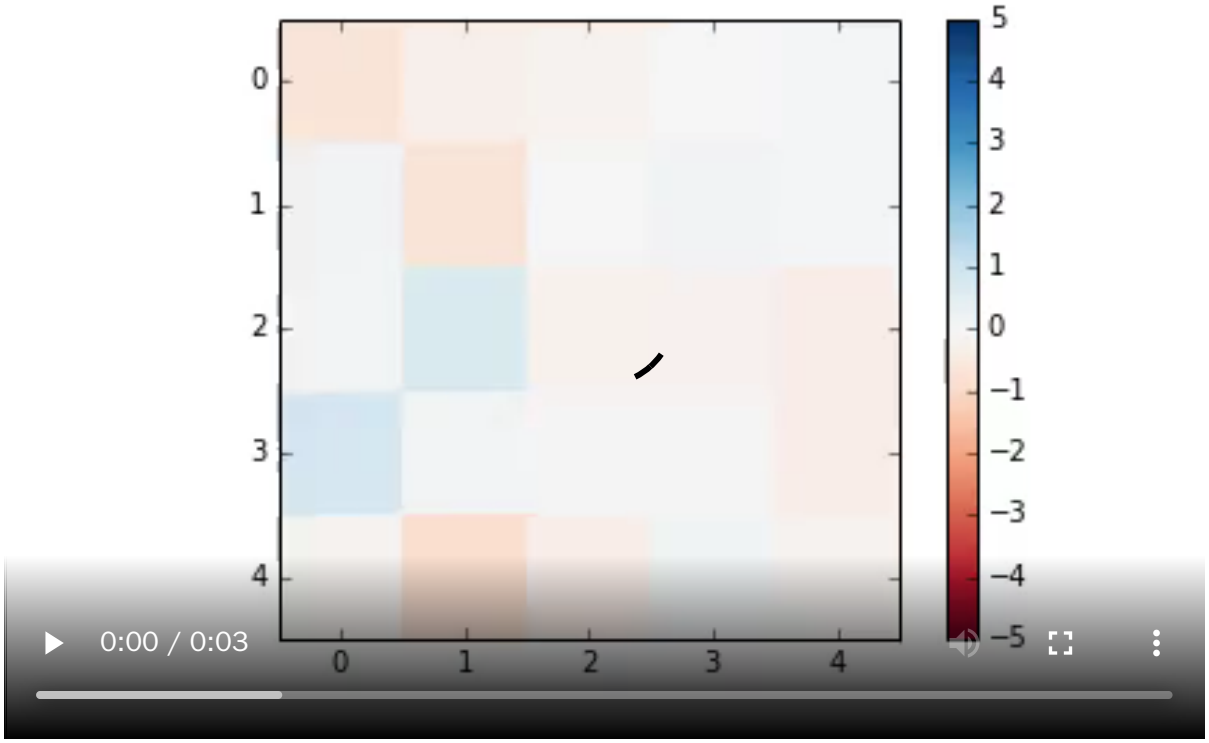
Theorem

Let $\rho(\mathbf{A})$ be the spectral radius of the matrix $\mathbf{A}$, defined as

$$\rho(\mathbf{A}) = \max\{|\lambda_1|, ..., |\lambda_d|\}.$$

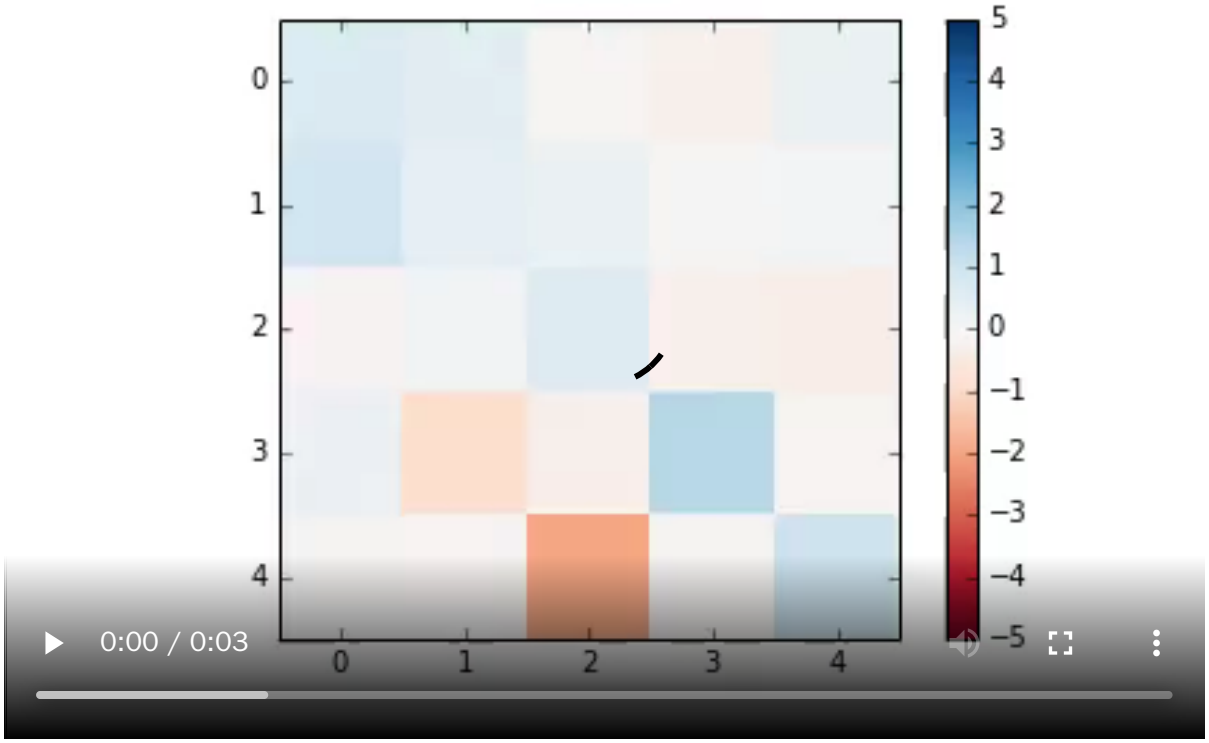
We have:

- if $\rho(\mathbf{A}) < 1$ then $\lim_{n\to\infty} ||\mathbf{A}^n|| = \mathbf{0}$ (= vanishing activations),
- if $\rho(\mathbf{A}) > 1$ then $\lim_{n\to\infty} ||\mathbf{A}^n|| = \infty$ (= exploding activations).



$\rho(\mathbf{A}) < 1, \mathbf{A}^n$ vanish.

Credits: Stephen Merety, Explaining and illustrating orthogonal initialization for recurrent neural networks, 2016.



$\rho(\mathbf{A}) > 1, \mathbf{A}^n$ explode.

Credits: Stephen Merety, Explaining and illustrating orthogonal initialization for recurrent neural networks, 2016.

