Natural Language Processing
Recurrent Neural Networks

Felipe Bravo-Marquez

May 30, 2023

Recurrent Neural Networks

- While representations derived from convolutional networks offer some sensitivity to word order, their order sensitivity is restricted to mostly local patterns, and disregards the order of patterns that are far apart in the sequence.
- Recurrent neural networks (RNNs) allow representing arbitrarily sized sequential inputs in fixed-size vectors, while paying attention to the structured properties of the inputs [Goldberg, 2016].
- RNNs, particularly ones with gated architectures such as the LSTM and the GRU, are very powerful at capturing statistical regularities in sequential inputs.

The RNN Abstraction

- We use $\vec{x}_{i:j}$ to denote the sequence of vectors $\vec{x}_i, \ldots, \vec{x}_j$.
- On a high-level, the RNN is a function that takes as input an arbitrary length ordered sequence of n d_{in}-dimensional vectors $\vec{x}_{1:n} = \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$ ($\vec{x}_i \in \mathcal{R}^{d_{in}}$) and returns as output a single d_{out} dimensional vector $\vec{y}_n \in \mathcal{R}^{d_{out}}$:

$$\begin{gathered} \vec{y}_n = RNN(\vec{x}_{1:n}) \\ \vec{x}_i \in \mathcal{R}^{d_{in}} \quad \vec{y}_n \in \mathcal{R}^{d_{out}} \end{gathered} \tag{1}$$

- This implicitly defines an output vector $\vec{y}_i$ for each prefix $\vec{x}_{1:i}$ of the sequence $\vec{x}_{i:n}$.
- We denote by RNN^* the function returning this sequence:

$$\begin{gathered} \vec{y}_{1:n} = RNN^*(\vec{x}_{1:n}) \\ \vec{y}_i = RNN(\vec{x}_{1:i}) \\ \vec{x}_i \in \mathcal{R}^{d_{in}} \quad \vec{y}_n \in \mathcal{R}^{d_{out}} \end{gathered} \tag{2}$$

The RNN Abstraction

- The output vector $\vec{y}_n$ is then used for further prediction.
- For example, a model for predicting the conditional probability of an event e given the sequence $\vec{x}_{1:n}$ can be defined as the j-th element in the output vector resulting from the softmax operation over a linear transformation of the RNN encoding:

$$p(e = j | \vec{x}_{1:n}) = \text{softmax}(RNN(\vec{x}_{1:n}) \cdot W + \vec{b})_{[j]}$$

- The RNN function provides a framework for conditioning on the entire history without resorting to the Markov assumption which is traditionally used for modeling sequences.

The RNN Abstraction

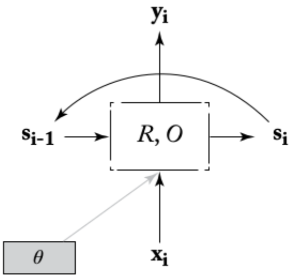
- The RNN is defined recursively, by means of a function R taking as input a state vector $\vec{s}_{i-1}$ and an input vector $\vec{x}_i$ and returning a new state vector $\vec{s}_i$.
- The state vector $\vec{s}_i$ is then mapped to an output vector $\vec{y}_i$ using a simple deterministic function $O(\cdot)$.
- The base of the recursion is an initial state vector, $\vec{s}_0$, which is also an input to the RNN.
- For brevity, we often omit the initial vector s_0 , or assume it is the zero vector.
- When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs $\vec{x}_i$ as well as the dimensions of the outputs $\vec{y}_i$.

The RNN Abstraction

$$
\begin{aligned}
RNN^*(\vec{x}_{1:n}; \vec{s}_0) &= \vec{y}_{1:n} \\
\vec{y}_i &= O(\vec{s}_i) \\
\vec{s}_i &= R(\vec{s}_{i-1}, \vec{x}_i) \\
\vec{x}_i \in \mathcal{R}^{d_{in}}, \quad \vec{y}_i \in \mathcal{R}^{d_{out}}&, \quad \vec{s}_i \in \mathcal{R}^{f(d_{out})}
\end{aligned} \tag{3}
$$

- The functions *R* and *O* are the same across the sequence positions.
- The RNN keeps track of the states of computation through the state vector $\vec{s}_i$ that is kept and being passed across invocations of *R*.

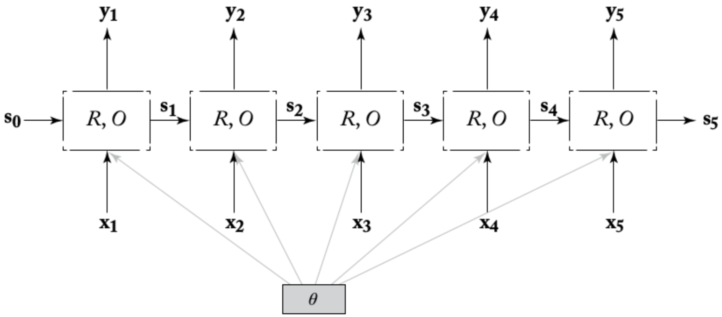
The RNN Abstraction



- This presentation follows the recursive definition, and is correct for arbitrarily long sequences.

The RNN Abstraction

- For a finite sized input sequence (and all input sequences we deal with are finite) one can unroll the recursion.



