Faculty of Computer Science, Dalhousie University *27-Nov-2025*

CSCI 4152/6509 — Natural Language Processing

Lecture 17: Deep Learning Architectures for NLP

Location: Studley LSC-Psychology P5260 Instructor: Vlado Keselj
Time: 14:35 – 15:55

Previous Lecture

Neural Network Models

- Neural networks and deep learning
- Overview of Large Language Models
- Foundations of Deep Learning
  - From Naïve Bayes to Perceptron
- Perceptron as an Artificial Neuron and Computation
- Feedforward Neural Network and Matrix Computation

P0 Topics Discussion (4)

- Additional discussion of individual projects as proposed in P0 submissions (part 4)
- Projects discussed: P-18

Slide notes:

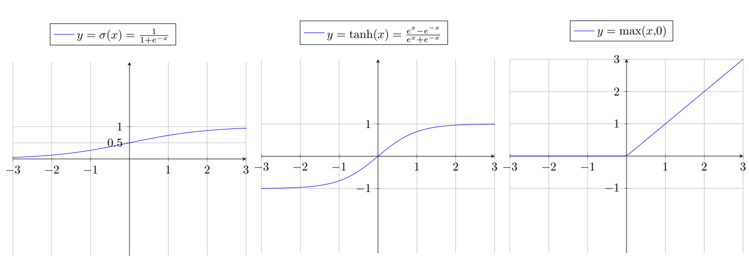
Activation Function

- must be non-linear
  - otherwise, the whole neural network would collapse into one neuron
- should be monotonically non-decreasing
- useful to be differentiable and relatively simple for speed of training
- Best known activation functions: sigmoid, tanh, ReLU (Rectified Linear Unit)

Slide notes:

Common Activation Functions

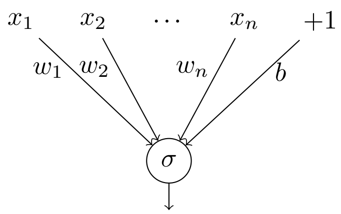
Sigmoid tanh ReLU



Slide notes:

Binary Classification with One Layer

- same as binary logistic regression

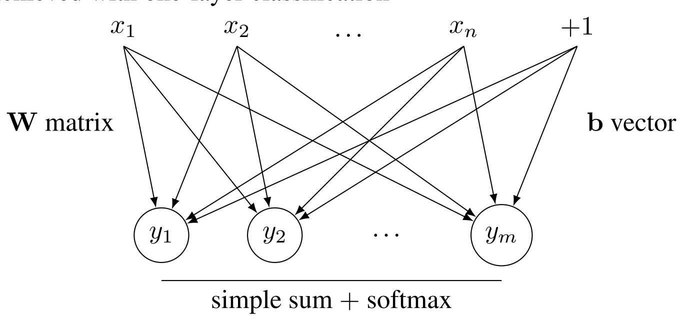


$$y = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b})$$

Slide notes:

Multinomial Logistic Regression

- achieved with one-layer classification



$$\mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Slide notes:

Softmax Function

- Softmax transforms numbers into positive domain using e^x; i.e., $\exp(x)$, function, and normalizing numbers into a probability distribution

$$\text{softmax}(\mathbf{x}) = [\frac{\exp(x_1)}{\sum_{i=1}^{n} \exp(x_i)}, \frac{\exp(x_2)}{\sum_{i=1}^{n} \exp(x_i)}, \cdots \frac{\exp(x_n)}{\sum_{i=1}^{n} \exp(x_i)}]$$

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

- Example from Jurafsky and Martin:

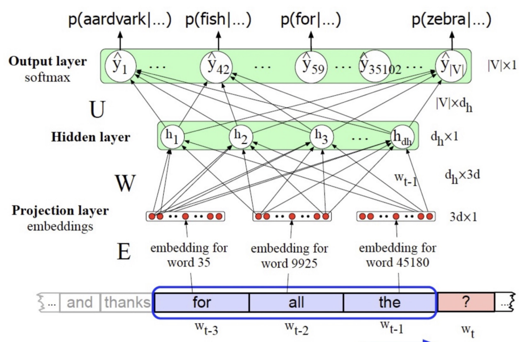
$$\mathbf{x} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(x) = [0.055, 0.09, 0.006, 0.099, 0.74, 0.01]$$

