UE 803 - Data Science for NLP

Lecture 15: Neural Sequence Processing

Claire Gardent - CNRS / LORIA

Sequence Processing

- Sequence Processing in NLP
    - Which sequences ?
    - Which tasks ?
- Recurrent Neural Networks
    - Basic RNN
    - RNN with Selective memories: LSTMs and GRUs
    - Bi-directional RNNs

Sequence Processing in NLP

Why are sequences important ?

- Speech is a sequence of sounds
- Texts are sequences of words

"This morning I took my cat for a walk."

Sequence Processing Tasks

- Language modeling
- Part of speech tagging
- Chunking
- Named Entity Recognition

Language Modeling

Determines the probability of a ***sequence of words***

$$P(W) = P(w1, w2, w3...wn)$$

Computed using the chain Rule of Probability

$$P(w1, w2, \ldots, wn) = \prod_i P(wi|w1, w2, \ldots, ii-1)$$

Can be used to predict the next word

France is where I grew up and where I now work. I speak fluent ??

Ex (in english)

$P_1 = P($"a quick brown dog"$)$

$P_2 = P($"dog quick a brown"$)$

$P_3 = P($"un chien quick brown"$)$

$P_4 = P($"un chien brun rapide"$)$

- $P_1 > P_2 > P_3 > P_4$

Example

$P(\text{its water is so transparent}) =$
$P(\text{its}) \times P(\text{ water } | \text{ its}) \times P(\text{ is } | \text{ its water })$
$\times P(\text{so } | \text{ its water is }) \times (\text{ transparent } | \text{ its water is so})$

Language Modeling

LM is a key component of many applications

- Machine Translation

$$P(\text{high winds tonight}) > P(\text{large winds tonight})$$

- Spell Correction:
  Sentence: *The office is about fifteen minuets from my house*

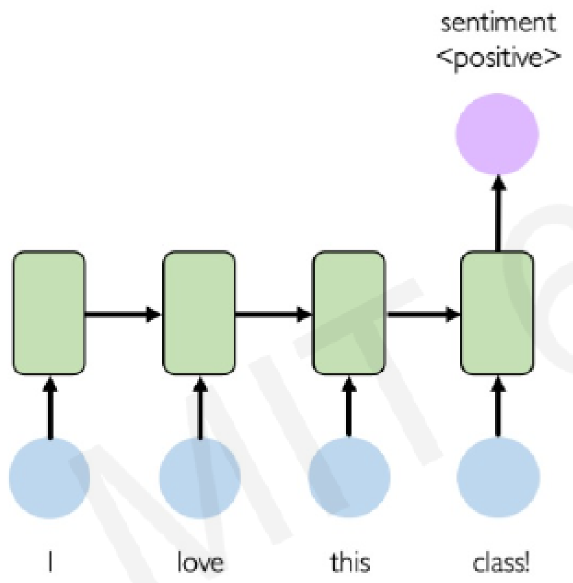
$$P(\text{about fifteen minutes from}) > P(\text{about fifteen minuets from})$$

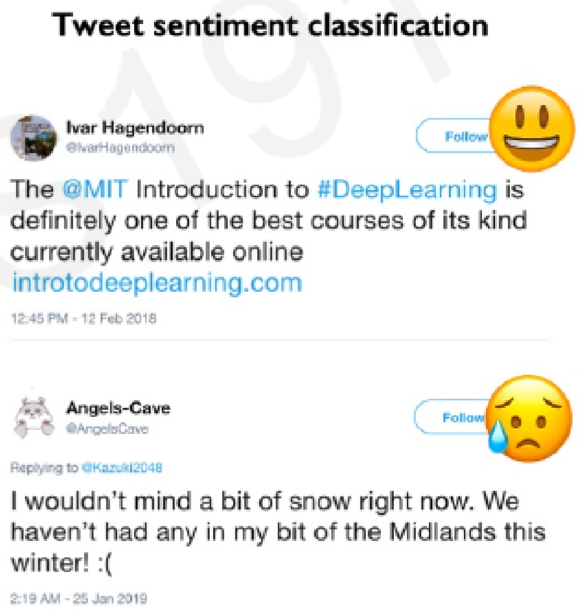
- Speech Recognition:

$$P(\text{I saw a van}) >> P(\text{eyes awe of an})$$

Summarization, question- answering, etc., etc.!

Sentiment Analysis




Tweet sentiment classification

Ivar Hagendoorn
@IvarHagendoorn
Follow

The @MIT Introduction to #DeepLearning is definitely one of the best courses of its kind currently available online introtodeeplearning.com

12:45 PM - 12 Feb 2018

Angels-Cave
@AngelsCave
Follow

Replying to @Kazuki2048

I wouldn't mind a bit of snow right now. We haven't had any in my bit of the Midlands this winter! :(

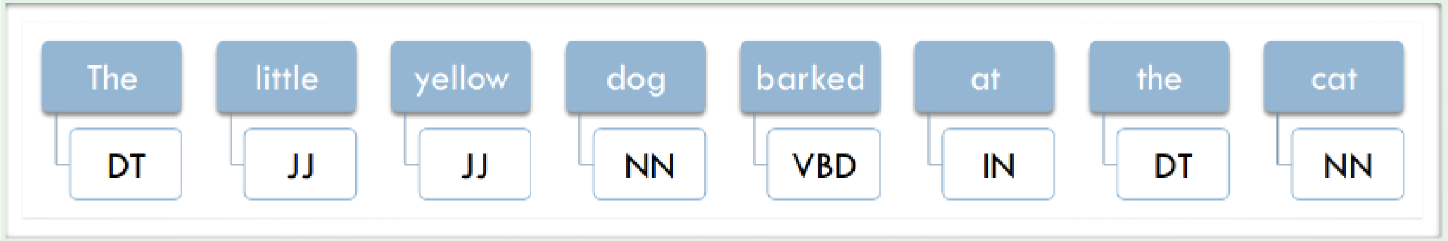
2:19 AM - 25 Jan 2019

MIT 6.S191

Tagging sequences of word with a sentiment

POS tagging

Example

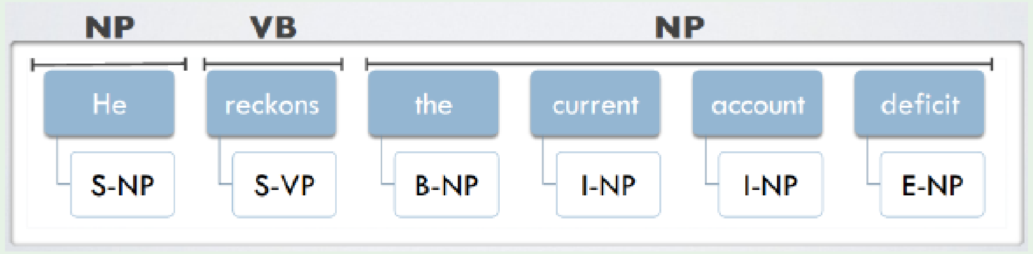


(from Stanislas Lauly)