- The parameters θ highlight the fact that the same parameters are shared across all time steps.
- Different instantiations of *R* and *O* will result in different network structures.

The RNN Abstraction

- We note that the value of $\vec{s}_i$ (and hence $\vec{y}_i$) is based on the entire input $\vec{x}_1, \ldots, \vec{x}_i$.
- For example, by expanding the recursion for $i = 4$ we get:

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

- Thus, $\vec{s}_n$ and $\vec{y}_n$ can be thought of as encoding the entire input sequence.
- The job of the network training is to set the parameters of R and O such that the state conveys useful information for the task we are tying to solve.

Elman Network or Simple-RNN

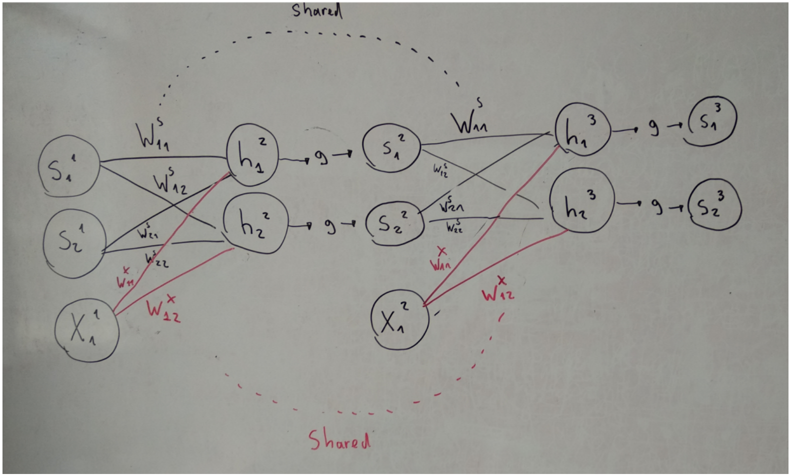
- After describing the RNN abstraction, we are now in place to discuss the simplest instantiations of it.
- Recall that we are interested in a recursive function $\vec{s}_i = R(\vec{x}_i, \vec{s}_{i-1})$ such that $\vec{s}_i$ encodes the sequence $\vec{x}_{1:n}$.
- The simplest RNN formulation is known as an Elman Network or Simple-RNN (S-RNN).

Elman Network or Simple-RNN

$$\begin{aligned} \vec{s}_i &= R_{SRNN}(\vec{x}_i, \vec{s}_{i-1}) = g(\vec{s}_{i-1} W^s + \vec{x}_i W^x + \vec{b}) \\ \vec{y}_i &= O_{SRNN}(\vec{s}_i) = \vec{s}_i \\ \vec{s}_i, \vec{y}_i \in \mathcal{R}^{d_s}, &\quad \vec{x}_i \in \mathcal{R}^{d_x}, \quad W^x \in \mathcal{R}^{d_x \times d_s}, \quad W^s \in \mathcal{R}^{d_s \times d_s}, \vec{b} \in \mathcal{R}^{d_s} \end{aligned} \tag{4}$$

- The state $\vec{s}_i$ and the input $\vec{x}_i$ are each linearly transformed.
- The results are added (together with a bias term) and then passed through a nonlinear activation function g (commonly tanh or ReLU).
- The Simple RNN provides strong results for sequence tagging as well as language modeling.

Elman Network or Simple-RNN



RNN Training

- An unrolled RNN is just a very deep neural network.
- The same parameters are shared across many parts of the computation.
- Additional input is added at various layers.
- To train an RNN network, need to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph.
- Then use the backward (backpropagation) algorithm to compute the gradients with respect to that loss.

RNN Training

- This procedure is referred to in the RNN literature as backpropagation through time (BPTT).
- The RNN does not do much on its own, but serves as a trainable component in a larger network.
- The final prediction and loss computation are performed by that larger network, and the error is back-propagated through the RNN.
- This way, the RNN learns to encode properties of the input sequences that are useful for the further prediction task.
- The supervision signal is not applied to the RNN directly, but through the larger network.

RNN usage-patterns: Acceptor

- A common RNN usage-pattern is the acceptor.
- This pattern is used for text classification.
- The supervision signal is only based on the final output vector $\vec{y}_n$.
- The RNN's output vector $\vec{y}_n$ is fed into a fully connected layer or an MLP, which produce a prediction.
- The error gradients are then backpropagated through the rest of the sequence.
- The loss can take any familiar form: cross entropy, hinge, etc.

RNN usage-patterns: Acceptor

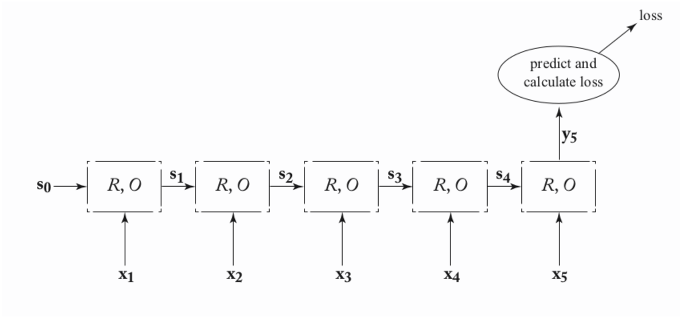


Figure: Acceptor RNN training graph.

