Natural Language Processing CSCI 4152/6509 — Lecture 21 Neural Network Models for NLP; Parsing NLP

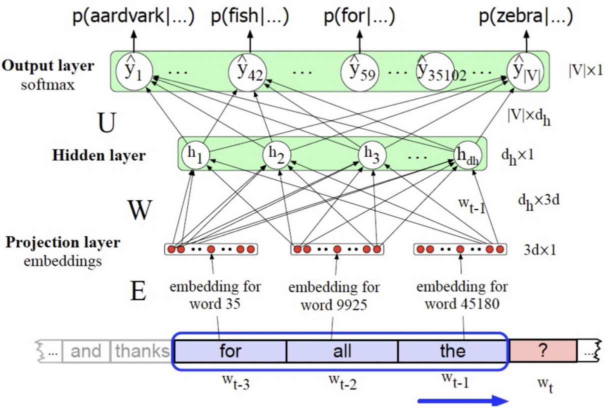
Instructors: Vlado Keselj
Time and date: 16:05 – 17:25, 21-Nov-2023
Location: Rowe 1011

Previous Lecture

Neural networks and deep learning

- Applications
- Some main developments
- Large deep learning models
- Exponential growth in size of LLMs
- Biological neuron, perceptron, feed-forward network
- Activation functions, softmax function

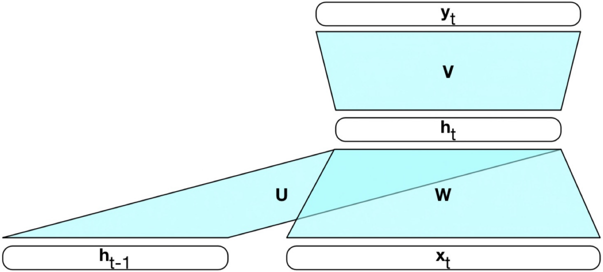
Neural Language Model



(Jurafsky and Martin)
The model has limited history, similarly to n-gram model

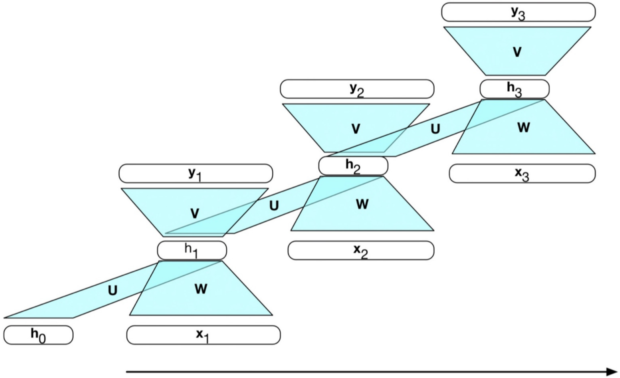
Recurrent Neural Networks (RNN)

- Simple recurrent neural network presented as a feedforward network (Jurafsky and Martin, Figure 9.3)
- RNN is trained as a Language model by providing the next word as output



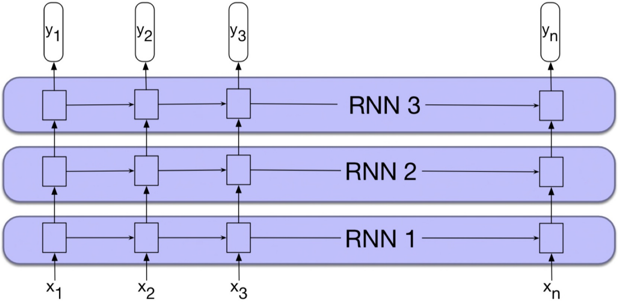
RNN Unrolled in Time

- RNN unrolled in time; more clear view of training (Jurafsky and Martin, Figure 9.5)



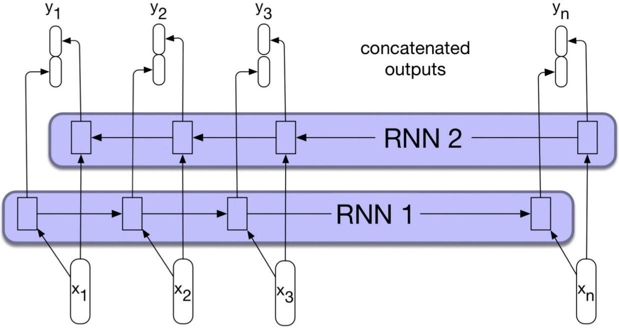
Stacked RNN

- Stacked RNN: Output from lower level is input to higher level; top level is final output (Jurafsky and Martin, Figure 9.10)



