Lecture 5

Recurrent Networks and Graph Networks

Part 1: Recurrent Neural Networks

Machine Learning for Structured Data

Vlad Niculae · LTL, UvA · https://vene.ro/mlsd

1. **Recurrent Neural Networks**
    - Gated RNN
    - Bidirectional RNN
    - Multi-layer RNN
2. **Graph Neural Networks**
    - RNN vs GNN
    - Permutation equivariance
    - GNN Variants

Last time: convolutions for sequences and grids

We saw how to encode

- variable-length sequences
- variable-size grids (images)

using layers of **convolutions** with small, learned sliding filters.

First layers capture local phenomena (ngrams, edges).

Deeper layers get increasingly “global” by combining lower level features.

Today: a completely different approach to handle variable-length data.

Recurrent Neural Networks (RNN)

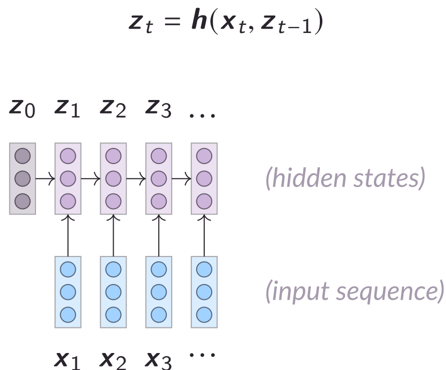
Remember: recurrence is when something invokes itself (e.g., a function calls itself).

Example:

$$\begin{aligned} & \text{sum}([a_1, a_2, a_3, \ldots, a_n]) \\ = a_1 + & \text{sum}([a_2, a_3, \ldots, a_n]) \\ = a_1 + a_2 + & \text{sum}([a_3, \ldots, a_n]) \\ = & \ldots \end{aligned}$$

Recurrent Neural Networks (RNN)

Recurrently encoding a sequence of input vectors $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \to (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$:



The simplest RNN is the Elman RNN:

$$\boldsymbol{z}_t = \phi\left(\underbrace{\boldsymbol{W}\boldsymbol{x}_t}_{\text{lin. func. of inputs}} + \underbrace{\boldsymbol{U}\boldsymbol{z}_{t-1}}_{\text{lin. func. of prev. state}} + \boldsymbol{b}\right)$$

Each hidden state depends on the previous ones. Therefore, cannot parallelize, must compute in order $\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots$

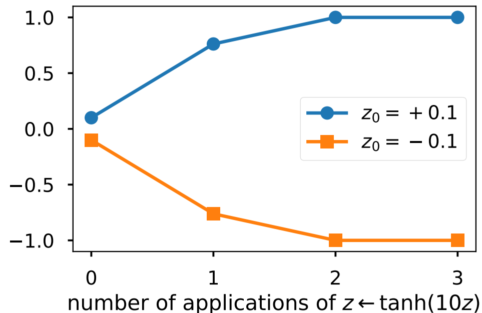
The initial state $\boldsymbol{z}_0$ is a fixed parameter.

The final state $\boldsymbol{z}_n$ has seen the entire sequence.

Gated RNN

Unless sequences are very short,
the simple RNN can get very unstable.

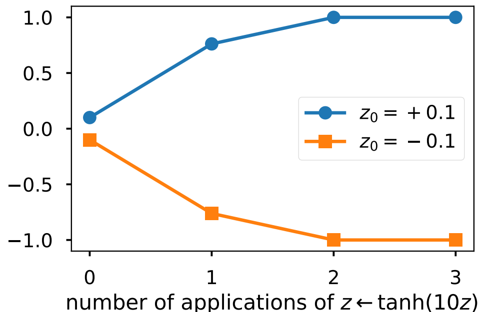
Assume $d = 1$ (single scalars) and all $x_t = 0$,
then $z_t = \phi(wz_{t-1})$.



Gated RNN

Unless sequences are very short,
the simple RNN can get very unstable.

Assume $d = 1$ (single scalars) and all $x_t = 0$,
then $z_t = \phi(w z_{t-1})$.



Idea: only update the state sometimes:

$$z_t = \begin{cases} h(x_t, z_{t-1}), & \text{under some condition} \\ z_{t-1}, & \text{otherwise} \end{cases}$$

A gated RNN is (basically) an RNN which also predicts, at each step, whether to update its state or not.

Hard and soft gating

Say we want to compute z as a choice between two numbers:

$$z = \begin{cases} c, & \text{if } g = 0 \\ d, & \text{if } g = 1 \end{cases}$$

where g serves as a “gate” that depends on some condition.