Neural Language Model

Slide notes:

Neural Language Model

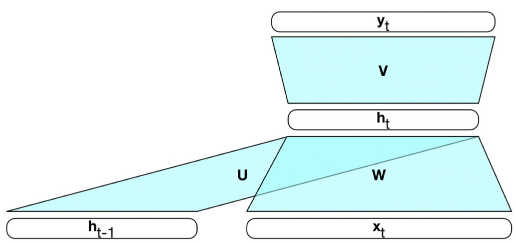


(Jurafsky and Martin)
The model has limited history, similarly to n-gram model

Slide notes:

Recurrent Neural Networks (RNN)

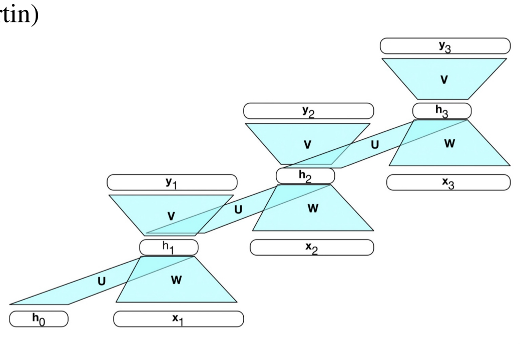
- Simple recurrent neural network presented as a feedforward network (Jurafsky and Martin)
- RNN is trained as a Language model by providing the next word as output



Slide notes:

RNN Unrolled in Time

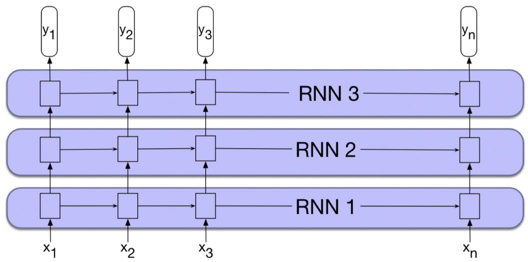
- RNN unrolled in time; more clear view of training (Jurafsky and Martin)



Slide notes:

Stacked RNN

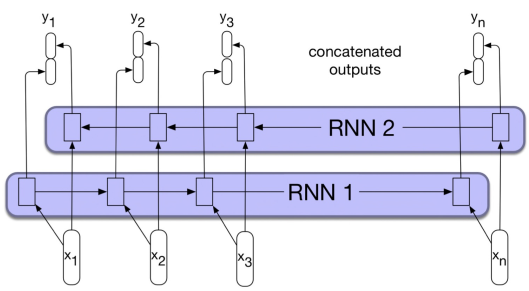
- Stacked RNN: Output from lower level is input to higher level; top level is final output (Jurafsky and Martin)



Slide notes:

Bidirectional RNN

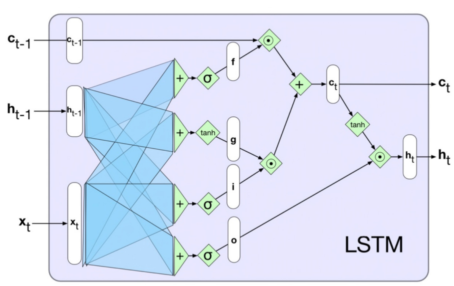
- Bidirectional RNN; trained forward and backward with concatenated output (Jurafsky and Martin)
- Output can be used for sequence labeling, for example



Slide notes:

LSTM — Long Short-Term Memory

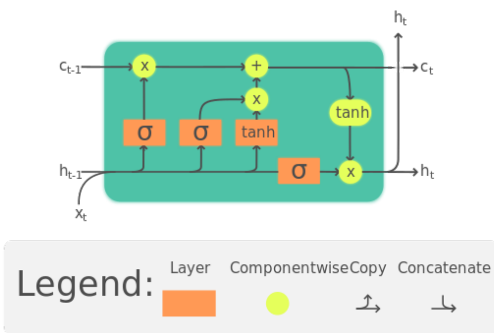
- LSTM: x_t is input, h_{t-1} is previous hidden state, c_{t-1} is previous long-term context, h_t and c_t is output (Jurafsky and Martin)