Orthogonal initialization

If $\mathbf{A}$ is orthogonal, then it is diagonalizable and all its eigenvalues are equal to -1 or 1. In this case, the norm of

$$\mathbf{A}^n = \mathbf{S}\mathbf{\Lambda}^n\mathbf{S}^{-1}$$

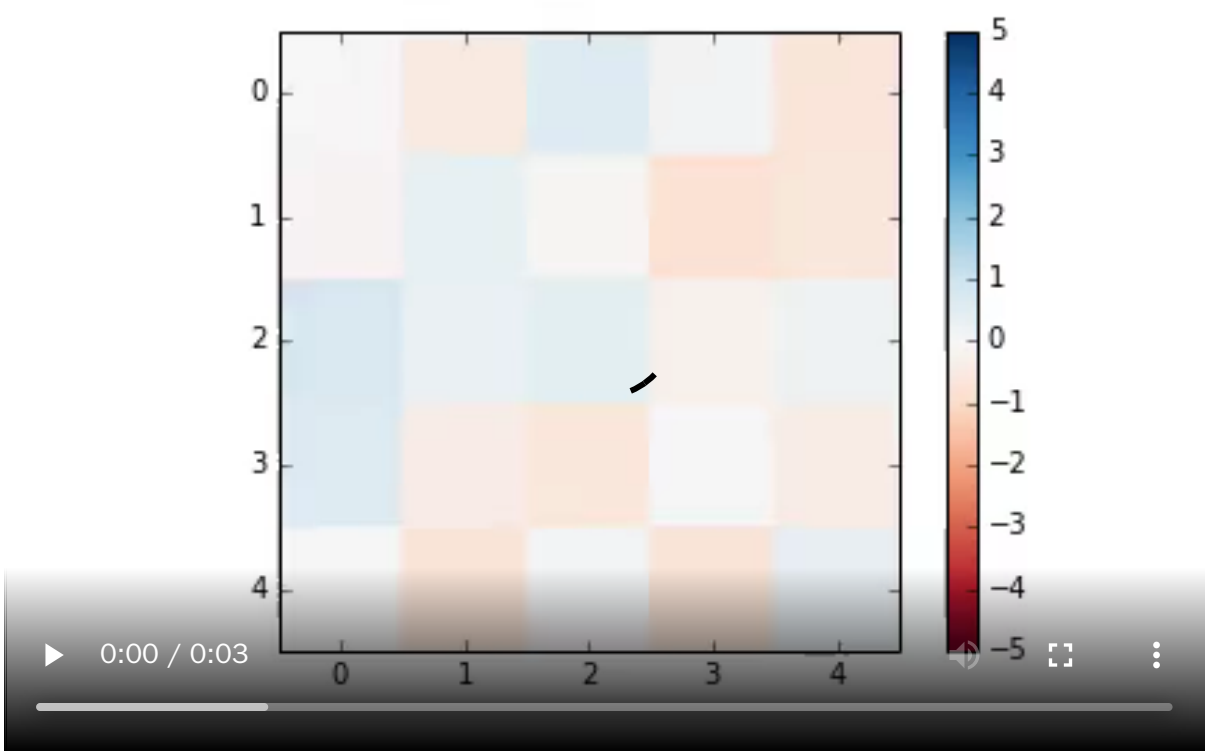
remains bounded.

- Therefore, initializing $\mathbf{W}$ as a random orthogonal matrix will guarantee that activations will neither vanish nor explode.
- In practice, a random orthogonal matrix can be found through the SVD decomposition or the QR factorization of a random matrix.
- This initialization strategy is known as orthogonal initialization.

In Tensorflow's `Orthogonal` initializer:

```
# Generate a random matrix
a = random_ops.random_normal(flat_shape, dtype=dtype, seed=self.seed)
# Compute the qr factorization
q, r = gen_linalg_ops.qr(a, full_matrices=False)
# Make Q uniform
d = array_ops.diag_part(r)
q *= math_ops.sign(d)
if num_rows < num_cols:
  q = array_ops.matrix_transpose(q)
return self.gain * array_ops.reshape(q, shape)
```

Credits: Tensorflow, tensorflow/python/ops/init_ops.py.



$\mathbf{A}$ is orthogonal.

Credits: Stephen Merety, Explaining and illustrating orthogonal initialization for recurrent neural networks, 2016.

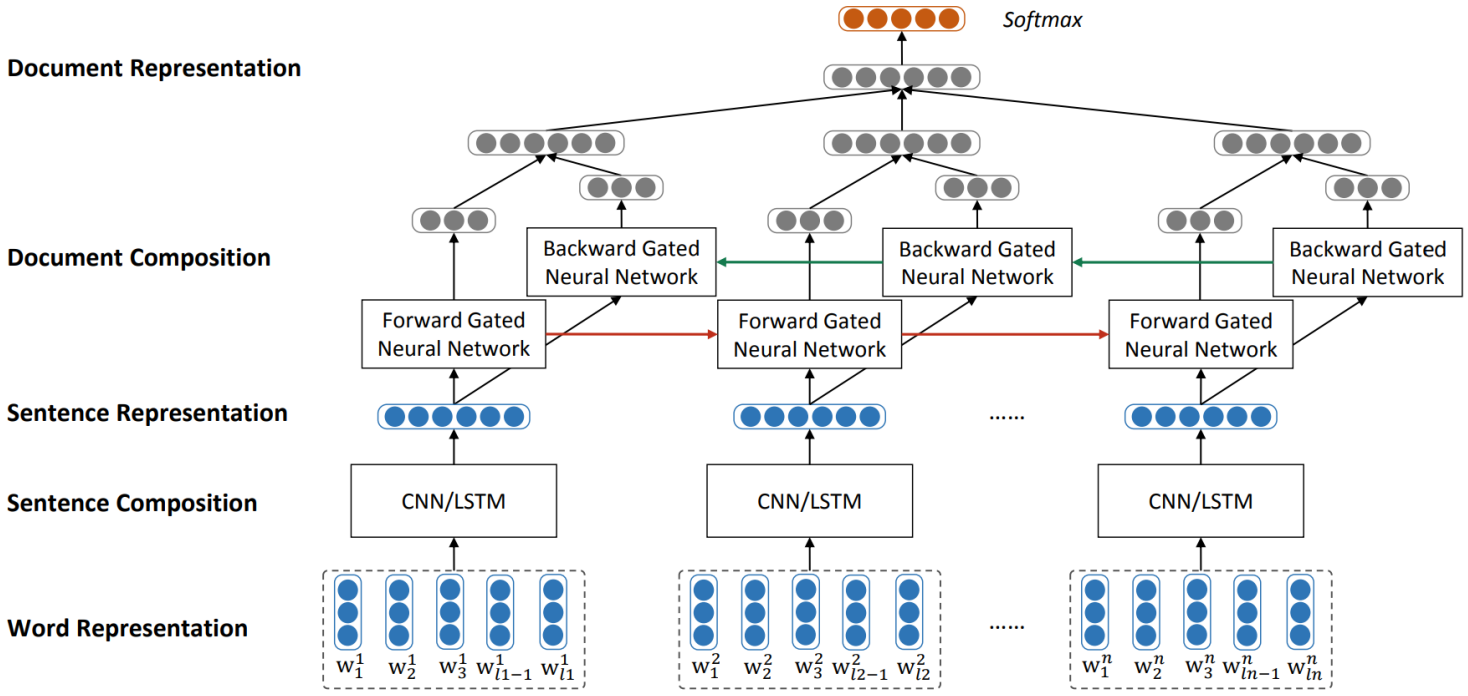
Exploding activations are also the reason why squashing non-linearity functions (such as $\tanh$) are preferred in RNNs.