- Example 1: an RNN that reads the characters of a word one by one and then use the final state to predict the part-of-speech of that word.
- Example 2: an RNN that reads in a sentence and, based on the final state decides if it conveys positive or negative sentiment.

RNN usage-patterns: Transducer

- Another option is to treat the RNN as a transducer, producing an output $\hat{t}_i$ for each input it reads in.
- This pattern is very handy for sequence labeling tasks (e.g, POS tagging, NER).
- We compute a local loss signal (e.g., cross-entropy) $L_{\text{local}}(\hat{t}_i, t_i)$ for each of the outputs $\hat{t}_i$ based on a true label t_i.
- The loss for unrolled sequence will then be: $L(\hat{t}_{i:n}, t_{i:n}) = \sum_i L_{\text{local}}(\hat{t}_i, t_i)$, or using another combination rather than a sum such as an average or a weighted average

RNN usage-patterns: Transducer

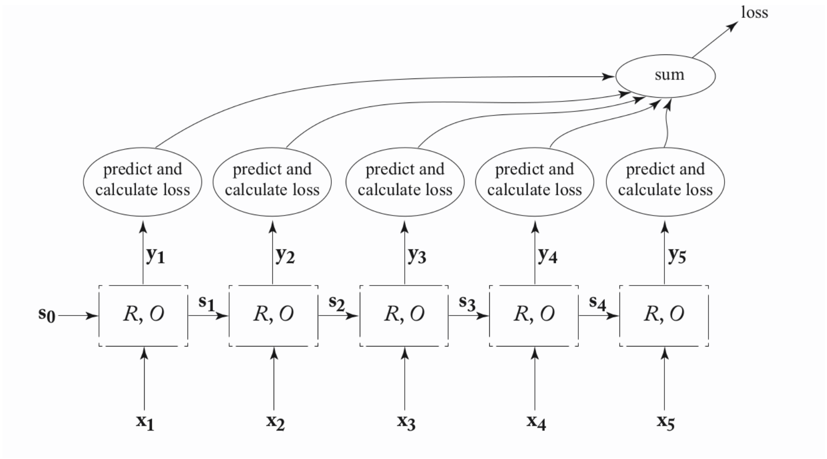

loss
sum
predict and calculate loss
predict and calculate loss
predict and calculate loss
predict and calculate loss
predict and calculate loss
y1
y2
y3
y4
y5
s0
R, O
s1
R, O
s2
R, O
s3
R, O
s4
R, O
x1
x2
x3
x4
x5

Figure: Transducer RNN training graph.

RNN usage-patterns: Transducer

- Example 1: a sequence tagger (e.g., NER, POS), in which we take $\vec{x}_{i:n}$ to be feature representations for the n words of a sentence, and t_i as an input for predicting the tag assignment of word i based on words $1 : i$.
- Example 2: language modeling, in which the sequence of words $x_{1:n}$ is used to predict a distribution over the $(i+1)th$ word.
- RNN-based language models are shown to provide vastly better perplexities than traditional language models.
- Using RNNs as transducers allows us to relax the Markov assumption that is traditionally taken in language models and HMM taggers, and condition on the entire prediction history.

Bidirectional RNNS (BIRNN)

- A useful elaboration of an RNN is a bidirectional-RNN (also commonly referred to as biRNN)
- Consider the task of sequence tagging over a sentence.
- An RNN allows us to compute a function of the *i*-th word x_i based on the past words $x_{1:i}$ up to and including it.
- However, the following words $x_{i+1:n}$ may also be useful for prediction,
- The biRNN allows us to look arbitrarily far at both the past and the future within the sequence.

Bidirectional RNNS (BIRNN)

- Consider an input sequence $\vec{x}_{1:n}$.
- The biRNN works by maintaining two separate states, s_i^f and s_i^b for each input position i.
- The forward state s_i^f is based on $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_i$, while the backward state s_i^b is based on $\vec{x}_n, \vec{x}_{n-1}, \ldots, \vec{x}_i$.
- The forward and backward states are generated by two different RNNs.
- The first RNN(R^f, O^f) is fed the input sequence $\vec{x}_{1:n}$ as is, while the second RNN(R^b, O^b) is fed the input sequence in reverse.
- The state representation $\vec{s}_i$ is then composed of both the forward and backward states.

Bidirectional RNNS (BIRNN)

- The output at position i is based on the concatenation of the two output vectors:

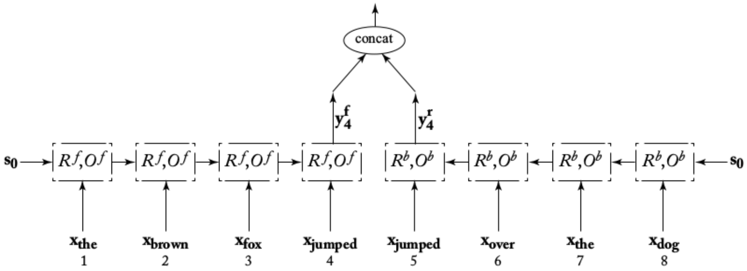
$$\vec{y}_i = [\vec{y}_i^f; \vec{y}_i^b] = [O^f(s_i^f); O^b(s_i^b)]$$

