Recurrent Neural Networks for Natural Language Processing

Katarzyna Baraniak

Polish-Japanese Academy of Information Technology

February 21, 2024

Agenda

- RNNs and their variants
  - Simple RNNs
  - LSTM
  - GRU
- RNN's architectures
- Textual input representation for RNNs
- RNNs as language models
- RNNs for other NLP tasks

Introduction

Language is a sequence of words and has temporal structure[JM23].
Recurrent neural networks seems to be more natural choice than feed forward networks.
They address some challenges of natural language processing are:

- text does not have fixed length
- the order of words matter
  I have to read this book.
  I have this book to read.

Table of Contents

RNNs and their variants
- Simple RNNs
- LSTM
- GRU

RNN's architectures

Textual input representation for RNNs

RNNs as language models

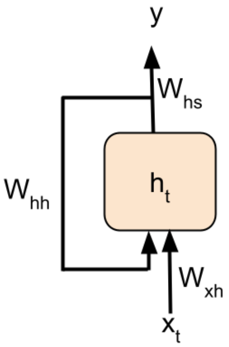
RNNs for other NLP tasks

Recurrent Neural Networks

Any network that have a circular connection passing it's output to the new input is called Recurrent Neural Network (RNN) [RHW88]. This type networks are able to process the sequence of any length and capture the long distance dependencies in sequences.

Simple Recurrent Neural Network

The input is a sequence $X = \{x_1, x_2, x_3, \ldots, x_t\}$ where each element x_t is an input at timestep t. The network calculates the hidden state h_t and the output y. The idea is to add connection, which passes the value of the previous hidden state h_{t-1} to the current neuron's input.



Recurrent Neural Network's equation

Equations to calculate basic RNN's output at timestep t are:

$$\begin{aligned} h_t &= g_1(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ y_t &= g_2(W_{hs}h_t + b_s) \end{aligned} \tag{1}$$

h_t -the hidden state at time step t
y -the output of network
g_1 and g_2 - activation functions
h_{t-1} - the previous hidden state, $W_x h$ - weight matrix for input
W_{hh} - weight matrix for previous hidden state
W_{hs} - weight matrix for current hidden state
b_h and b_s - corresponding biases

Training RNN's

Recurrent neural networks are trained using a backpropagation through time [Wer90].

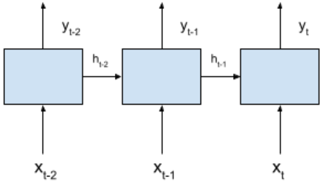


Figure: Unfolded RNN

- network is unfolded, so it contains t inputs and outputs.
- Every instance shares the same parameters.
- The error is calculated at each time step and backpropagated through all previous instances of unfolded network.
- Calculated weights changes are summed together.
- Weights are updated.

Problems with simple RNNs

vanishing gradient - when running a backpropagation method to train recurrent neural network long-term gradients may get smaller and smaller, tending to zero. Due tu finite precision numbers gradient may "vanish".
exploding gradient - the opposite, gradient becomes excessively large

Long short-term memory networks

Long short-term memory networks are RNNs that were designed to deal with vanishing gradient.
This neural networks provide short-term memory that "remembers" chosen details of input for many steps.

The model of the LSTM cell

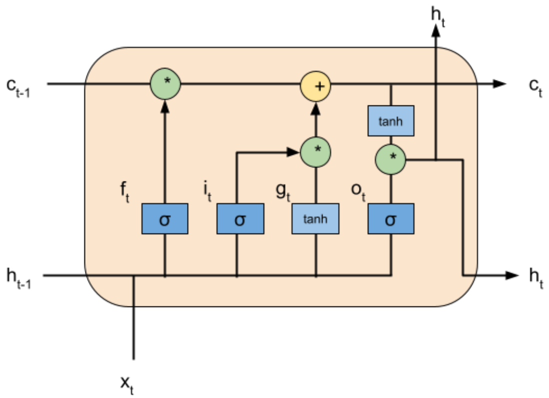


Figure: LSTM cell model

There are various types of LSTMs, usually with modifications of the forget gate or the output gate.