- They avoid recurrent states from exploding by upper bounding $||\mathbf{h}_t||$.
- (At least when running the network forward.)

Applications

(some)

Sentiment analysis



Document-level modeling for sentiment analysis (= text classification), with stacked, bidirectional and gated recurrent networks.

Credits: Duyu Tang et al, Document Modeling with Gated Recurrent Neural Network for Sentiment Classification, 2015.

Language models

Model language as a Markov chain, such that sentences are sequences of words $\mathbf{w}_{1:T}$ drawn repeatedly from

$$p\left(\mathbf{w}_t \middle| \mathbf{w}_{1:t-1}\right).$$

This is an instance of sequence synthesis, for which predictions are computed at all time steps t.

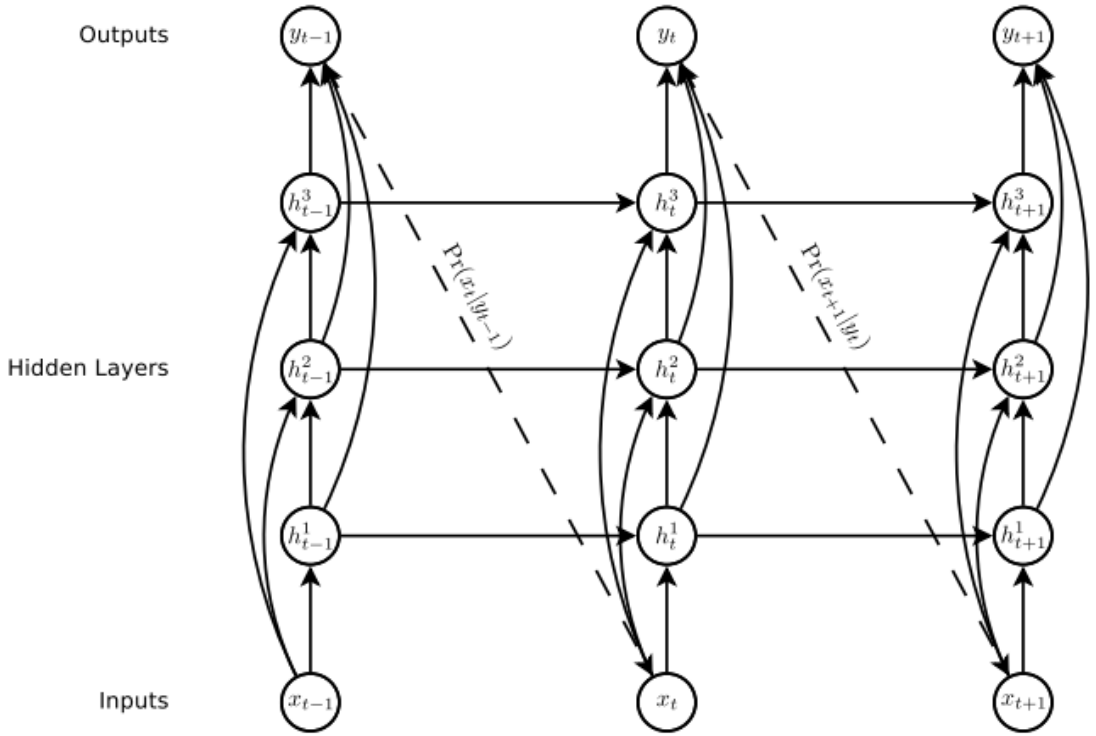


Figure 1: **Deep recurrent neural network prediction architecture.** The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