Hard and soft gating

Say we want to compute z as a choice between two numbers:

$$z = \begin{cases} c, & \text{if } g = 0 \\ d, & \text{if } g = 1 \end{cases}$$

where g serves as a "gate" that depends on some condition.

We can write this using multiplication and addition:

$$\begin{aligned} z = (1 - g) \cdot c \\ + g \cdot d \end{aligned}$$

Hard and soft gating

Say we want to compute z as a choice between two numbers:

$$z = \begin{cases} c, & \text{if } g = 0 \\ d, & \text{if } g = 1 \end{cases}$$

where g serves as a "gate" that depends on some condition.

We can write this using multiplication and addition:

$$\begin{aligned} z = (1 - g) \cdot c \\ + g \cdot d \end{aligned}$$

Can this expression handle soft gating $0 < g < 1$? Consider $g = 0.5$.

Hard and soft gating

Say we want to compute z as a choice between two numbers:

$$z = \begin{cases} c, & \text{if } g = 0 \\ d, & \text{if } g = 1 \end{cases}$$

where g serves as a "gate" that depends on some condition.

We can write this using multiplication and addition:

$$\begin{aligned} z = (1 - g) \cdot c \\ + g \cdot d \end{aligned}$$

Can this expression handle soft gating $0 < g < 1$? Consider $g = 0.5$.

Yes: we get a combination between the two choices. $g = 0.5 \rightarrow$ take the average.

The Gated Recurrent Unit (GRU)

The GRU uses two gates:

- the *update* gate $\boldsymbol{g}_t$
- the *reset* gate $\boldsymbol{r}_t$

The GRU main recurrence updates the hidden state as[1].

$$\begin{aligned} \boldsymbol{z}_t = (1 - \boldsymbol{g}_t) \odot \tilde{\boldsymbol{z}}_t \\ + \ \boldsymbol{g}_t \odot \boldsymbol{z}_{t-1} \end{aligned}$$

where $\odot$ is elementwise multiplication; $\tilde{\boldsymbol{z}}_t$ is the candidate for new hidden state.

[1] Notation: elementwise product $[\boldsymbol{u} \odot \boldsymbol{v}]_i = u_i v_i$

The Gated Recurrent Unit (GRU)

The GRU uses two gates:

- the *update* gate $\boldsymbol{g}_t = \sigma\left(\boldsymbol{W}_g \boldsymbol{x}_t + \boldsymbol{U}_g \boldsymbol{z}_{t-1} + \boldsymbol{b}_g\right)$
- the *reset* gate $\boldsymbol{r}_t = \sigma\left(\boldsymbol{W}_r \boldsymbol{x}_t + \boldsymbol{U}_r \boldsymbol{z}_{t-1} + \boldsymbol{b}_r\right)$

The GRU main recurrence updates the hidden state as[1].

$$\begin{aligned}\boldsymbol{z}_t = & \,(1 - \boldsymbol{g}_t) \odot \tilde{\boldsymbol{z}}_t \\ & + \boldsymbol{g}_t \odot \boldsymbol{z}_{t-1}\end{aligned}$$

where $\odot$ is elementwise multiplication; $\tilde{\boldsymbol{z}}_t$ is the candidate for new hidden state.

The candidate state is computed as

$$\tilde{\boldsymbol{z}}_t = \tanh\left(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}(\boldsymbol{r}_t \odot \boldsymbol{z}_{t-1}) + \boldsymbol{b}\right)$$

[1] Notation: elementwise product $[\boldsymbol{u} \odot \boldsymbol{v}]_i = u_i v_i$

LSTM and other RNNs

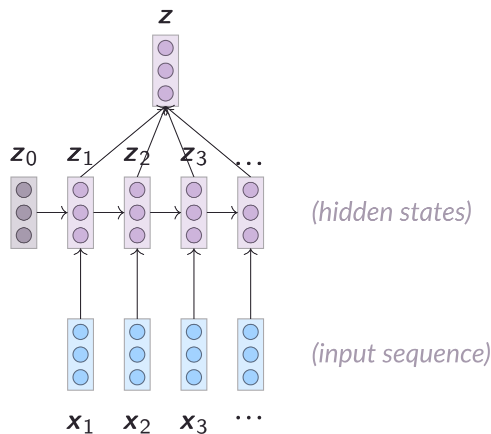
There are other kinds of gated RNNs.

