Introduction to Recurrent Neural Networks (RNNs) and their Applications (in biomedical signal processing)

learn_bAIome

15-17 October 2024

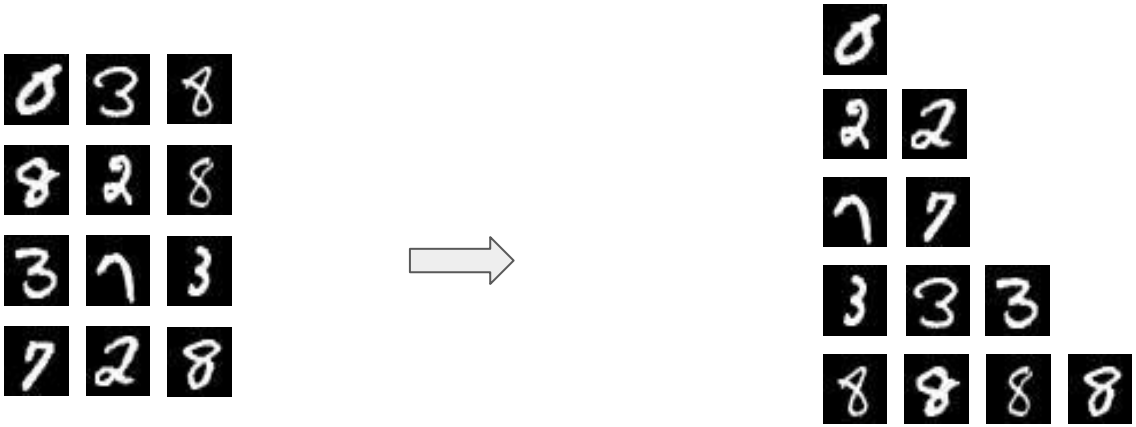
Fatemeh Hadaeghi

Email: f.hadaeghi@uke.de

INSTITUTE OF COMPUTATIONAL NEUROSCIENCE
UNIVERSITY MEDICAL CENTER HAMBURG

UKE HAMBURG bAIome

Task 1:

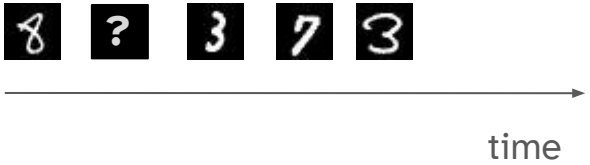


Task 2:

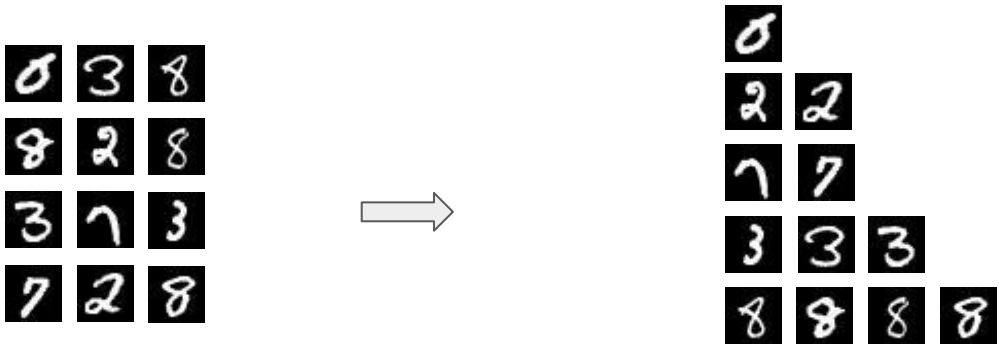
Task 3:

time

T3:

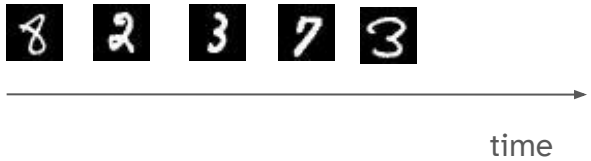


Task 1: Classify these digits

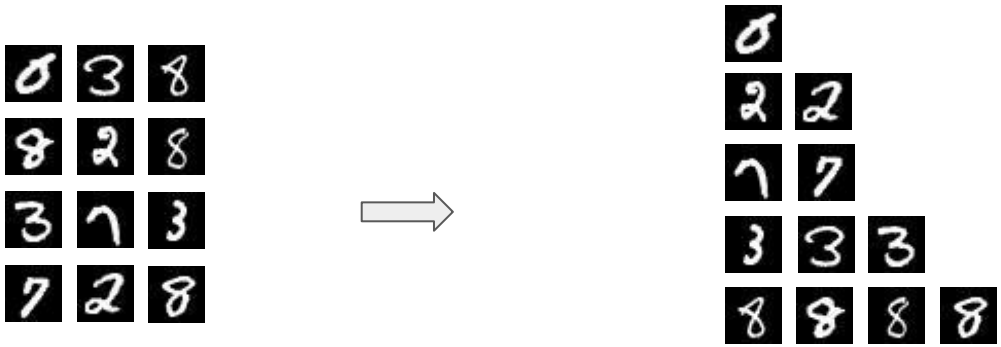


Task 2: Guess the next digits

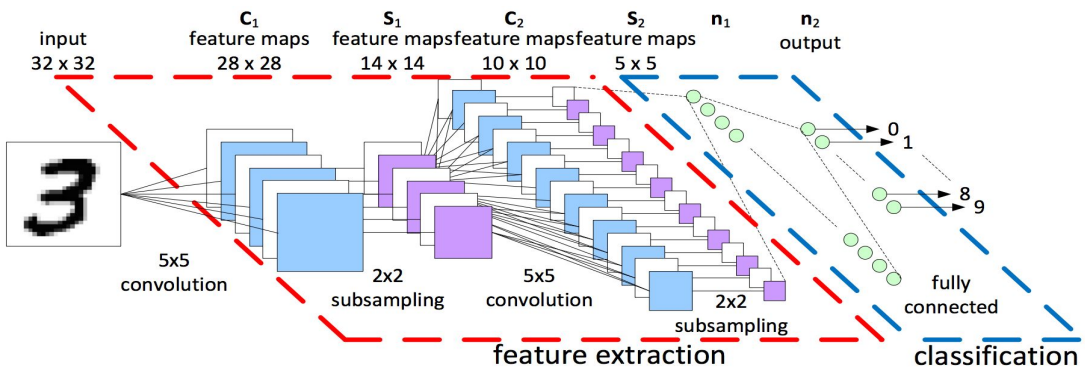
Task 3: Memorize these digits



T1: Classify these digits



Feed-forward Neural Networks



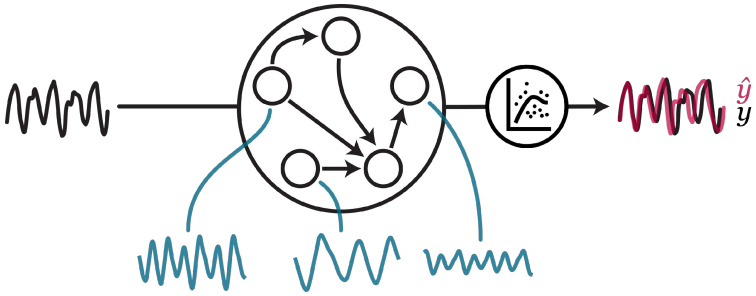
[https://www.kaggle.com]

T2: Guess the next digits

T3: Memorize these digits

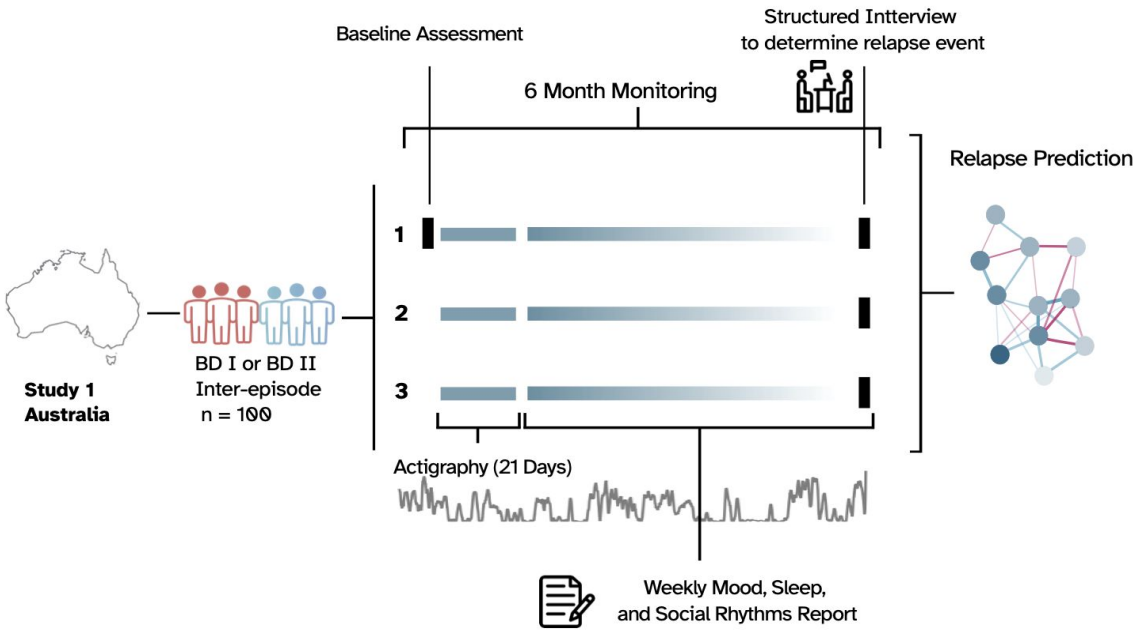
time

Recurrent Neural Networks



RNNs for Sequence Modelling and Signal Processing

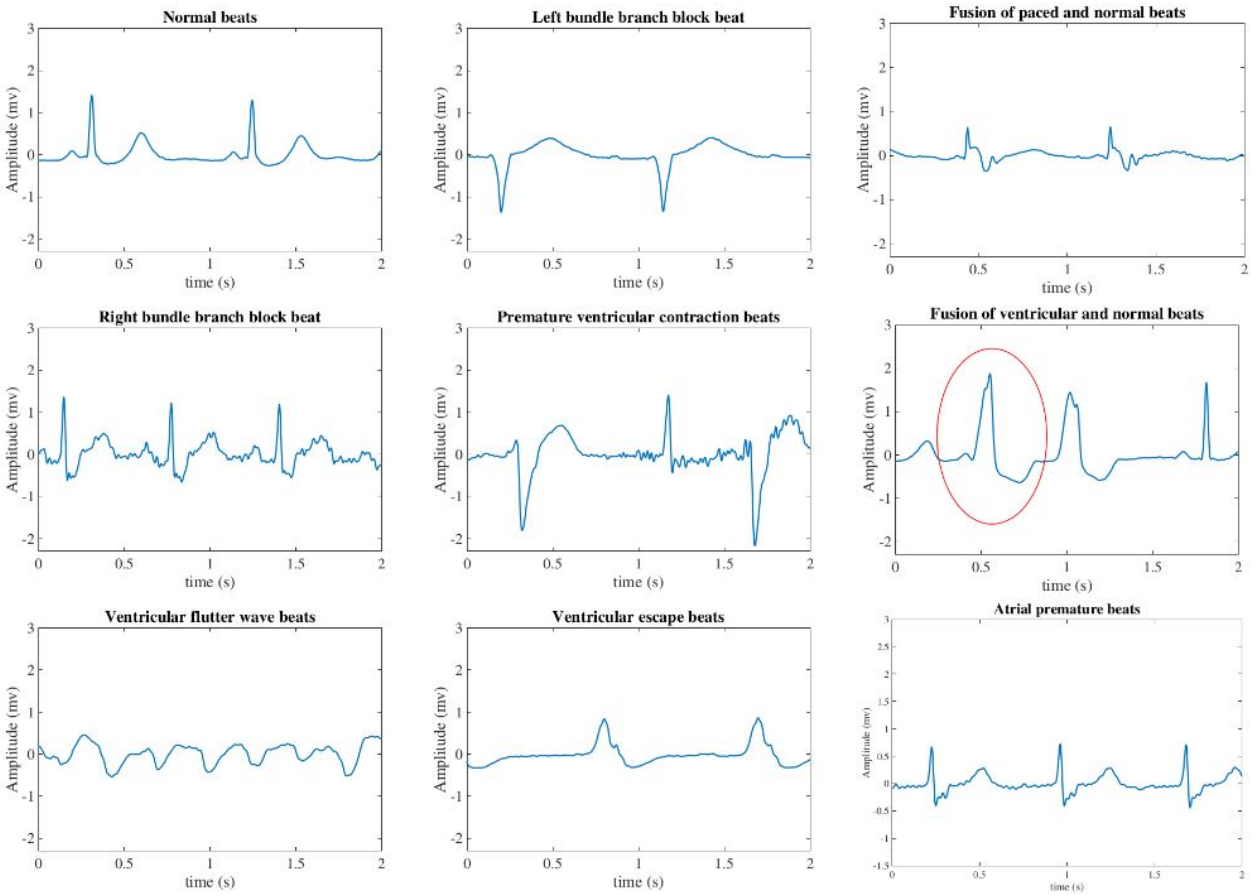
Biological Signal (Data) Processing Tasks