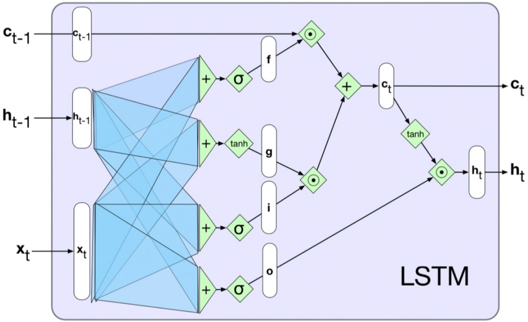
Bidirectional RNN

- Bidirectional RNN; trained forward and backward with concatenated output (Jurafsky and Martin, Figure 9.11)
- Output can be used for sequence labeling, for example



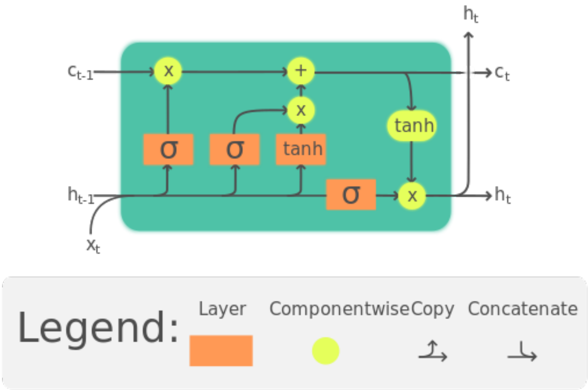
LSTM — Long Short-Term Memory

- LSTM: x_t is input, h_{t-1} is previous hidden state, c_{t-1} is previous long-term context, h_t and c_t is output (Jurafsky and Martin, Figure 9.13)



LSTM Cell

- Another view of LSTM cell (source Wikipedia)

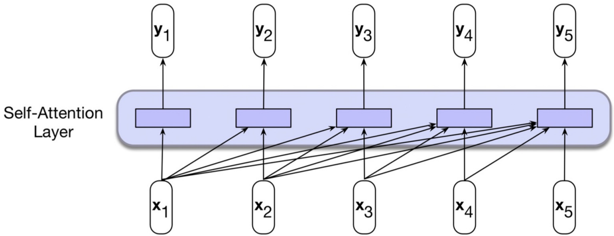


Transformers

- Transformers map a sequence of input vectors to a sequence of output vectors of the same length

$$\begin{array}{cccc} x_1 & x_2 & \ldots & x_n \\ \hline \downarrow & \downarrow & \vdots & \downarrow \\ \hline y_1 & y_2 & \ldots & y_n \end{array}$$

Self-Attention Layer



(Jurafsky and Martin)

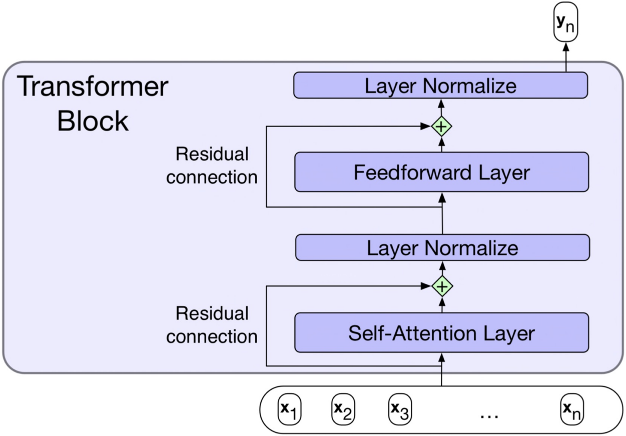
Self-Attention Training

$$\mathit{score}(x_i, x_j) = x_i \cdot x_j$$

$$\alpha_{ij} = \mathrm{softmax}(\mathit{score}(x_i, x_j)) \quad \forall j \leq i$$

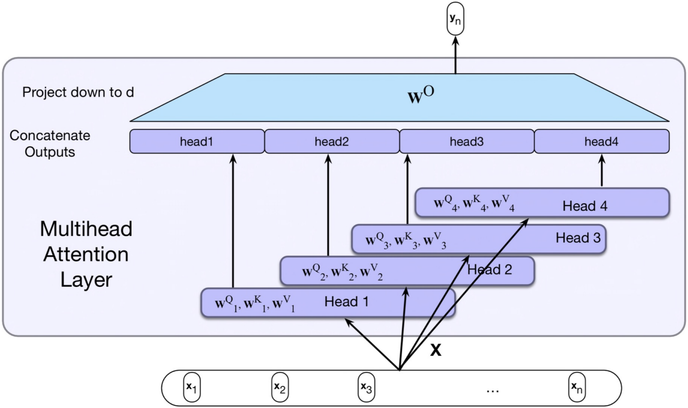
$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$

