Recurrent neural nets.

Victor Kitov

v.v.kitov@yandex.ru

Intro

- Sequences
  - words: sequences of symbols
  - sentences: sequences of words
  - documents: sequences of words
- Need fixed vector representation for prediction!
- Bag-of-words allows to do that:
  - one-hot encoding: indicator, TF, TF-IDF models
  - embeddings: get average embedding for sentence/document.

Problem of bag-of-words approach

- Problem: bag-of-words completely ignores word order.
  - information loss!
- Recurrent neural nets account for positions of all elements in sequence!
  - output fixed size sequence representation
  - this feature representation is feature extraction for later model.
    - e.g. MLP.

Recurrent neural net (RNN)

- Consider input sequence $\mathbf{x_{i:j}} := \mathbf{x_i}, ...\mathbf{x_j}$, $\mathbf{x_i} \in \mathbb{R}^{d_{in}}$.
- RNN outputs single vector $\widehat{\mathbf{y}}_\mathbf{n} \in \mathbb{R}^{d_{out}}$:

$$\widehat{\mathbf{y}}_\mathbf{n} = RNN(\mathbf{x_{1:n}})$$

- This implicitly defines RNN* with sequential output:

$$\widehat{\mathbf{y}}_\mathbf{1:n} = RNN^*(\mathbf{x_{1:n}})$$
$$\widehat{\mathbf{y}}_\mathbf{i} = RNN(\mathbf{x_{1:i}})$$

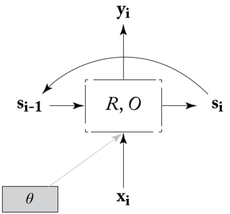
- Comments:
  - RNN shrinks history $\mathbf{x_{1:n}}$ to fixed size vector $\mathbf{y_n}$.
  - No Markov assumption: all info is aggregated!

Technical details of RNN

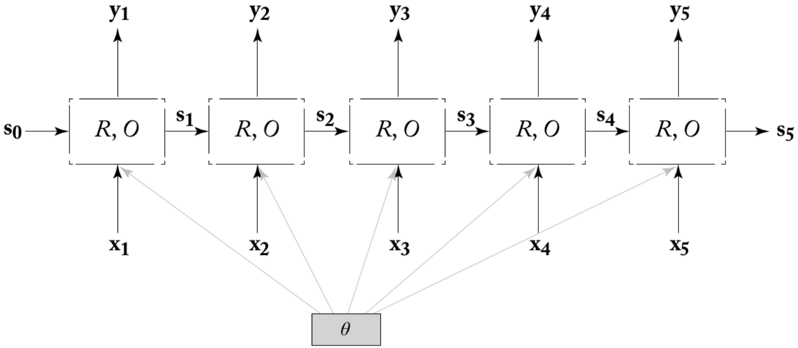
$$RNN^*(\mathbf{x_{1:n}}, \mathbf{s_0}) = \mathbf{y_{1:n}}$$
$$\widehat{\mathbf{y}}_\mathbf{i} = O(\mathbf{s_i})$$
$$\mathbf{s_i} = R(\mathbf{s_{i-1}}, \mathbf{x_i})$$

$$\mathbf{x_i} \in \mathbb{R}^{d_{in}}, \mathbf{y_i} \in \mathbb{R}^{d_{out}}, \mathbf{s_i} \in \mathbb{R}^{d_{state}}$$

Typical usage: $O(\mathbf{s}) \equiv \mathbf{s}$, $d_{state} = d_{out}$, $s_0 = \mathbf{0}$.



Unrolled RNN



$$\begin{aligned} s_4 &= R(s_3, x_4) = R(R(s_2, x_3), x_4) \\ &= R(R(R(s_1, x_2), x_3), x_4) = R(R(R(R(s_0, x_1), x_2), x_3), x_4) \end{aligned}$$

Training

Training: unroll RNN and use parameter sharing.