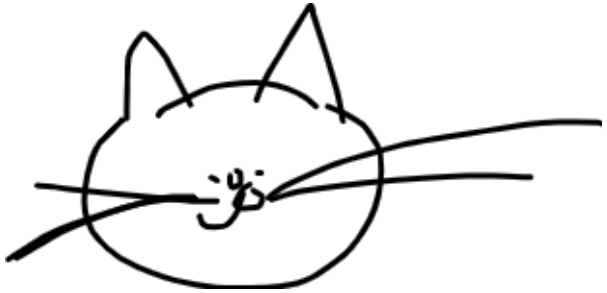
Credits: Alex Graves, Generating Sequences With Recurrent Neural Networks, 2013.

```
maxs-mbp:tweet-generator maxwoolf$ python3
Python 3.6.4 (default, Jan  6 2018, 11:51:59)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

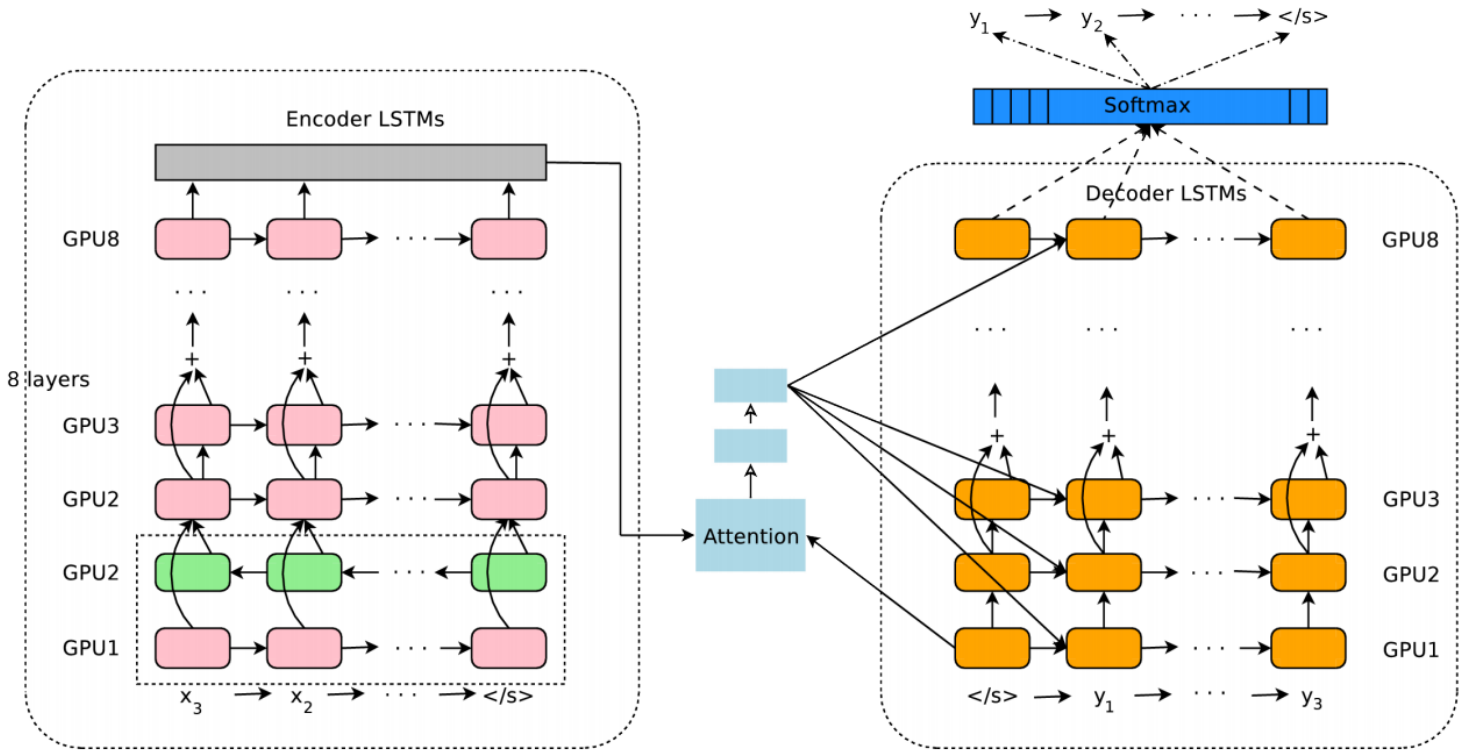
Sequence synthesis

The same generative architecture applies to any kind of sequences.

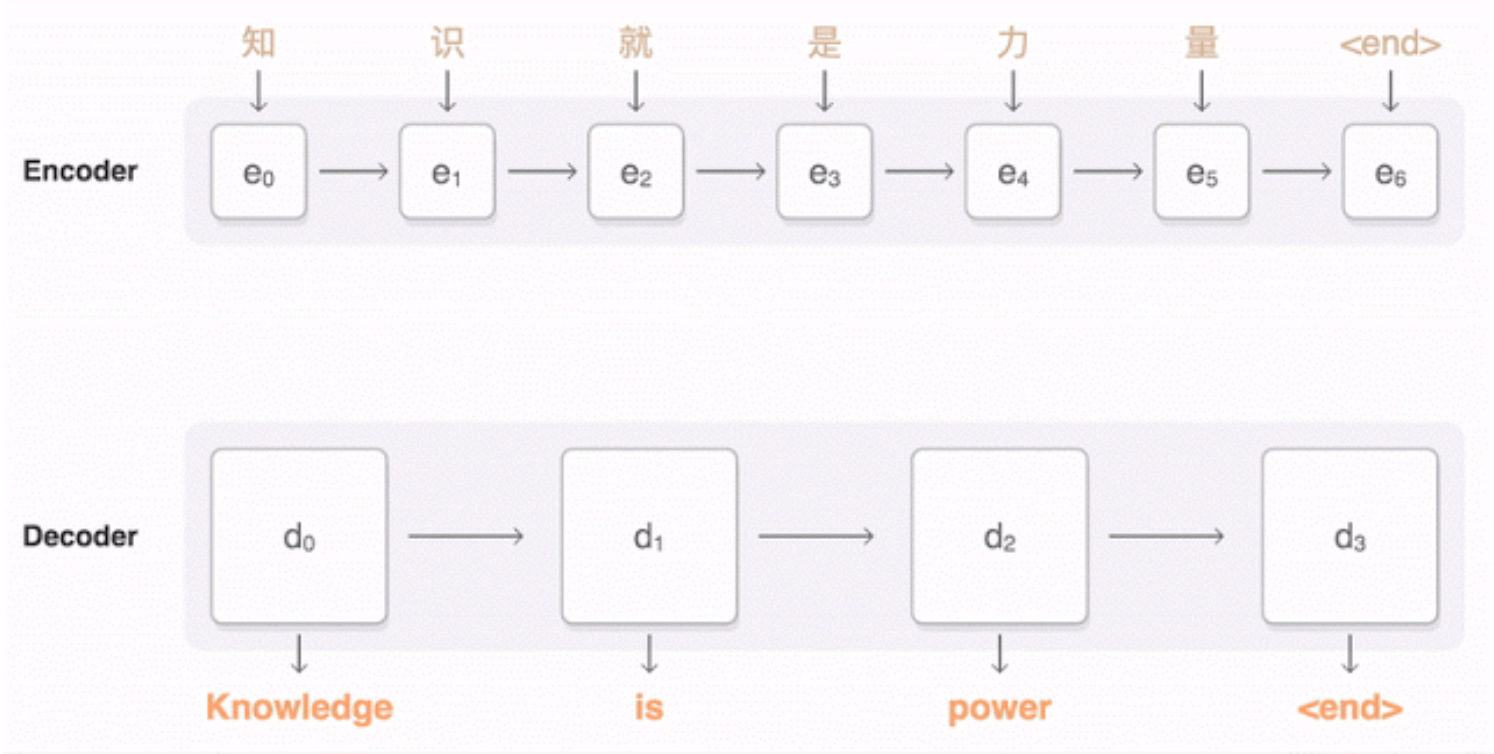
E.g., `sketch-rnn-demo` sketches defined as sequences of strokes.



Neural machine translation



Credits: Yonghui Wu et al, Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016.



Credits: Yonghui Wu et al, Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, 2016.

Text-to-speech synthesis

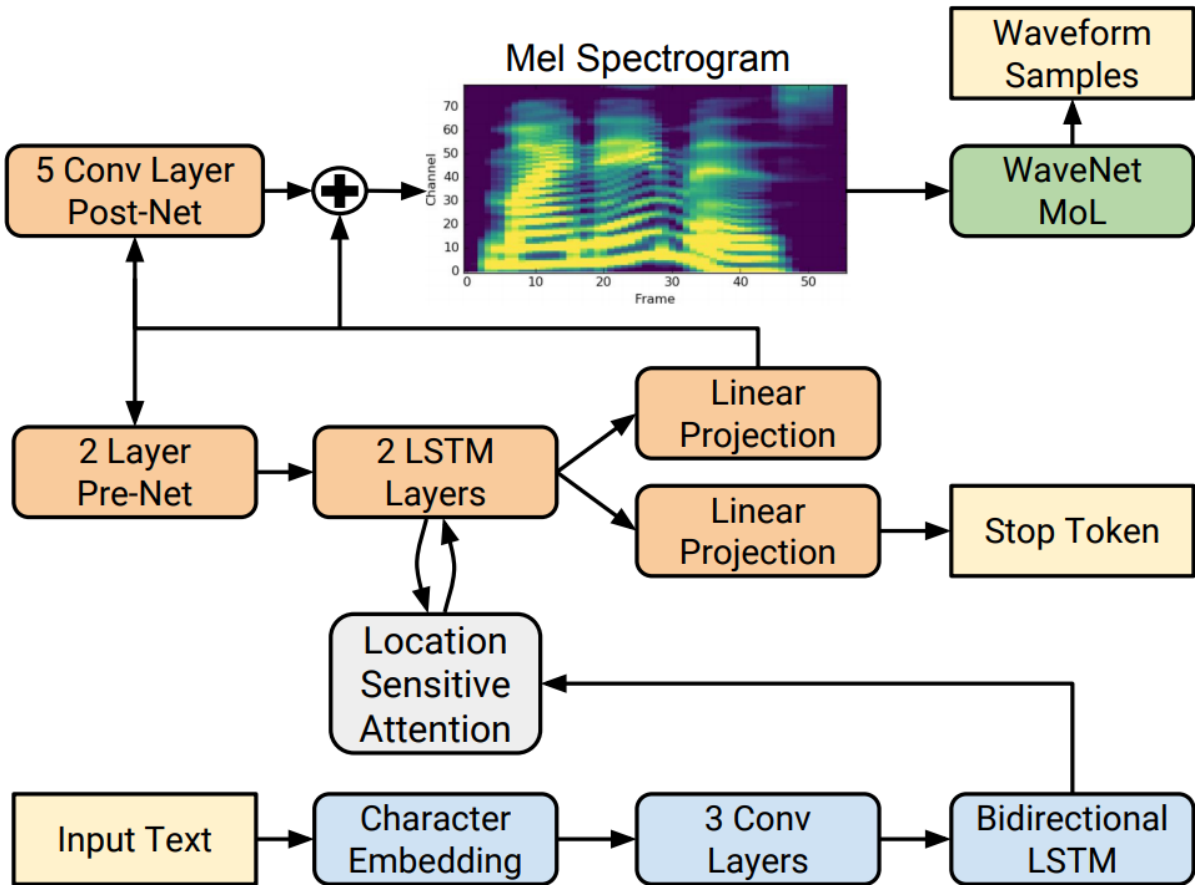
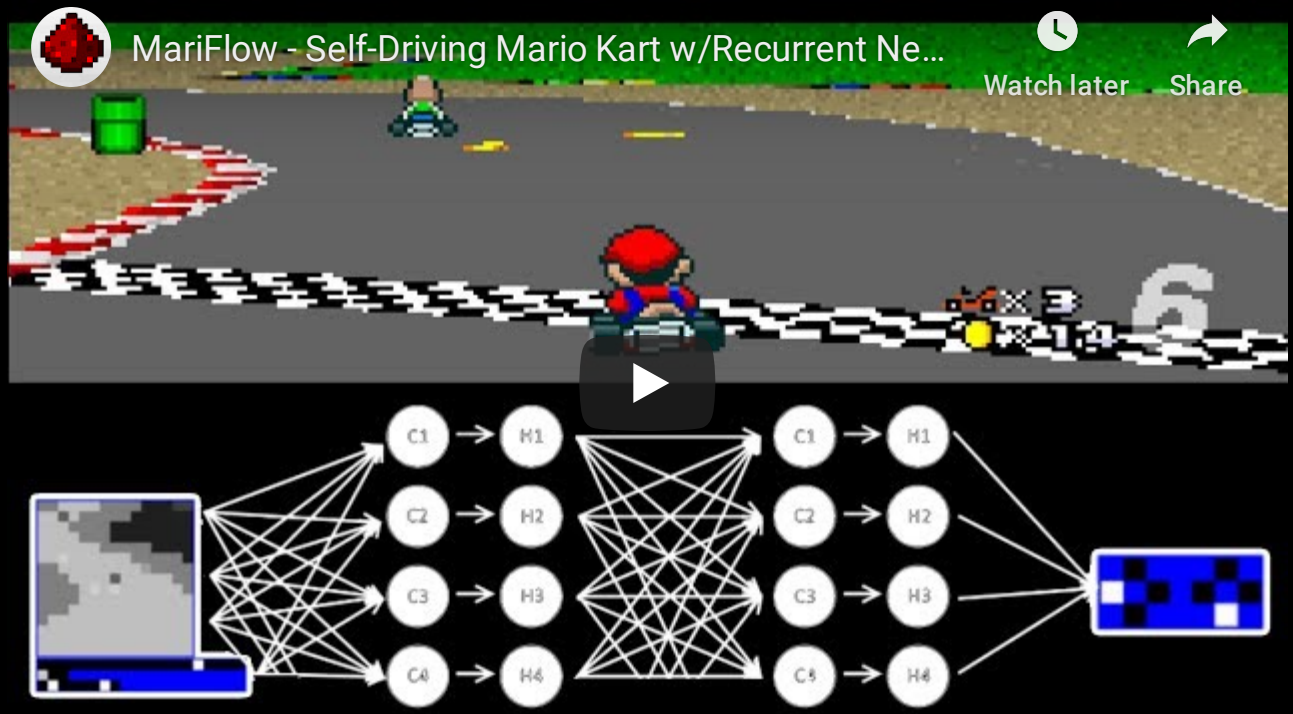


Image credits: Shen et al, 2017. arXiv:1712.05884.



Learning to control