Long short-term memory networks

The function for the LSTM [HS97]:

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hi}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned} \tag{2}$$

where $\odot$ Hadamard (elementwise) product operator

LSTM - forget gate

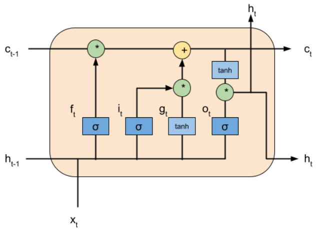


Figure: LSTM cell model

- $f_t = \sigma(W_{if} x_t + b_{if} + W_{hf} h_{t-1} + b_{hf})$
- **f_t is the forget gate which sets the weight of previous cell state from 0 to 1 using a sigmoid function**
- h_{t-1} - the hidden state at the time step $t-1$
- x_t is the input at the time t.
- W_i, W_h and b_h - input weights, hidden state weights and bias

LSTM - input gate

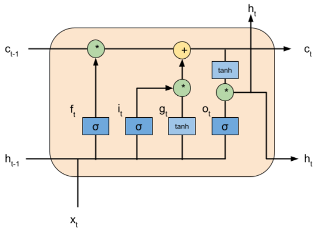


Figure: LSTM cell model

- $i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$
- **i_t is the input gate which controls how much of the new input is let in using sigmoid function**
- h_{t-1} - the hidden state at the time step $t-1$
- x_t is the input at the time t.
- W_i, W_h and b_h - input weights, hidden state weights and bias

LSTM - cell gate

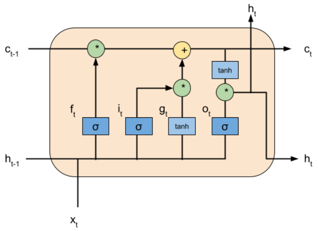


Figure: LSTM cell model

- $g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hi}h_{t-1} + b_{hg})$
- g_t is the cell gate which is responsible for an update of the information coming from the input
- h_{t-1} - the hidden state at the time step $t-1$
- x_t is the input at the time t.
- W_i, W_h and b_h - input weights, hidden state weights and bias

LSTM - output gate

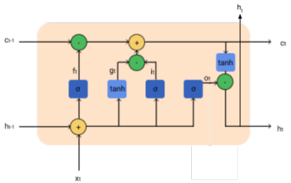


Figure: LSTM cell model

- $o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$
- **o_t is an output gate that controls the output of the hidden state.**
- h_{t-1} - the hidden state at the time step $t-1$
- x_t is the input at the time t.
- W_i, W_h and b_h - input weights, hidden state weights and bias

LSTM -cell state and hidden state

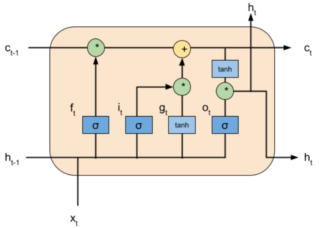
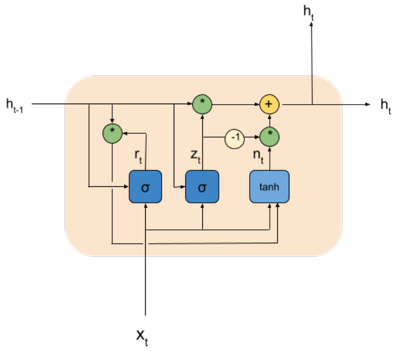


Figure: LSTM cell model

- $c_t = f_t \odot c_{t-1} + i_t \odot g_t$
- $h_t = o_t \odot \tanh(c_t)$
- **c_t is the cell state that preserves LSTM from the exploding gradient**
- **h_t is the hidden state at the time step t**
- cell state and hidden state is controlled by other gates

Gated Recurrent Unit Network (GRU)[CVMG+14]

Easier to train than the LSTM, without loses on the performance.



In this network the interpolation between the hidden state h_t and the previous state h_{t-1} is modulated only by the update gate. They also have a reset gate to control the hidden state. There is no cell state.

Gated Recurrent Unit Network

Reset gate r_t is computed by

$$r_t = \sigma\left(W_{ir}x_t + b_{iz} + W_{hr}h_{(t-1)} + b_{hr}\right), \tag{3}$$

the update gate z_t is computed by:

$$z_t = \sigma\left(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}\right), \tag{4}$$

the new gate n_t is defined by:

$$n_t = \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{(t-1)} + b_{hn})), \tag{5}$$

and finally, the hidden state h_t is computed by following:

$$h_t = z_t \odot h_{(t-1)} + (1 - z_t) \odot n_t, \tag{6}$$

where x is the input, h_{t-1} is the previous hidden state, W_i and W_h, b are learned input weights, recurrent weights and bias.