A famous one is the "long-short-term memory" or LSTM – we won't cover its internal details; the intuitions are similar.

There are some important subtle differences that are still being studied; e.g., formally LSTM can learn to "count" while GRU cannot.[2]

[2] Weiss et al (ACL 2018) "On the practical computational power of finite precision RNNs for language recognition"

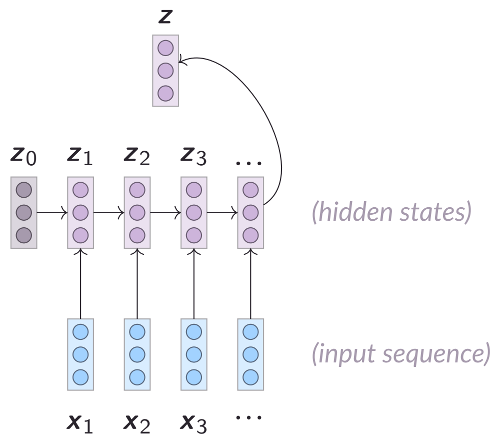
Pooling an RNN output



How to extract a single vector to represent the entire sequence?

- Pool (e.g., max-pool) all states.

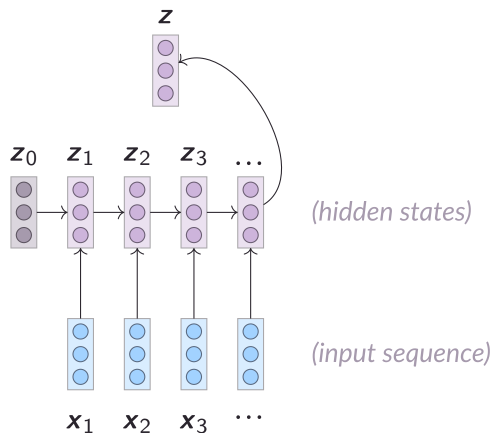
Pooling an RNN output



How to extract a single vector to represent the entire sequence?

- Pool (e.g., max-pool) all states.
- Take the last state, $z = z_n$.
  - may have "recency bias" in favor of the end of the sequence

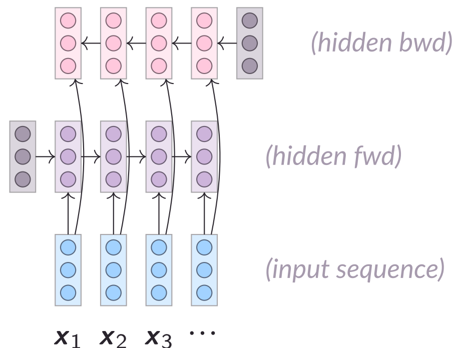
Pooling an RNN output



How to extract a single vector to represent the entire sequence?

- Pool (e.g., max-pool) all states.
- Take the last state, $z = z_n$.
  - may have “recency bias” in favor of the end of the sequence
  - Mitigate this by going in both directions?

Bidirectional RNN

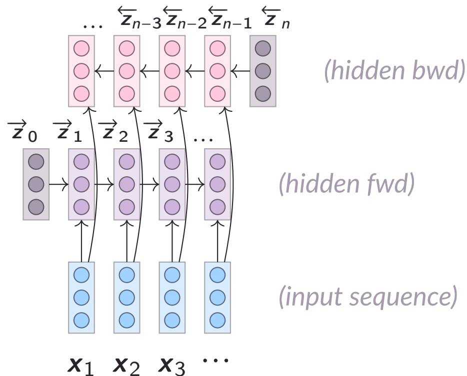


Same as two RNNs: one from left to right, one from right to left.

Concatenate $\boldsymbol{z}_t = [\overrightarrow{\boldsymbol{z}}_t, \overleftarrow{\boldsymbol{z}}_t]$ to get a representation of word t.

Concatenate $\boldsymbol{z} = [\overrightarrow{\boldsymbol{z}}_n, \overleftarrow{\boldsymbol{z}}_1]$ to get a representation of entire sequence.

Bidirectional RNN

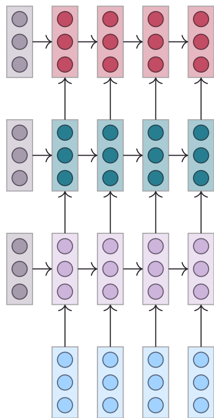


Same as two RNNs: one from left to right, one from right to left.

Concatenate $\boldsymbol{z}_t = [\overrightarrow{\boldsymbol{z}}_t, \overleftarrow{\boldsymbol{z}}_t]$ to get a representation of word t.