Relapse Prediction in Depression Spectrum

Input: actigraphy time-series of length T

Output: a single True/False value



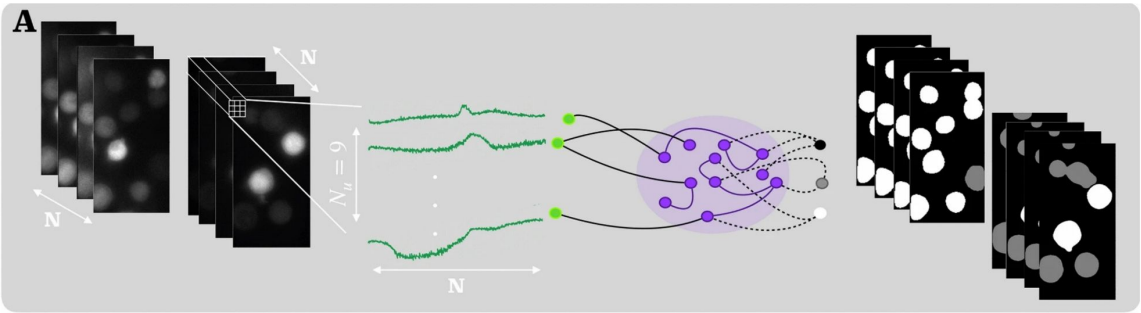
(F. Hadaeghi, 2019)

ECG Monitoring

Input: ECG time-series of length T_x

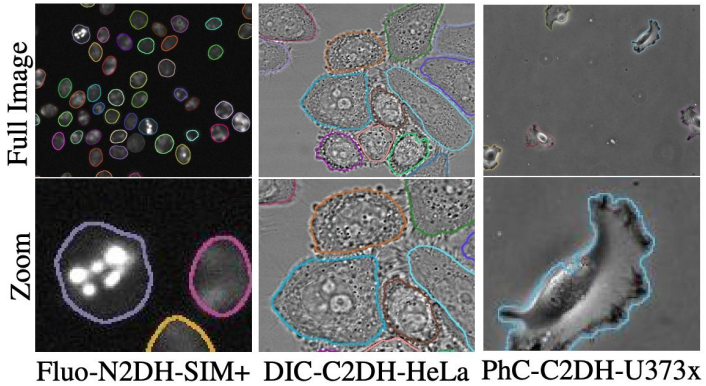
Output: a sequence of length T_y

Cell Segmentation in Live-Cell Ca^{2+} Fluorescence Microscopy



(F. Hadaeghi, 2021)

Microscopy Cell Segmentation

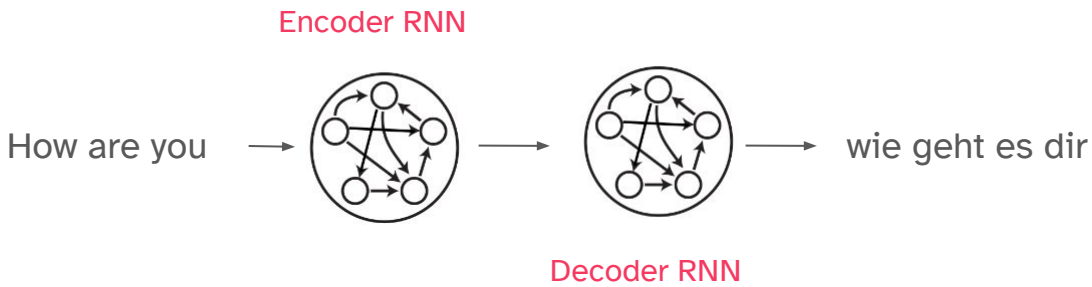


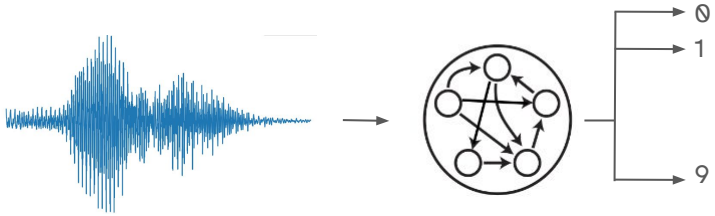
(A. Arbelle and T. Riklin Raviv, 2019)

Other clinically relevant tasks:

- MEG classification
- Sleep stage classification in heart rate data
- Sleep stage classification using EEG signals
- Sentiment analysis in medical texts

Beyond medical applications:
RNNs for Speech Recognition and Machine Translation





RNNs for Machine Translation

RNNs with attention mechanism

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

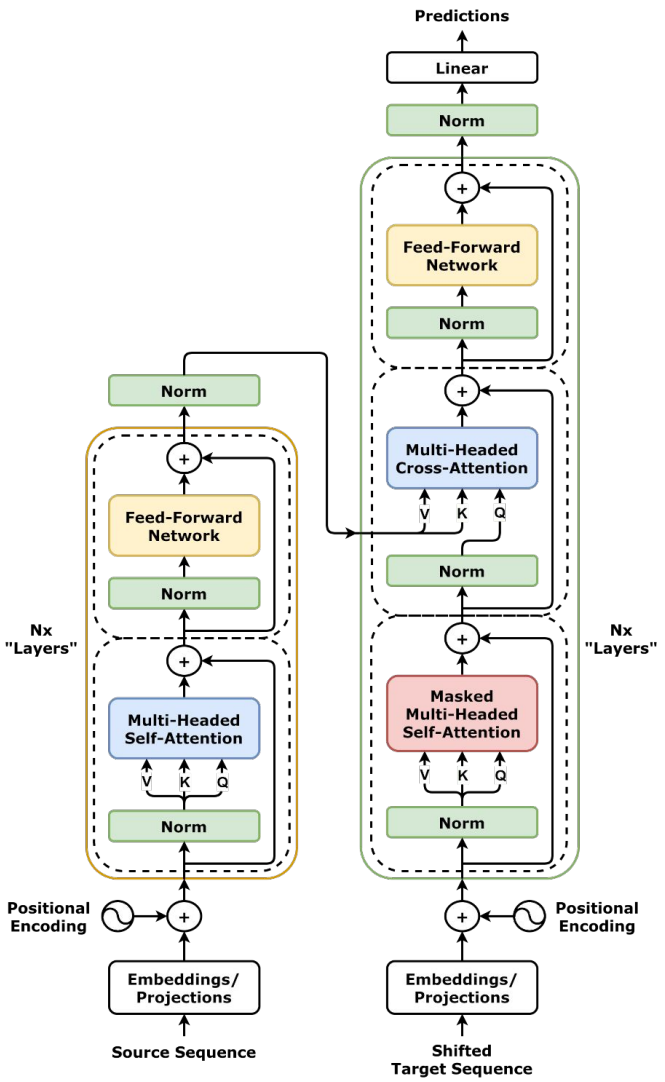
KyungHyun Cho **Yoshua Bengio***
Université de Montréal

ABSTRACT

Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine translation aims at building a single neural network that can be jointly tuned to maximize the translation performance. The models proposed recently for neural machine translation often belong to a family of encoder–decoders and encode a source sentence into a fixed-length vector from which a decoder generates a translation. In this paper, we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder–decoder architecture, and propose to extend this by allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly. With this new approach, we achieve a translation performance comparable to the existing state-of-the-art phrase-based system on the task of English-to-French translation. Furthermore, qualitative analysis reveals that the (soft-)alignments found by the model agree well with our intuition.

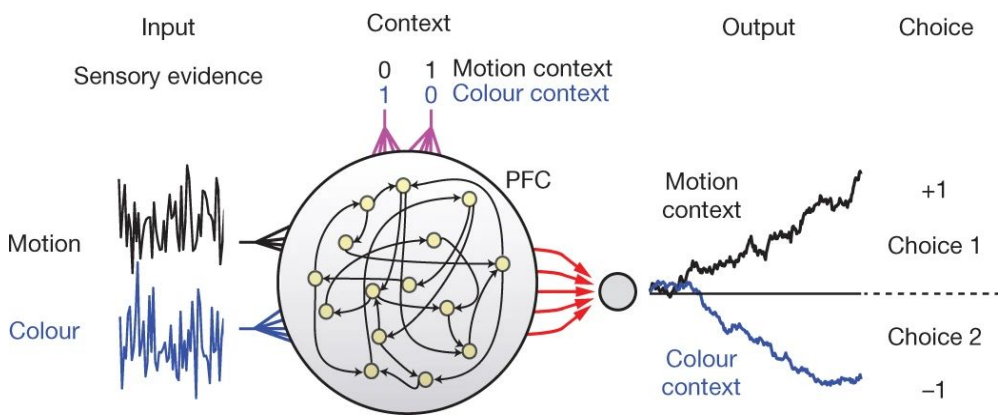
RNNs vs. Transformers

No RNNs any more!

