Natural Language Processing

Neural Network Language Models

Tianxing He

goosehe@cs.washington.edu

Outline

- Basics of neural network (~35min, tough, involves math)
- Feedforward NN LM
- Recurrent NN LM
- Word2vec (briefly, if time permits)

Brief Review: logistic regression (LR)

- To understand today's content in neural network, it will be super helpful to review the basic formulations from the LR model.

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

$$z = w \cdot x + b$$

$$P(y = 1|x) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Review: A general recipe for multi-class classification

- Before we dive into NN and NLP, let's review a general recipe for multi-class classification. It's super important for understanding of almost everything covered in these two lectures!

- Take 3-class sentiment classification as an example.

This restaurant is great! –> positive

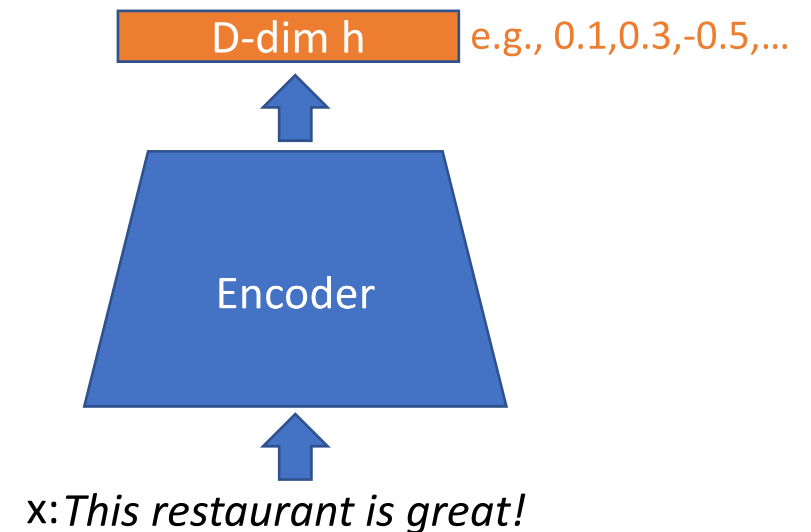
The food is okay. -> neutral

I hate this dish! -> negative

A general recipe for classification: Encode, Predict, Train

- Step1: Encode
- Assume we have an encoder(e.g., a neural network)

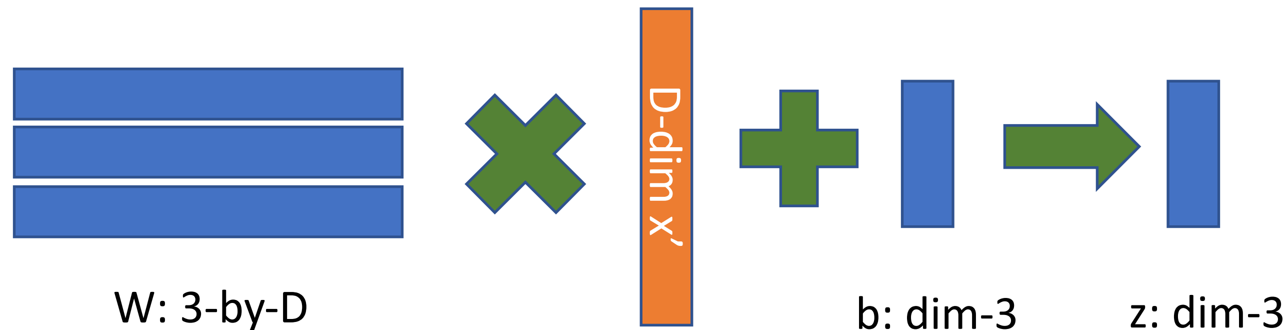
which maps the input x to a D-dim vector h.



How to realize this NN encoder will be clear soon!

A general recipe for classification: Encode, Predict, Train

- Step2: Predict
- We apply a linear transform to reduce enc(x) to a 3-dim vector, each dimension represents one class.