Table of Contents

RNNs and their variants
Simple RNNs
LSTM
GRU

RNN’s architectures

Textual input representation for RNNs

RNNs as language models

RNNs for other NLP tasks

Common RNN's architectures

RNNs can be combined into more complicated architectures

- stacked RNNs
- bidirectional RNNs
- encoder decoder

Stacked RNNs

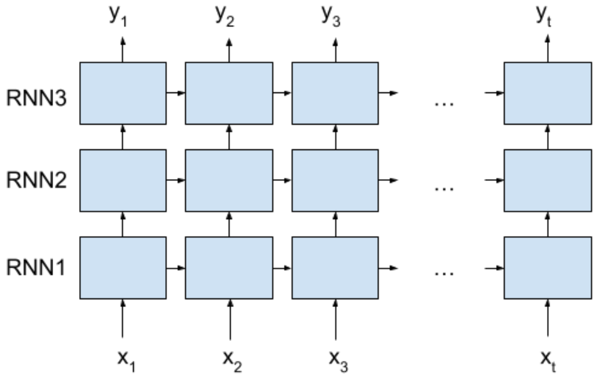


Figure: Stacked RNN. The output of the first layers is passed directly to the next layers.

Bi-directional RNNs

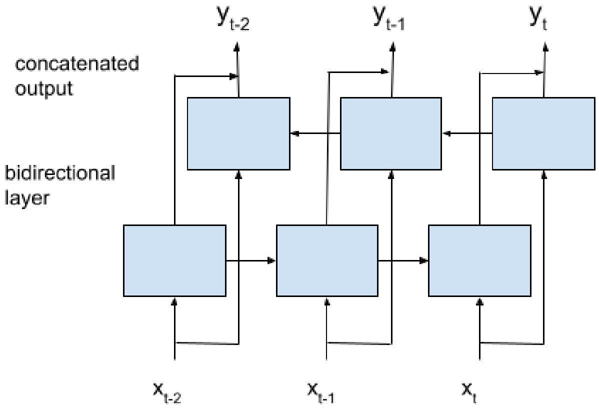


Figure: Bidirectional RNN. There are two layers that reads the input sequence in both directions. The output of the first layer and second layer is concatenated.

Encoder-decoder RNNs

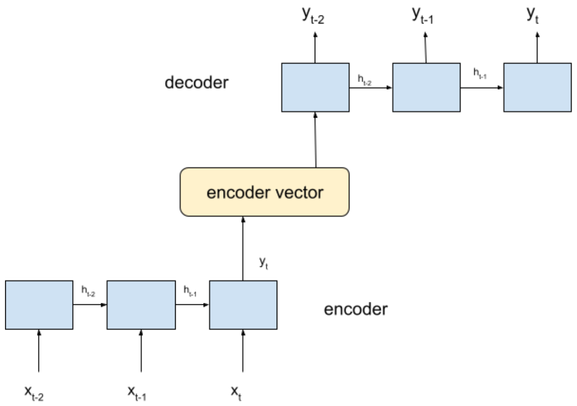


Figure:

RNN's architectures by input and output type

RNNs may be categorised by input and output type:

RNN's architectures by input and output type

RNNs may be categorised by input and output type:

- vector-to-sequence input is a single element, the output is generated at each time-step and used as input for the next time step for example sequence generation tasks

RNN's architectures by input and output type

RNNs may be categorised by input and output type:

- vector-to-sequence input is a single element, the output is generated at each time-step and used as input for the next time step for example sequence generation tasks
- sequence-to-vector - the input is a sequence and there is only one output. This architecture is used for sequence classification tasks like sentiment analysis

RNN's architectures by input and output type

RNNs may be categorised by input and output type:

- vector-to-sequence input is a single element, the output is generated at each time-step and used as input for the next time step for example sequence generation tasks
- sequence-to-vector - the input is a sequence and there is only one output. This architecture is used for sequence classification tasks like sentiment analysis
- sequence-to-sequence with input and output of the same length - This is mostly used for sequence tokens classification tasks like named entity recognition

RNN's architectures by input and output type

RNNs may be categorised by input and output type:

- vector-to-sequence input is a single element, the output is generated at each time-step and used as input for the next time step for example sequence generation tasks
- sequence-to-vector - the input is a sequence and there is only one output. This architecture is used for sequence classification tasks like sentiment analysis
- sequence-to-sequence with input and output of the same length - This is mostly used for sequence tokens classification tasks like named entity recognition
- sequence-to-sequence (encoder-decoder) with input and output of different lengths - for the task where we generate a new sequence based on the input sequence, for example machine translation

Table of Contents

RNNs and their variants
Simple RNNs
LSTM
GRU

RNN's architectures

Textual input representation for RNNs

RNNs as language models

RNNs for other NLP tasks

Textual input representation for RNNs

Text cannot be directly passed to neural networks. Wee need to create it's meaningful representation

- one-hot encoded vectors
- word embeddings (word2vec)

One-hot encoded vectors

Each input token is represented as a vector of zeros and single one representing the current token.
Vector's length is equal to the number of token in vocabulary.