Tagging sequences of word with a Part-of-Speech (PoS) tag

Chunking

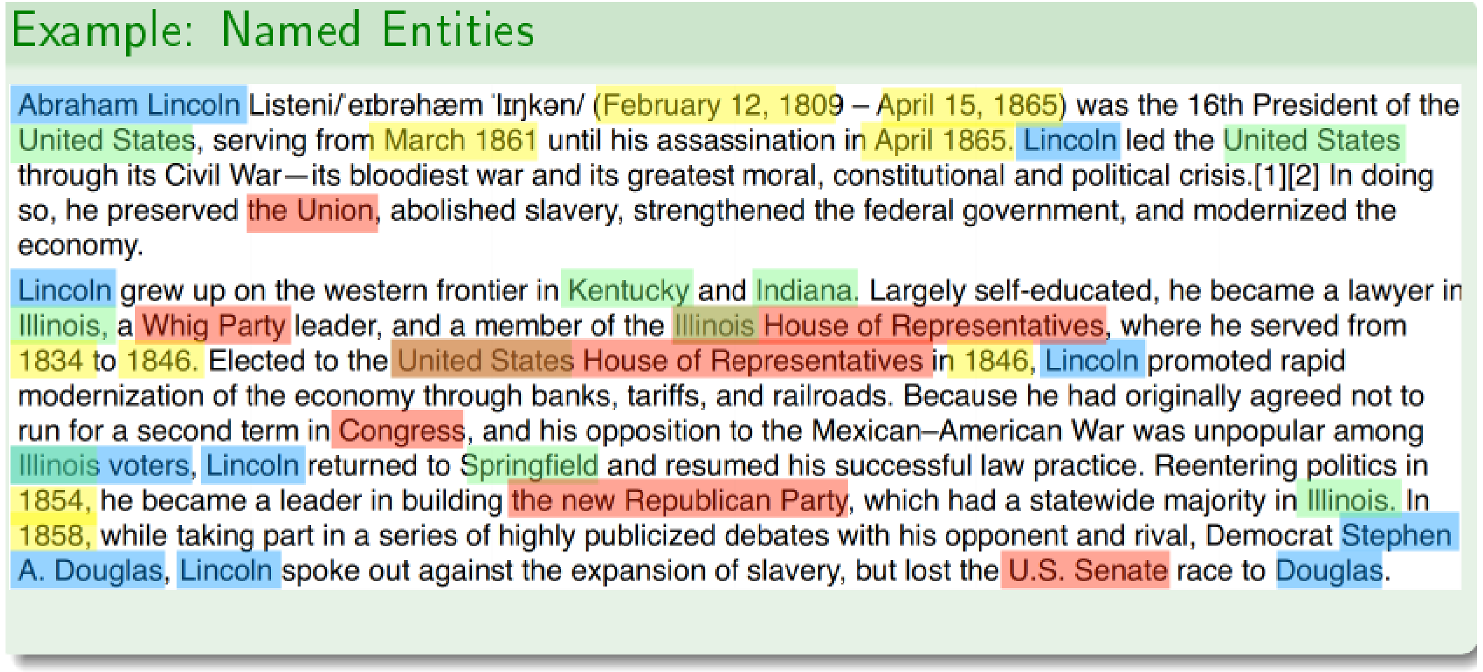
Example



(from Stanislas Lauly)

Tagging chunks (sequences of words) with a syntactic tag

Named Entity Recognition



Example: Named Entities

Abraham Lincoln Listeni/ˈeɪbrəhæm ˈlɪŋkən/ (February 12, 1809 – April 15, 1865) was the 16th President of the United States, serving from March 1861 until his assassination in April 1865. Lincoln led the United States through its Civil War—its bloodiest war and its greatest moral, constitutional and political crisis.[1][2] In doing so, he preserved the Union, abolished slavery, strengthened the federal government, and modernized the economy.

Lincoln grew up on the western frontier in Kentucky and Indiana. Largely self-educated, he became a lawyer in Illinois, a Whig Party leader, and a member of the Illinois House of Representatives, where he served from 1834 to 1846. Elected to the United States House of Representatives in 1846, Lincoln promoted rapid modernization of the economy through banks, tariffs, and railroads. Because he had originally agreed not to run for a second term in Congress, and his opposition to the Mexican–American War was unpopular among Illinois voters, Lincoln returned to Springfield and resumed his successful law practice. Reentering politics in 1854, he became a leader in building the new Republican Party, which had a statewide majority in Illinois. In 1858, while taking part in a series of highly publicized debates with his opponent and rival, Democrat Stephen A. Douglas, Lincoln spoke out against the expansion of slavery, but lost the U.S. Senate race to Douglas.

Tagging sequences of word with a semantic tag (PERSON, LOCATION etc.)

Named Entity Recognition

exemple

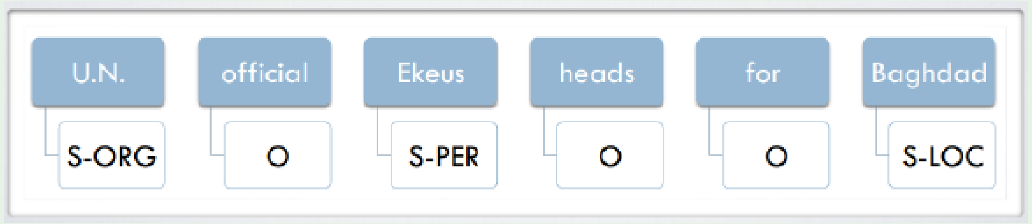


Figure: (from Stanislas Lauly)