- The output takes into account both the past and the future.
- The biRNN encoding of the ith word in a sequence is the concatenation of two RNNs, one reading the sequence from the beginning, and the other reading it from the end.
- We define $biRNN(\vec{x}_{1:n}, i)$ to be the output vector corresponding to the ith sequence position:

$$biRNN(\vec{x}_{1:n}, i) = \vec{y}_i = [RNN^f(\vec{x}_{1:i}); RNN^b(\vec{x}_{n:i})]$$

Bidirectional RNNS (BIRNN)

- The vector $\vec{y}_i$ can then be used directly for prediction, or fed as part of the input to a more complex network.
- While the two RNNs are run independently of each other, the error gradients at position i will flow both forward and backward through the two RNNs.
- Feeding the vector $\vec{y}_i$ through an MLP prior to prediction will further mix the forward and backward signals.


concat
$\mathbf{y}^{\mathbf{f}}_4$
$\mathbf{y}^{\mathbf{r}}_4$
$\mathbf{s}_0$
R^f, O^f
R^f, O^f
R^f, O^f
R^f, O^f
R^b, O^b
R^b, O^b
R^b, O^b
R^b, O^b
$\mathbf{s}_0$
$\mathbf{x}_{\mathbf{the}}$ 1
$\mathbf{x}_{\mathbf{brown}}$ 2
$\mathbf{x}_{\mathbf{fox}}$ 3
$\mathbf{x}_{\mathbf{jumped}}$ 4
$\mathbf{x}_{\mathbf{jumped}}$ 5
$\mathbf{x}_{\mathbf{over}}$ 6
$\mathbf{x}_{\mathbf{the}}$ 7
$\mathbf{x}_{\mathbf{dog}}$ 8

- Note how the vector $\vec{y}_4$, corresponding to the word **jumped**, encodes an infinite window around (and including) the focus vector $\vec{x}_{jumped}$.

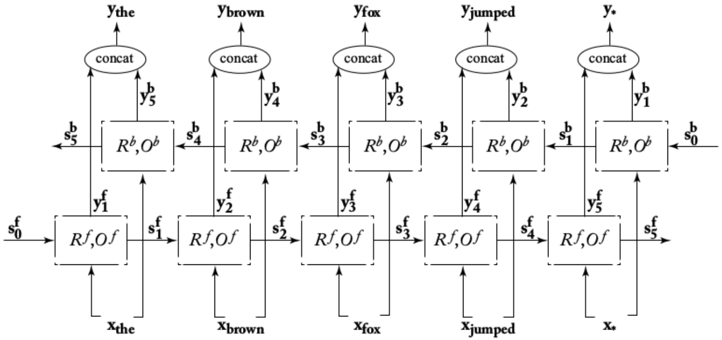
Bidirectional RNNS (BIRNN)

- Similarly to the *RNN* case, we also define $biRNN^*(\vec{x}_{1:n})$ as the sequence of vectors $\vec{y}_{1:n}$:

$$biRNN^*(\vec{x}_{1:n}) = \vec{y}_{1:n} = biRNN(\vec{x}_{1:n}, 1), \ldots, biRNN(\vec{x}_{1:n}, n)$$

- The n output vectors $\vec{y}_{1:n}$ can be efficiently computed in linear time by first running the forward and backward RNNs, and then concatenating the relevant outputs.

Bidirectional RNNS (BIRNN)

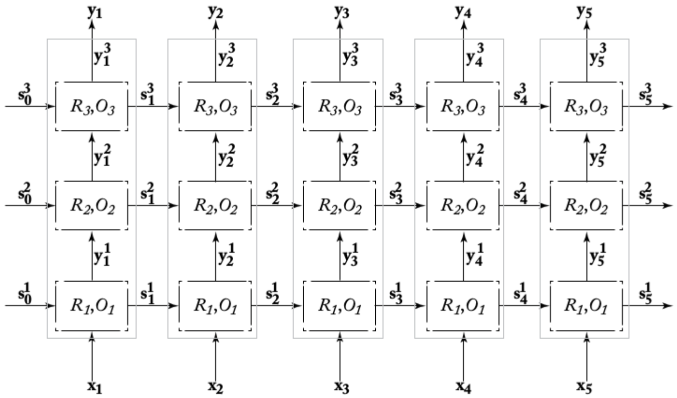


- The biRNN is very effective for tagging tasks, in which each input vector corresponds to one output vector.
- It is also useful as a general-purpose trainable feature-extracting component, that can be used whenever a window around a given word is required.

Multi-layer (stacked) RNNS

- RNNs can be stacked in layers, forming a grid.
- Consider k RNNs, $RNN_1, \ldots, RNN_k$, where the jth RNN has states $\vec{s}^{\,j}_{1:n}$ and outputs $\vec{y}^{\,j}_{1:n}$.
- The input for the first RNN are $\vec{x}_{1:n}$.
- The input of the jth RNN ($j \geq 2$) are the outputs of the RNN below it, $\vec{y}^{\,j-1}_{1:n}$.
- The output of the entire formation is the output of the last RNN, $\vec{y}^{\,k}_{1:n}$.
- Such layered architectures are often called deep RNNs.

Multi-layer (stacked) RNNS



- It is not theoretically clear what is the additional power gained by the deeper architecture.
- It was observed empirically that deep RNNs work better than shallower ones on some tasks (e.g., machine translation).