Concatenate $\boldsymbol{z} = [\overrightarrow{\boldsymbol{z}}_n, \overleftarrow{\boldsymbol{z}}_1]$ to get a representation of entire sequence.

Multi-layer RNN



Since the RNN outputs an sequence of hidden states, we can feed these states as inputs to another RNN.

And so on.

Unlike for CNN, multiple layers don't pass information further: a single layer is enough to aggregate the entire sequence.

But it can work better in practice.

Machine Learning for Structured Data

Vlad Niculae · LTL, UvA · https://vene.ro/mlsd

Recurrent Networks and Graph Networks

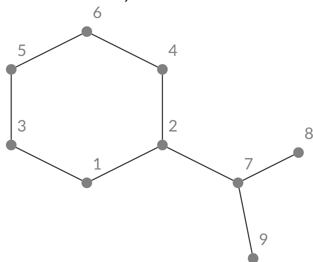
1. Recurrent Neural Networks
   - Gated RNN
   - Bidirectional RNN
   - Multi-layer RNN
2. **Graph Neural Networks**
   - RNN vs GNN
   - Permutation equivariance
   - GNN Variants

Encoding general graphs

Graph-structured data: proteins, molecules, social networks, etc.

A graph $\mathcal{G} = (V, E)$:

- $V = \{1, \ldots, n\}$ is the set of nodes.
- $E \subseteq V \times V$ are the edges, e.g., $(u, v) \in E$ means an edge from u to v
- Directed vs undirected graphs: in a nutshell, undirected means $(u, v) \in E \iff (v, u) \in E$.
- the adjacenty matrix $\boldsymbol{A} \in \{0, 1\}^{n \times n}$ encodes the set of edges E: $a_{uv} = 1 \iff (u, v) \in E$.



Each node can have a *type* (e.g., carbon, hydrogen, ...).

For simplicity, we assume all edges are of the same type.

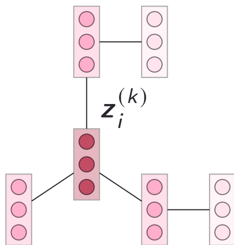
Graph datasets

Two main scenarios, but the tools we use are the same

1. Each data point $\boldsymbol{x}^{(i)}$ is a graph.
   - e.g., molecule solubility, malicious software detection, protein classification, …
   - can be given as a sequence of node labels $(x_1^{(i)}, \ldots, x_{n_i}^{(i)})$ and an adjacency matrix $\boldsymbol{A}^{(i)}$
   - this is what you have in assignment 1
2. Data points are parts of one big graph.
   - e.g., node classification (classifying bots on twitter), link prediction (instagram follow suggestions), community detection, …
   - much harder to set up experiments, dev set/test set, etc.

Node representations with graph neural nets

Encoding a **graph** of input vectors $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \to (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$:



- We apply an iterative process.
- At iteration 0, $\boldsymbol{z}_i^{(0)} = \boldsymbol{x}_i$ (the input embedding)
- At each iteration, a node's embedding is updated as a function of the embeddings of its neighbors, i.e., message passing along the edges:

$$\boldsymbol{m}_i^{(k)} = \sum_{j \in N(i)} \boldsymbol{z}_j^{(k)}$$

$$\boldsymbol{z}_i^{(k+1)} = \phi\left(\boldsymbol{W}_{\text{self}} \boldsymbol{z}_i^{(k)} + \boldsymbol{W}_{\text{neigh}} \boldsymbol{m}_i^{(k)} + \boldsymbol{b}\right)$$

- Apply this update in parallel for every node, then repeat.

Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_j$$

Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_j$$

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
(n =num. nodes, d =embedding dimension).

The ith column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node i.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} =$$

Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_j$$

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
(n =num. nodes, d =embedding dimension).

The ith column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node i.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} = \sum_j a_{ij} \boldsymbol{z}_j =$$

Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_j$$

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
(n =num. nodes, d =embedding dimension).

The ith column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node i.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} = \sum_j a_{ij} \boldsymbol{z}_j = \boldsymbol{m}_i$$

Efficiently computing the messages

The message received by each node is a sum of its neighbors' embeddings:

$$\boldsymbol{m}_i = \sum_{j \in N(i)} \boldsymbol{z}_j$$

Denote by $\boldsymbol{Z} \in \boldsymbol{R}^{n \times d}$ the matrix of stacked node embeddings,
(n =num. nodes, d =embedding dimension).

