Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 10: Recurrent models

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Linearized transformer

The model

Up to now, we introduced two classes of models for dealing with sequences:

1. **Convolutional** models are parameter-efficient, they have linear complexity in the sequence length, but they are less efficient at handling *long-range* dependencies.
2. **Transformers** can handle long-range dependencies efficiently, but they require the addition of positional embeddings and they have a quadratic complexity in the sequence length. For autoregressive generation with a KV cache, the size of the cache grows linearly in the sequence length.

We would like an operator which combines the benefits of the two approaches - recently, **recurrent networks** (RNNs) have gained interest.

We begin with a generalization of the attention layer (**linearized attention layer**) that can be written in a recurrent form.

Consider again the SA layer with a generic scalar-valued attention function $\alpha(\cdot, \cdot)$ instead of the dot product:

$$\mathbf{h}_i = \frac{\sum_{j=1}^{n} \alpha\left(\mathbf{q}_i, \mathbf{k}_j\right) \mathbf{v}_j}{\sum_{j=1}^{n} \alpha\left(\mathbf{q}_i, \mathbf{k}_j\right)} \tag{1}$$

where for the standard SA, $\alpha(\mathbf{x}, \mathbf{y}) = \exp(\mathbf{x}^\top \mathbf{y})$.

Katharopoulos, A., Vyas, A., Pappas, N. and Fleuret, F., 2020. **Transformers are RNNs: Fast autoregressive transformers with linear attention**. In *ICML* (pp. 5156-5165). PMLR.

Any non-negative α is a valid similarity function. In machine learning, these are known as **kernel functions**. Most kernel functions (e.g., polynomial, Gaussian) can be written as a generalized dot product:

$$\alpha(\mathsf{x}, \mathsf{y}) = \phi(\mathsf{x})^\top \phi(\mathsf{y}) \tag{2}$$

for some function $\phi : \mathbb{R}^c \rightarrow \mathbb{R}^e$ performing a feature expansion. A **linearized attention** layer is obtained by using (2) in the attention layer.

Working out the calculation (see details in the book and in the paper, the key point is that $\phi(\mathbf{q}_i)$ does not depend on j):

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{n} \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{n} \phi(\mathbf{k}_j)} \tag{3}$$

This is the **linearized attention** layer. By itself, computing (3) for all tokens has complexity $\mathcal{O}(n(e^2 + ev))$, which is linear in the sequence length and advantageous whenever $n < e^2$. However, the usefulness of the layer becomes apparent if we focus on a **causal** variant.

Linearized transformer

A recurrent formulation

A causal variant is obtained by masking all elements such that $j > i$:

Attention memory $\mathbf{S}_i$

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{i} \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_i)^\top \sum_{j=1}^{i} \phi(\mathbf{k}_j)} = \frac{\phi(\mathbf{q}_i)^\top \mathbf{S}_i}{\phi(\mathbf{q}_i)^\top \mathbf{z}_i} \quad (4)$$

Normalizer memory $\mathbf{z}_i$

We can rewrite the two *memories* recursively as:

$$S_i = S_{i-1} + \phi(k_i)v_i^\top \tag{5}$$

$$z_i = z_{i-1} + \phi(k_i) \tag{6}$$

where the base case of the recurrence is given by their initialization:

$$S_0 = 0 \tag{7}$$

$$z_0 = 0 \tag{8}$$

These four equations together with the (4) provide the complete **recurrent** variant of the layer.

- Similarly to the implementation of batch normalization, this layer is **stateful**: $\mathbf{S}_i$ and $\mathbf{z}_i$ change for every token processed, and they are reinitialized at the end of the sequence.
- The update step (5)-(6) has constant-time complexity, similarly to a convolutional layer.
- Can we implement this layer in parallel?

We call any layer of this form a **recurrent** layer.

Recurrent layers

General formulation

To provide a general definition, let us abstract away some key properties:

1. First, we need a fixed-size **state** which encodes all useful information up to the current element in the sequence. We denote it generically as $\mathbf{s}_i$, and from now on we assume it is a vector.
2. Second, we need a **transition function** (recurrence) that updates the state vector based on the previous value and the value of the current token, which we denote as $f(\mathbf{s}_{i-1}, \mathbf{x}_i)$.
3. Third, we need a **readout function** to compute the output for the i-th element of the sequence. We denote it as $g(\mathbf{s}_i, \mathbf{x}_i)$.