Semantic Role Labelling

Example

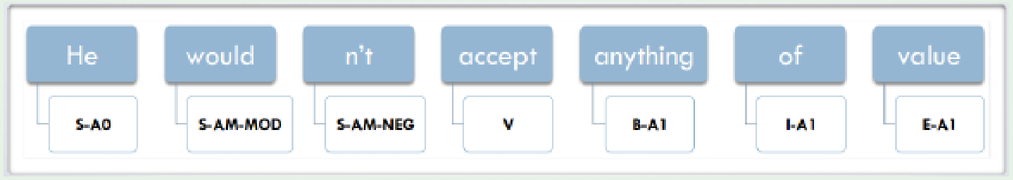
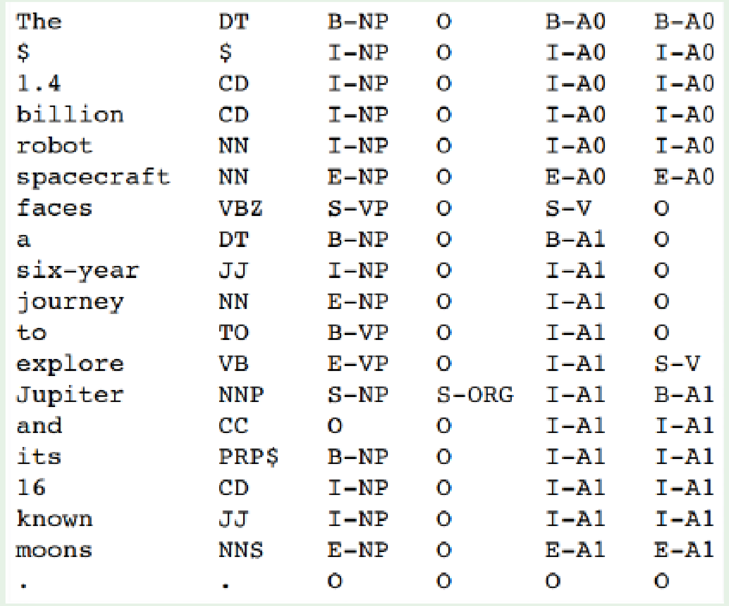


Figure: (from Stanislas Lauly)

Data labelled with multiple annotations

Example: SPMRL format



The	DT	B-NP	O	B-A0	B-A0
$	$	I-NP	O	I-A0	I-A0
1.4	CD	I-NP	O	I-A0	I-A0
billion	CD	I-NP	O	I-A0	I-A0
robot	NN	I-NP	O	I-A0	I-A0
spacecraft	NN	E-NP	O	E-A0	E-A0
faces	VBZ	S-VP	O	S-V	O
a	DT	B-NP	O	B-A1	O
six-year	JJ	I-NP	O	I-A1	O
journey	NN	E-NP	O	I-A1	O
to	TO	B-VP	O	I-A1	O
explore	VB	E-VP	O	I-A1	S-V
Jupiter	NNP	S-NP	S-ORG	I-A1	B-A1
and	CC	O	O	I-A1	I-A1
its	PRP$	B-NP	O	I-A1	I-A1
16	CD	I-NP	O	I-A1	I-A1
known	JJ	I-NP	O	I-A1	I-A1
moons	NNS	E-NP	O	E-A1	E-A1
.	.	O	O	O	O

B (Begin), I (Inside), E (End), S (Single)

Recurrent Neural Networks

Variable Lengths

The weather is great

The weather is good but not great

When the weather is good, I like to good for long walks in the woods.

Natural language sequences have ***variable lengths***

Long-term Dependencies

France is where I grew up and where I now work. I speak fluent __

Information from the distant pass restricts possible continuations

There are ***long-term dependencies*** between words

Sequence Order

The weather was good, not bad at all

The weather was bad, not good at all

Order matters

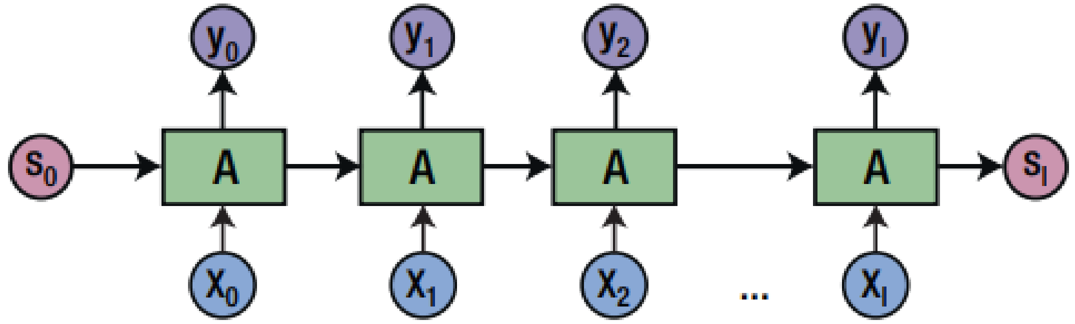
Recurrent Networks (RNN)

Recurrent neural network are well-suited for sequence processing (and therefore for NLP) because they naturally

- handle variable-length sequences
- track long-term dependencies
- maintain information about order

Recurrent Neural Networks

- A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor
- In other words RNNs pass on information left-to-right thru a sequence; they "memorise" the preceding context



RNN Input and Output

Input

- an input unit x_i e.g., a word vector
- a hidden state h_j

Output

- an output $\hat{y_i}$
- a new hidden state h_i

$x_i, h_i, h_j, \hat{y_i}$ are vectors

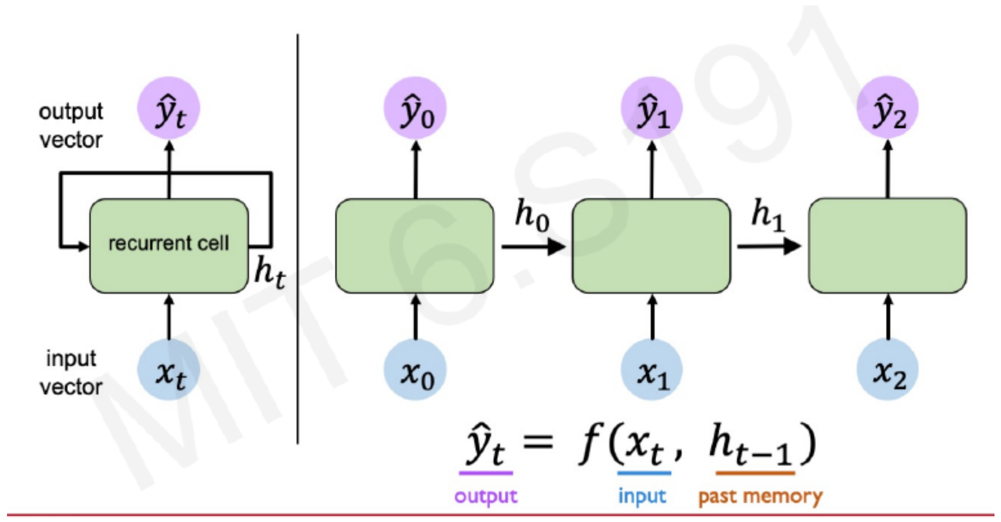
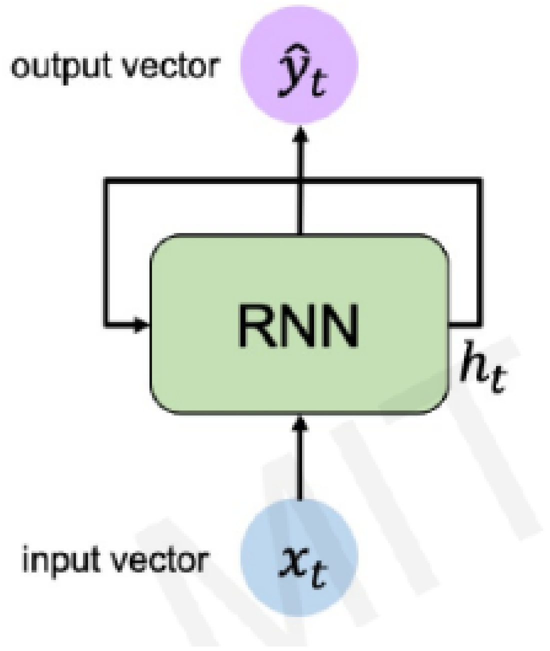