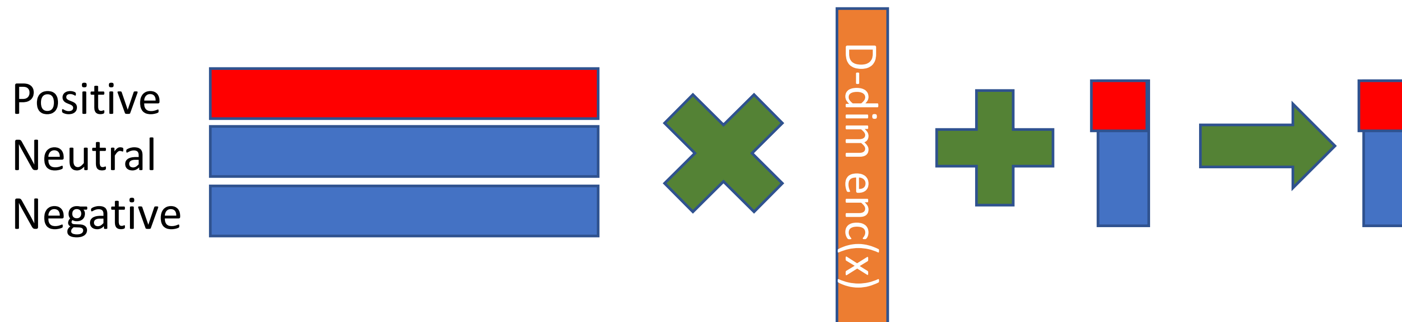
Note: the h and b here are column vectors.

$$z = W^{\text{cls}} h + b^{\text{cls}}$$

<- a 3-by-D linear transform

Comparison with logistic regression

Linear–transform for multi-class prediction



If we focus on one dimension(red), it's the same as binary LR!

LR for binary classification

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

$$z = w \cdot x + b$$

LR slide, page40

Encode, Predict, Train

- In order to do maximum likelihood training, we need a probability distribution.
- Now, we use the Softmax operation to map z to P(y|x).

$$\mathrm{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k}\exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k}\exp(z_i)}\right]$$

$$\mathrm{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k}\exp(z_j)} \quad 1 \le i \le k$$

<- k is the number of classes.

Example:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\mathrm{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Connection of softmax and sigmoid

- Softmax can regarded as a multi-class version of sigmoid

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

- Sigmoid is a softmax of z and 0

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^{z}}{e^{z}+e^{0}}$$

LR slide, page42

Encode, Predict, Train

- The training part is the same with LR (in high-level)!
- Assume we have a dataset $\{x_i, y_i\}$.
- We use the cross-entropy loss:

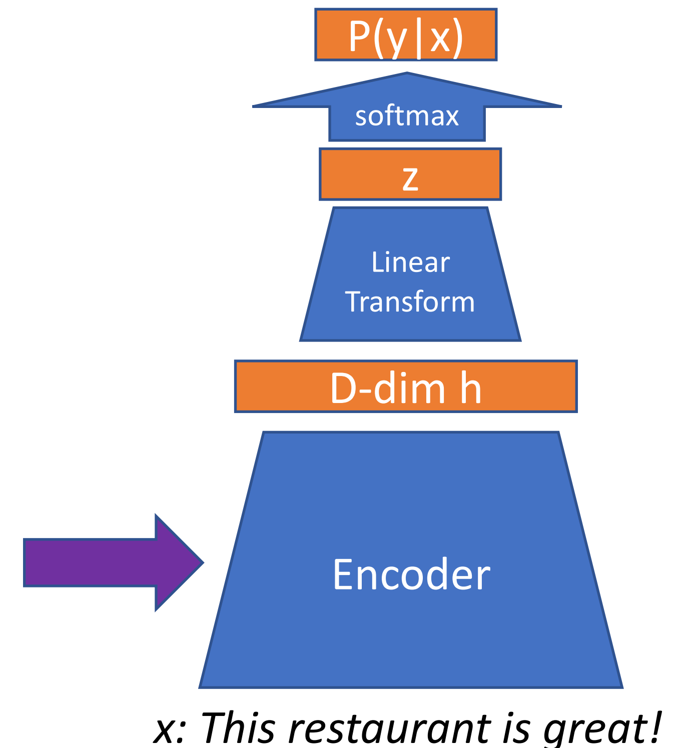
$$L_{\mathrm{CE}} = \sum_i -\log P(y = y_i | x_i)$$

- Assuming the model is differentiable, we use stochastic gradient descent to train the parameters θ. (W^{cls} and b^{cls} are part of θ)

$$\theta^{t+1} = \theta^t - \frac{\partial}{\partial \theta} L_{\mathrm{CE}}(\text{mini-batch}\{x_i, y_i\})$$

Review of the model: What's left?