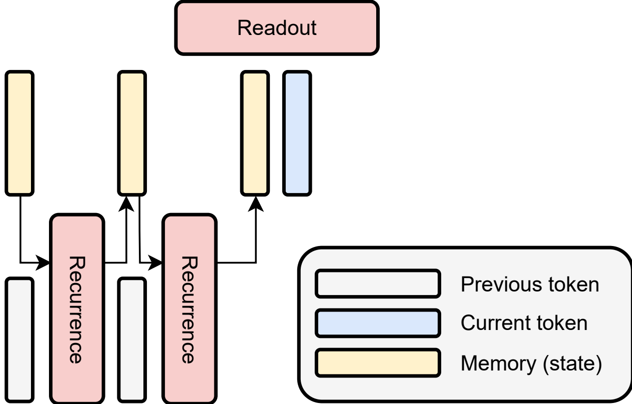

Figure 1: Overview of a recurrent layer: past tokens are shown in gray, current input token in blue, the memory state in yellow.

Based on the previous definition, given a sequence of tokens $\mathbf{x}_1, \mathbf{x}_2, \ldots$, a recurrent layer can be written as:

$$\mathbf{s}_i = f(\mathbf{s}_{i-1}, \mathbf{x}_i) \tag{9}$$

$$\mathbf{h}_i = g(\mathbf{s}_i, \mathbf{x}_i) \tag{10}$$

where $\mathbf{s}_0 = \mathbf{0}$. The size of the state vector, e, and the size of the output vector $\mathbf{h}_i \sim (o)$ are hyper-parameters. We call f the **state transition** function and g the **readout** function.

Recurrent neural networks (RNNs) are built by stacking multiple layers of this form.

- RNNs are fundamentally input-driven dynamical systems.
- They are causal layers by definition.
- In control engineering, they are known as **state-space models**.

For tasks in which causality is unnecessary, **bidirectional layers** can be defined. In a bidirectional layer we initialize two recurrent layers (with separate parameters), one of which processes the sequence left-to-right, and the second one right-to-left. Their output states are then concatenated to provide the final output.

Recurrent layers

Vanilla recurrent layers

Historically, recurrent layers were instantiated by considering two fully-connected layers as transition and readout functions:

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \phi(\mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i) \tag{11}$$

$$g(\mathbf{s}_i, \mathbf{x}_i) = \mathbf{C}\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{12}$$

The layer has four trainable matrices $\mathbf{A} \sim (e, e)$, $\mathbf{B} \sim (e, c)$, $\mathbf{C} \sim (o, e)$, and $\mathbf{D} \sim (o, c)$, where c is the input dimensionality (the size of each token). This is known generally as *the* recurrent layer, or vanilla recurrent layer, or Elman recurrent network.

This family of recurrent layers has multiple issues, e.g.:

- The computation cannot be paralellized and it must be executed with a for-loop operation. Even on recent GPUs with customized kernels, this is computationally expensive compared to other types of layers.
- The gradient computation requires to back-propagate through all transition functions (**backpropagation through time**, BPTT), which might create instabilities due to the non-linearity (see the full computation in the book).

As a reason, they fell out of favour compared to alternative attention-based models.

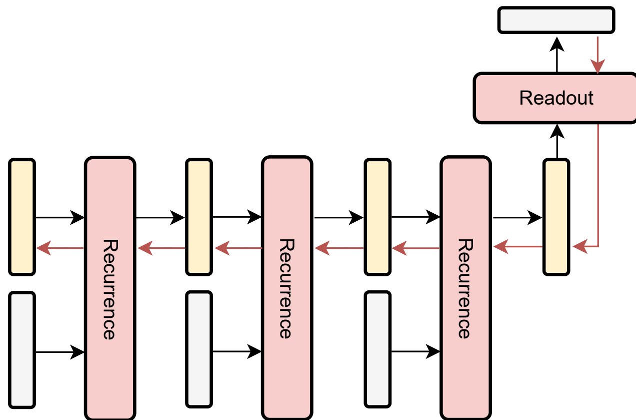

Figure 2: Backward pass for a recurrent layer: the adjoint values have to be propagated through all the transition steps. Each state then contributes a single term to the full gradient of the parameters.

Recurrent layers

Gated recurrent layers

One issue of RNNs is that the entire state gets overwritten at each transition. However, for many sequences, only a few elements of these transitions are important and we may prefer to sparsify the transition step. This can be controlled by the introduction of so-called **gate** elements.

We consider the simplest form of gated RNN, called **light gated recurrent unit** (Li-GRU), having a single gate.