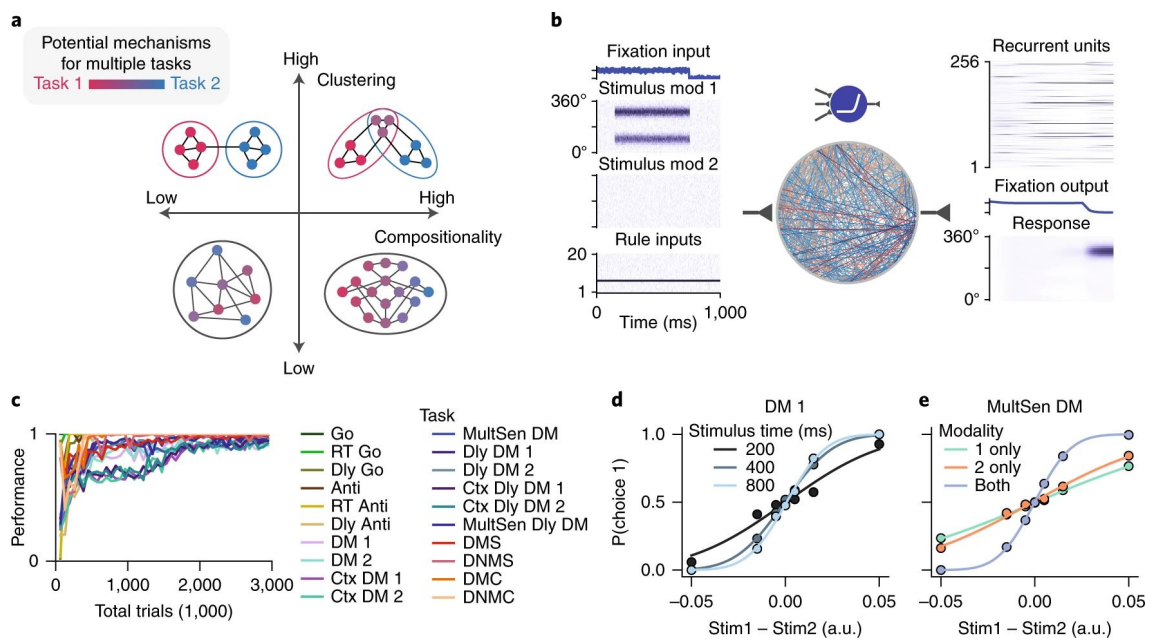


(Vaswani et al, 2017)

RNNs for Computational Modelling

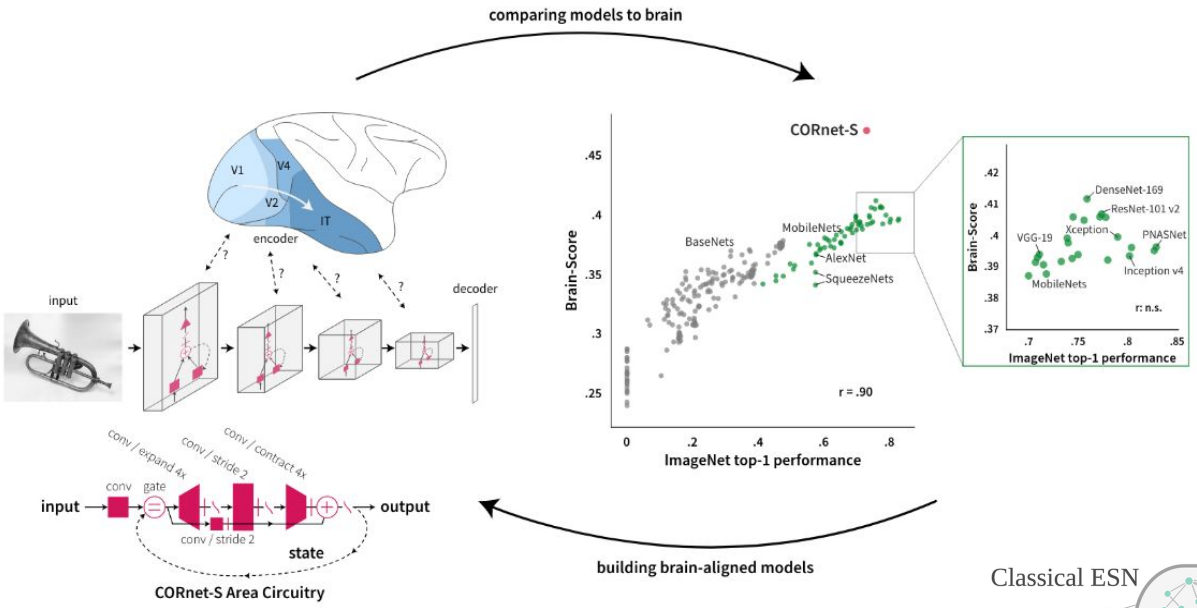


(V. Mante, et al. 2013)

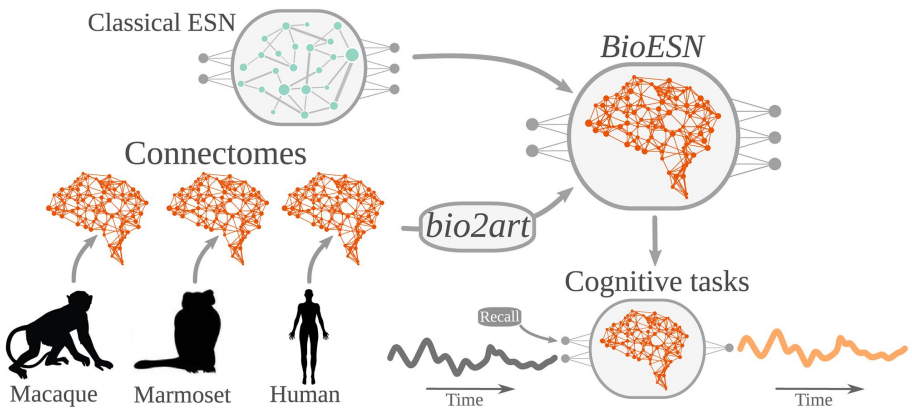


(R. Yang, et al. 2019)

RNNs for Computational Modelling

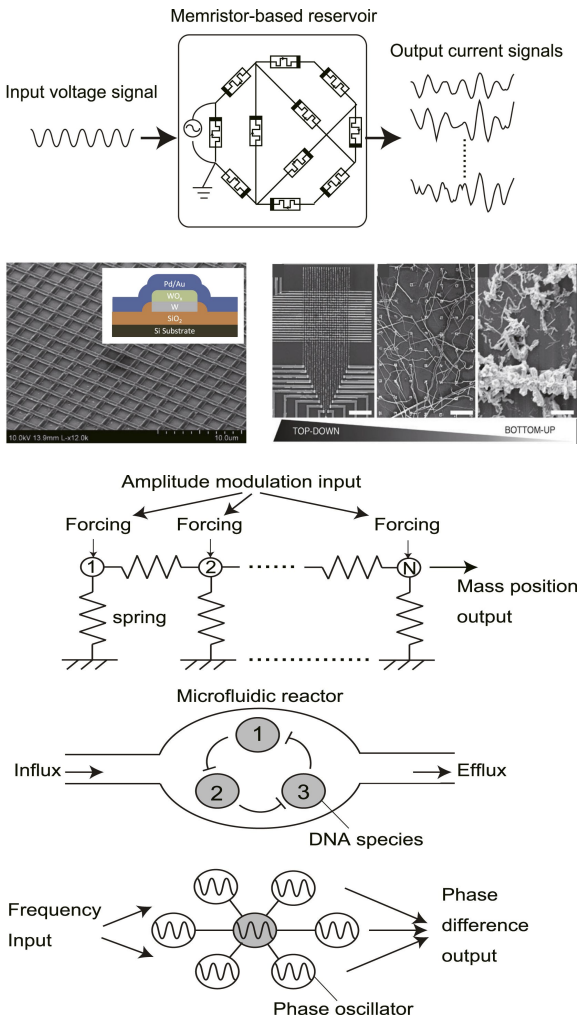


(J. Kubilius, et al., 2019).

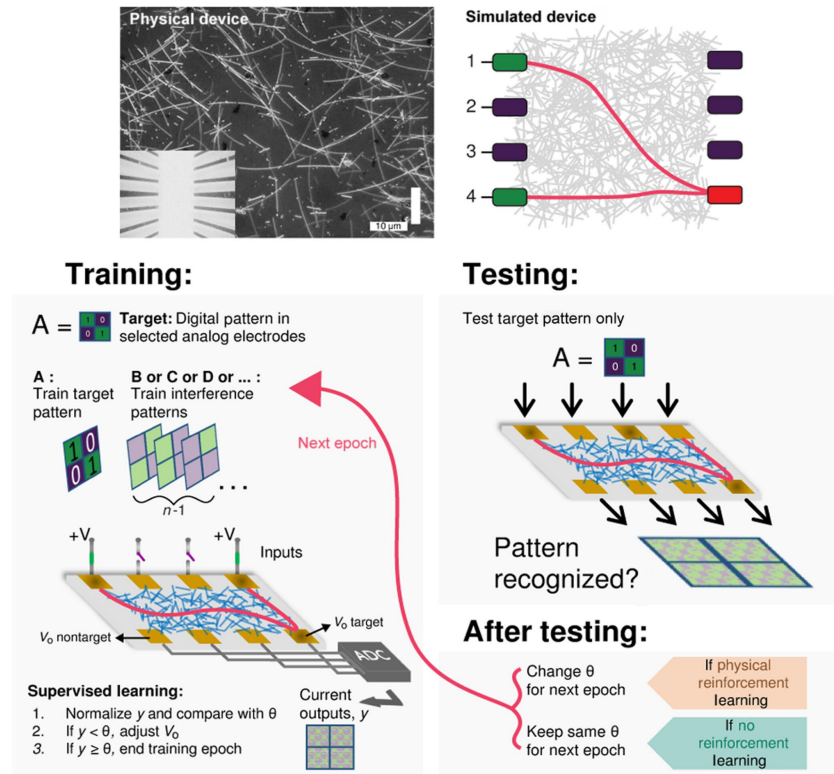


F. Damicelli, et al. 2021

RNNs for Neuromorphic Computing



(G. Tanaka, et al. 2019)



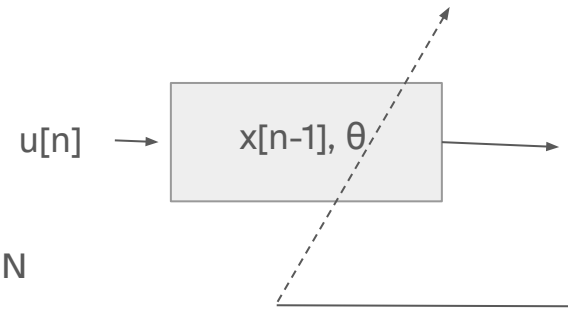
(A. Loeffler, et al. 2023)

RNN: Design and Training

Notation (based on Discrete-Time Dynamics)

Supervised Learning

A set of (u, y) is given.



y[n]

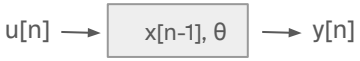
$$\hat{y}[n] = g(u[n], x[n-1], \theta)$$

$$\text{error} = y[n] - \hat{y}[n]$$

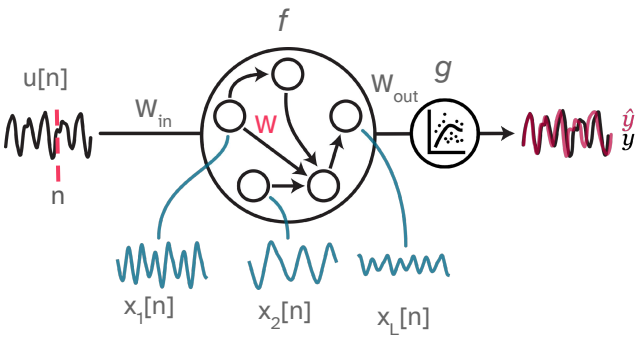
Time steps: n = 1, 2, 3, ..., N

Notation (based on Discrete-Time Dynamics)

Time steps: n = 0, 1, 2, 3, ..., N



y[n]= g(u[n], x[n-1], θ)



Input, output, hidden states:

$\mathbf{x}[n] = f(W\,\mathbf{x}[n-1] + W_{in}\,\mathbf{u}[n] + b)$,
$y[n] = g(W_{out}\,\mathbf{x}[n])$

Input: u[n]

General case: with K input channels: $\mathbf{U}[n] = [u_1[n], u_2[n], ..., u_K[n]]$

Hidden/activation state: $\mathbf{x}[n] = [x_1[n], x_2[n], ..., x_L[n]]$

L: number of neurons in the recurrent layer

W_{in} : input-to-hidden weights, a vector of size K × L
W: weights in the hidden layer, a matrix of size L × L
b: a vector of biases with L entities
f(.): nonlinear function; e.g., tanh, sigmoid, ReLU

Notation (based on Discrete-Time Dynamics)