Slide notes:

LSTM Cell

– Another view of LSTM cell (source Wikipedia)



Slide notes:

Transformers

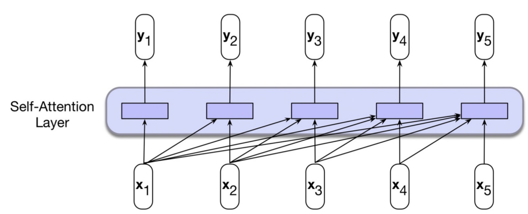
– Transformers map a sequence of input vectors to a sequence of output vectors of the same length

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

– Appeared around 2017

Slide notes:

Self-Attention Layer



(Jurafsky and Martin)

Slide notes:

Self-Attention Training

- A simplified view:

$$score(x_i, x_j) = x_i \cdot x_j$$

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

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

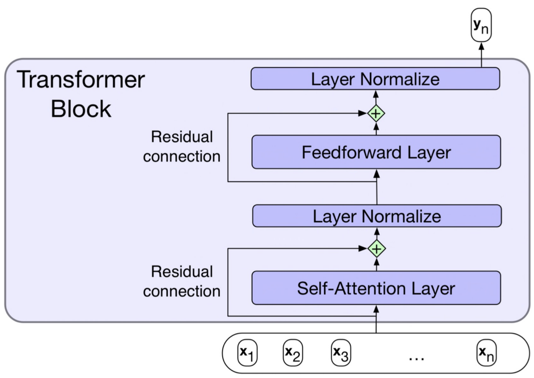
Slide notes:

Actual Attention Computation

– Three separate roles for each word vector:

1. query: current token being compared to preceding tokens
2. key: preceeding token being compared to the current
3. value: value of a preceding token that gets weighted

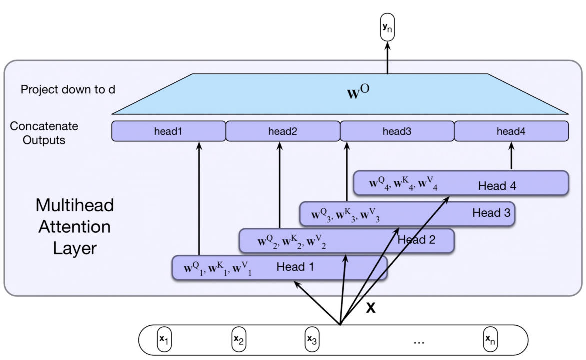
Transformer Block



(Jurafsky and Martin)

Slide notes:

Multihead Attention Layer



(Jurafsky and Martin)

Slide notes:

Encoding Word Positions in Transformers

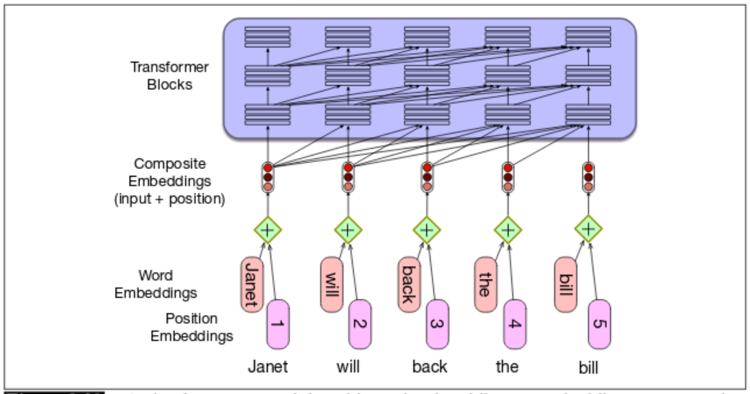


Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

from: Jurafsky and Martin, 3rd ed. draft

Slide notes:

Training Transformer as a Language Model

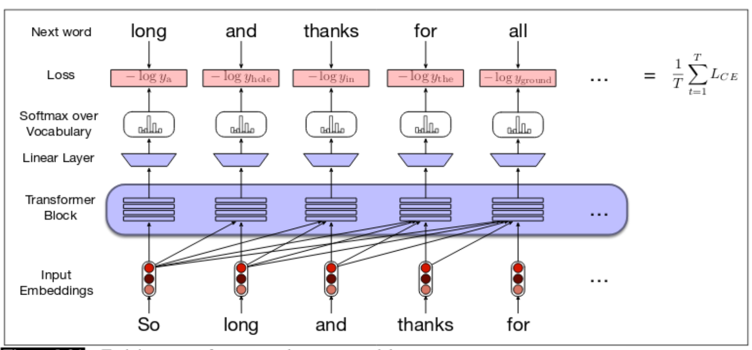


Figure 9.21 Training a transformer as a language model.

from: Jurafsky and Martin, 3rd ed. draft

Slide notes:

Text Completion with Transformers

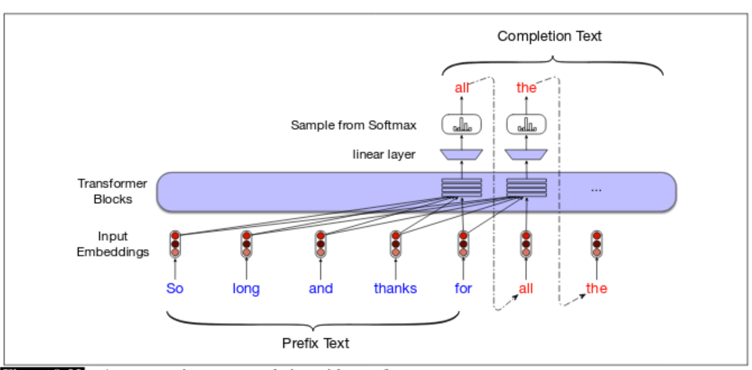


Figure 9.22 Autoregressive text completion with transformers.

from: Jurafsky and Martin, 3rd ed. draft

Part IV

Parsing

In this part, we will move a level above in processing natural languages—parsing, or syntactic processing. For some practical purposes, we will start with an brief introduction to the Prolog programming language.

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

18 A Brief Introduction to Prolog

In this section, we will first go over a brief Prolog review. Prolog is described in some more details in the lab tutorial.

Slide notes:

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

An implicative normal form is a mathematical logic formula, which is a conjunction of smaller formulae called clauses, where each clause is in the following form:

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

where p_i and q_i are simple logical statements called propositions.

Note: Just as a reminder, the operator $\land$ is the logical AND, operator $\lor$ is the logical OR, and the operator $\Rightarrow$ is the logical "implies" operator.

If $m \leq 1$, then the clause is called a **Horn clause**.

When resolution is applied to two Horn clauses, the result is again a Horn clause. Inference on Horn clauses is relatively efficient.

Rules

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

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

It is expressed in Prolog as:

```
q1 :- p1, p2, ..., p_n.
```

Facts

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

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

is expressed in Prolog as:

```
p1, p2, ..., p_n.
```

or

```
:- p1, p2, ..., p_n.
```

and it is called a **fact.**

Running Prolog

It is covered in more details in the lab how to run Prolog interpreter. We use a Prolog interpreter called SWI Prolog and it is available on the `timberlea` server. The lab also covers how to write a program, load it and execute it using interpreter.

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, e.g., named `file.prolog` and load the file using the command `['file.prolog']`. The Prolog interpreter uses prompt '`?-`'. After loading the file, on Prolog prompt, type:

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

After this there is some difference between Prolog interpreters. The newest SWI-Prolog will simply print 'true' and go back to the prompt. The previous version of SWI-Prolog would print 'Yes' waiting for user input. The user should type semicolon (`;`) and then the Prolog prompt would appear.

Try `faster(X,franklin).` and `faster(X,Y).` in the similar fashion (keep pressing the semicolon if user input is required until the Prolog prompt is obtained in the both cases).

Slide notes:

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

Variable names in Prolog start with an uppercase letter or an underscore character ('_]). The variable name _ (just an underscore) is special because it denotes a special, so-called *anonymous* variable. Two occurrences of this variable can represent arbitrary different values, and there is no connection between them. This variable is used a placeholder in terms for part that is generally ignored.

Slide notes:

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).
```