Ravanelli, M., Brakel, P., Omologo, M. and Bengio, Y., 2018. **Light gated recurrent units for speech recognition**. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2), pp. 92-102.

A gating function is a layer that outputs values in the range $[0, 1]$ that can be used to "mask" the input. As an example, a gate over the state can be obtained by a fully-connected layer with a sigmoid activation function:

$$\gamma(\mathbf{s}_{i-1}, \mathbf{x}_i) = \sigma\left(\mathbf{V}\mathbf{s}_{i-1} + \mathbf{U}\mathbf{x}_i\right)$$

If $\gamma_i \approx 0$, the i-th feature of the state should be kept untouched, while if $\gamma_1 \approx 1$, we should propagate its updated value as output.

We can obtain a gated layer by combining a gate with the transition function:

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \overbrace{\gamma(\mathbf{s}_{i-1}, \mathbf{x}_i) \odot (\mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i)}^{\text{New values}} + \underbrace{(1 - \gamma(\mathbf{s}_{i-1}, \mathbf{x}_i)) \odot \mathbf{s}_{i-1}}_{\text{Old values}}$$

This can be seen as a soft (differentiable) approximation to a "real" gate having only binary values, or as a convex combination of the original layer and a skip connection.

Other famous models in this category include the **gated recurrent unit** (GRU) with two gates and the **long-short term memory** (LSTM) network with three gates.

LSTMs were the first gated variant to be introduced in the literature, and for a long time they have been the most successful deep architecture for processing sequences. Research on LSTM models is still very active.

Beck, M., Pöppel, K., Spanring, M., Auer, A., Prudnikova, O., Kopp, M., Klambauer, G., Brandstetter, J. and Hochreiter, S., 2024. **xLSTM: Extended Long Short-Term Memory**. *arXiv preprint arXiv:2405.04517*.

Structured SSMs

A linear transition function

Suppose we remove the nonlinearity in the transition function:

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i \tag{13}$$

$$g(\mathbf{s}_i, \mathbf{x}_i) = \mathbf{C}\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{14}$$

We call this a **state space model** (improperly, as every RNN is an SSM) or better a *structured* SSM, as we will see that the transition matrix needs to be properly constrained to make this work. An SSM layer is "less expressive" than a vanilla layer, but this can be recovered by adding activation functions after the output, or by interleaving these layers with token-wise MLPs.

Orvieto, A., De, S., Gulcehre, C., Pascanu, R. and Smith, S.L., 2023. **On the universality of linear recurrences followed by nonlinear projections**. *arXiv preprint arXiv:2307.11888*.

Interest in structured SSMs models started in 2020 with the introduction of the HiPPO (**High-Order Polynomial Projection Operator**) layer, a theoretical construction for $\mathbf{A}$ to compress one-dimensional input sequences according to some pre-defined reconstruction criterion.

A family of neural networks built by a stack of SSM layers based on the HiPPO theory followed, leading to the **Structured State Space for Sequence Modeling** (S4) layer in 2021 and the simplified S4 model (S5) in 2022.

We focus our analysis on a simplified variant known as the **linear recurrent unit** (LRU).

Orvieto, A., et al., 2023. **Resurrecting recurrent neural networks for long sequences.** In *ICML* (pp. 26670-26698). PMLR.

Because of linearity, the recurrence has a closed form solution:

$$\mathbf{s}_i = \sum_{j=1}^{i} \mathbf{A}^{i-j}\mathbf{B}\mathbf{x}_j \tag{15}$$

We can exploit this equation in two ways: as a convolution, or as a parallel scan. First, let us aggregate all coefficients with respect to the input sequence into a rank-3 tensor:

$$K = \text{stack}\left(\mathbf{A}^{n-1}\mathbf{B}, \mathbf{A}^{n-2}\mathbf{B}, \ldots, \mathbf{A}\mathbf{B}, \mathbf{B}\right)$$

We can compute all outputs via a single 1D convolution of filter size equal to the length of the sequence (a *long* convolution) between the input sequence stacked into a single matrix $\mathsf{X} \sim (n, c)$ and K:

$$\mathsf{S} = \mathrm{Conv1D}(\mathsf{X}, K)$$

Hence, the SSM layer can be interpreted as a convolution. This was exploited in the original structured SSM models by working in a frequency domain where convolution is a multiplication.

Gu, A., et al., 2021. **Combining recurrent, convolutional, and continuous-time models with linear state space layers**. *Advances in neural information processing systems*, 34, pp. 572-585.