Figure from MIT 6S191 Course

RNN Hidden State



Input Vector

$$x_t$$

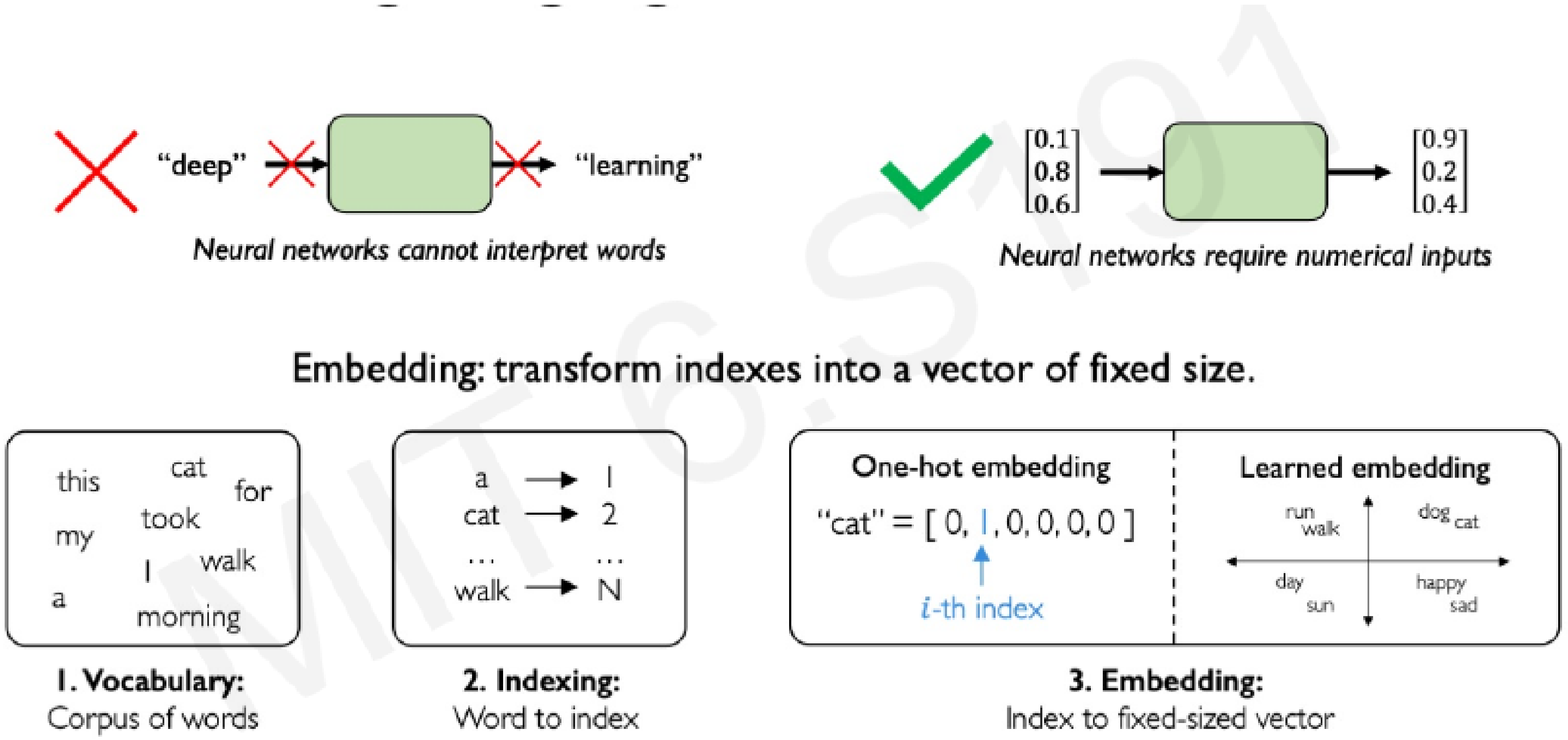
Output Hidden State

$$h_t = tanh(W_{hh}^\top h_{t-1} + W_{xh}^\top x_t)$$

Output

$$\hat{y}_t = softmax(W_{hy}^\top h_t)$$

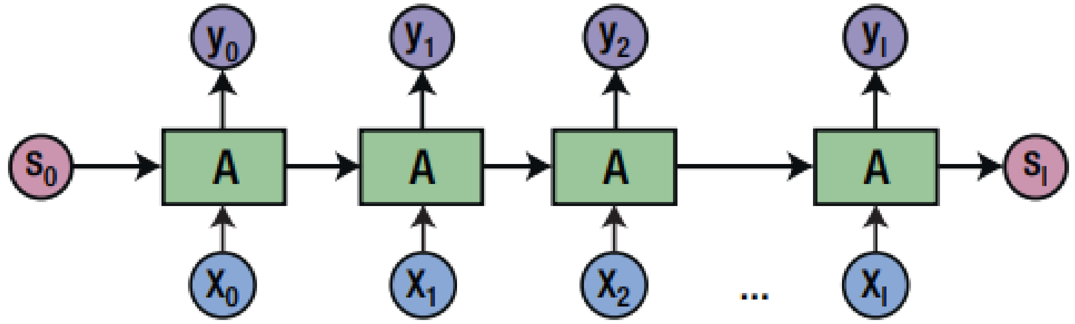
RNN Input



"deep"
"learning"
Neural networks cannot interpret words
$\begin{bmatrix}0.1\\0.8\\0.6\end{bmatrix}$
$\begin{bmatrix}0.9\\0.2\\0.4\end{bmatrix}$
Neural networks require numerical inputs
Embedding: transform indexes into a vector of fixed size.
this cat for my took I walk a morning
a → 1
cat → 2
... ...
walk → N
One-hot embedding
"cat" = [0, 1, 0, 0, 0, 0]
i-th index
Learned embedding
run walk dog cat day sun happy sad
1. Vocabulary:
Corpus of words
2. Indexing:
Word to index
3. Embedding:
Index to fixed-sized vector
MIT 6.S191