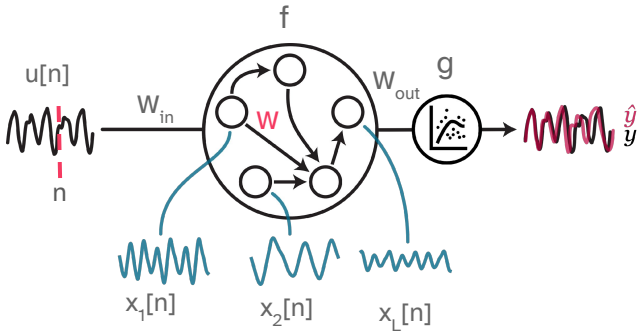
$$u[n] \longrightarrow \boxed{x[n-1], \theta} \longrightarrow y[n]$$

$$y[n] = g(u[n], x[n-1], \theta)$$

Time steps: $n = 0, 1, 2, 3, \ldots, N$

Input, output, hidden states:

$\mathbf{x}[n] = f(W\,\mathbf{x}[n-1] + W_{in}\,\mathbf{u}[n] + b)$,
$y[n] = g(W_{out}\,\mathbf{x}[n])$



Output: $y[n]$

General case: with M output channels: $\mathbf{Y}[n] = [y_1[n], y_2[n], \ldots, y_M[n]]$

W_{out}: hidden-to-output weights, a vector of size $L \times M$

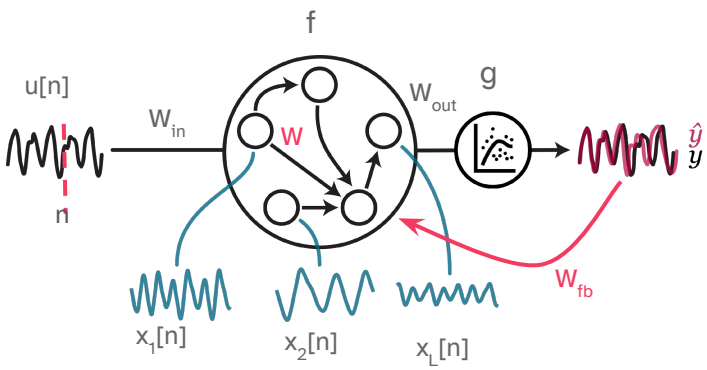
g(.): linear or nonlinear function

Notation

Feedback:

This type of output feedback is necessary when the task does not involve a simple input-to-output transformation, where the output is directly driven by the input signal. Instead, it applies to scenarios where the output is actively produced by the RNN, even in the absence of an input signal.

$$\mathbf{x}[n] = f(W\,\mathbf{x}[n-1] + W_{in}\,\mathbf{u}[n] + W_{fb}y[n-1] + b),$$
$$y[n] = g(W_{out}\,\mathbf{x}[n])$$

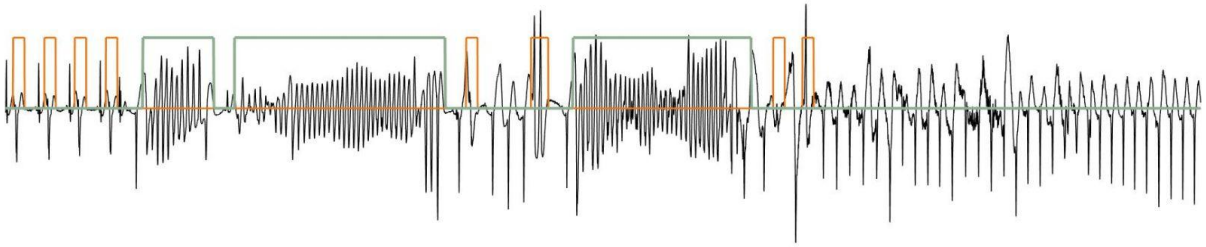


Notation

Task Formulation Scenarios and Temporal Learning Setups:

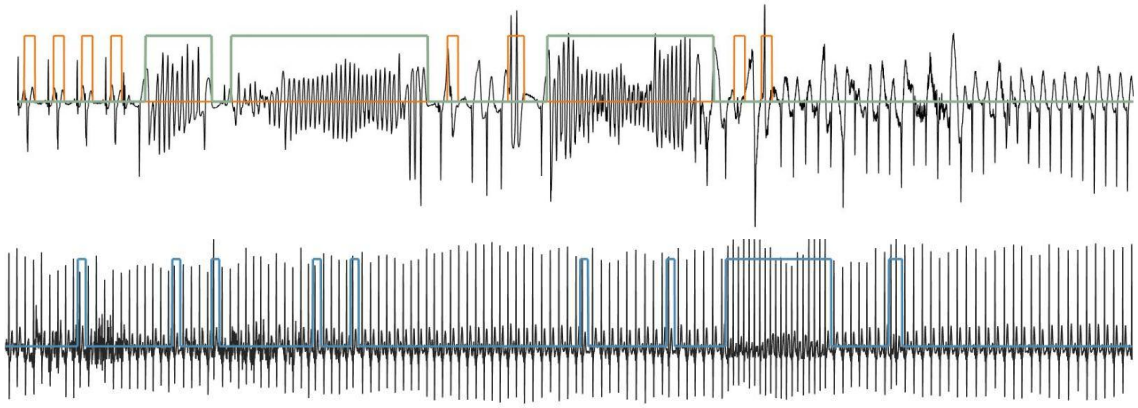
- **A single long input/output:** (u[n], y[n])

Challenge: one has to assess stationarity (and ergodicity) of time-series before designing the task



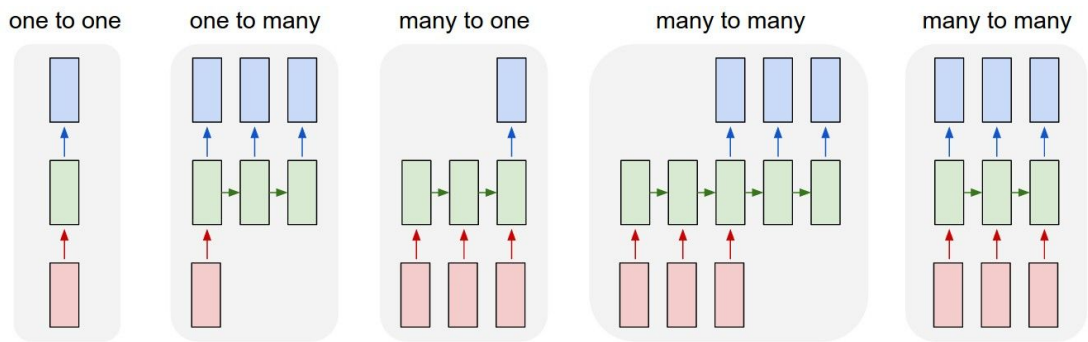
- **Multiple (subjects) long input/output:** ($u^i[n]$, $y^i[n]$)

This formulation fits better with the non-stationary time-series



Notation

RNN implementation types:



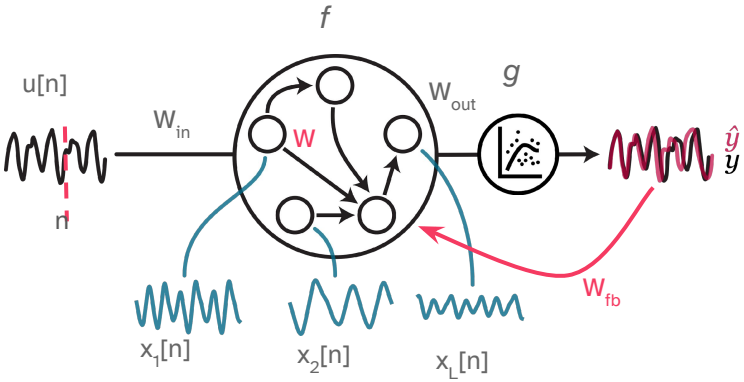
(Andrej Karpathy's blog, 2015)

- **One-to-one:** Transform one signal to another one with the same length, e.g., denoise a low-quality ECG recording
- **one/zero-to-many:** Pattern generation, e.g., models of CPG, or synthetic EEG signals
- **Many-to-one:** Pattern detection tasks, e.g., given the cardiac beat, find its R peak or label the beat type (healthy/pathological)
- **Many-to-many (different input-output length):** Sequential event detection, e.g., seizure detection in EEG
- **Many-to-many (same input-output length):** Real-time time-series segmentation and labeling, e.g., ECG monitoring and labeling P wave, QRS complex, and T wave

Training RNNs

How to optimizing the weight matrices, and the bias vector, such that for a given test dataset, i.e, ($u_{test}[n]$, $y_{test}[n]$) the network's output, $\hat{y}[n]$, closely approximates $y_{test}[n]$ based on a chosen loss metric.

Loss functions



- **Quadratic loss**

Case 1: only last time point matters

$$\mathcal{L}(\hat{y}[n], y[n]) = ||\hat{y}[n] - y[n]||^2$$

Case 2: predictions throughout the entire sequence matter

$$\Sigma_{n=1,\ldots,k} a_n \mathcal{L}_0(\hat{y}[n], y[n])$$

Loss functions

For convenience, let's denote $y[n]$ as y and $\hat{y}[n]$ as $\hat{y}$ and use i to indicate i-th training sample

- **Binary Cross-Entropy Loss**

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right)$$

- **Cross-Entropy Loss (Categorical Cross-Entropy)**

$$\mathcal{L}_{\text{Cross-Entropy}} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

- **Negative Log Likelihood Loss (NLL Loss)**

$$\mathcal{L}_{\text{NLL}} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)\cdot$$

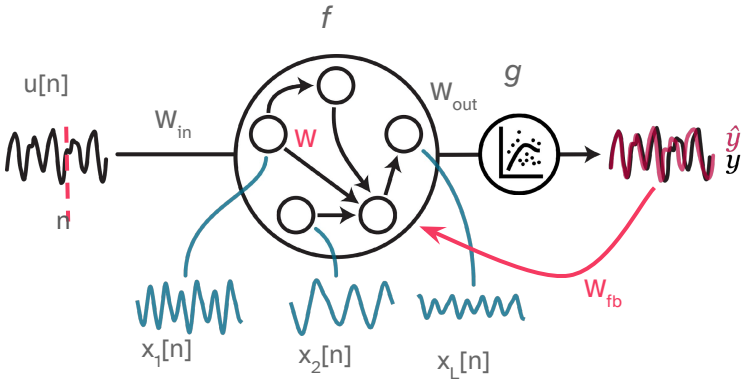
- **Connectionist Temporal Classification (CTC) Loss**

$$\mathcal{L}_{\text{CTC}} = -\log(p(y \mid x))$$

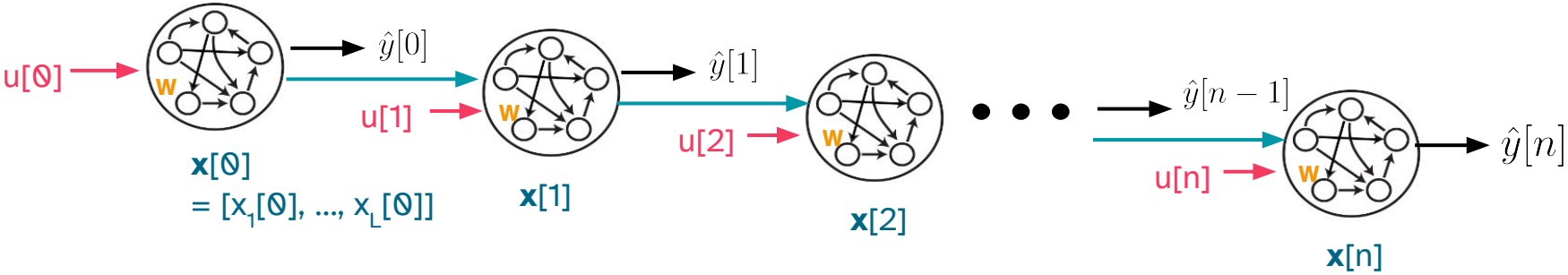
- **Hinge Loss**

$$\mathcal{L}_{\text{Hinge}} = \frac{1}{N}\sum_{i=1}^{N}\max(0, 1 - y_i \cdot \hat{y}_i)$$