Transformer Block



(Jurafsky and Martin)

Multihead Attention Layer



(Jurafsky and Martin)

Encoding Word Positions in Transformers

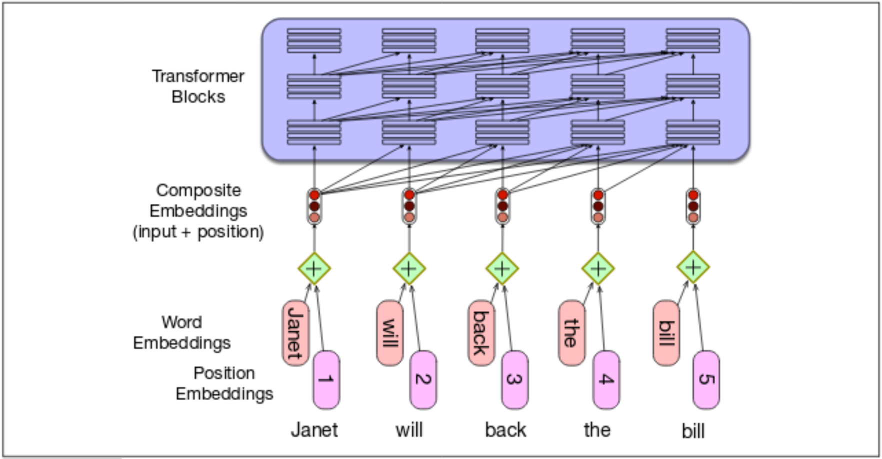


Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

from: Jurafsky and Martin, 3rd ed. draft

Training Transformer as a Language Model

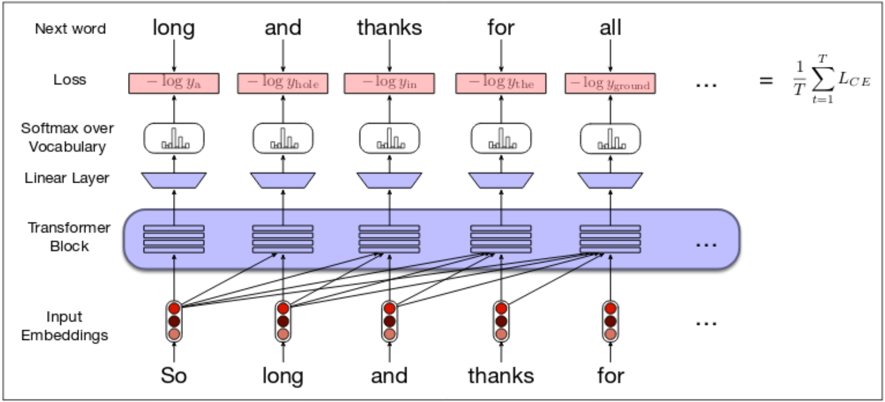


Figure 9.21 Training a transformer as a language model.

from: Jurafsky and Martin, 3rd ed. draft

Text Completion with Transformers

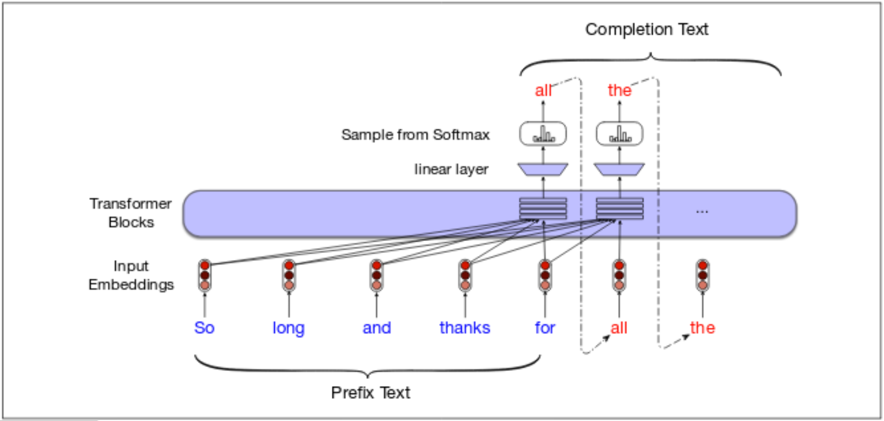


Figure 9.22 Autoregressive text completion with transformers.

from: Jurafsky and Martin, 3rd ed. draft

Parsing Natural Languages

- Must deal with possible ambiguities
- Decide whether to make a phrase structure or dependency parser
- When parsing NLP, there are generally two approaches:
  1. Backtracking to find all parse trees
  2. Chart parsing
- Prolog provides a very expressive way to NL parsing
- FOPL is also used to represent semantics