A recurrent network playing Mario Kart.

Beyond sequences

*An increasingly large number of **people are defining the networks procedurally in a data-dependent way (with loops and conditionals)**, allowing them to change dynamically as a function of the input data fed to them. It's really **very much like a regular program, except it's parameterized**.*

Yann LeCun (Director of AI Research, Facebook, 2018)

Neural computers

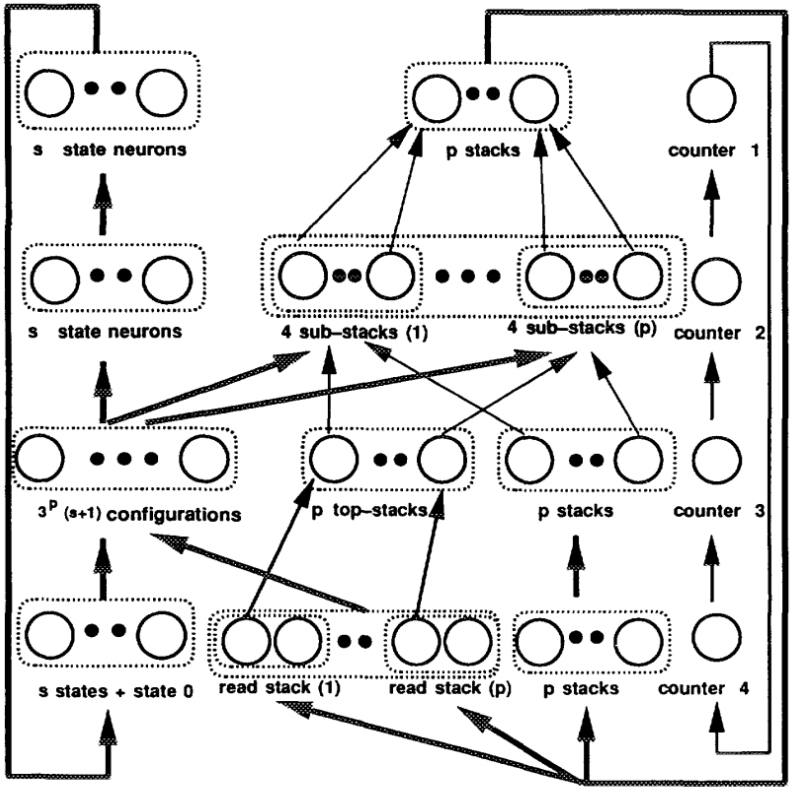
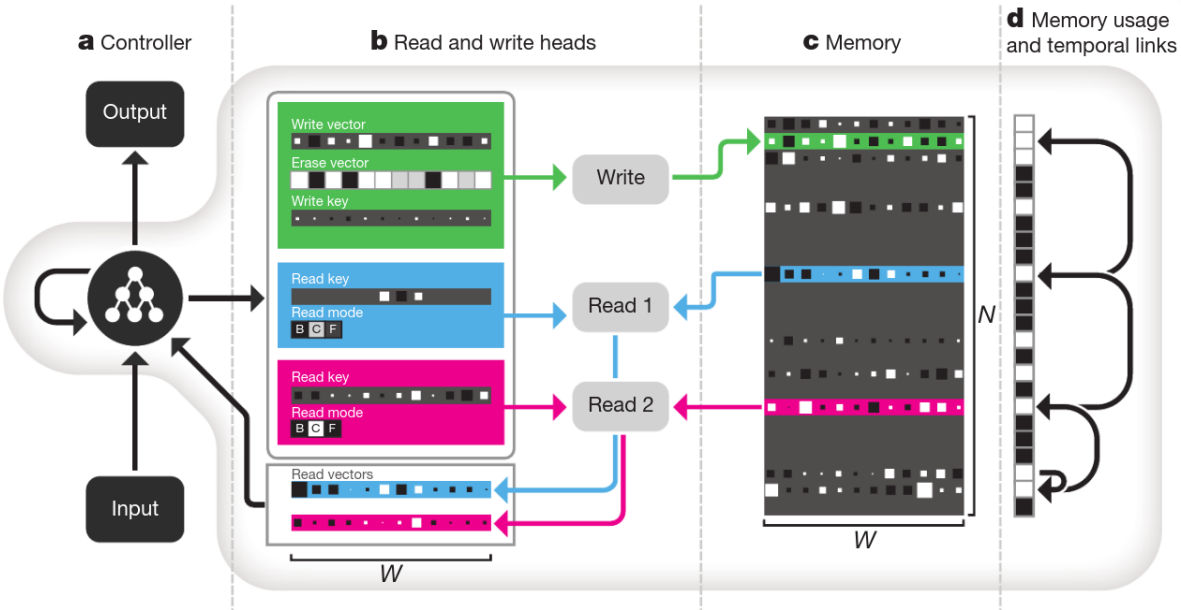


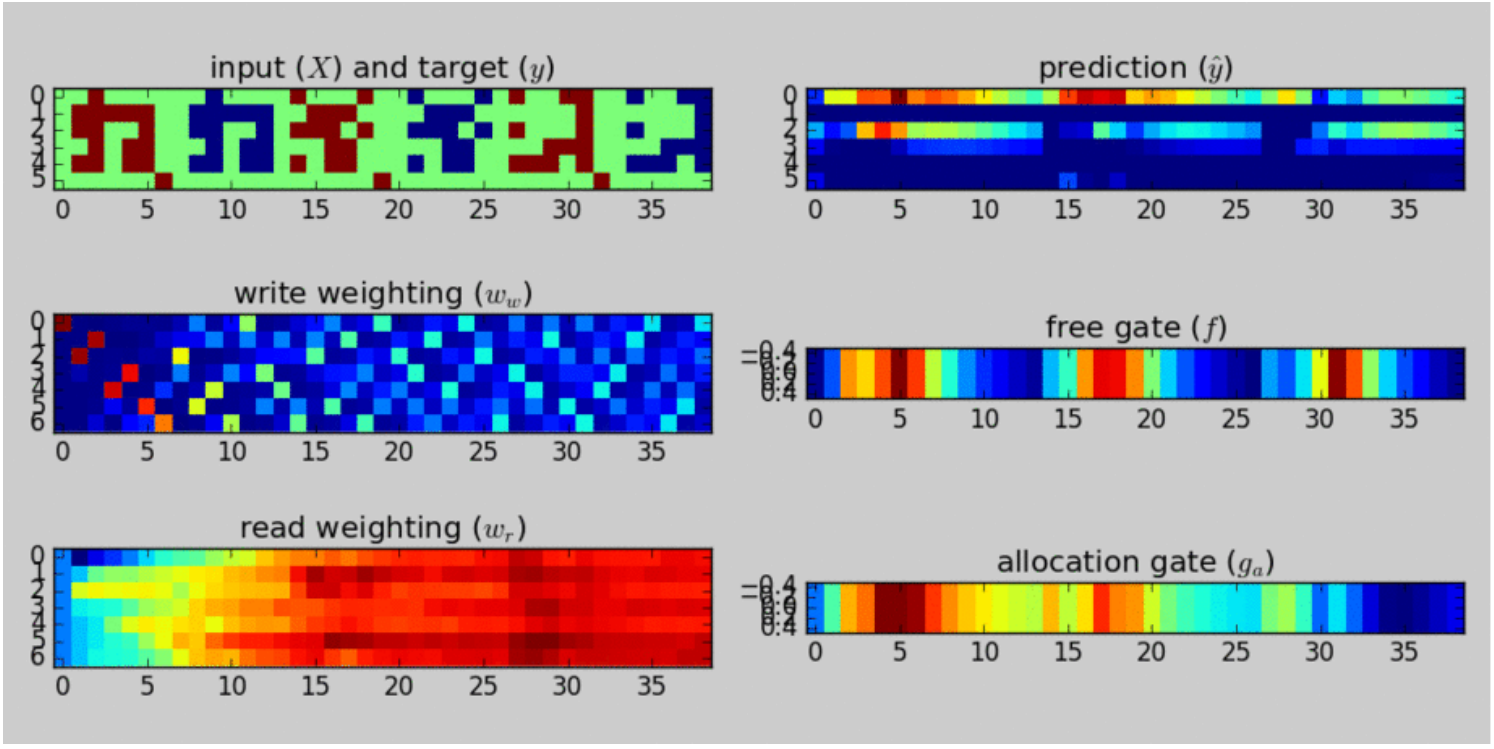
FIG. 1. The universal network.

Any Turing machine can be simulated by a recurrent neural network (Siegelmann and Sontag, 1995)



Networks can be coupled with memory storage to produce neural computers:

- The controller processes the input sequence and interacts with the memory to generate the output.
- The read and write operations attend to all the memory addresses.