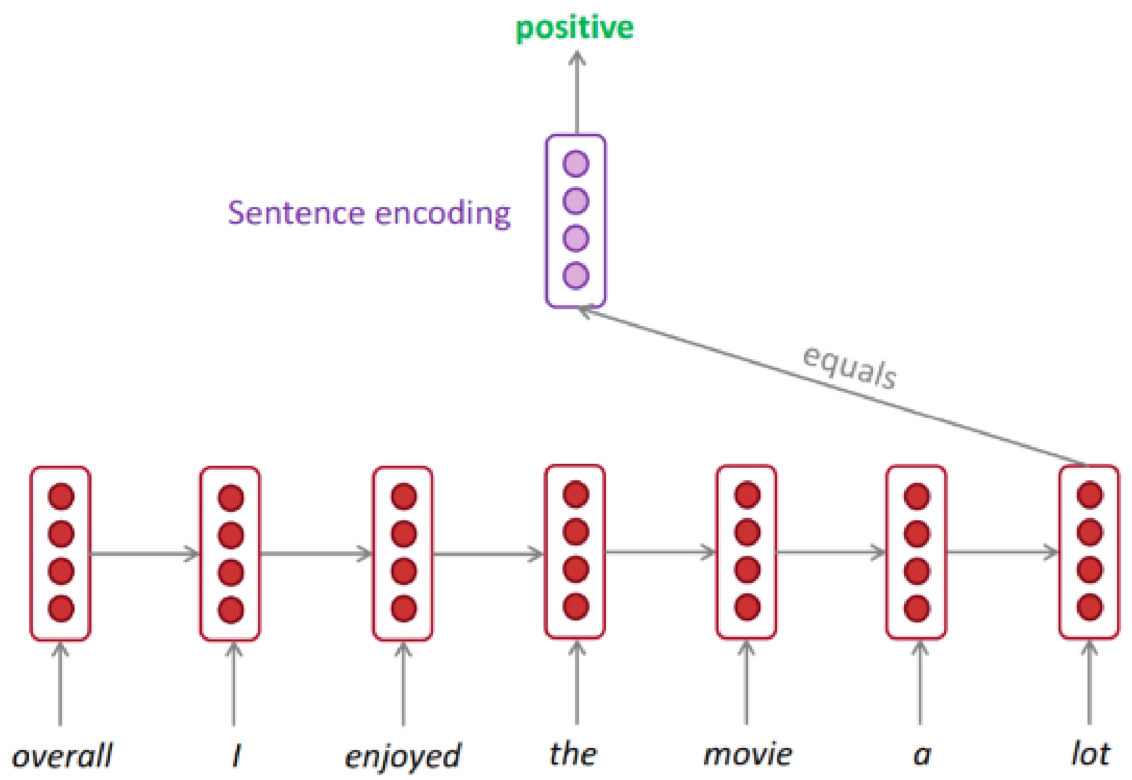
Figure from MIT 6S191 Course

RNN used for classification

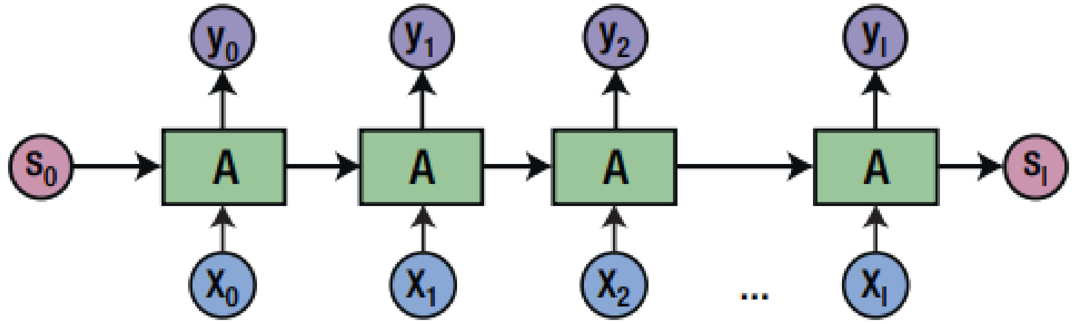


- Use the last hidden state of the RNN (S_l) as sentence representation
- Use a softmax layer to project this ***last hidden state*** into a layer whose values will specify a probability distribution over the set of possible output classes