One-hot encoded vectors

Each input token is represented as a vector of zeros and single one representing the current token.
Vector's length is equal to the number of token in vocabulary.

Example

Corpus with two documents:
"I have to read this book"
"I have this book"
Word "I" can represented by vector: [1,0,0,0,0,0] word "have" [0,1,0,0,0,0] for both documents. Each vector is of vocabulary size length

Word embeddings

Dense vector representation of input tokens.
Can be trained together with RNN model or pretrained using dedicated methods: Word2vec, fastText, Glove.

Word embeddings

Dense vector representation of input tokens.
Can be trained together with RNN model or pretrained using dedicated methods: Word2vec, fastText, Glove.
Word meaning is learned directly from their distribution in text using **self-supervised** methods.

Word embeddings

Dense vector representation of input tokens.

Can be trained together with RNN model or pretrained using dedicated methods: Word2vec, fastText, Glove.

Word meaning is learned directly from their distribution in text using **self-supervised** methods.

Example of **representation learning**.

Word embeddings

Dense vector representation of input tokens.
Can be trained together with RNN model or pretrained using dedicated methods: Word2vec, fastText, Glove.
Word meaning is learned directly from their distribution in text using **self-supervised** methods.
Example of **representation learning**.
Words that are synonyms or have quite similar meaning are represented by vectors that are close to each other.
Using such vectors it is possible to make operations on words:

Example

Poland + capital = Warsaw

First introduced as neural language models [BDVJ03] .

Word2Vec

word2vec [MSC$^+$13] [MCCD13].

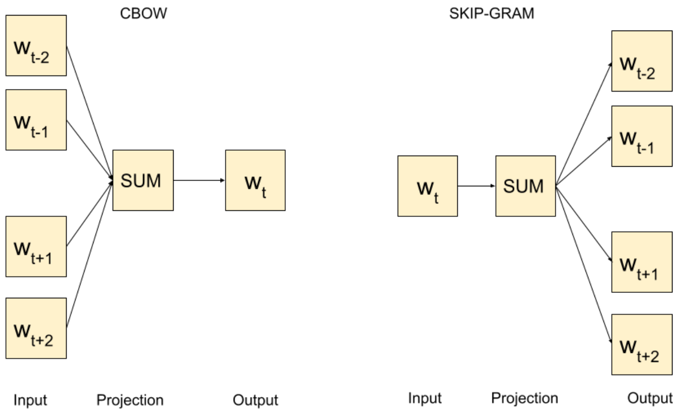


Figure: Architecture of word2vec cbow and skip-gram models. w_t is a target word, $\{w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}\}$ are context words.

Some problems with word embeddings

- large vocabulary
- how to represent new words
- preprocessing
- word-sense disambiguation

Table of Contents

RNNs and their variants
- Simple RNNs
- LSTM
- GRU

RNN's architectures

Textual input representation for RNNs

RNNs as language models

RNNs for other NLP tasks

Language models

What is a language model?
The general idea of language models is to predict the probability of the next word in a sequence based on the given context, usually previous words. Example:

Language models

What is a language model?
The general idea of language models is to predict the probability of the next word in a sequence based on the given context, usually previous words. Example:

$$\text{P(Madrit}|\text{ The largest city in Spain is)}$$

Language models

What is a language model?
The general idea of language models is to predict the probability of the next word in a sequence based on the given context, usually previous words. Example:

P(Madrit| The largest city in Spain is)

Language model assign such conditional probability to each possible word creating the distribution for the whole vocabulary. Using a chain rule we can assign the conditional probability for the whole sequence

$$P(w_{1:K}) = \prod_{k=1...K} p(w_k | w_{1:k-1}) \tag{7}$$

RNNs as language models

RNN based language model **predicts the next word based on the current input word and previous hidden state**.
Comparing to the simpler language models RNNs predicts next word based on the whole historical context instead of limit of n-1 previous words.
It also does not have fixed window size as in feed forward neural networks.

Inference - how to find next word?