Gated Architectures

- So far we have seen only one instantiation of the RNN: the simple RNN (S-RNN) or Elman network.
- This model is hard to train effectively because of the **vanishing gradients** problem.
- Error signals (gradients) in later steps in the sequence diminish quickly in the backpropagation process.
- Thus, they do not reach earlier input signals, making it hard for the S-RNN to capture long-range dependencies.
- Gating-based architectures, such as the LSTM [Hochreiter and Schmidhuber, 1997] and the GRU [Cho et al., 2014] are designed to solve this deficiency.

The vanishing gradients problem in Recurrent Neural Networks

- Intuitively, recurrent neural networks can be thought of as very deep feed-forward networks, with shared parameters across different layers.
- For the Simple-RNN, the gradients then include repeated multiplication of the matrix W, making it very likely for the values to vanish or explode.
- The gating mechanism mitigate this problem to a large extent by getting rid of this repeated multiplication of a single matrix.

Gated Architectures

- Consider the RNN as a general purpose computing device, where the state $\vec{s}_i$ represents a finite memory.
- Each application of the function R reads in an input $\vec{x}_{i+1}$, reads in the current memory $\vec{s}_i$, operates on them in some way, and writes the result into memory.
- This results in a new memory state $\vec{s}_{i+1}$.
- An apparent problem with the S-RNN architecture is that the memory access is not controlled.
- At each step of the computation, the entire memory state is read, and the entire memory state is written.

Gated Architectures

- How does one provide more controlled memory access?
- Consider a binary vector $\vec{g} \in [0,1]^n$.
- Such a vector can act as a **gate** for controlling access to *n*-dimensional vectors, using the hadamard-product operation $\vec{x} \odot \vec{g}$.
- The hadamard operation is the same as the element-wise multiplication of two vectors:

$$\vec{x} = \vec{u} \odot \vec{v} \Leftrightarrow \vec{x}_{[i]} = \vec{u}_{[i]} \cdot \vec{v}_{[i]} \quad \forall i \in [1,n]$$

Gated Architectures

- Consider a memory $\vec{s} \in \mathcal{R}^d$, an input $\vec{x} \in \mathcal{R}^d$ and a gate $\vec{g} \in [0,1]^d$.
- The following computation:

$$\vec{s}' \leftarrow \vec{g} \odot \vec{x} + (\vec{1} - \vec{g}) \odot (\vec{s})$$

- Reads the entries in $\vec{x}$ that correspond to the $\vec{1}$ values in $\vec{g}$, and writes them to the new memory $\vec{s}'$.
- Locations that weren't read to are copied from the memory $\vec{s}$ to the new memory $\vec{s}'$ through the use of the gate $(\vec{1} - \vec{g})$.

Gated Architectures

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

- This gating mechanism can serve as a building block in our RNN.
- Gate vectors can be used to control access to the memory state $\vec{s}_i$.
- We are still missing two important (and related) components:
  1. The gates should not be static, but be controlled by the current memory state and the input.
  2. Their behavior should be learned.
- This introduced an obstacle, as learning in our framework entails being differentiable (because of the backpropagation algorithm).
- The binary 0-1 values used in the gates are not differentiable.

Gated Architectures

- A solution to this problem is to approximate the hard gating mechanism with a soft—but differentiable—gating mechanism.
- To achieve these differentiable gates , we replace the requirement that $\vec{g} \in [0,1]^n$ and allow arbitrary real numbers, $\vec{g}' \in \mathcal{R}^n$.
- These real numbers are then pass through a sigmoid function $\sigma(\vec{g}')$.
- This bounds the value in the range $(0,1)$, with most values near the borders.

Gated Architectures

- When using the gate $\sigma(\vec{g}') \odot \vec{x}$, indices in $\vec{x}$ corresponding to near-one values in $\sigma(\vec{g}')$ are allowed to pass.
- While those corresponding to near-zero values are blocked.
- The gate values can then be conditioned on the input and the current memory.
- And can be trained using a gradient-based method to perform a desired behavior.
- This controllable gating mechanism is the basis of the LSTM and the GRU architectures.
- At each time step, differentiable gating mechanisms decide which parts of the inputs will be written to memory and which parts of memory will be overwritten (forgotten).

LSTM

- The Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997] was designed to solve the vanishing gradients problem.
- It was the the first architecture to introduce the gating mechanism.
- The LSTM architecture explicitly splits the state vector $\vec{s}_i$ into two halves: 1) memory cells and 2) working memory.
- The memory cells are designed to preserve the memory, and also the error gradients, across time, and are controlled through differentiable gating components[1].
- At each input state, a gate is used to decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten.

[1] Smooth mathematical functions that simulate logical gates.