RNN for Sentiment Analysis

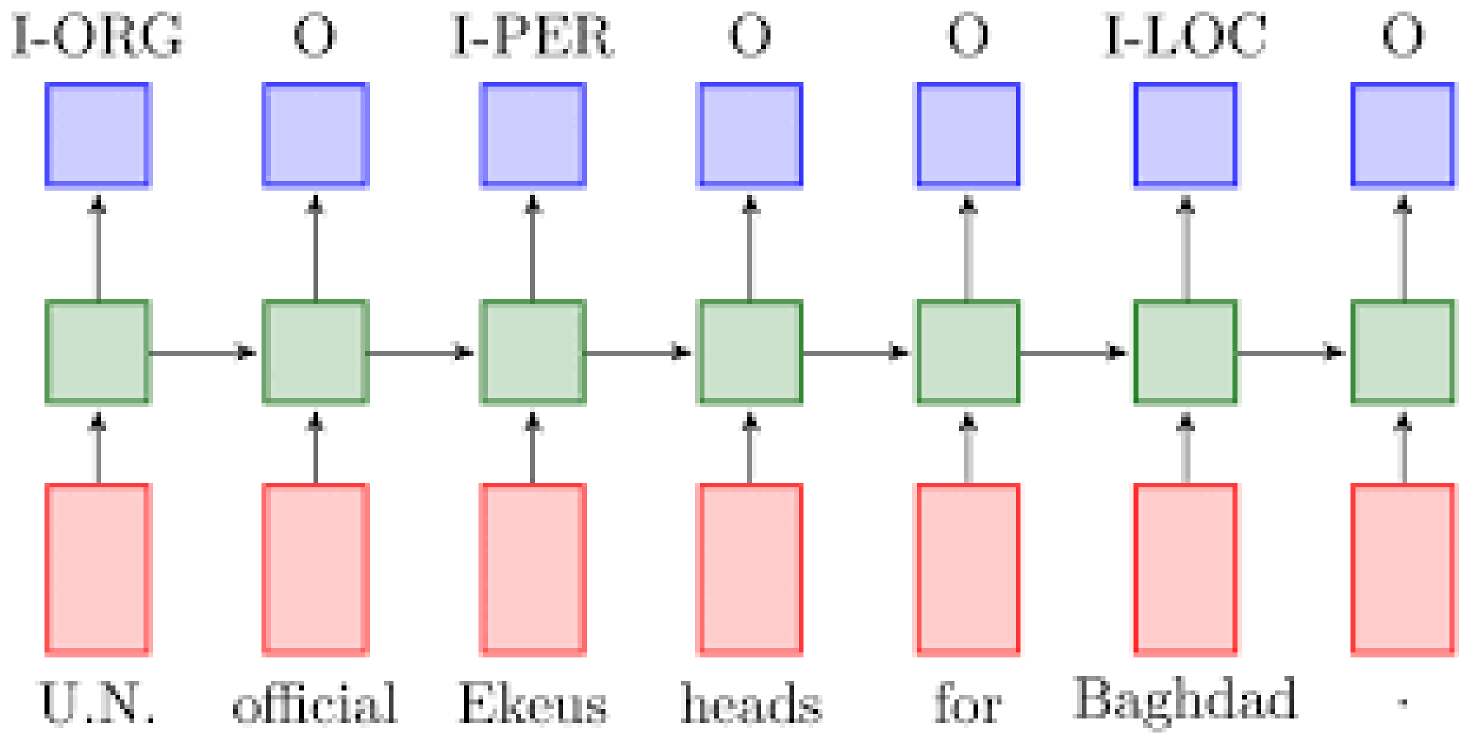


RNN used for sequence tagging



Use a softmax layer to project ***each hidden state*** of the RNN into a layer whose values will specify a probability distribution over the set of possible output classes

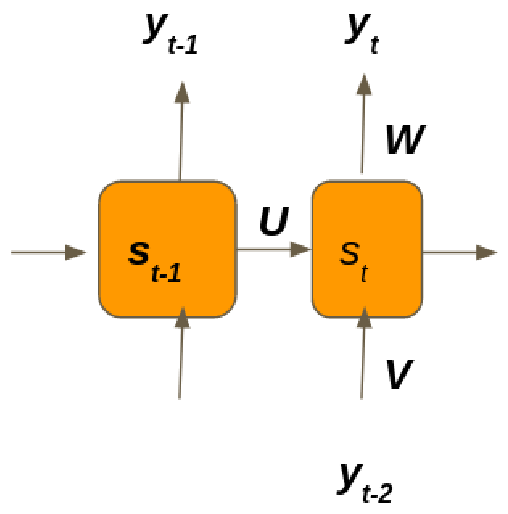
At each encoding step the RNN ***outputs a prediction*** in addition to a new hidden state.



RNN used for Sequence Tagging

At each time step

- a new hidden state (s_t) is computed based on the previous hidden state and on the current input .
- a prediction is output based on the new hidden state and on a softmax layer



$$
\begin{aligned}
s_t &= tanh(U * s_{t-1} + V * y_{t-1}) \\
y_t &= softmax(W * s_t)
\end{aligned}
$$

RNN, LSTM and GRU

Vanishing and Exploding Gradients

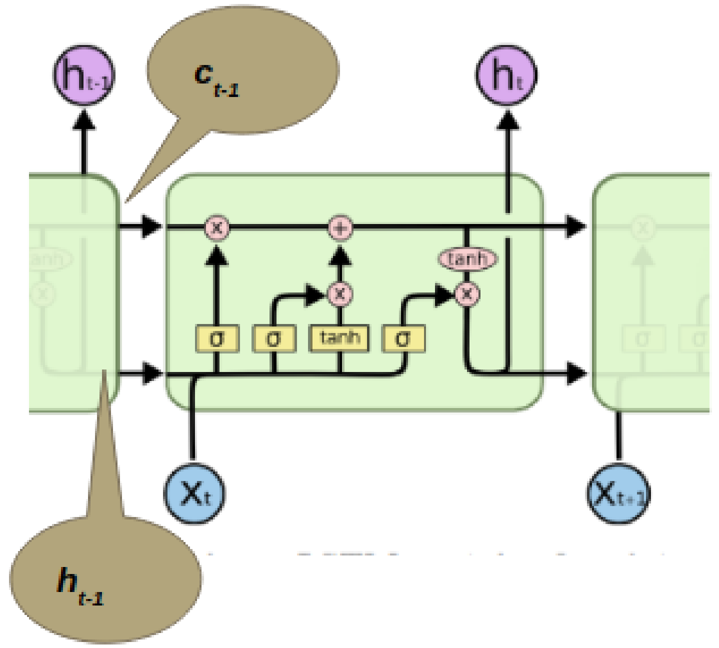
- In practice, RNNs cannot handle long input because of the Vanishing and Exploding Gradients issue [Bengio et al. 1994] (the gradients become either very large or very small so that either learning crashes or never converges).

- RNNs with selective memories were introduced to adress that issue: instead of passing the previous hidden state as is, these new RNNs learn to decide what information should be memorised and what information can be forgotten.

- Two main type of such RNNs:

    - LSTM Long Short Term Memory Network
    - GRU Gated Recurrent Unit

LSTM



$$\begin{aligned}
& \tilde{c}_t &=& \; tanh(W_c * [h_{t-1}, x_t]) \\
\text{update} \quad & u_t &=& \; \sigma(W_t * [h_{t-1}, x_t]) \\
\text{forget} \quad & f_t &=& \; \sigma(W_f * [h_{t-1}, x_t]) \\
\text{output} \quad & o_t &=& \; \sigma(W_o * [h_{t-1}, x_t]) \\
& c_t &=& \; u_t * \tilde{c}_t + f_t * c_{t-1} \\
& h_t &=& \; o_t * tanh(c_t)
\end{aligned}$$

- The ***cell*** keeps track of what is kept, forgotten and updated
- Gates modulate which information gets through.
  The ***gates sigmoid layer*** outputs numbers between 0 and 1, describing how much of each component should be let through (0 = “forget”, 1 = “keep”).

LSTM

- Create a candidate cell

$$\tilde{c}_t = tanh(W^\top h_{t-1} + W^\top x_t)$$

- Apply a gate to this candidate cell and to the previous cell c_{t-1} and sum the result to create a new cell c_t

$$u_t = \sigma(W_u^\top h_{t-1} + W_u^\top x_t)$$

$$f_t = \sigma(W_f^\top h_{t-1} + W_f^\top x_t)$$

$$c_t = u_t * \tilde{c}_t + f_t * \tilde{c}_{t-1}$$

- Create the new hidden state h_t by applying a third gate to the new cell

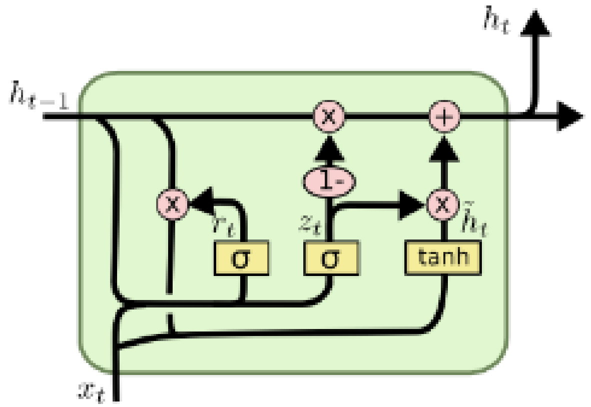
$$o_t = \sigma(W_o^\top h_{t-1} + W_o^\top x_t)$$

$$c_t = o_t * tanh(c_t)$$

See Colah's Blog for a detailed explanation.