- called **backpropagation through time** (BPTT)
- Variant: unroll RNN for all non-intersecting subsequences of given sequence of given length.

init s_0

for i in $0, 1, ...n/k - 1$:
    $\mathbf{\hat{y}_{ki+1:ki+k}} = RNN^*(\mathbf{x_{ki+1:ki+k}}, \mathbf{s_{ki}})$
    calculate loss $\sum_{j=ki+1}^{ki+k} L(\mathbf{\hat{y}_j}, \mathbf{y_j})$
    backpropagate gradients, update weights

Common use-cases of RNN

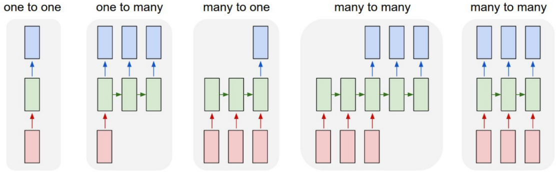
- **Acceptor**: output prediction in $\widehat{\mathbf{y}}_{\mathbf{n}}$.
  - e.g. read sentence and output its polarity probabilities.
- **Transducer**: tag sequence $x_1, ...x_n$ with RNN outputs $y_1, ...y_n$. Loss function:

$$\mathcal{L}\left(\widehat{\mathbf{y}}_{\mathbf{1:n}}, \mathbf{y}_{\mathbf{1:n}}\right) = \sum_{i=1}^{n} L\left(\widehat{\mathbf{y}}_{\mathbf{i}}, \mathbf{y}_{\mathbf{i}}\right)$$

  - e.g. POS tagging, language modelling.
  - summarization: for each sentence classify whether to include it into summary or not.
- **Encoder**: encode input sequence representation as $\widehat{\mathbf{y}}_{\mathbf{n}}$, e.g.:
  - machine translation: translation generation done with another "decoding" RNN
    decode starting from $\mathbf{s_0} = \widehat{\mathbf{y}}_{\mathbf{n}}$.

NN & RNN architectures

NN & RNN architectures:



Examples, where these architectures arise:

- **one to one:** classical classification, image classification.
- **one to many:** image captioning, story generation based on topic.
- **many to one:** text classification, sentiment analysis.
- **many to many:** machine translation, summarization.
- **synced many to many:** POS tagging, activity detection on video.

Table of Contents

Bidirectional RNN

- Bidirectional RNN consists of 2 RNNs
  - forward RNN $\left(R^f, O^f\right)$ with state $\mathbf{s_i^f}$,$i = \overline{1, n}$
  - backward RNN $\left(R^b, O^b\right)$ with state $\mathbf{s_i^b}$, $i = \overline{1, n}$
- Forward RNN goes in forward direction $\mathbf{x_1}, \mathbf{x_2}...\mathbf{x_n}$.
- Backward RNN goes in backward direction $\mathbf{x_n}, \mathbf{x_{n-1}}...\mathbf{x_1}$.
- At each moment i we have 2 states:
  1. $\mathbf{s_i^f} = F_1(\mathbf{x_1}, \mathbf{x_2}...\mathbf{x_i})$
  2. $\mathbf{s_i^b} = F_2(\mathbf{x_i}, \mathbf{x_{i+1}}, ...\mathbf{x_n})$

Bidirectional RNN

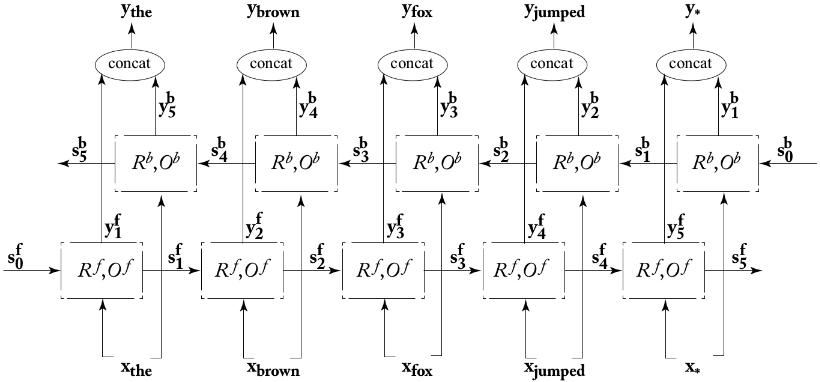
- So we can output

$$biRNN(\mathbf{x_{1:n}}, i) = \widehat{\mathbf{y}}_{\mathbf{i}} = [\widehat{\mathbf{y}}_{\mathbf{i}}^{\mathbf{f}}; \widehat{\mathbf{y}}_{\mathbf{i}}^{\mathbf{b}}] = [RNN^f(\mathbf{x_{1:i}});\ RNN^b(\mathbf{x_{n:i}})]$$