LSTM

- Mathematically, the LSTM architecture is defined as:

$$
\begin{aligned}
\boldsymbol{s_j} = R_{\text{LSTM}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) &= [\boldsymbol{c_j}; \boldsymbol{h_j}] \\
\boldsymbol{c_j} &= \boldsymbol{f} \odot \boldsymbol{c_{j-1}} + \boldsymbol{i} \odot \boldsymbol{z} \\
\boldsymbol{h_j} &= \boldsymbol{o} \odot \tanh(\boldsymbol{c_j}) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xi}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hi}}) \\
\boldsymbol{f} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xf}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hf}}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xo}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{ho}}) \\
\boldsymbol{z} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{h_{j-1}} \boldsymbol{W^{hz}})
\end{aligned}
$$

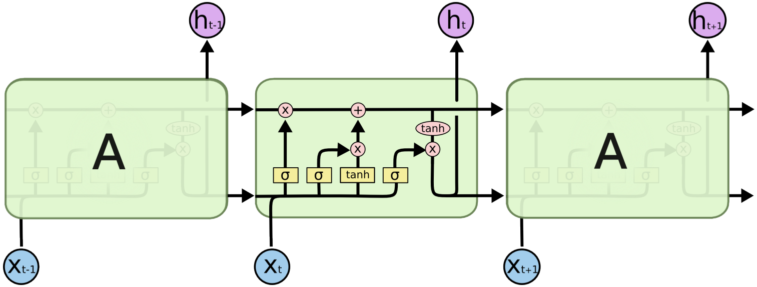
$$
\boldsymbol{y_j} = O_{\text{LSTM}}(\boldsymbol{s_j}) = \boldsymbol{h_j}
$$

$$
\boldsymbol{s_j} \in \mathbb{R}^{2 \cdot d_h},\ \ \boldsymbol{x_i} \in \mathbb{R}^{d_x},\ \ \boldsymbol{c_j}, \boldsymbol{h_j}, \boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{z} \in \mathbb{R}^{d_h},\ \ \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_h},\ \ \boldsymbol{W^{h\circ}} \in \mathbb{R}^{d_h \times d_h}.
$$

LSTM

- The state at time j is composed of two vectors, $\vec{c}_j$ and $\vec{h}_j$, where $\vec{c}_j$ is the memory component and $\vec{h}_j$ is the hidden state component.
- There are three gates, $\vec{i}$, $\vec{f}$, and $\vec{o}$, controlling for **i**nput, **f**orget, and **o**utput.
- The gate values are computed based on linear combinations of the current input $\vec{x}_j$ and the previous state $\vec{h}_{j-1}$, passed through a sigmoid activation function.
- An update candidate $\vec{z}$ is computed as a linear combination of $\vec{x}_j$ and $\vec{h}_{j-1}$, passed through a tanh activation function (to push the values to be between -1 and 1).
- The memory $\vec{c}_j$ is then updated: the forget gate controls how much of the previous memory to keep ($\vec{f} \odot \vec{c}_{j-1}$), and the input gate controls how much of the proposed update to keep ($\vec{i} \odot \vec{z}$).
- Finally, the value of $\vec{h}_j$ (which is also the output $\vec{y}_j$) is determined based on the content of the memory $\vec{c}_j$, passed through a tanh nonlinearity and controlled by the output gate.
- The gating mechanisms allow for gradients related to the memory part $\vec{c}_j$ to stay high across very long time ranges.

LSTM



[2]source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

LSTM

- Intuitively, recurrent neural networks can be thought of as very deep feed-forward networks, with shared parameters across different layers.
- For the Simple-RNN, the gradients then include repeated multiplication of the matrix W.
- This makes the gradient values to vanish or explode.
- The gating mechanism mitigate this problem to a large extent by getting rid of this repeated multiplication of a single matrix.
- LSTMs are currently the most successful type of RNN architecture, and they are responsible for many state-of-the-art sequence modeling results.
- The main competitor of the LSTM RNN is the GRU, to be discussed next.

GRU

- The LSTM architecture is very effective, but also quite complicated.
- The complexity of the system makes it hard to analyze, and also computationally expensive to work with.
- The gated recurrent unit (GRU) was introduced in [Cho et al., 2014] as an alternative to the LSTM.
- It was subsequently shown in [Chung et al., 2014] to perform comparably to the LSTM on several (non textual) datasets.
- Like the LSTM, the GRU is also based on a gating mechanism, but with substantially fewer gates and without a separate memory component.

GRU

$$
\begin{aligned}
\boldsymbol{s_j} = R_{\text{GRU}}(\boldsymbol{s_{j-1}}, \boldsymbol{x_j}) &= (\mathbf{1} - \boldsymbol{z}) \odot \boldsymbol{s_{j-1}} + \boldsymbol{z} \odot \tilde{\boldsymbol{s_j}} \\
\boldsymbol{z} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xz}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sz}}) \\
\boldsymbol{r} &= \sigma(\boldsymbol{x_j} \boldsymbol{W^{xr}} + \boldsymbol{s_{j-1}} \boldsymbol{W^{sr}}) \\
\tilde{\boldsymbol{s_j}} &= \tanh(\boldsymbol{x_j} \boldsymbol{W^{xs}} + (\boldsymbol{r} \odot \boldsymbol{s_{j-1}}) \boldsymbol{W^{sg}})
\end{aligned}
$$

$$
\boldsymbol{y_j} = O_{\text{GRU}}(\boldsymbol{s_j}) = \boldsymbol{s_j}
$$

$$
\boldsymbol{s_j}, \tilde{\boldsymbol{s_j}} \in \mathbb{R}^{d_s},\ \boldsymbol{x_i} \in \mathbb{R}^{d_x},\ \boldsymbol{z}, \boldsymbol{r} \in \mathbb{R}^{d_s},\ \boldsymbol{W^{x\circ}} \in \mathbb{R}^{d_x \times d_s},\ \boldsymbol{W^{s\circ}} \in \mathbb{R}^{d_s \times d_s}.
$$

GRU

- One gate $\vec{r}$ is used to control access to the previous state $\vec{s}_{j-1}$ and compute a proposed update $\tilde{\vec{s}}_j$.
- The updated state $\vec{s}_j$ (which also serves as the output $\vec{y}_j$) is then determined based on an interpolation of the previous state $\vec{s}_{j-1}$ and the proposal $\tilde{\vec{s}}_j$.
- The proportions of the interpolation are controlled using the gate $\vec{z}$.
- The GRU was shown to be effective in language modeling and machine translation.
- However, the jury is still out between the GRU, the LSTM and possible alternative RNN architectures, and the subject is actively researched.
- For an empirical exploration of the GRU and the LSTM architectures, see [Jozefowicz et al., 2015].