1. the input sequence $X = [x_1, \ldots, x_t, \ldots, x_N]$ - a series of input token representation (a token may be a word, a character, a subword represented as one-hot vector)
2. at each step model retrieves the embedding of a token from embedding matrix
   $e_t = Ex_t$
3. it combines the current token embedding with the previous hidden state and compute the new hidden state
   $h_t = g(Uh_{t-1} + We_t)$
4. the the output layer is calculated and passed through softmax to get the probability distribution over the next token w_i.
   $P(y_t = w_i | x_t, y_{1:t-1}) = softmax(Vh_t)$
5. The goal is to find the most probable word from all possibilities
   $$y_t = argmaxP(y|x) \tag{8}$$

Training RNN as language model

- training is done on a corpus of text
- self supervised algorithm: no extra labeling is needed
- natural sequence of words is used
- train the model to minimize the error in predicting the true next word in training sequence
- cross entropy is used as loss, measuring the difference between true and predicted probability distribution at time t the negative log probability model calculates for the next word in a training sequence
  $L_{CE} = -log\, \hat{y}_t[w_{t+1}]$
- teacher forcing - in next step the model takes as input the true word sequence, not the predicted one

Table of Contents

RNNs and their variants

- Simple RNNs
- LSTM
- GRU

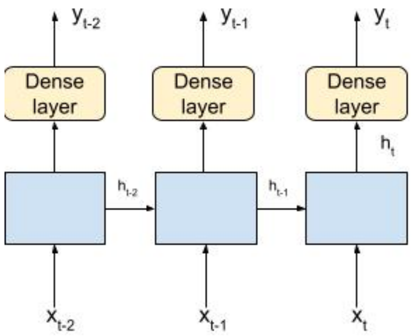
RNN's architectures

Textual input representation for RNNs

RNNs as language models

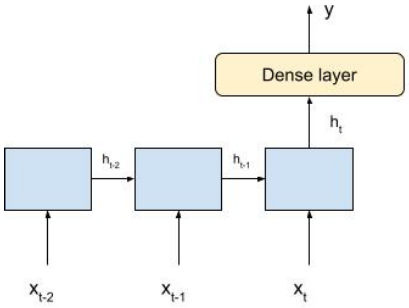
RNNs for other NLP tasks

RNN's for sequence tagging



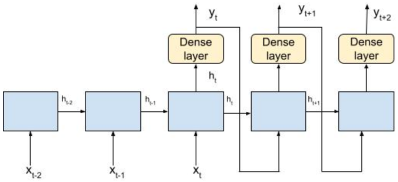
- sequence to sequence model
- for each input token the model predicts exactly one output label
- at each step RNN reads embedding of the current word calculates hidden layer, pass it through output layer and softmax that generates the distribution over labels set, then choose label with maximum value as a prediction
- NLP tasks: Named entity recognition, Part of speech tagging

RNN's for sequence classification



- sequence-to-vector
- for the whole sequence predicts one label
- RNN reads the whole sequence. the last hidden state is representation of the whole sequence and is processed through output layer
- end-to-end learning
- pooling - instead of the last hidden state, take all and do pooling
- NLP tasks: Sentiment analysis, spam detection etc

RNN's for text generation



- example of generative AI
- input can be a short text or longer text and the aim is to generate the new sequence of words based on the input
- autoregressive generation
- tasks: text generation, question answering, storytelling, chatbots, machine translation

Summary

- idea of RNNs
- LSTM and GRU variants
- word embeddings as input representation for RNNs
- various RNNs architecture
- application of RNNs for NLP

References I

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin, *A neural probabilistic language model*, Journal of machine learning research **3** (2003), no. Feb, 1137–1155.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio, *Learning phrase representations using rnn encoder-decoder for statistical machine translation*, arXiv preprint arXiv:1406.1078 (2014).

Sepp Hochreiter and Jürgen Schmidhuber, *Long short-term memory*, Neural computation **9** (1997), no. 8, 1735–1780.

Daniel Jurafsky and James Martin, *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, vol. 3, 2023.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, *Efficient estimation of word representations in vector space*, arXiv preprint arXiv:1301.3781 (2013).

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean, *Distributed representations of words and phrases and their compositionality*, NIPS, Curran Associates, Inc., 2013, pp. 3111–3119.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, *Learning representations by back-propagating errors*, p. 696–699, MIT Press, Cambridge, MA, USA, 1988.

Paul J Werbos, *Backpropagation through time: what it does and how to do it*, Proceedings of the IEEE **78** (1990), no. 10, 1550–1560.