The ith column of the adjacency matrix $\boldsymbol{a}_i$ encodes the (in-)neighbors of node i.

$$\boldsymbol{a}_i^\top \boldsymbol{Z} = \sum_j a_{ij} \boldsymbol{z}_j = \boldsymbol{m}_i$$

Compute all messages at once:

$$\boldsymbol{M} = \boldsymbol{A}^\top \boldsymbol{Z}$$

RNN vs GNN

The sequence (chain) graph is also a graph, we could use a GNN.

RNN: sequential updates

t=1

t=2

t=3

RNN vs GNN

The sequence (chain) graph is also a graph, we could use a GNN.

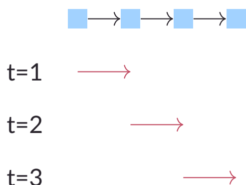
RNN: sequential updates

GNN: parallel local updates

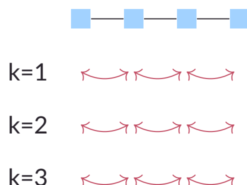
RNN vs GNN

The sequence (chain) graph is also a graph, we could use a GNN.

RNN: sequential updates



GNN: parallel local updates

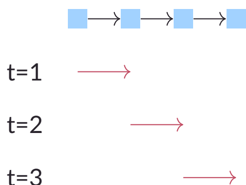


- Propagates through entire sequence with L “messages”.
- Embeddings only aware of nodes to the left (without bidirectional “hack”)
- Defined for sequences only (some extensions possible).

RNN vs GNN

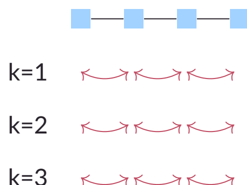
The sequence (chain) graph is also a graph, we could use a GNN.

RNN: sequential updates



- Propagates through entire sequence with L “messages”.
- Embeddings only aware of nodes to the left (without bidirectional “hack”)
- Defined for sequences only (some extensions possible).

GNN: parallel local updates



- After k iterations, every node got updates from its neighborhood up to k steps away.
- Can be used for any graph.

Pooling

As defined, a GNN gives us rich embeddings of every node.

To get a single embedding of the entire graph, we turn again to pooling.

Unlike for RNNs, there is no single node that could be taken as representative of the entire graph (especially if k is small and the graph is wide).

We turn to the kind of pooling used for CNNs:

1. average pooling: $\boldsymbol{z} = \frac{1}{n}(\boldsymbol{z}_1 + \ldots + \boldsymbol{z}_n)$
2. max pooling: $[\boldsymbol{z}]_j = \max([\boldsymbol{z}_1]_j, \ldots, [\boldsymbol{z}_n]_j)$

Permutation equivariance

The structure of a graph doesn't change if we number the nodes in another order.

The output of a GNN should not change either.

Mathematically, given a graph represented as $(\boldsymbol{X}, \boldsymbol{A})$, for any permutation matrix $\boldsymbol{P}$, a GNN satisfies

$$\mathrm{GNN}(\boldsymbol{P}\boldsymbol{X}, \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\top) = \boldsymbol{P}\,\mathrm{GNN}(\boldsymbol{X}, \boldsymbol{A}).$$

GNN variants

Many variations can be built on top of this idea.

- The update $\boldsymbol{z}_i^{(k+1)} = \phi(\boldsymbol{W}_{\text{self}}\boldsymbol{z}_i^{(k)} + \boldsymbol{W}_{\text{neigh}}\boldsymbol{m}_i^{(k)} + \boldsymbol{b})$ resembles an RNN.
  $\longrightarrow$ gated variants (GGNN)!
- Separate weight matrices per iteration $(\boldsymbol{W}^{(k)}_{\{\text{self,neigh}\}}, \boldsymbol{b}^{(k)})$
- Supporting different edge types:
  - first, notice that $\boldsymbol{W}_{\text{neigh}} \sum_j \boldsymbol{z}_j = \sum_j \boldsymbol{W}_{\text{neigh}}\boldsymbol{z}_j$.
  - then, if $e(i,j)$ is the type of the edge from i to j, we could compute $\sum_j \boldsymbol{W}_{e(i,j)}\boldsymbol{z}_j$.
- Different normalization over neighbors (more next time).

Today we have seen:

1. **Recurrent Neural Networks**

   Gated RNN

   Bidirectional RNN

   Multi-layer RNN

2. **Graph Neural Networks**

   RNN vs GNN

   Permutation equivariance

   GNN Variants