Training RNNs: Backpropagation Through Time (BPTT)



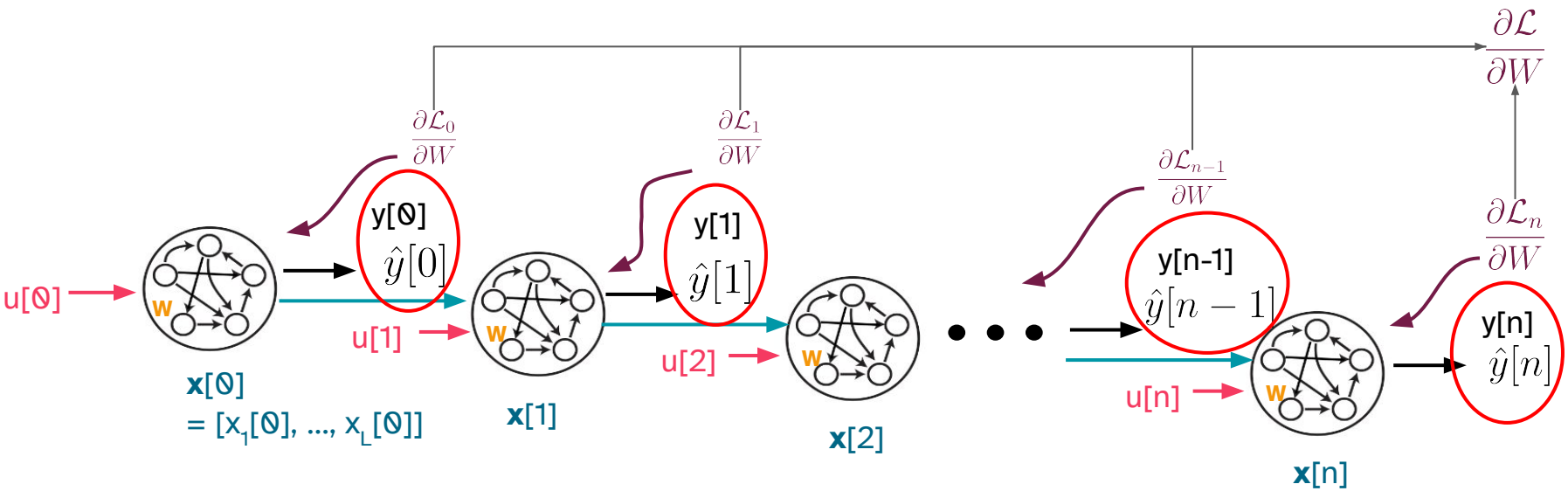
Forward path



Training RNNs: Backpropagation Through Time (BPTT)

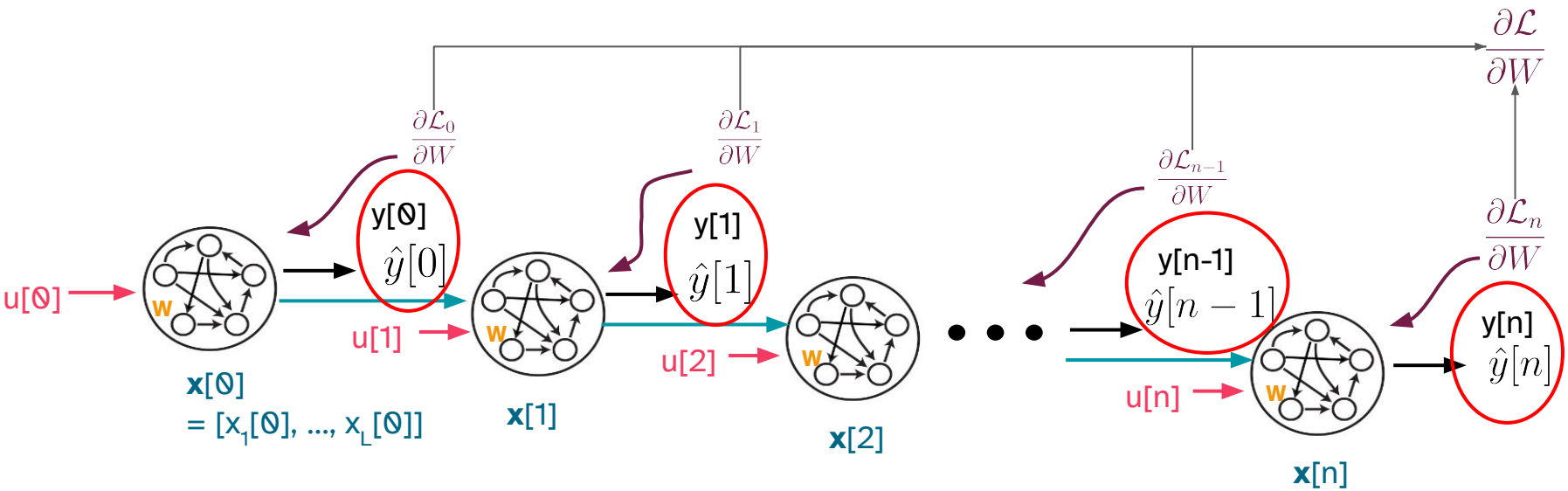
Backward path

$$\mathcal{L} = \sum_{k=1}^{N} \mathcal{L}_k$$



Challenges in Training RNNs

- Computational Complexity
- Difficulty in Learning Long-Term Dependencies
- Exploding Gradients
- Vanishing Gradients



Challenges in Training RNNs

- **Computational Complexity**

RNNs are sequential by nature, meaning that the computation of the current step depends on the previous step. This lack of parallelization makes RNNs slower to train, especially when working with very long sequences.

Solutions

- ❑ Techniques like truncated backpropagation through time (TBPTT) are often used to limit the number of time steps over which gradients are propagated, speeding up the training process.
- ❑ Reservoir computing (RC) paradigm introduces an inexpensive method to design and train RNNs (we will discuss it a few slides later)

Challenges in Training RNNs

- **Difficulty in Learning Long-Term Dependencies**

The stack of temporal copies in unfolding the RNN must have a finite depth (also referred to as “horizon”) to enable the backpropagation algorithm. Simple RNN architectures often fail to capture patterns that span a large number of time steps or to solve multi-timescale problem.

Solution

- ❑ LSTMs, GRUs, and more complex architectures (e.g., Transformers) are designed to handle longer sequences by explicitly controlling which information is retained or forgotten. (we will discuss LSTMs and RGUs a few slides later)

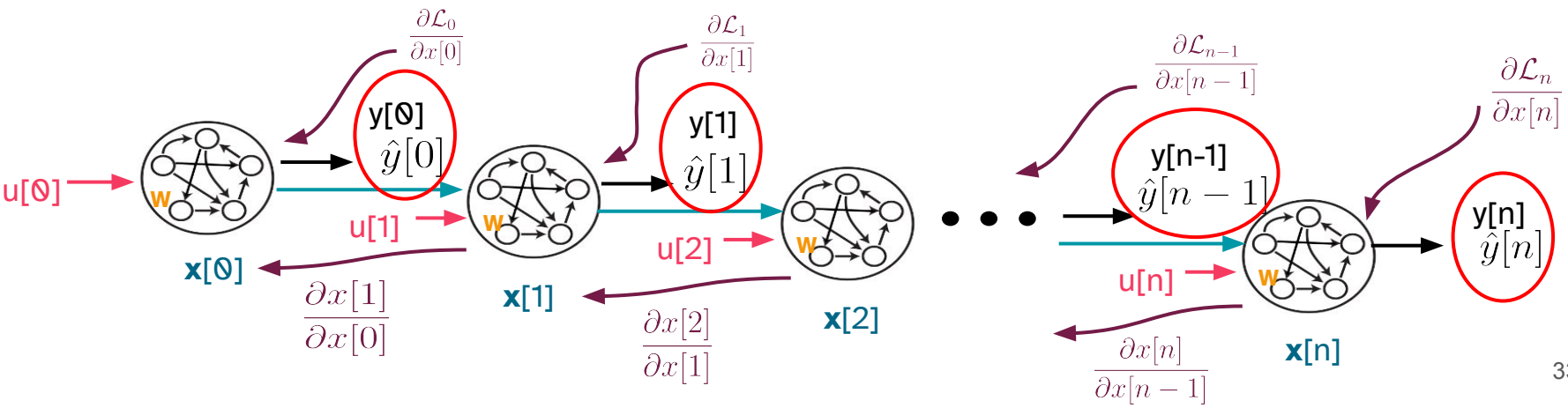
Challenges in Training RNNs

- **Exploding and Vanishing Gradients**

To train RNNs the Gradient of the loss function, $\mathcal{L} = \sum_{k=1}^{N} \mathcal{L}_k$ is computed with respect to the weights in the hidden layer: $\frac{\partial \mathcal{L}}{\partial W}$

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial x[n]} \cdot \sum_{k=1}^{n} \frac{\partial x[n]}{\partial x[k]} \cdot \frac{\partial x[k]}{\partial W}$$

$$\frac{\partial x[k]}{\partial W} = f'(Wx[k-1] + W_{in}u[k] + b) \cdot x[k-1]^\top$$



Challenges in Training RNNs

- **Exploding and Vanishing Gradients**

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial x[n]} \cdot \sum_{k=1}^{n} \frac{\partial x[n]}{\partial x[k]} \cdot \frac{\partial x[k]}{\partial W}$$

$$\frac{\partial \mathcal{L}}{\partial x[n]} = \frac{\partial \mathcal{L}}{\partial x[N]} \cdot \prod_{k=n}^{N} \frac{\partial x[k]}{\partial x[k-1]}$$

$$\frac{\partial x[k]}{\partial x[k-1]} = W^{\top} \cdot \mathrm{diag}(f'(Wx[k-1] + W_{in}u[k] + b))$$

Thus, the total gradient over multiple time steps depends on the product of many Jacobian matrices, each corresponding to $\frac{\partial x[k]}{\partial x[k-1]}$:

$$\prod_{k=n}^{N} \frac{\partial x[k]}{\partial x[k-1]} = \prod_{k=n}^{N} W^{\top} \cdot \mathrm{diag}(f'(Wx[k-1] + W_{in}u[k] + b))$$

Vanishing Gradients

$$\prod_{k=n}^{N} \frac{\partial x[k]}{\partial x[k-1]} = \prod_{k=n}^{N} W^{\top} \cdot \mathrm{diag}(f'(Wx[k-1] + W_{in}u[k] + b))$$

The **vanishing gradient problem** occurs when the norms of these Jacobian matrices are less than 1. If the eigenvalues of W are less than 1, the gradients shrink exponentially as they propagate backward through time:

$$\left\| \prod_{k=n}^{N} W^{\top} \right\| \approx \|W^{\top}\|^{N-n}$$

This results in very small gradients, making it hard for the network to learn long-term dependencies.

If $||W|| < 1$ the gradient decays exponentially, leading to vanishing gradients:

$$\frac{\partial \mathcal{L}}{\partial W} \approx 0$$

Exploding Gradients

$$\prod_{k=n}^{N} \frac{\partial x[k]}{\partial x[k-1]} = \prod_{k=n}^{N} W^{\top} \cdot \mathrm{diag}(f'(Wx[k-1] + W_{in}u[k] + b))$$

The **exploding gradient problem** occurs when the norms of the Jacobian matrices are greater than 1. If the eigenvalues of W are greater than 1, the gradients grow exponentially as they propagate backward through time:

$$\left\| \prod_{k=n}^{N} W^{\top} \right\| \approx \|W^{\top}\|^{N-n}$$

This results in very large gradients, which can lead to unstable training and numerical overflow.