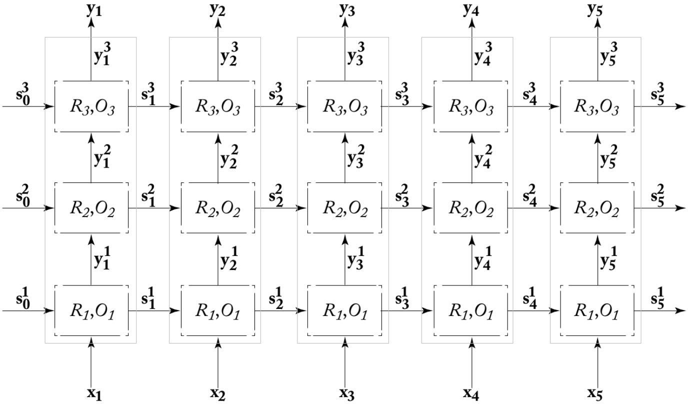
$$biRNN^*(\mathbf{x_{1:n}}) = \mathbf{y_{1:n}} = [biRNN(\mathbf{x_{1:n}}, 1); ...; biRNN(\mathbf{x_{1:n}}, n)]$$

  - encoding takes into account past and future!
- biRNN is very effective for tagging sequences (e.g. POS tagging).

biRNN illustration



Stacked RNN



- Output of previous layer RNN is input to next layer.
- Empirically stacked RNNs work better than single layer RNNs.
- biRNNs can also be stacked.

Table of Contents

Bag-of-words RNN

Bag-of-words RNN:

$$\mathbf{s_i} = \mathbf{s_{i-1}} + \mathbf{x_i}$$
$$\mathbf{y_i} = \mathbf{s_i}$$

- $\mathbf{x_i}$: input vector
- $\mathbf{s_i}$: hidden layer
- $\mathbf{y_i}$: output vector

Order of words does not matter, not very informative.

Simple RNN (S-RNN)

Simple RNN (S-RNN)[1]:

$$\mathbf{s_i} = g_s\left(W_s\mathbf{s_{i-1}} + V_s\mathbf{x_i} + \mathbf{b_s}\right)$$
$$\mathbf{y_i} = g_y(W_y\mathbf{s_i} + \mathbf{b_y})$$

- $\mathbf{x_i}$: input vector
- $\mathbf{s_i}$: hidden layer
- $\mathbf{y_i}$: output vector
- W_s, V_s, W_y: parameter matrices
- $\mathbf{b_s}, \mathbf{b_y}$: parameter vectors
- $g_s(\cdot), g_y(\cdot)$: activation functions

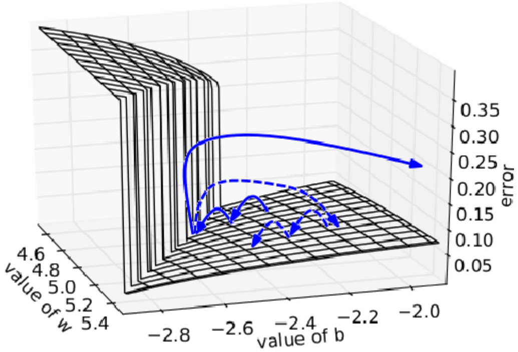
[1] also called Elman network

Properties of S-RNN

- S-RNN is sensitive to the order of the inputs.
- Due to recurrent multiplications by W_s is subject to:
    - exploding gradient problem
        - solved by gradient clipping
    - vanishing gradient problem
        - solved by gated models and memory models

Exploding gradient problem

Exploding gradient problem:



Exploding gradient problem

Solutions:

- add regularization
- gradient clipping: clip norm of gradient by threshold.
    - if $\|\nabla_\theta L(\widehat{y}_i, y_i)\| < t$

$$\theta \to \theta - \varepsilon \nabla_\theta L(\widehat{\mathbf{y}}_\mathbf{i}, \mathbf{y}_\mathbf{i})$$

    - else

$$\theta \to \theta - \varepsilon \frac{t}{\|\nabla_\theta L(\widehat{\mathbf{y}}_\mathbf{i}, \mathbf{y}_\mathbf{i})\|} \nabla_\theta L(\widehat{\mathbf{y}}_\mathbf{i}, \mathbf{y}_\mathbf{i})$$

Vanishing gradients

- Repetitive multiplication of state by the same matrix W_s and saturating non-linearities also cause net to forget past quickly due to vanishing gradients.
- Ways to combat this:

1. Initialize $W_s = I$,$b_s = 0$, $g_s = ReLu$.
    - so initially network sums information
    - will change behavior after training if needed
2. Better solution: use LSTM model.