A differentiable neural computer being trained to store and recall dense binary numbers. Upper left: the input (red) and target (blue), as 5-bit words and a 1 bit interrupt signal. Upper right: the model's output

Programs as neural nets

The topology of a recurrent network unrolled through time is dynamic.

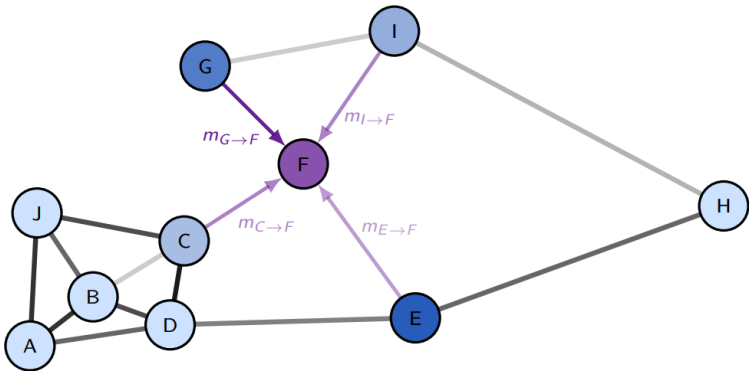
It depends on:

- the input sequence and its size
- a graph construction algorithms which consumes input tokens in sequence to add layers to the graph of computation.

This principle generalizes to:

- arbitrarily structured data (e.g., sequences, trees, graphs)
- arbitrary graph construction algorithm that traverses these structures (e.g., including for-loops or recursive calls).

Neural message passing



Algorithm 1 Message passing neural network

Require: $N \times D$ nodes $\mathbf{x}$, adjacency matrix A