If $||W|| > 1$ the gradient grows exponentially, leading to exploding gradients:

$$\frac{\partial \mathcal{L}}{\partial W} \to \infty$$

Mitigating Vanishing and Exploding Gradients

- **Vanishing gradients** can be mitigated using architectures like Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRU), which include gating mechanisms to preserve gradients over time.
- **Exploding gradients** are commonly addressed by techniques like **gradient clipping**, which restricts the size of gradients to a predefined threshold.

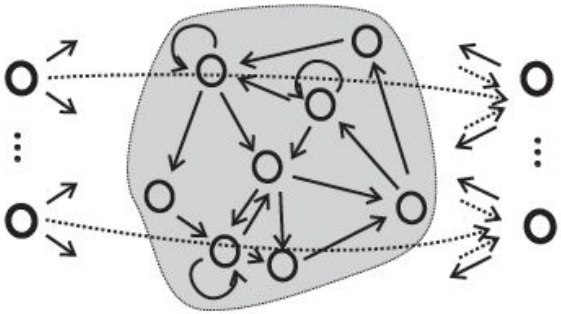
Reservoir Computing

Basic Idea

- The RC paradigm suggest that **fixed random hidden layers** within RNNs, combined with nonlinear neuron activation functions, create a broad range of representations for input stimuli.

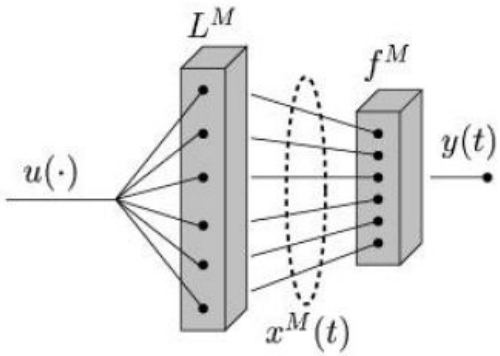
- By solely training a linear read-out mechanism on these representations, tasks can be solved efficiently, exploiting the richness of the generated representations.

Echo-State-Network (ESN)



(H. Jaeger and H. Haas, 2004)

Liquid-State-Machine(LSM)



(W. Maass and H. Markram, 2004)

Advantages

- **Simple Training**: Only the output layer is trained, making it computationally efficient.
- **Memory**: Natural time-series processing abilities due to reservoir dynamics.
- **Non-linearity**: High-dimensional projection allows complex patterns to emerge.

When and where RC is applied?

- In **neuroscience**, RC models serve as easily accessible representations of neuronal circuits due to their cost-effective training.
    - A variety of transfer functions, from **spiking neuron models** to **population-based models**, can be chosen for each network node.
- **Beyond neuroscience**, recent advances in material science and nanotechnology have made reservoir computing an increasingly popular approach for implementing computations on unconventional hardware platforms, such as photonic circuits and memristive materials and devices

How RC Works

Problem Statement:

Given: A training input/output sequence (u(1), y(1)), ..., (u(N), y(N)).

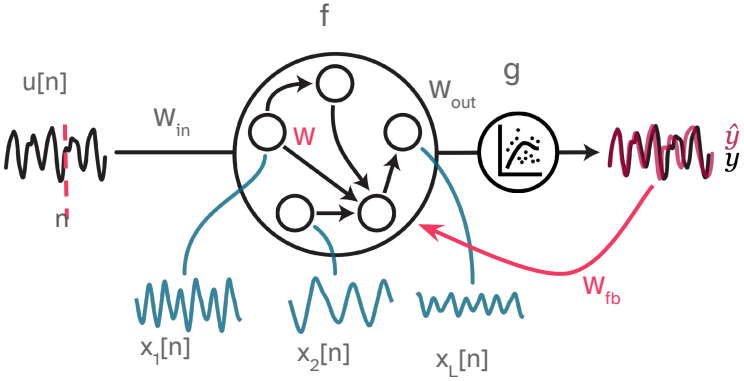
Wanted: A trained ESN (W_{in}, W, W_{fb}, W_{out}) whose output $\hat{y}(n)$ approximates the teacher output y(n), when the ESN is driven by the training input u(n).

1. Create an untrained network (W_{in}, W, W_{fb}) which has the **echo state property**, and whose internal units exhibit mutually interestingly different dynamics when excited. (Not an easy task!)
2. Sample network training dynamics, **x**[n]: $\mathbf{x}[n] = f(W\,\mathbf{x}[n-1] + W_{in}\,\mathbf{u}[n] + W_{fb}y[n-1] + b)$
3. Now you have a large matrix, x with size of L × N, and a desired output vector with N elements. Compute the output weights by solving a linear regression problem, e.g, via pseudo inverse method:

$$W_{out} = y\mathbf{Z}^T(\mathbf{Z}\mathbf{Z}^T + \gamma^2\mathbb{I})^{-1}$$
$$\mathbf{Z} = [\mathbf{x}; \mathbf{u}]$$

4. Inference:

$$\hat{y}(n) = g(W_{out}[\mathbf{x}[n]; \mathbf{u}[n+1]])$$



Best Practices for Applying Reservoir Computing

- **Input weights**
  - Randomly generated with uniform (or Gaussian) distribution
  - A single scaling value is used to scale all the columns

- **Reservoir size, L**
  - The bigger the reservoir, the easier it is to find a linear combination of the signals to approximate the desired output
  - It's lower bound: the number of independent real values that the reservoir must remember from the input to accomplish the task

- **Reservoir transfer function, *f(.)***
  - ESN: tanh, ReLU, Leaky integrator neurons
  - LSM: LIF, Izhikevich, Hodgkin-Huxley

Best Practices for Applying Reservoir Computing

- **RNN weights**
  - Generated sparse, with nonzero elements with either a symmetrical uniform, discrete bi-valued, or normal distribution centered around zero
  - Each neuron is typically connected to 10 other neurons to ensure sparsity
  - Once the matrix is generated, its **spectral radius** is normalized and scaled with a scaling factor < 1. (This is related to echo-state property)
- **Random noise**
  - Is added to transfer functions to produce diverse activities
- **Regularization parameter, γ**
  - is determined via cross-validation

A useful Python package to implement RC

- Echoes: https://fabridamicelli.github.io/echoes/

echoes Search GitHub 36 8

echoes

High level API for machine learning with Echo State Networks (ESN) - work in progress!.

Check out the examples for a quick start and What are Echo State Networks? section for a little intro about Echo State Networks.

The library is scikit-learn compatible, thus you can directly use sklearn utils, such as `GridSearchCV`. Moreover, it tries to stick to the intuitions that you might bring from using sklearn. For example, models can be initialized without passing arguments (but kwargs are enforced if you choose to pass any); attributes generated during fitting are stored with trailing underscore; etc.

Installation

You can install the package via pip

```
pip install echoes
```

Table of contents

Installation
Citing
Requirements
Dependencies
Tests

Reservoir Computing in Action: A Real-World Use Case

Cardiac Beat Classification (MIT-BIH arrhythmia dataset)