GRU

A simpler way to decide what to forget and what to memorize



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

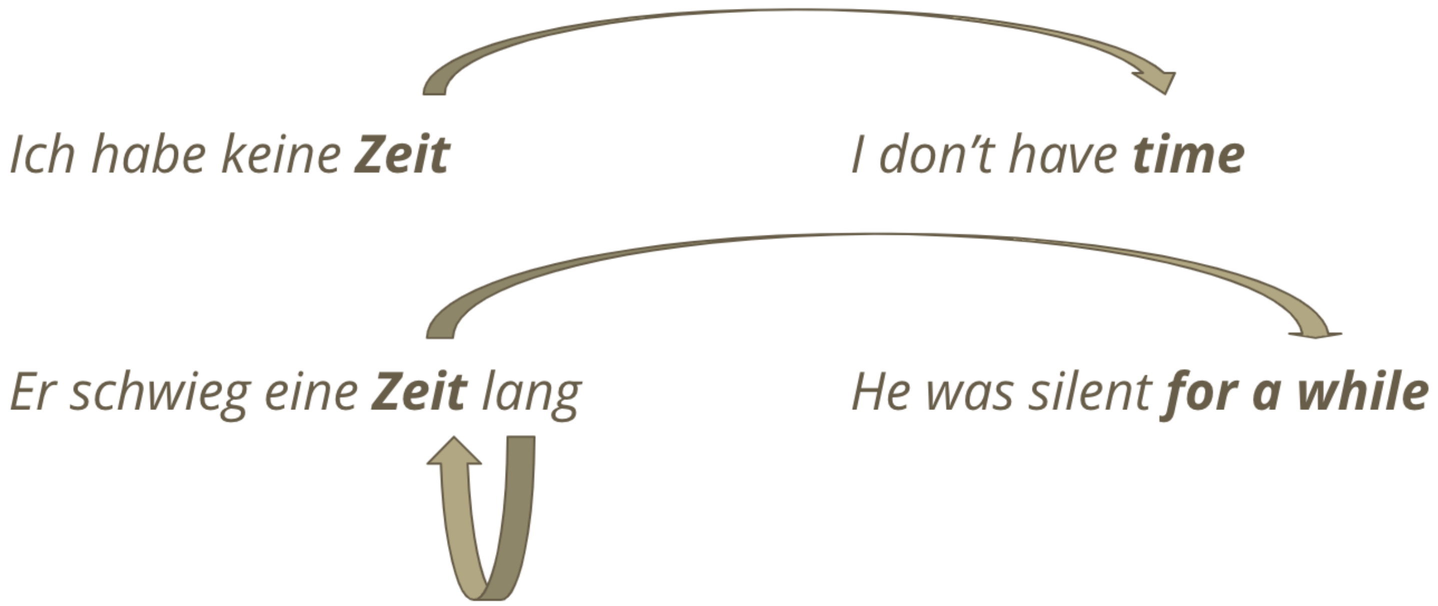
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- Creates a candidate state using a reset gate (r_t)
- Applies an update gate (z_t) to that candidate state
- Applies the corresponding opposite gate ($1 - z_t$) to the previous state
- Add the updated candidate state and previous state to create the output state

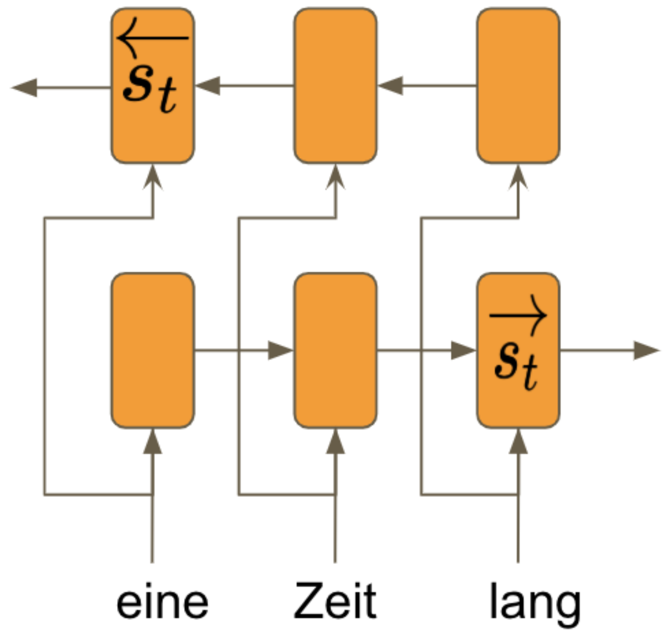
RNNs and Directionality

Bi-LSTM

Both left and right context matter


*Ich habe keine **Zeit***
*I don't have **time***
*Er schwieg eine **Zeit** lang*
*He was silent **for a while***

Bidirectional RNN

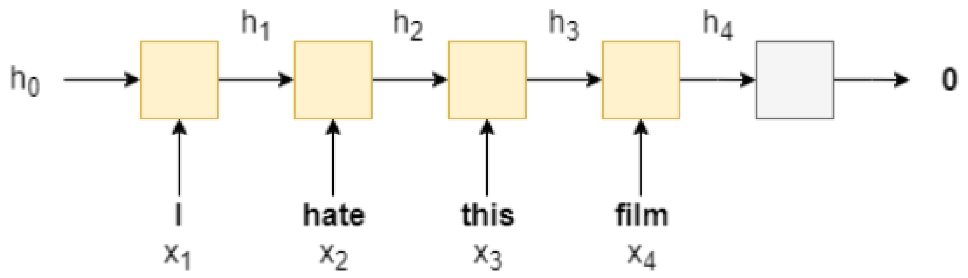


Input representation

$$[\overrightarrow{s_t}, \overleftarrow{s_t}]$$

- Forward RNN encodes left context
- Backward RNN encodes right context
- Forward and backward states are concatenated

RNNs in Pytorch



- Input: a sequence of words, X={x1,...,xT}
- The RNN processes each word , one at a time.
- Each time, the RNN takes as input the current word x_t and the hidden state from the previous word, h_{t-1} , to produce the next hidden state, h_t .
- The final hidden state, h_4 is passed through a linear layer, f, to produce the output class, $\hat{y} = f(h_4)$.

The same RNN is used for all words. The initial hidden state, h_0 , is a tensor initialized to all zeros.