- We did not talk about the encoder!
- It's time to introduce neural network!

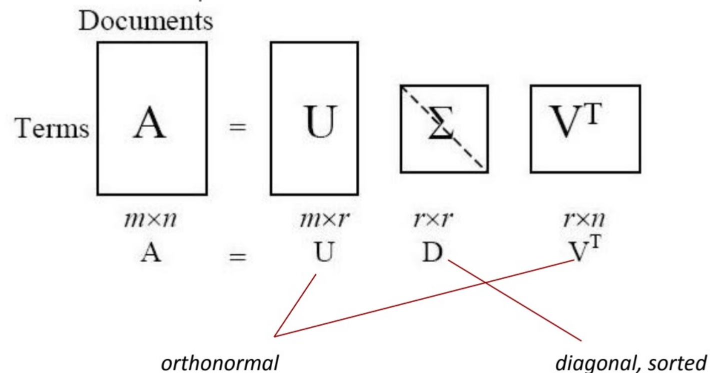


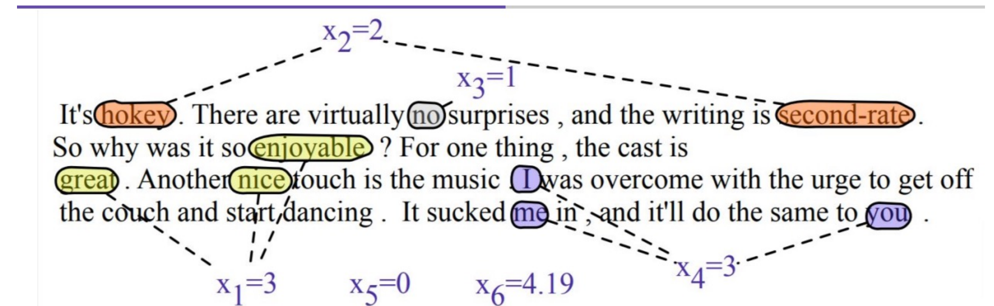
Philosophy (mind-set) of Neural Networks for NLP

- In previous lectures, we talked about smart ways for extracting features for word/sentence.
- They need some level of algorithm design or hand crafting.

Singular Value Decomposition (SVD)

- Solution idea:
  - Find a projection into a low-dimensional space (~300 dim)
  - That gives us a best separation between features





Var	Definition	Value
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

Yulia Tsvetkov 52 Undergrad NLP 202

Philosophy (mind-set) of Neural Networks for NLP

- When using neural networks, we leave these smart feature extraction techniques behind, and just feed (almost) raw data into the NN.
- And we let NN and SGD "learn" a good feature extraction from data.

- Instead, what we care about now are:
- 1: Use a powerful NN architecture
- 2: Use large amounts of data
- 3: Use a right learning objective

<- our focus in these two lectures!
Let's start with the simple ones

Bag of words as input

- First we need to encode the input x as a vector…

- Bag of words is a simple way to encode a sentence:
- a |V|-dim vector, the i-th dimension indicates whether the i-th word in V(vocabulary) exists in x.

- *This restaurant is great!* Will be mapped to:
- 0(a) 0(the) … 0(that) 1(this) 0 …. 0(amazing) 1(great) 0 …… <- We denote this vector as $\tilde{x}$.