Sentiment Classification with RNNs

- The simplest use of RNNs is as acceptors: read in an input sequence, and produce a binary or multi-class answer at the end.
- RNNs are very strong sequence learners, and can pick-up on very intricate patterns in the data.
- An example of naturally occurring positive and negative sentences in the movie-reviews domain would be the following:
- Positive: It's not life-affirming—it's vulgar and mean, but I liked it.
- Negative: It's a disappointing that it only manages to be decent instead of dead brilliant.

Sentiment Classification with RNNs

- Note that the positive example contains some negative phrases (not life affirming, vulgar, and mean).
- While the negative example contains some positive ones (dead brilliant).
- Correctly predicting the sentiment requires understanding not only the individual phrases but also the context in which they occur, linguistic constructs such as negation, and the overall structure of the sentence.

Sentiment Classification with RNNs

- The sentence-level sentiment classification task is modelled using an RNN-acceptor.
- After tokenization, the RNN reads in the words of the sentence one at a time.
- The final RNN state is then fed into an MLP followed by a softmax-layer with two outputs (positive and negative).
- The network is trained with cross-entropy loss based on the gold sentiment labels.

$$\begin{aligned} p(label = k \mid \vec{w}_{1:n}) &= \hat{\vec{y}}_{[k]} \\ \hat{\vec{y}} &= softmax(MLP(RNN(\vec{x}_{1:n}))) \\ \vec{x}_{1:n} &= E_{[w_1]}, \ldots, E_{[w_n]} \end{aligned} \tag{5}$$

Sentiment Classification with RNNs

- The word embeddings matrix E is initialized using pre-trained embeddings learned over a large external corpus using an algorithm such as Word2vec or Glove with a relatively wide window.
- It is often helpful to extend the model by considering bidirectional RNNs.
- For longer sentences, [Li et al., 2015] found it useful to use a hierarchical architecture, in which the sentence is split into smaller spans based on punctuation.
- Then, each span is fed into a bidirectional RNN.
- Sequence of resulting vectors (one for each span) are then fed into an RNN acceptor.
- A similar hierarchical architecture was used for document-level sentiment classification in [Tang et al., 2015].

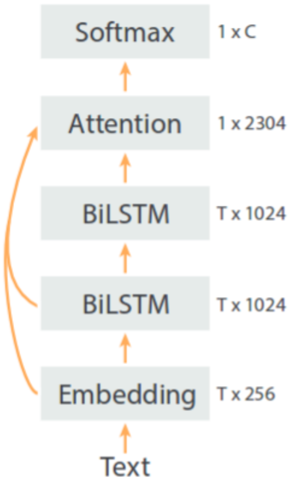
Twitter Sentiment Classification with LSTMS Emojis

- An emoji-based distant supervision model for detecting sentiment and other affective states from short social media messages was proposed in [Felbo et al., 2017].
- Emojis are used as a distant supervision approach for various affect detection tasks (e.g., emotion, sentiment, sarcasm) using a large corpus of 634M tweets with 64 emojis.
- A neural network architecture is pretrained with this corpus.
- The network is an LSTM variant formed by an embedding layer, 2 bidrectional LSTM layers with normal skip connections and temporal average pooling-skip connections.

DeepEmoji

I love mom's cooking	😋 49.1%	😍 8.8%	❤️ 3.1%	☺️ 3.0%	♡ 2.9%
I love how you never reply back..	😒 14.0%	😑 8.3%	😠 6.3%	😐 5.4%	💔 5.1%
I love cruising with my homies	😎 34.0%	👌 6.6%	✌️ 5.7%	😌 4.1%	💯 3.8%
I love messing with yo mind!!	😜 17.2%	😈 11.8%	😏 8.0%	😉 6.4%	🙊 5.3%
I love you and now you're just gone..	💔 39.1%	😔 11.0%	😞 7.3%	😪 5.3%	😢 4.5%
This is shit	😠 7.0%	😡 6.4%	😞 6.0%	😒 6.0%	😤 5.8%
This is the shit	🎧 10.9%	🎶 9.7%	👌 6.5%	😎 5.7%	😏 4.8%

DeepEmoji



Softmax
1 x C
Attention
1 x 2304
BiLSTM
T x 1024
BiLSTM
T x 1024
Embedding
T x 256
Text

Twitter Sentiment Classification with LSTMS Emojis

- Authors propose the chain-thaw transfer-learning approach in which the pretrained network is fine-tuned for the target task.
- Here, each layer is individually fine-tuned in each step with the target gold data, and then they are all fine-tuned together.
- The model achieves state-of-the-art results the detection of emotion, sentiment, and sarcasm.
- The pretrained network is released to the public.
- A demo of the model: `https://deepmoji.mit.edu/`.