RNN in Pytorch

Specifying the network configuration

```
class RNN(nn.Module):
    def __init__(self):
        super().__init__()

        # Here we define the network layers

        # An embedding layer which assigns
        # to each word in the vocabulary an embedding of size embed_size
        self.embed = nn.Embedding(len(vocab), embed_size)

        # A RNN layer to process each input token (represented by its embedding)
        self.rnn = nn.RNN(embed_size, hidden_size)

        # Drop out layer for regularisation
        self.dropout = nn.Dropout(0.3)

        # Fully connected layer mapping the last hidden state to a prediction
        self.decision = nn.Linear(hidden_size, len(set(labels)))
```

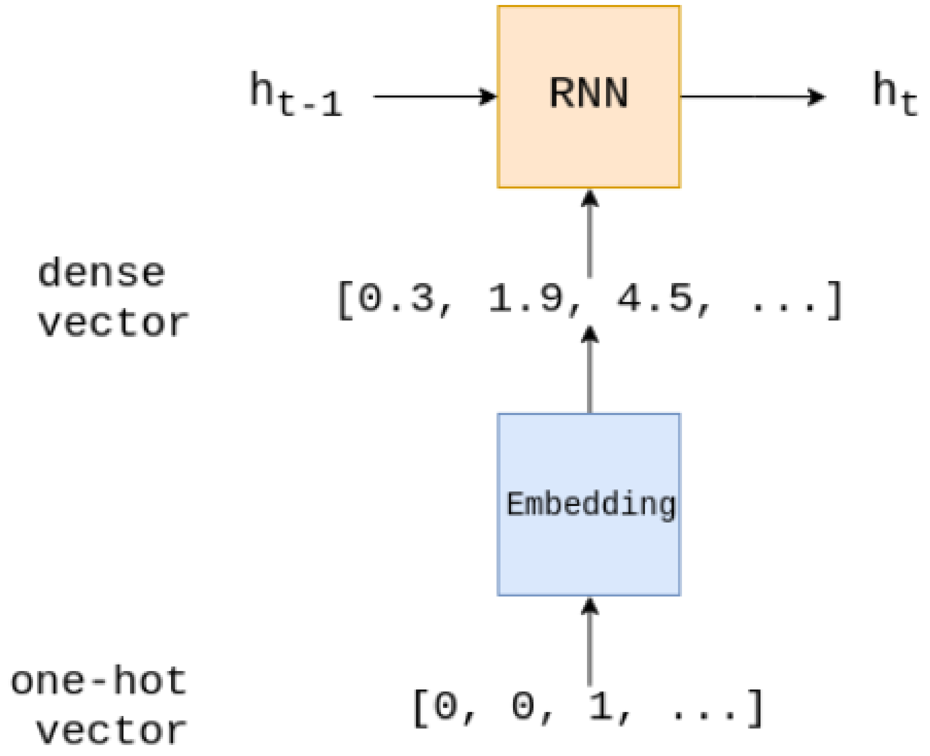
Embedding Layer

nn.Embedding

- A lookup table that stores embeddings (vectors) of a fixed dictionary.
- The input to this module is a list of indices
- The output is the corresponding list of word embeddings

```
# Creating the look up table
emb_table = nn.Embedding(len(vocab), embed_size)

# Retrieving the embedding for token x
embedding = emb_table(x)
```



RNN Layer

nn.GRU

- takes as input a word embedding and the previous hidden state
- outputs a new hidden state

```
# A recurrent (GRU) layer to process each input token (represented by its embedding)
# The GRU network takes as input the embedding of the current word
# and the previous hidden state

        self.rnn = nn.RNN(embed_size, hidden_size)
```

Output layer

```
# Fully connected layer mapping a hidden state to a vector of size the number of classes
self.decision = nn.Linear(hidden_size, len(set(labels)))
```

RNN in Pytorch

The `forward` method is called when examples are input to the model.

It applies to ***batches*** .

```
def forward(self, x):
        # Retrieve the embeddings of the current token x
        embed = self.embed(x)

       # Run the RNN on x's embedding
       output, hidden = self.rnn(embed)

       # Apply dropout (for regularisation)
       drop = self.dropout(hidden)

       # Return a prediction
       return self.decision(drop.squeeze(0))
```

Embedding Layer Input and Output Size

```
def forward(self, x):
   embed = self.embed(x)
```

- The input is processed in batches (multiple sentences)
- Each batch `x` is a tensor of size ***(sentence length, batch size)***
- The embedding layer assigns each input token a dense vector representation of size 'embedding dim'
- `embed`, the output of the embedding layer, is a tensor of size ***(sentence length, batch size, embedding dim)***

RNN Layer Input and Output Size

```
output, hidden = self.rnn(embed)
```

- The RNN layer returns 2 tensors `output` and `hidden`
- `output` is the concatenation of the hidden states for each time step. It has size ***(sentence length, batch size, hidden dim)***
- `hidden` is the final hidden state and has size ***(1, batch size, hidden dim)***

Output Layer Input and Output Size

```
# Apply the fully connected layer to the output of the dropout
# Input: (1, batch size, hidden dim)
# self.decision(drop): (batch size, 1)
# Output: (batch size)

return self.decision(drop.squeeze(0))
```

- The squeeze method is used to remove a dimension of size 1.
- The squeeze is needed as the predictions are initially size (batch size, 1), and we need to remove the dimension of size 1 as PyTorch expects the predictions input to the criterion function (`BCEWithLogitsLoss`) to be of size (batch size).

Training

Iterates over all examples, one batch at a time.

```
def train(model, iterator, optimizer, criterion):
```

The `iterator` iterates over the data and returns a batch of examples (indexed and converted into tensors) at each iteration.

The `optimizer` updates the parameters using stochastic gradient descent (SGD). The first argument is the parameters will be updated by the optimizer, the second is the learning rate

```
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=1e-3)
```

Training

The `criterion` is the loss function, here binary cross entropy with logits.

Given a model which outputs an unbound real number (a logit), the BCEWithLogitsLoss criterion converts the logits to a number between 0 and 1 (using the sigmoid or logit function) and calculate the loss using binary cross entropy.

```
criterion = nn.BCEWithLogitsLoss()
```

```python
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:
        # Zero the gradients.
        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

```
# Zero the gradients.
optimizer.zero_grad()
```

Each parameter in a model has a grad attribute which stores the gradient calculated by the criterion.

PyTorch does not automatically remove (or "zero") the gradients calculated from the last gradient calculation, so they must be manually zeroed

Evaluate

- Similar to train but the parameters are not updated.
    - There is no need to zero the gradients,
    - No back-propagation, no optimisation
- model.eval() turns off dropout and batch normalization.

Evaluate

```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

   # no gradients are computed
    with torch.no_grad():

        for batch in iterator:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)

            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Useful Links

- Lecture Slides on RNNs by Graham Neubig (CMU)
  Nice graphical explanation of how learning works for RNNs (backpropagation through time)
- Video of G. Neubig Class
- A well documented blog on sequence tagging