```
h ←Embed(x)
for t = 1, ..., T do
    m ← Message(A, h)
    h ← VertexUpdate(h, m)
end for
r = Readout(h)
return Classify(r)
```

Even though the graph topology is dynamic, the unrolled computation is fully differentiable. The program is trainable.

Credits: Henrion et al, 2017.

Graph neural network for object detection in point clouds

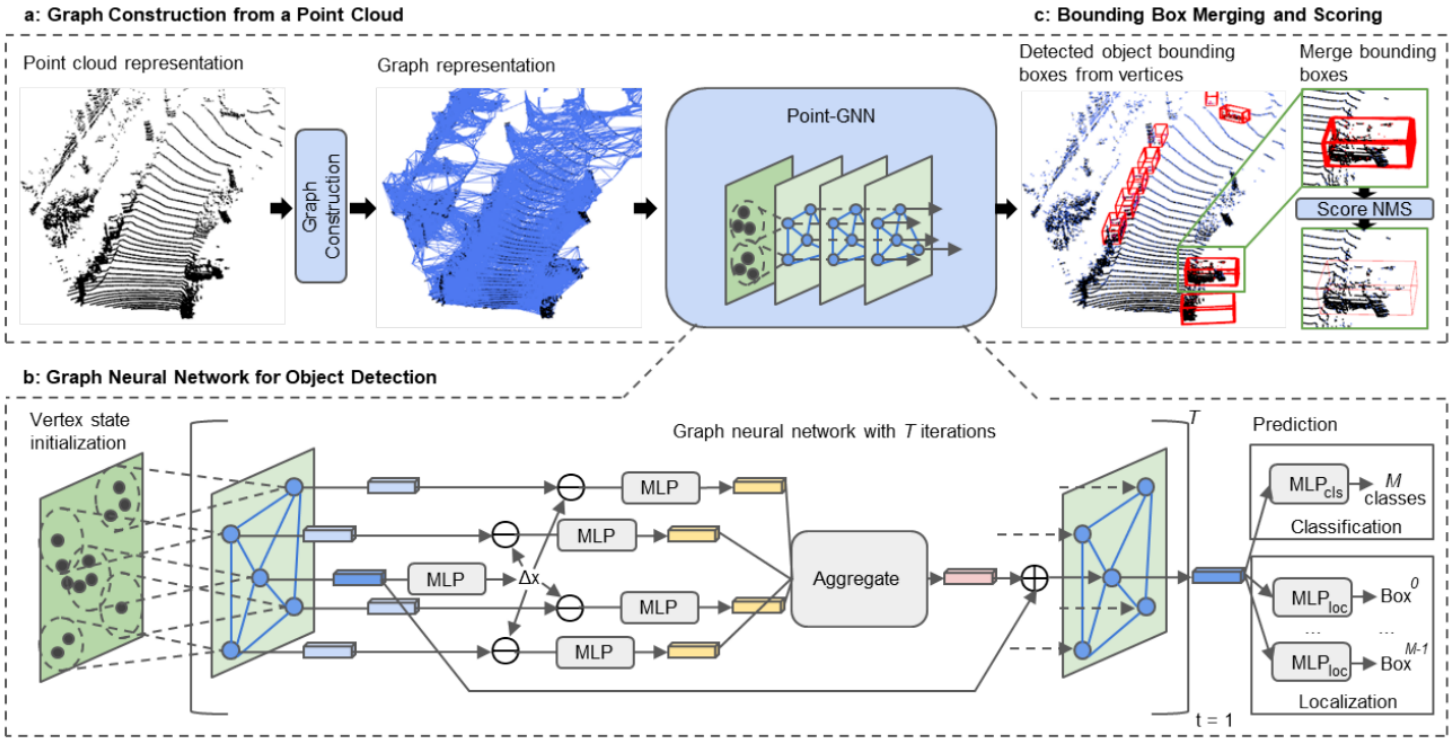
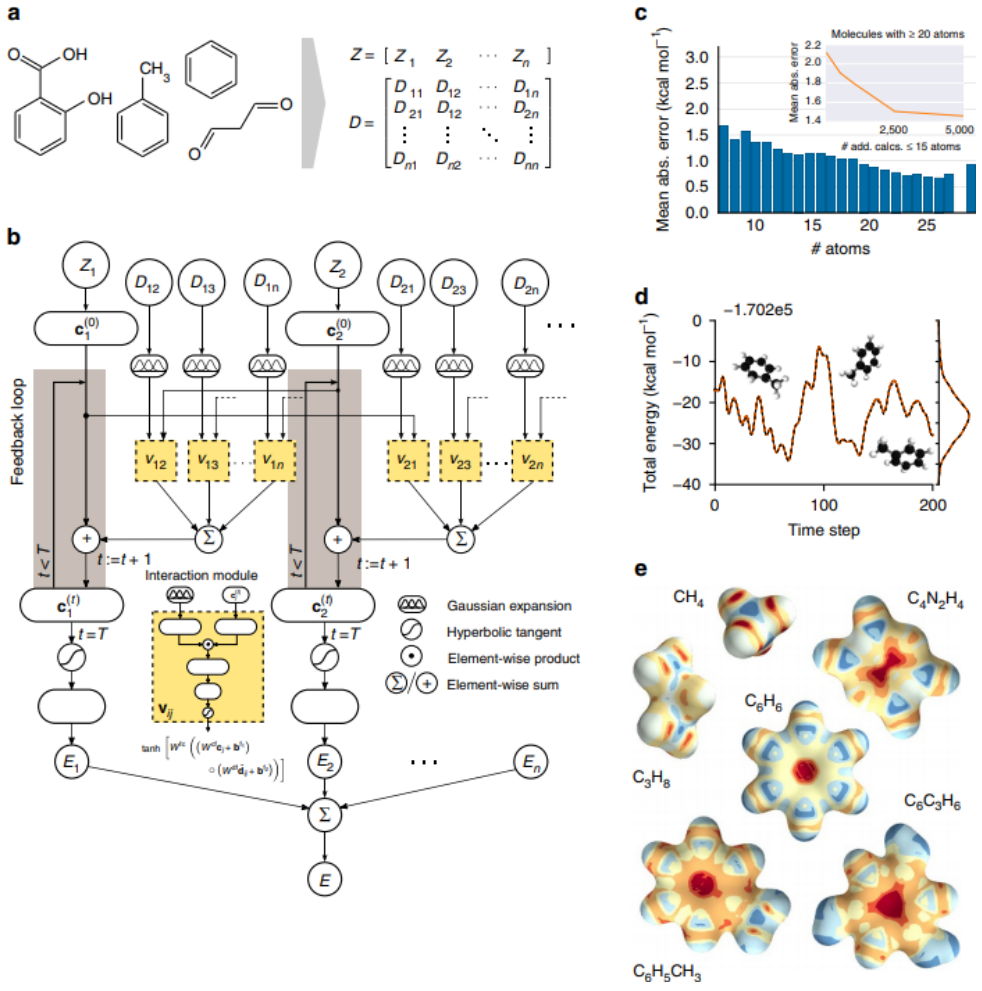


Figure 2. The architecture of the proposed approach. It has three main components: (a) graph construction from a point cloud, (b) a graph neural network for object detection, and (c) bounding box merging and scoring.

Credits: Shi and Rajkumar, Point-GNN, 2020.

Quantum chemistry with graph networks



Credits: Schutt et al, 2017.

Learning to simulate physics with graph networks