DeepEmoji

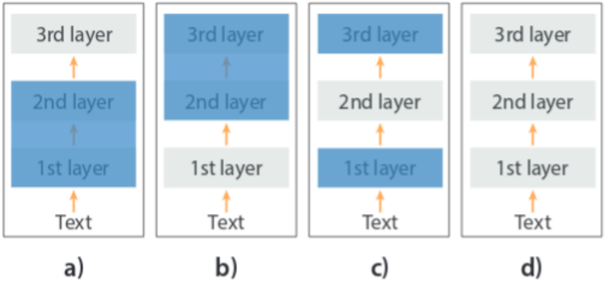


Figure 2: Illustration of the chain-thaw transfer learning approach, where each layer is fine-tuned separately. Layers covered with a blue rectangle are frozen. Step a) tunes any new layers, b) then tunes the 1st layer and c) the next layer until all layers have been fine-tuned individually. Lastly, in step d) all layers are fine-tuned together.

Bi-LSTM CRF

- An alternative approach to the transducer for sequence labeling is to combine a BI-LSTM with a Conditional Random Field (CRF).
- This produces a powerful tagger [Huang et al., 2015] called BI-LSTM-CRF, in which the LSTM acts as feature extractor, and the CRF as a special layer that models the transitions from one tag to another.
- The CRF layers performs a global normalization over all possible sequences which helps to find the optimum sequence.
- In a CRF we model the conditional probability of the output tag sequence given the input sequence:

$$P(s_1, \ldots, s_m | x_1, \ldots, x_m) = P(s_{1:m} | x_{1:m})$$

Bi-LSTM CRF

- We do this by defining a feature map

$$\vec{\Phi}(x_{1:m}, s_{1:m}) \in \mathcal{R}^d$$

  that maps an entire input sequence $x_{1:m}$ paired with an entire tag sequence $s_{1:m}$ to some d-dimensional feature vector.
- Then we can model the probability as a log-linear model with the parameter vector $\vec{w} \in \mathcal{R}^d$:

$$P(s_{1:m}|x_{1:m}; \vec{w}) = \frac{\exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}))}{\sum_{s'_{1:m} \in S^m} \exp(\vec{w} \cdot \vec{\Phi}(x_{1:m}, s'_{1:m}))}$$

  where $s'_{1:m}$ ranges over all possible output sequences.
- We can view the expression $\vec{w} \cdot \vec{\Phi}(x_{1:m}, s_{1:m}) = \text{score}_{crf}(x_{1:m}, s_{1:m})$ as a score of how well the tag sequence fits the given input sequence.
- Recall from the CRF lecture that $\vec{\Phi}(x_{1:m}, s_{1:m})$ was created with manually designed features.

Bi-LSTM CRF

- The idea of the LSTM CRF is to replace $\vec{\Phi}(x_{1:m}, s_{1:m})$ with the output of an LSTM transducer (automatically learned features):

$$\text{score}_{lstm-crf}(x_{1:m}, s_{1:m}) = \sum_{i=0}^{m} W_{[s_{i-1}, s_i]} \cdot \text{LSTM}^*(x_{1:m})_{[i]} + \vec{b}[s_{i-1}, s_i]$$

  where $W_{[s_{i-1}, s_i]}$ corresponds to a transition score from tag s_{i-1} to s_i, $\vec{b}[s_{i-1}, s_i]$ is a bias term for the transition and $\text{LSTM}^*(x_{1:m})_{[i]}$ comes from the hidden state of the Bi-LSTM at timestep i.
- All these parameters are learned jointly.
- Note that the transition matrix W is position independent.
- The forward-backward algorithm is used during training and the Viterbi algorithm during decoding.
- More info in[3] and[4].

[3] https://pytorch.org/tutorials/beginner/nlp/advanced_tutorial.html

[4] https://www.depends-on-the-definition.com/sequence-tagging-lstm-crf/

Questions?

Thanks for your Attention!

References I

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014).
Learning phrase representations using rnn encoder-decoder for statistical machine translation.
arXiv preprint arXiv:1406.1078.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014).
Empirical evaluation of gated recurrent neural networks on sequence modeling.
arXiv preprint arXiv:1412.3555.

Felbo, B., Mislove, A., Søgaard, A., Rahwan, I., and Lehmann, S. (2017).
Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm.
In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1615–1625.

Goldberg, Y. (2016).
A primer on neural network models for natural language processing.
J. Artif. Intell. Res.(JAIR), 57:345–420.

Hochreiter, S. and Schmidhuber, J. (1997).
Long short-term memory.
Neural computation, 9(8):1735–1780.

References II

Huang, Z., Xu, W., and Yu, K. (2015).
Bidirectional lstm-crf models for sequence tagging.
arXiv preprint arXiv:1508.01991.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015).
An empirical exploration of recurrent network architectures.
In *International conference on machine learning*, pages 2342–2350.

Li, J., Luong, M.-T., Jurafsky, D., and Hovy, E. (2015).
When are tree structures necessary for deep learning of representations?
arXiv preprint arXiv:1503.00185.

Tang, D., Qin, B., and Liu, T. (2015).
Document modeling with gated recurrent neural network for sentiment classification.
In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 1422–1432.