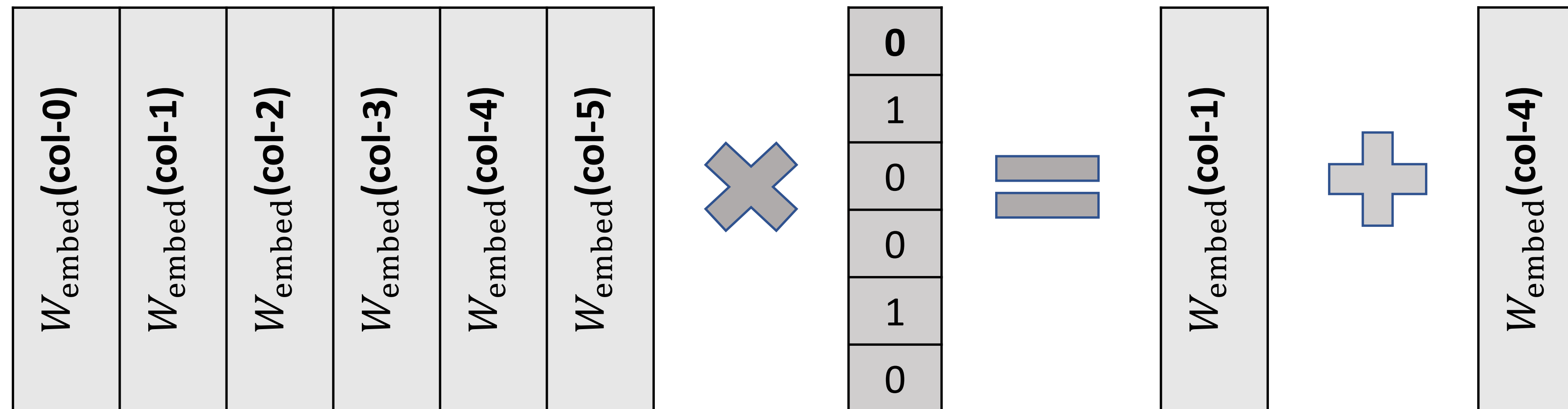
- Note: We can easily extend bag-of-words to bag-of-bigrams, which is |V|^2-dim.

Input dimension reduction via word embedding

- |V| is usually very large!! (at least 10k) We want to reduce it to a reasonable dimension D (e.g., 512).

- A simple way to do this is to multiply it with a D-by-|V| word embedding matrix:

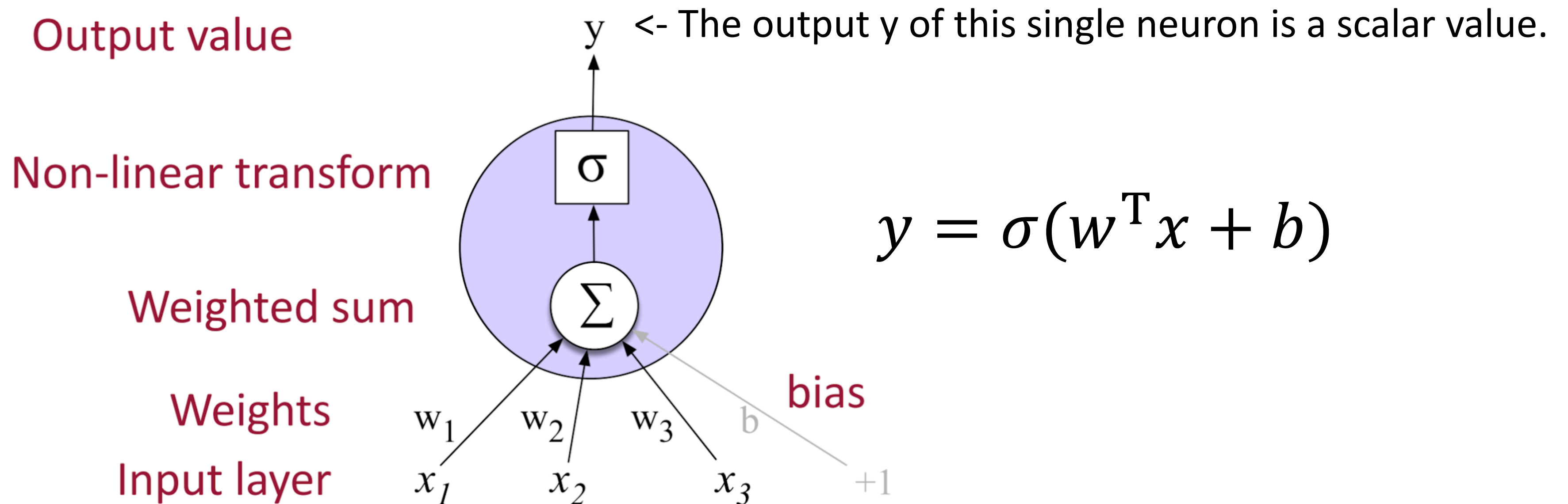
$$h^0 = W^{\text{embed}} \tilde{x}$$



- Note: The difference with LSA is that here the word embedding matrix is treated as part of the parameters of the NN model, and is learned by SGD.