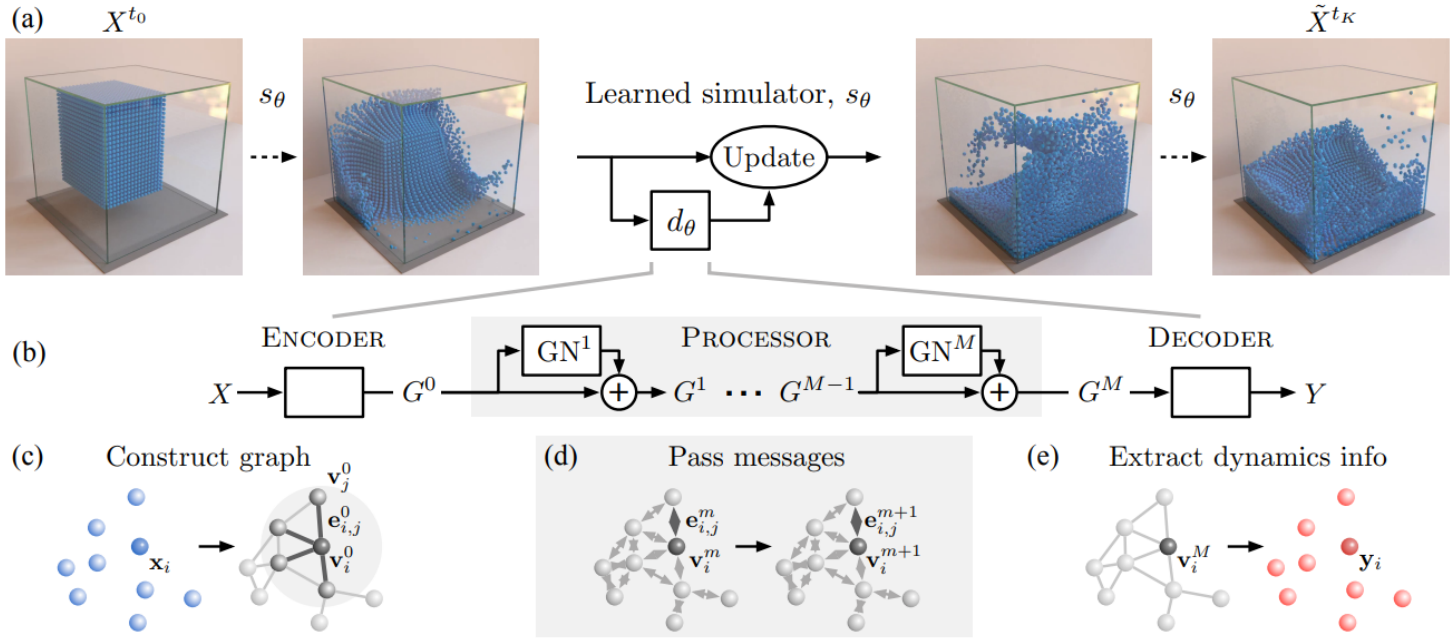
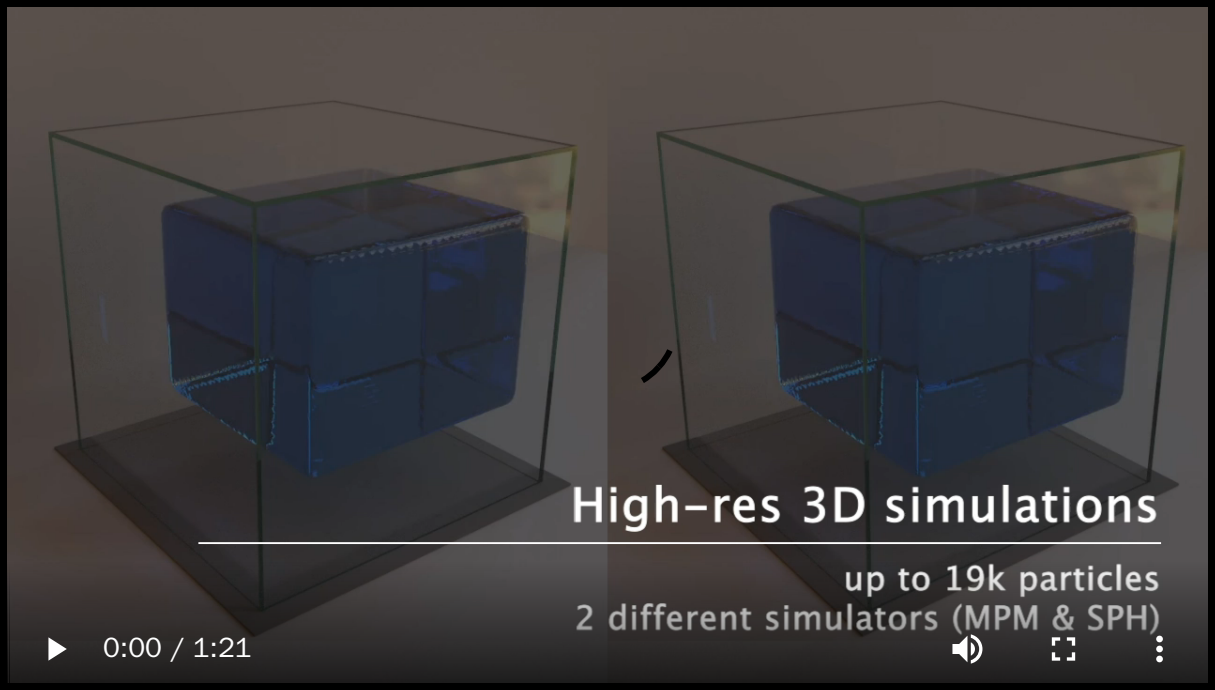


Figure 2. **(a)** Our GNS predicts future states represented as particles using its learned dynamics model, d_θ, and a fixed update procedure. **(b)** The d_θ uses an "encode-process-decode" scheme, which computes dynamics information, Y, from input state, X. **(c)** The ENCODER constructs latent graph, G^0, from the input state, X. **(d)** The PROCESSOR performs M rounds of learned message-passing over the latent graphs, $G^0, \ldots, G^M$. **(e)** The DECODER extracts dynamics information, Y, from the final latent graph, G^M.

Credits: Sanchez-Gonzalez et al, 2020.



Credits: Sanchez-Gonzalez et al, 2020.

The end.

References

- Kyunghyun Cho, "Natural Language Understanding with Distributed Representation", 2015.