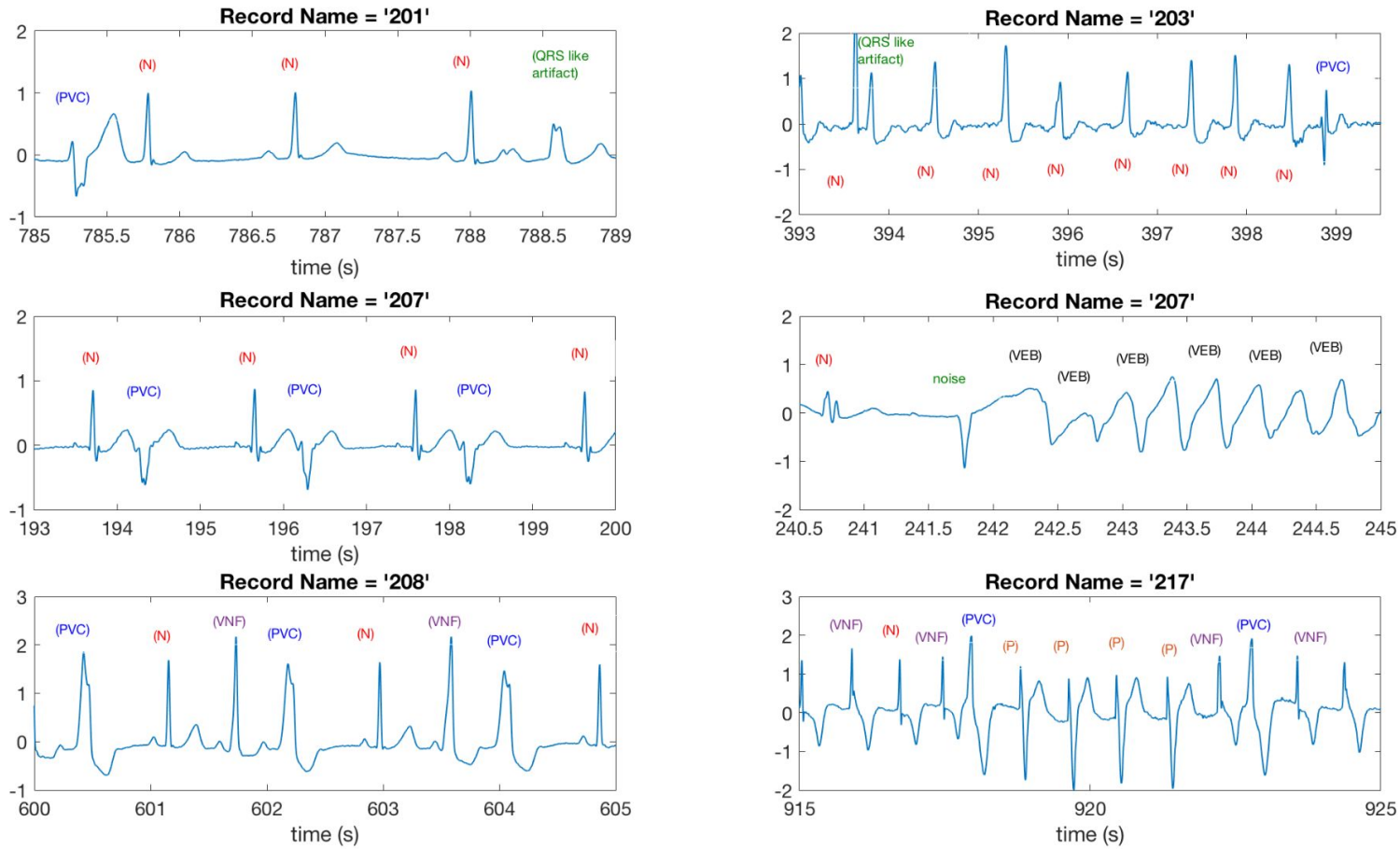
(N): Normal/Non ectopic

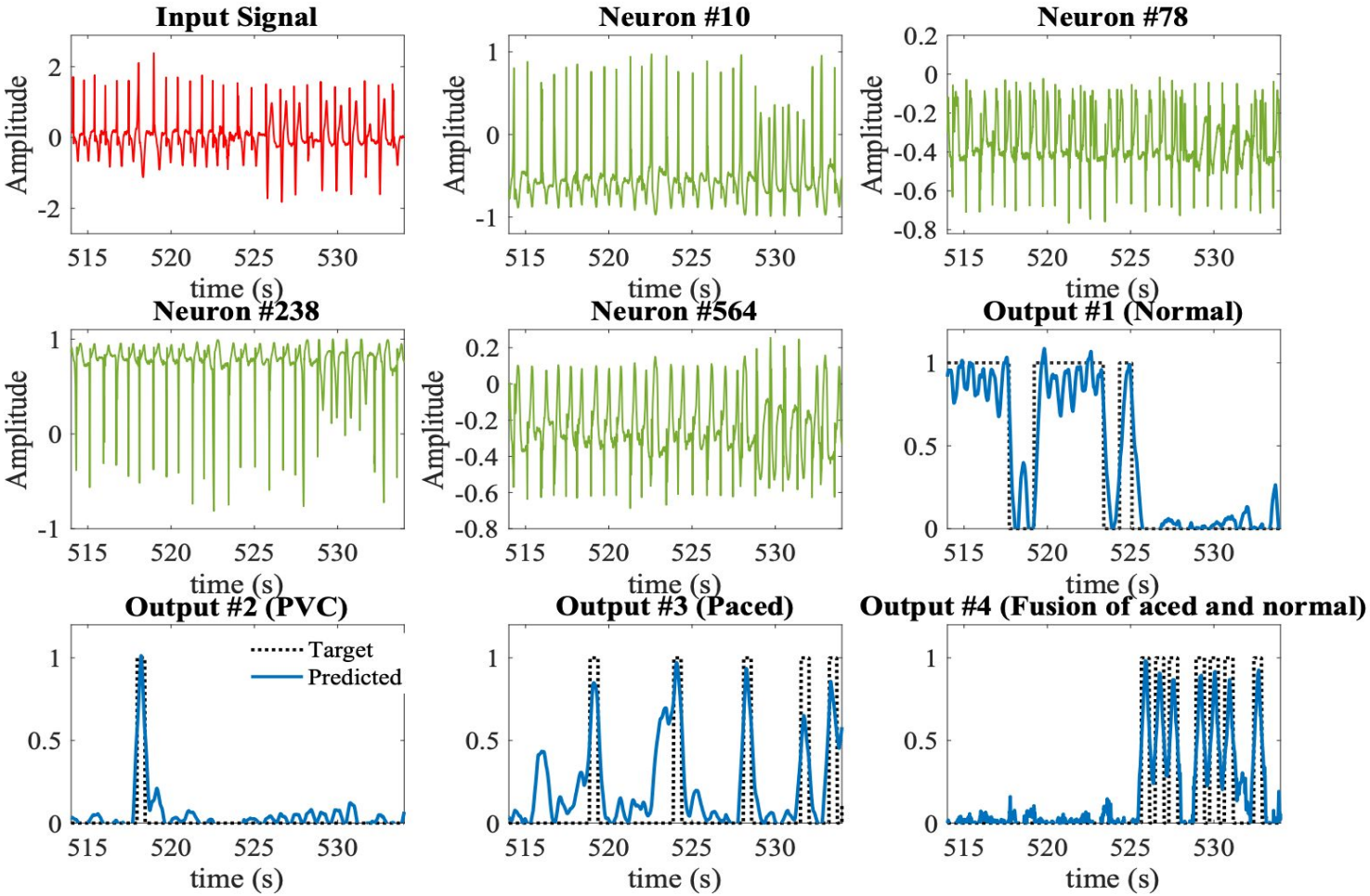
(PVC): Premature Ventricular Contraction

(VEB): Ventricular Escape Beat

(VNF): Fusion of Ventricular and Normal

(P): Paced Beats





Hadaeghi, Fatemeh. "Reservoir computing models for patient-adaptable ECG monitoring in wearable devices." *arXiv preprint arXiv:1907.09504* (2019).

Reservoir Computing in Action: A Real-World Use Case

Table 2: Patient-specific control parameters and performance evaluation of the reservoir computing model on MIT-BIH arrhythmia database.

Data #	**Pathological Classes**	**[Test time slot, #Beats, Regularization parameter, Ratio of scaling factors ($\frac{a}{s}$), τ]**	**[Acc Se P F1-score]**
'104'	Class I: Paced beats (# beats = 380) Class II: Fusion of paced and normal beats (# beats = 151)	[[22.5,30] min, 554, 1e-5, no weights, 0.91] [[22.5,30] min, 554, 1e-6, 3.0, 0.83]	Class I: [0.9928 0.9947 0.9947 0.9947] Class II: [0.9856 0.9934 0.9554 0.9740]
'106'	Class I: Premature ventricular contraction (# beats= 148)	[[17.5,25] min, 501, 0.001, 1.5, 0.52]	Class I: [0.9940 0.9932 0.9866 0.9899]
'119'	Class I: Premature ventricular contraction (# beats= 156)	[[17.5,25] min, 499, 1e-5, no weights, 1.0]	Class I: [0.9980 1.0000 0.9936 0.9968]
'200'	Class I: Premature ventricular contraction (# beats= 205)	[[12.5,20] min, 663, 0.01, no weights, 0.5]	Class I: [0.9910 0.9951 0.9761 0.9855]

Hadaeghi, Fatemeh. "Reservoir computing models for patient-adaptable ECG monitoring in wearable devices." *arXiv preprint arXiv:1907.09504* (2019).

Reservoir Computing: Challenges and Limitations

Challenges:

- Sensitive to hyperparameter tuning (spectral radius, input scaling).
- Reservoir dynamics can sometimes be hard to interpret.
- Scaling to large systems.

Limitations:

- Fixed reservoir: no adaptation during training.
- May underperform on tasks requiring long-term memory compared to LSTMs.

LSTM
(Long Short-Term Memory Networks)

Basic Idea

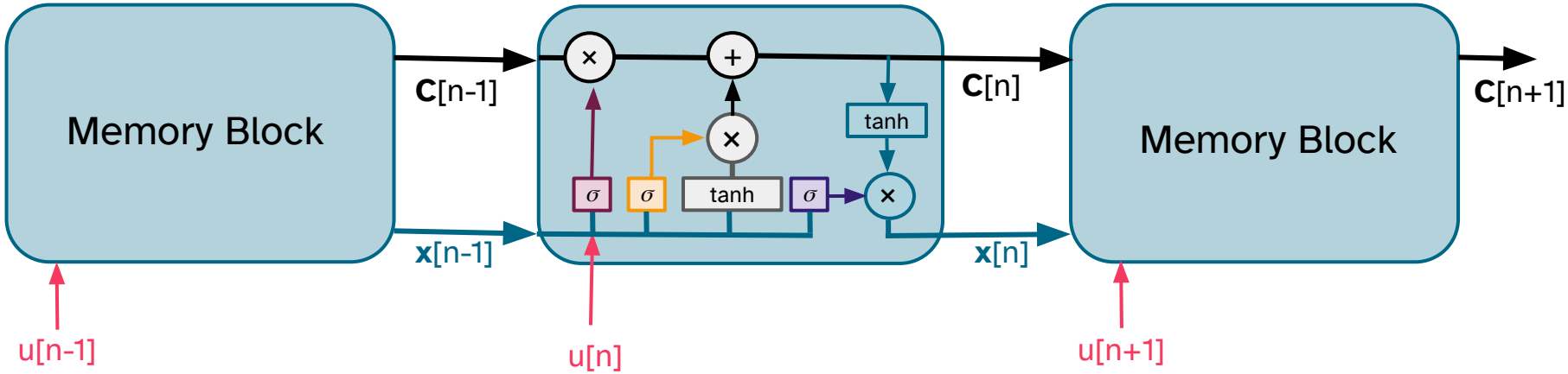
Traditional RNNs struggle with learning long-term dependencies due to vanishing or exploding gradients. **LSTM** introduces a memory cell to **preserve information** over time, allowing it to remember relevant data for longer periods.

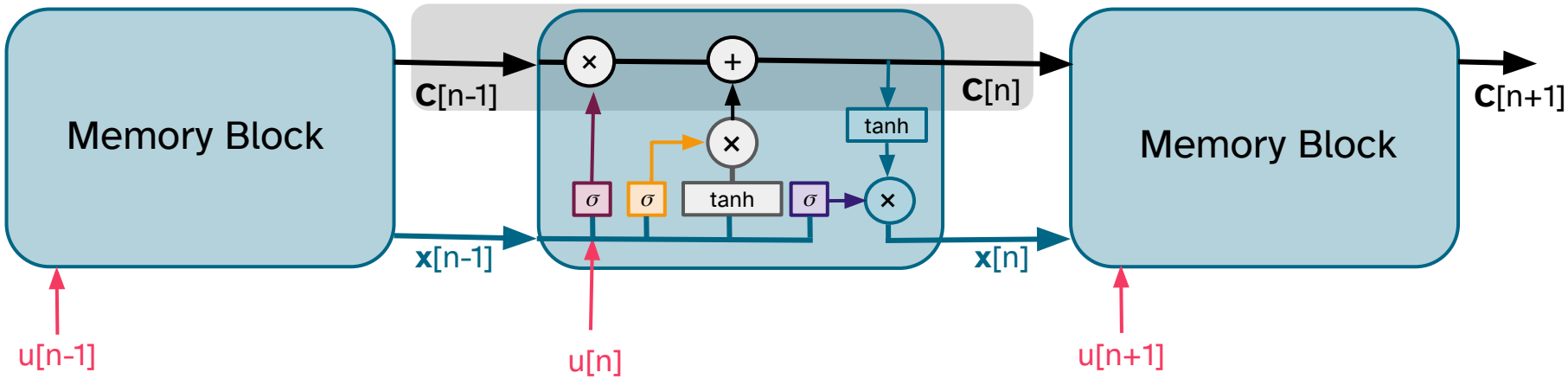
Communicated by Ronald Williams

Long Short-Term Memory

Sepp Hochreiter
Fakultät für Informatik, Technische Universität München, 80290 München, Germany

Jürgen Schmidhuber
IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland



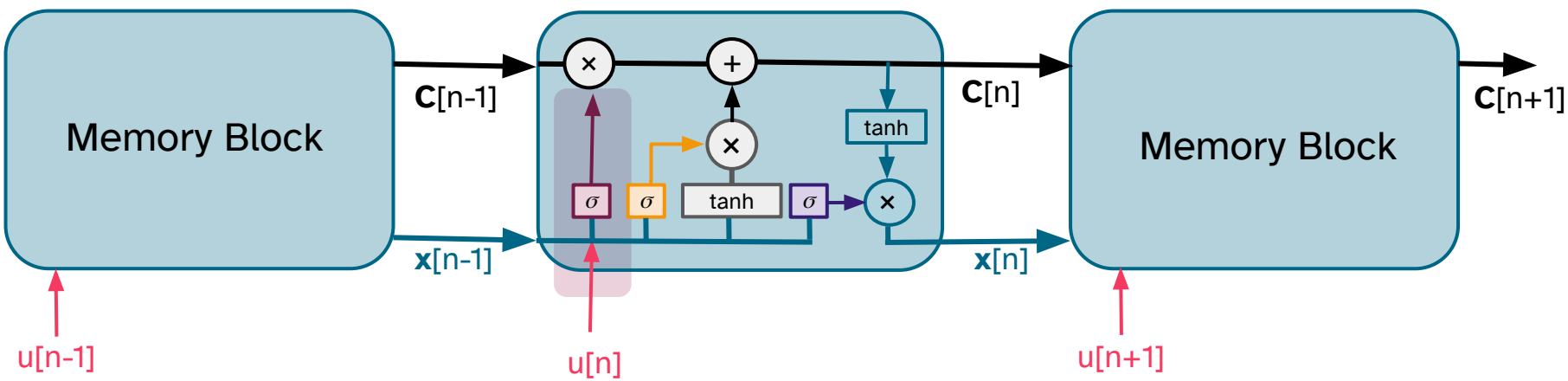


Cell State, C[n]:

- A "memory", represented by te the horizontal line in the diagram, that runs through the entire network, modified through carefully controlled gates.