Structured SSMs

Associative scans

Consider a sequence of elements $(x_1, x_2, \ldots, x_n)$, and a binary associative operation $\star$. We want to compute all partial applications:

$$x_1, \quad x_1 \star x_2, \quad x_1 \star x_2 \star x_3, \quad \ldots, \quad x_1 \star x_2 \star \cdots \star x_n$$

This can be done trivially by an iterative algorithm which computes the elements one-by-one, adding one element at every iteration. However, we can devise an efficient *parallel* algorithm by exploiting the associativity of the operator.

The key intuition is that multiple pairs of elements can be computed in parallel and then aggregated recursively.

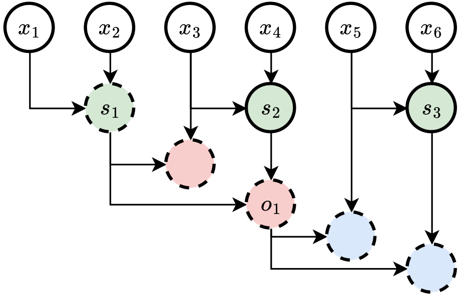

Figure 3: Parallel scan on a sequence of six elements: circles of the same color can be computed in parallel; dashed circles are the outputs of the parallel scan.

Consider a sequence of 6 elements $x_1, x_2, x_3, x_4, x_5, x_6$. We will denote by $\hat{x}_i$ the i-th prefix we want to compute. We first aggregate pairs of adjacent values as:

$$
\begin{aligned}
s_1 &= x_1 \star x_2 \rightarrow \hat{x}_2 \\
s_2 &= x_3 \star x_4 \\
s_3 &= x_5 \star x_6
\end{aligned}
$$

where we use arrows to denote outputs of the algorithm.

We now perform a second level of aggregations:

$$s_1 \star x_3 \rightarrow \hat{x}_3$$
$$o_1 = s_1 \star s_2 \rightarrow \hat{x}_4$$

And finally:

$$o_1 \star x_5 \rightarrow \hat{x}_5$$
$$o_1 \star s_3 \rightarrow \hat{x}_6$$

While this looks strange (we made 7 steps instead of 5), the three blocks of computations can be trivially parallelized if we have access to 3 separate threads. By organizing the set of computations in a balanced fashion, we can compute the parallel scan in $\mathcal{O}(T \log n)$, where T is the cost of the binary operator $\star$.

An example of implementation is the associative scan function in JAX.

https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.associative_scan.html

The transition function in a linear SSM is an example of an all-prefix-sums problem. We define the elements of our sequence as pairs $x_i = (\mathsf{A}, \mathsf{B}\mathsf{x}_i)$, and the binary operator as:

$$(\mathsf{Z}, \mathsf{z}) \star (\mathsf{V}, \mathsf{v}) = (\mathsf{VZ}, \mathsf{Vz} + \mathsf{v})$$

The prefixes of $\star$ are then given by:

$$x_1 \star x_2 \star \ldots \star x_i = (\mathsf{A}^i, \mathsf{s}_i)$$

Smith, J.T., Warrington, A. and Linderman, S.W., 2022. **Simplified state space layers for sequence modeling**. *arXiv preprint arXiv:2208.04933*.

Hence, running a parallel scan gives us the powers of A as the first elements of the output, and all the states of the layer as the second element of the output. The complexity of this operation is upper bounded by the complexity of $\mathsf{A}^{i-1}\mathsf{A}$, which scales as $\mathcal{O}(n^3)$. To make the entire procedure viable, we need to constrain A so that its powers can be computed more efficiently.

Structured SSMs

Diagonal SSMs

A common strategy to make the previous ideas feasible is to work with diagonal transition matrices. In this case, powers of A can be computed easily by taking powers of the diagonal entries in linear time.

In particular, a square matrix A is said to be **diagonalizable** if we can find another square (invertible) matrix P and a diagonal matrix Λ such that:

$$\mathsf{A} = \mathsf{P}\Lambda\mathsf{P}^{-1} \tag{16}$$

If a matrix can be diagonalized, its powers can be computed efficiently as:

$$\mathsf{A}^i = \mathsf{P}\Lambda^i\mathsf{P}^{-1}$$

Suppose that the transition matrix is diagonalizable. We can re-write the SSM in an equivalent form having a diagonal transition matrix. We first substitute the diagonalization and multiply on both sides by $\mathbf{P}^{-1}$:

$$\mathbf{P}^{-1}\mathbf{s}_i = \sum_{j=1}^{i} \Lambda^{i-j} \mathbf{P}\mathbf{B} \mathbf{x}_j$$