A neural unit for feature extraction

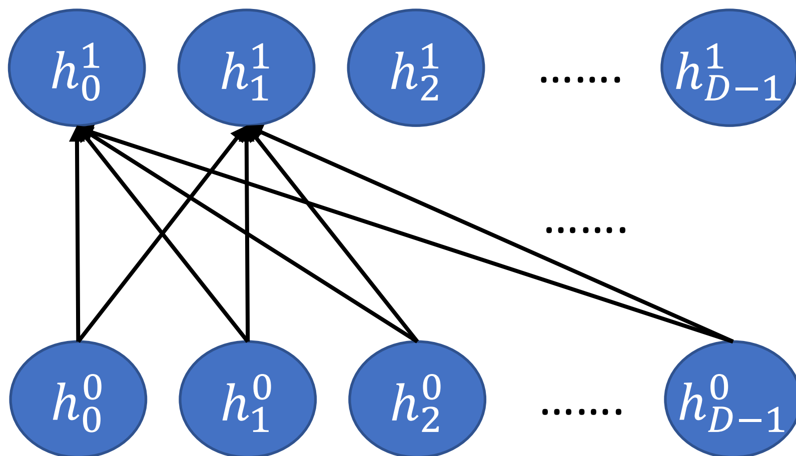
- Divide and conquer:
- In order to do the final prediction, we want to extract some easy binary feature first.
- *Example1: does x contain positive words (good, amazing, etc.) ?*
- *Example2: does x contain negation words (not, never, etc.) ?*
- This kind of low-level features can be extracted by a neural unit (aka., neuron), which is just a LR model !!



$$y = \sigma(w^{\mathrm{T}} x + b)$$

One hidden layer of neural network

- A layer of D neurons consists a hidden layer.



$$h^1 = \sigma(W^0 h^0 + b^0)$$

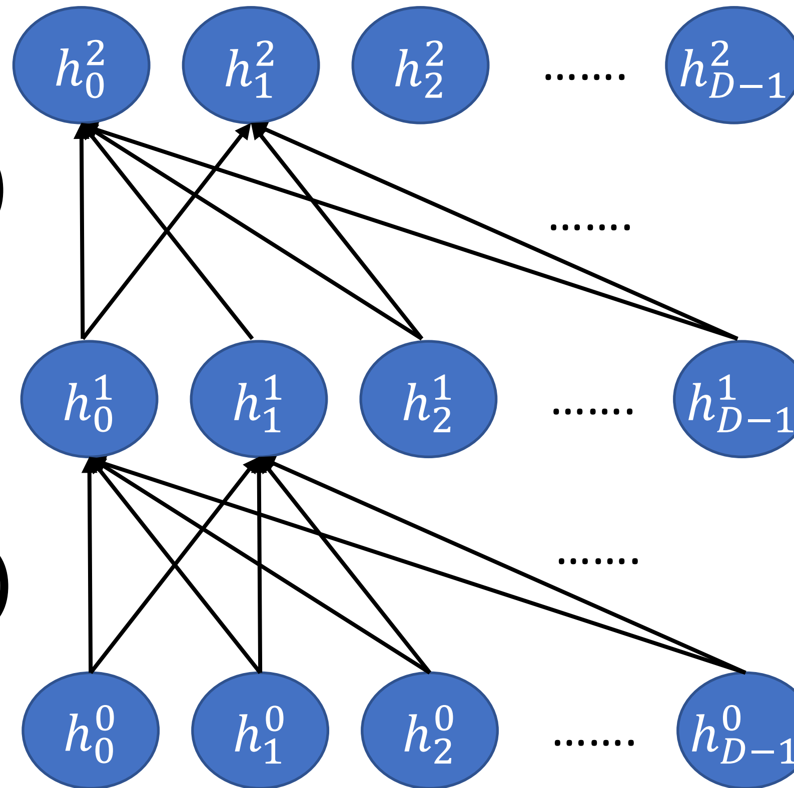
We aggregate the weights into W^0. The i-th row in W^0 corresponds to the weight w in the i-th neuron whose output is h_i^1.

Stacking multiple hidden layers

Notation: The "2" here does not mean squared. It means the second layer.

$$h^2 = \sigma(W^1 h^1 + b^1)$$

$$h^1 = \sigma(W^0 h^0 + b^0)$$



Intuition:
High-level feature
(semantic, etc.)

Low-level feature
(syntactic, etc.)

Raw feature
(n-gram, etc.)