Parsing with Prolog

- We will go over a brief Prolog review
  - more details are provided in the lab
- Implicative normal form:

$$p_1 \land p_2 \land \ldots \land p_n \Rightarrow q_1 \lor q_2 \lor \ldots \lor q_m$$

- If $m \leq 1$, then the clause is called a **Horn clause.**
- If resolution is applied to two Horn clauses, the result is again a Horn clause.
- Inference with Horn clauses is relatively efficient

Rules

A Horn clause with $m = 1$ is called a **rule**:

$$p_1 \land p_2 \land \ldots \land p_n \Rightarrow q_1$$

It is expressed in Prolog as: `q1 :- p1, p2, ..., p_n.`

Facts

A clause with $m = 0$ is called a **fact**:

$$p_1 \land p_2 \land \ldots \land p_n \Rightarrow \top$$

is expressed in Prolog as: `p1, p2, ..., p_n.`
or `:- p1, p2, ..., p_n.`
and it is called a **fact.**

Rabbit and Franklin Example

The ‘rabbit and franklin’ example in Prolog:

```
hare(rabbit).
turtle(franklin).
faster(X,Y) :- hare(X), turtle(Y).
```

Save the program in a file, load the file.

After loading the file, on Prolog prompt, type:

```
faster(rabbit,franklin).
```

Try: `faster(X,franklin).` and `faster(X,Y).`

Rabbit and Franklin Example

```
hare(rabbit).
turtle(franklin).
faster(X,Y) :- hare(X), turtle(Y).

?- faster(rabbit,franklin).
```

Rabbit and Franklin Example

```
hare(rabbit).
turtle(franklin).
faster(X,Y) :- hare(X), turtle(Y).

?- faster(X,franklin).
```

Rabbit and Franklin Example

```
hare(rabbit).
turtle(franklin).
faster(X,Y) :- hare(X), turtle(Y).

?- faster(X,Y).
```

Unification and Backtracking

- Two important features of Prolog: unification and backtracking
- Prolog expressions are generally mathematical symbolic expressions, called *terms*
- **Unification** is an operation of making two terms equal by substituting variables with some terms
- **Backtracking:** Prolog uses backtracking to satisfy given goal; i.e., to prove given term expression, by systematically trying different rules and facts, which are given in the program

Example in Unification and Backtracking

- What happens after we type:
  `?- faster(rabbit,franklin).`
- Prolog will search for a 'matching' fact or head of a rule:
  `faster(rabbit,franklin)` and
  `faster(X,Y) :- ...`
- 'Matching' here means **unification**
- After unifying `faster(rabbit,franklin)` and `faster(X,Y)` with substitution X←`rabbit` and Y←`franklin`, the rule becomes:

```
faster(rabbit,franklin) :-
   hare(rabbit), turtle(franklin).
```

Example (continued)

- Prolog interpreter will now try to satisfy predicates at the right hand side: `hare(rabbit)` and `turtle(franklin)` and it will easily succeed based on the same facts
- If it does not succeed, it can generally try other options through **backtracking**

Variables in Prolog

- Variable names start with uppercase letter or underscore ('_')
- _ is a special, *anonymous variable*
- Examples: `?- faster(rabbit,franklin).`

```
Yes ;
...
?- faster(rabbit,X).
X = franklin ;
...
?- hare(X).
X = rabbit ;
```

Lists (Arrays), Structures.

Lists are implemented as linked lists. Structures (records) are expressed as terms. Examples:
In program: `person(john,public,'123-456').`
Interactively: `?- person(john,X,Y).`
`[]` is an empty list.
A list is created as a nested term, usually a special function '`.`' (dot):

```
?- is_list(.(a, .(b, .(c, [])))).
```

List Notation

`(.(a, .(b, .(c, []))))` is the same as `[a,b,c]`

This is also equivalent to:

`[ a | [ b | [ c | [] ]]]`

or

`[ a, b | [ c ] ]`

A frequent Prolog expression is: `[H|T]`

where H is head of the list, and T is the tail, which is another list.

Example: Calculating Factorial

```
factorial(0,1).
factorial(N,F) :- N>0, M is N-1, factorial(M,FM),
    F is FM*N.
```

After saving in `factorial.prolog` and loading to Prolog:

```
?- ['factorial.prolog'].
% factorial.prolog compiled 0.00 sec, 1,000 bytes

Yes
?- factorial(6,X).

X = 720 ;
```

Example: List Membership

Example (testing membership of a list):

```
member(X, [X|_]).
member(X, [_|L]) :- member(X,L).
```