New state vector $\bar{\mathbf{s}}_i$

New input-state matrix $\bar{\mathbf{B}}$

Then we rewrite the readout function in terms of the new variable $\bar{\mathbf{s}}$:

New readout matrix $\bar{\mathbf{C}}$

$$\mathbf{y}_i = \mathbf{C}\mathbf{P}\,\bar{\mathbf{s}}_i + \mathbf{D}\mathbf{x}_i$$

Putting everything together:

$$\bar{\mathsf{s}}_i = \Lambda \bar{\mathsf{s}}_{i-1} + \bar{\mathsf{B}}\mathsf{x}_i \tag{17}$$

$$\mathsf{y}_i = \bar{\mathsf{C}}\bar{\mathsf{s}}_i + \mathsf{D}\mathsf{x}_i \tag{18}$$

Hence, whenever a diagonalization of A exists, we can always rewrite the SSM into an equivalent form having a diagonal transition matrix. In this case, we can directly train the four matrices $\Lambda = \text{diag}(\lambda)$, $\lambda \sim (e)$, $\bar{\mathsf{B}} \sim (e, c)$, $\bar{\mathsf{C}} \sim (o, e)$ and $\mathsf{D} \sim (o, c)$, with the diagonal matrix being parameterized by a single vector of dimension e.

Not all matrices can be diagonalized. However, an approximate diagonalization can always be found if one allows for matrices P and Λ to have complex-valued entries. Care must be taken to parameterize the values over the diagonal so that the eigenvalues of the transition matrix stay < 1 in absolute value, to avoid diverging dynamics. We do not cover these topics for brevity.

Orvieto, A., et al., 2023. **Resurrecting recurrent neural networks for long sequences**. In *ICML* (pp. 26670-26698). PMLR.

Structured SSMs

Selective SSMs

Despite their differences, all the models we saw are connected. For example, consider a linearized attention layer where we ignore the denominator:

$$\mathbf{S}_i = \mathbf{S}_{i-1} + \phi(\mathbf{k}_i)\mathbf{v}_i^\top \tag{19}$$

$$\mathbf{h}_i = \phi(\mathbf{q}_i)^\top \mathbf{S}_i \tag{20}$$

We now see that this has the form of a SSM layer, except that some matrices (e.g., $\mathbf{C} = \phi(\mathbf{q}_i)^\top$) are not fixed but they depend on the specific input token. This has inspired another class of SSM layers whose matrices are not constrained to be time-invariant, which have been called **selective** SSMs.

We consider the Mamba layer, which introduced the idea of selective SSMs. The core idea is to make some of the matrices of the SSM token-dependent:

$$\mathbf{s}_i = A(\mathbf{x}_i)\mathbf{s}_{i-1} + B(\mathbf{x}_i)\mathbf{x}_i \tag{21}$$

$$\mathbf{h}_i = C(\mathbf{x}_i)\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{22}$$

where $A(\bullet)$, $B(\bullet)$, and $C(\bullet)$ are linear projections of their input tokens. To make this feasible, the layer is applied to each channel of the input independently, and the transition matrix is selected as diagonal, so that all matrices of the SSM can be represented with a single vector of values.

Gu, A. and Dao, T., 2023. **Mamba: Linear-time sequence modeling with selective state spaces**. *arXiv preprint arXiv:2312.00752*.

This layer looses a simple parallel scan implementation and requires a customized hardware-aware implementation.

To make the overall architecture simpler, Mamba avoids alternating MLPs and SSMs, in favour of a gated archicture where an MLP is used to weight the outputs from the SSM. It also combines the model with the idea of multiple projections from transformers, and an additional depthwise convolution is added for improved flexibility.

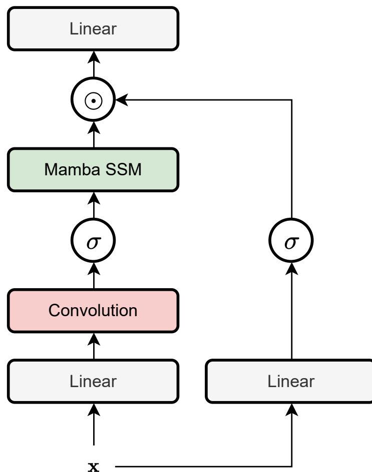

Figure 4: Mamba block (residual connections around the block and normalization are not shown). σ is the sigmoid function.

- **Chapter 13** of the book.