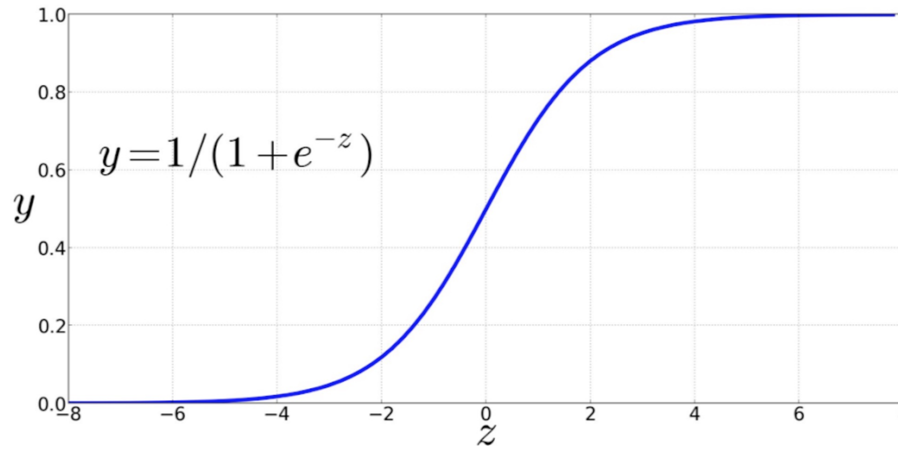
This is called a multi-layer perceptron (MLP) or a feedforward neural network.
It's the simplest type of neural network. (we will learn about more complicated ones in these two lectures)

Choice of activation function

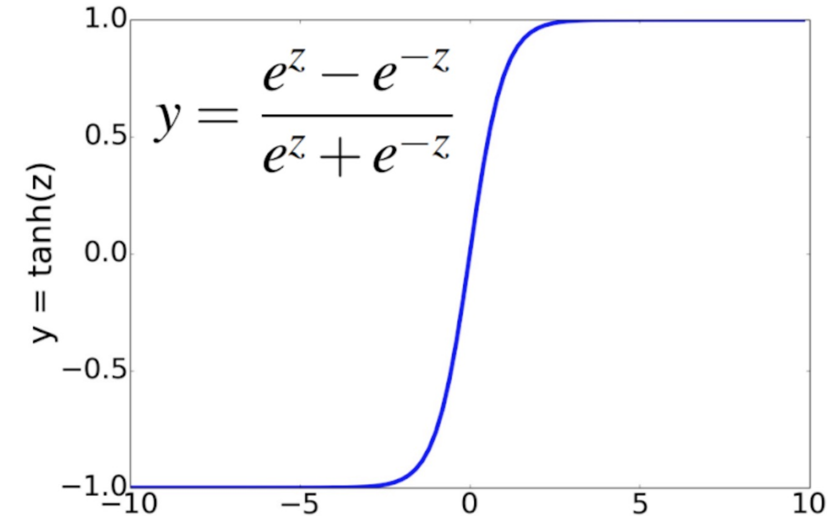
- The sigmoid function σ is one type of activation function.