Table of Contents

Gates

- Consider n dimensional vectors:
  - old state $\mathbf{s}$, update $\mathbf{x}$ and new state $\mathbf{s}'$.
- Gate $g \in \{0,1\} \in \mathbb{R}^n$ controls state positions where change is applied.
- Example ($\odot$ defines point-wise multiplication):

$$\underbrace{\begin{bmatrix}8\\11\\3\\7\\5\\15\end{bmatrix}}_{\mathbf{s}'} \leftarrow \underbrace{\begin{bmatrix}0\\1\\0\\0\\0\\1\end{bmatrix}}_{\mathbf{g}} \odot \underbrace{\begin{bmatrix}10\\11\\12\\13\\14\\15\end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix}1\\0\\1\\1\\1\\0\end{bmatrix}}_{(\mathbf{1}-\mathbf{g})} \odot \underbrace{\begin{bmatrix}8\\9\\3\\7\\5\\8\end{bmatrix}}_{\mathbf{s}}$$

- Problems:
  - gates need to be learned
  - piece-wise constant gates cannot be optimized.
- Solution: use sigmoid gate $\mathbf{g} = \sigma(f(\mathbf{x}, \mathbf{s}, \theta))$
  - θ: learned parameters
  - f: any differentiable function

Long short-term memory (LSTM) model

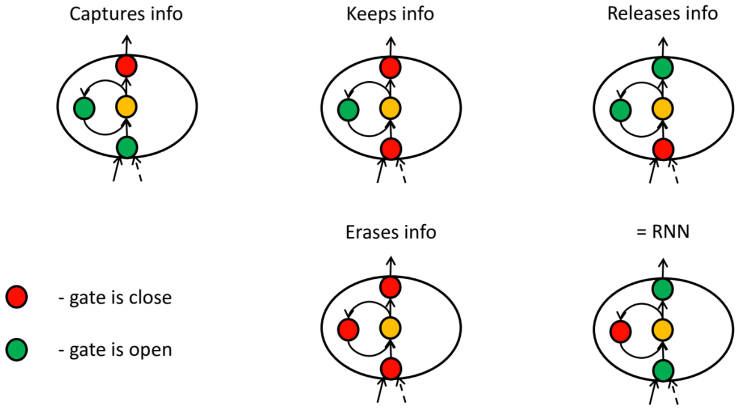
$$\mathbf{f_t} = \sigma\left(W_f \mathbf{x_t} + U_f \mathbf{h_{t-1}} + \mathbf{b_f}\right) \quad \text{forget gate}$$
$$\mathbf{i_t} = \sigma\left(W_i \mathbf{x_t} + U_i \mathbf{h_{t-1}} + \mathbf{b_i}\right) \quad \text{input gate}$$
$$\mathbf{o_t} = \sigma\left(W_o \mathbf{x_t} + U_o \mathbf{h_{t-1}} + \mathbf{b_0}\right) \quad \text{output gate}$$
$$\mathbf{c_t} = \mathbf{f_t} \odot \mathbf{c_{t-1}} + \mathbf{i_t} \odot tanh\left(W_c \mathbf{x_t} + U_c \mathbf{h_{t-1}} + \mathbf{b_c}\right) \quad \text{inner state}$$
$$\mathbf{h_t} = \mathbf{o_t} \odot tanh\left(\mathbf{c_t}\right) \quad \text{observed output}$$

$\mathbf{x_t}$-input, parameters:

- matrices: $W_f, U_f, W_i, U_i, W_o, U_o, W_c, U_c$
- vectors: b_f, b_i, b_o, b_c
- initialization: c_0, h_0

Illustration[2]

Input from below, output above, memory (yellow) in the middle.



[2]Illustration by Lobacheva Julia.

Comments

- Architecture excluded repetitive multiplication of state by the same matrix W_s (which cause vanishing and exploding gradients)
- Gating mechanisms allow for gradients related to c_t to stay high across long time ranges.
- It's recommended to initialize $\mathbf{b_f} = \mathbf{1}$
    - so initially neural net tries to remember everything