Gates:

- **Forget Gate f[n]:**
  - Decides what information to discard from the cell state.
  - $f[n] = \sigma(W_f[\mathbf{x}[n-1], u[n]] + b_f)$



- **Input Gate i[n]:**
  - Determines which new information to add to the cell state
  - $i[n] = \sigma(W_i[\mathbf{x}[n-1], u[n]] + b_i)$

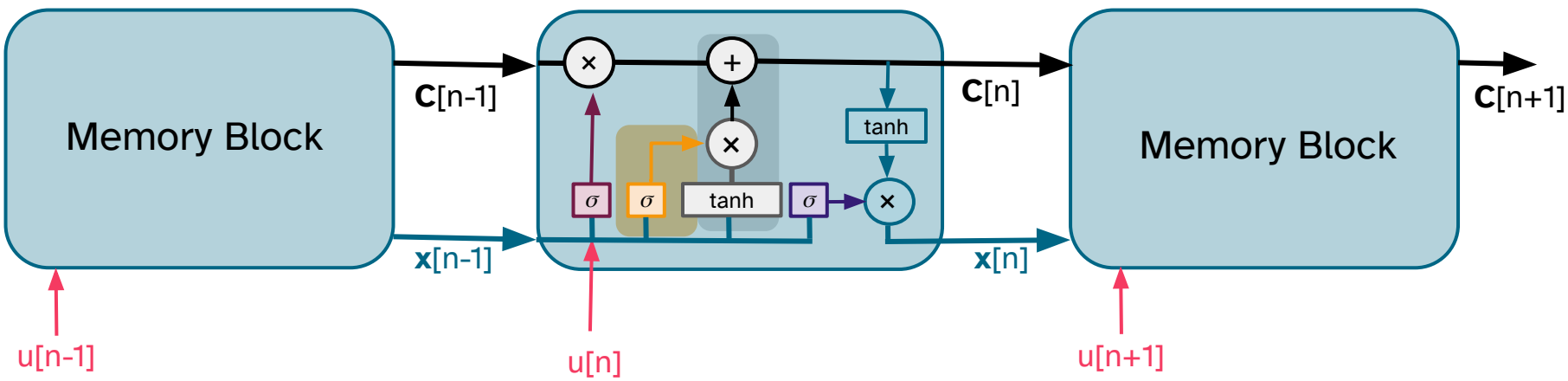
Cell State Update:

$$\tilde{C}[n] = \tanh(W_C[\mathbf{x}[n-1], u[n]] + b_C)$$

Cell State Equation:

The new cell state after applying the forget and input gates:

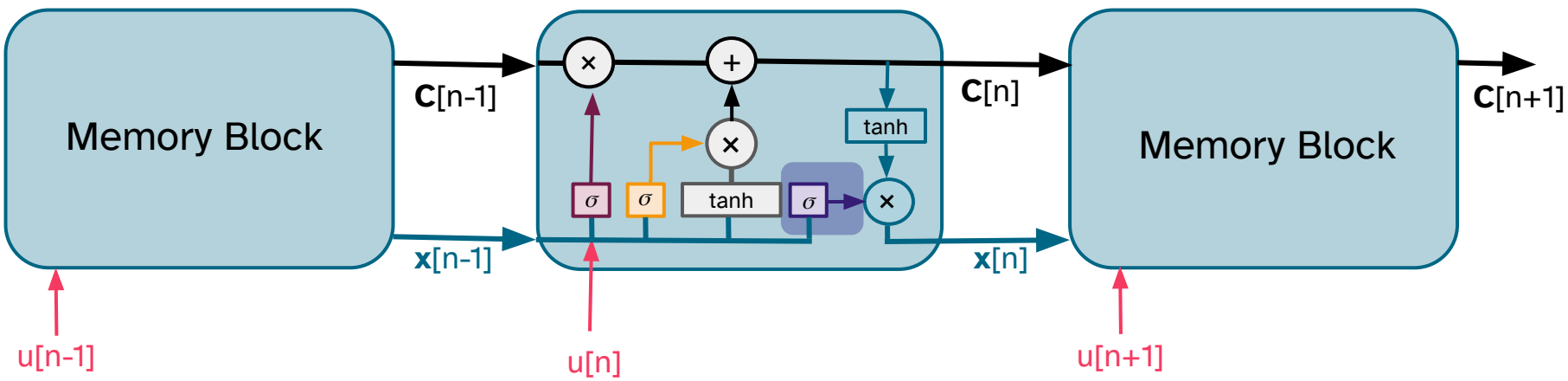
$$C[n] = f[n] \odot C[n-1] + i[n] \odot \tilde{C}[n]$$



- **Output Gate O[n]:**
  - Controls the final output from the memory cell.
  - $O[n] = \sigma(W_o[\mathbf{x}[n-1], u[n]] + b_o)$

Hidden State Update:

$$\mathbf{x}[n] = O[n] \odot \tanh(C[n])$$



Best Practices for Applying LSTMs

- Sequence length
  - **Short sequences**: Can be easier to train, but might lose long-term dependencies.
  - **Long sequences**: Capture more context but risk memory issues and slower training.

    Try **truncating or padding sequences** to a consistent length to balance complexity and performance.

- Optimizer
  - **Adam** is a popular choice for LSTMs due to its adaptive learning rate, which helps manage the learning dynamics better than standard SGD.
  - **Learning Rate**: Start with a small value and tune based on validation performance.

- Regularization techniques to manage overfitting
  - **Dropout**: Apply dropout between LSTM layers or to the input.
  - **Early Stopping**: Monitor the validation loss and stop training if it begins to increase.

Best Practices for Applying LSTMs

- Weight initialization
  - **Xavier** initialization is commonly used for LSTM weights to avoid vanishing/exploding gradients during backpropagation

- Handling exploding gradients
  - **Gradient Clipping**: Set a threshold for gradients, typically between 1-5, to prevent exploding gradients during training.

- Batch size
  - **Smaller batch sizes**: Can help capture sequence dynamics better but slow down training.
  - **Larger batch sizes**: Speed up training but may lead to less effective learning on complex time dependencies.

    Start with a batch size of **32 or 64** and tune based on performance.

LSTM: Challenges and Limitations

Challenges:

- Large number of hyperparameters (e.g., number of LSTM units, the learning rate, and the sequence length)
- Overfitting
- Long training times

Limitations:

- Interpretability
- High computational cost

Gated Recurrent Unit
GRU

Basic Idea

GRUs use fewer gates and parameters compared to LSTMs, making them computationally more efficient. Here, hidden units adaptively remember and forget.

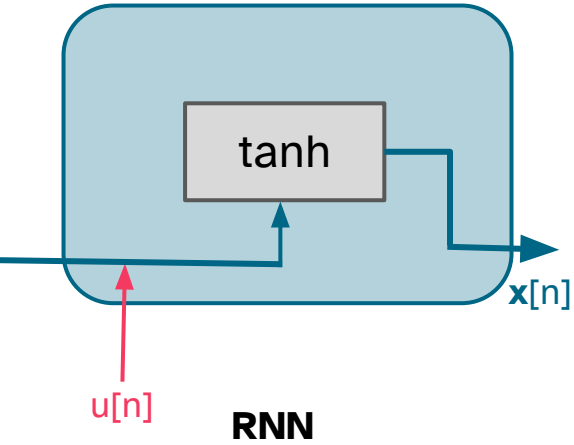
Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

Kyunghyun Cho
Bart van Merriënboer **Caglar Gulcehre**
Université de Montréal
firstname.lastname@umontreal.ca

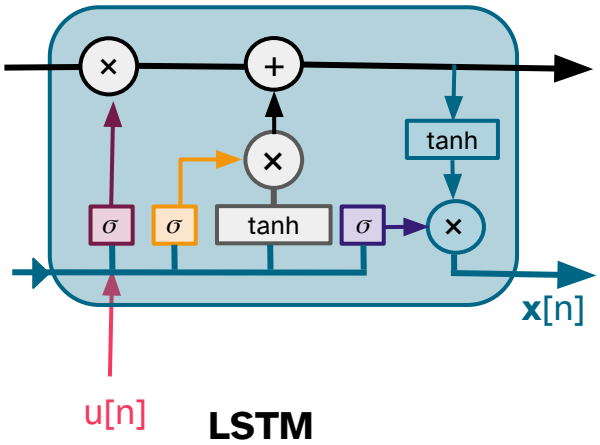
Dzmitry Bahdanau
Jacobs University, Germany
d.bahdanau@jacobs-university.de

Fethi Bougares **Holger Schwenk**
Université du Maine, France
firstname.lastname@lium.univ-lemans.fr

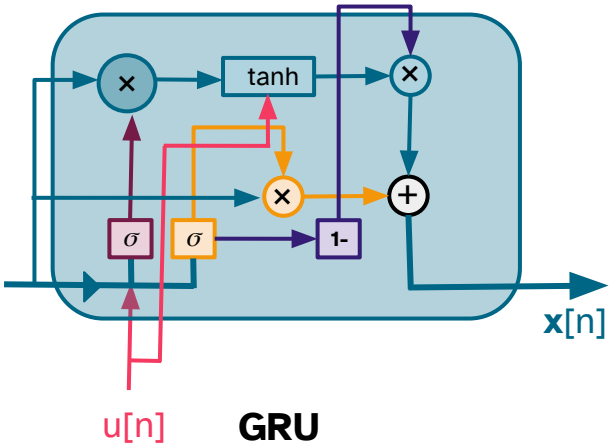
Yoshua Bengio
Université de Montréal, CIFAR Senior Fellow
find.me@on.the.web



RNN



LSTM



GRU

Update Gate z[n]:

- Controls how much of the past information needs to be passed to the future.
- $z[n] = \sigma(W_z[\mathbf{x}[n-1], u[n]] + b_z)$

Reset Gate r[n]:

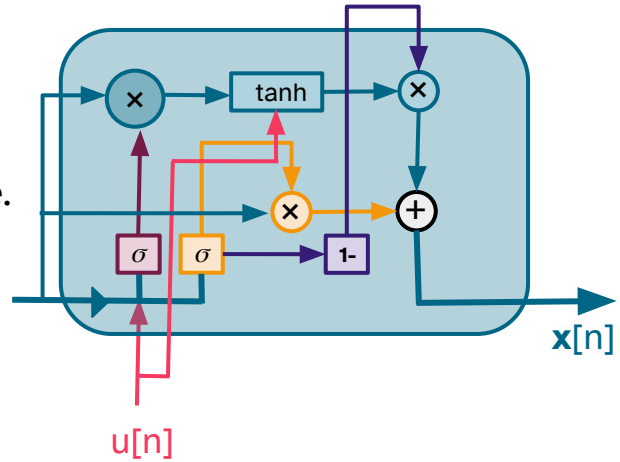
- Determines how much of the past information to forget or reset.
- $r[n] = \sigma(W_r[\mathbf{x}[n-1], u[n]] + b_r)$

Candidate Activation

- The potential new memory content.
- $\tilde{\mathbf{x}}[n] = \tanh(W[r[n] \odot \mathbf{x}[n-1], u[n]] + b)$

Final Memory x[n]:

- Combines past and present information using the update gate.
- $\mathbf{x}[n] = z[n] \odot \mathbf{x}[n-1] + (1 - z[n]) \odot \tilde{\mathbf{x}}[n]$



Best Practices for Applying GRUs

- Batch size
  - Use mini-batches and ensure consistent sequence lengths within a batch to speed up training.

- Regularization Techniques
  - **Dropout**: Apply dropout between GRU layers to prevent overfitting.
  - **Weight Decay**: Use L2 regularization for better generalization.

- Optimization
  - Use advanced optimizers like Adam or RMSProp for better gradient flow.
  - Consider gradient clipping to avoid exploding gradients.

GRU: Challenges and Limitations

Challenges:

- Sensitive to hyperparameter tuning
- GRUs can overfit when training on small datasets
- Difficulty capturing long-term dependencies

Limitations:

- High computational cost
- Challenge in parallelization

Recent Advances in RNN and State-Space Models

- Albert Gu and Tri Dao. **Mamba: Linear-time sequence modeling with selective state spaces**. arXiv preprint arXiv:2312.00752, 2024

- Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: **Reinventing rnns for the transformer era**. arXiv preprint arXiv:2305.13048, 2023.

- Leo Feng, Frederick Tung, Mohamed Osama Ahmed, Yoshua Bengio, Hossein Hajimirsadeghi. **Were RNNs All We Needed?** arXiv preprint arXiv:2410.01201, 2024