Sigmoid

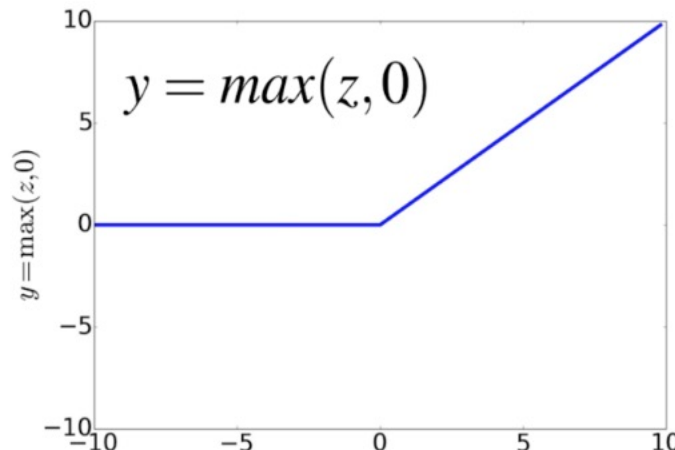
$$y = \sigma(z) = \frac{1}{1+e^{-z}}$$



tanh



ReLU
Rectified Linear Unit

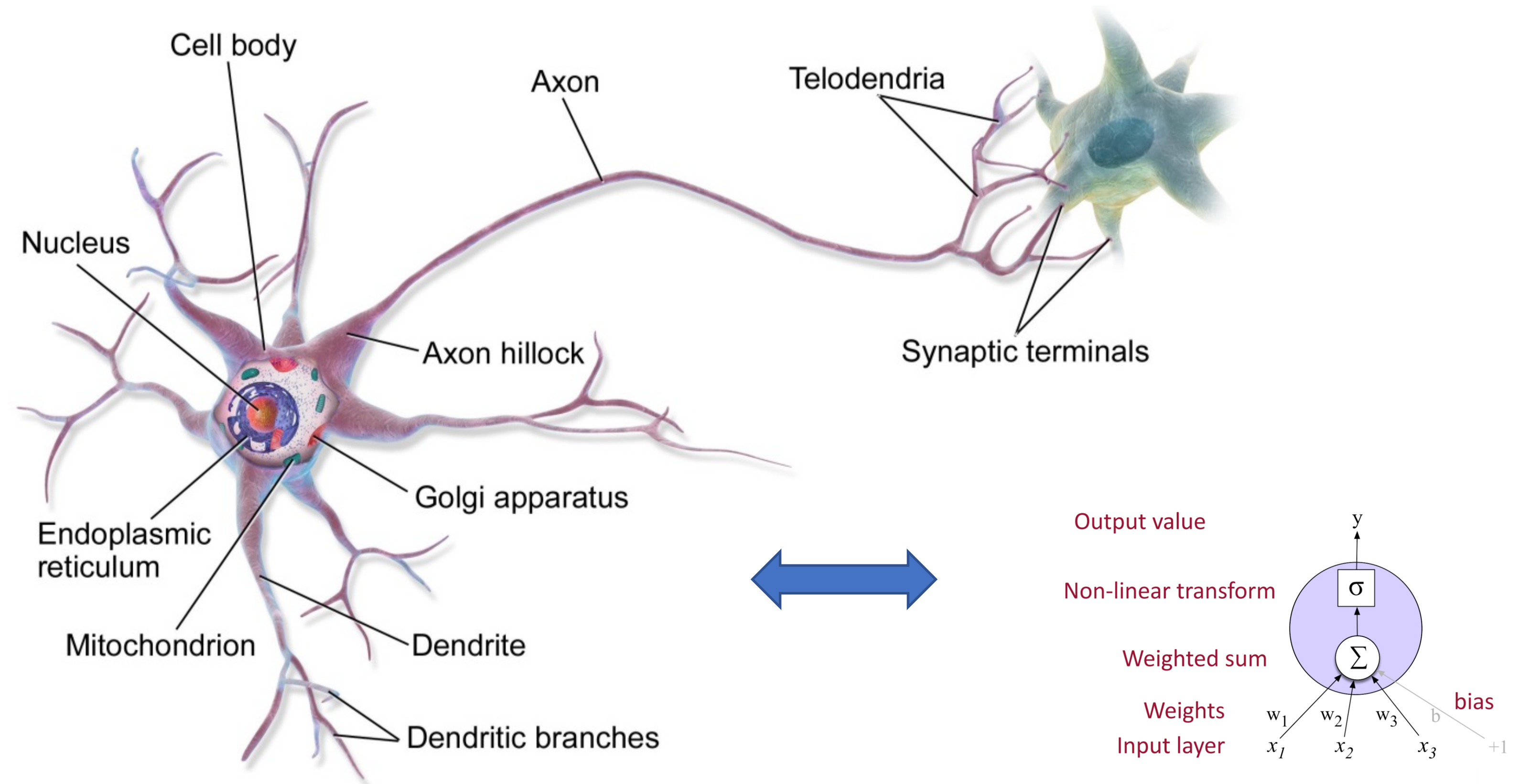


Tanh and ReLU have been empirically shown to outperform sigmoid.

Real neurons in brain

Remember that we are doing "artificial" neural network.

I don't think there's any transformer or lstm in our brain!



By BruceBlaus - Own work, CC BY 3.0,
https://commons.wikimedia.org/w/index.php?curid=28761830

The importance of non-linearity

A linear transform (e.g., $y = Wx$) can only give a linear decision boundary.
And the stacking of linear transforms (e.g., $y = W_1 W_2 W_3 x$) is still a linear transform.
The existence of non-linearity in NN is the key reason to make it powerful.

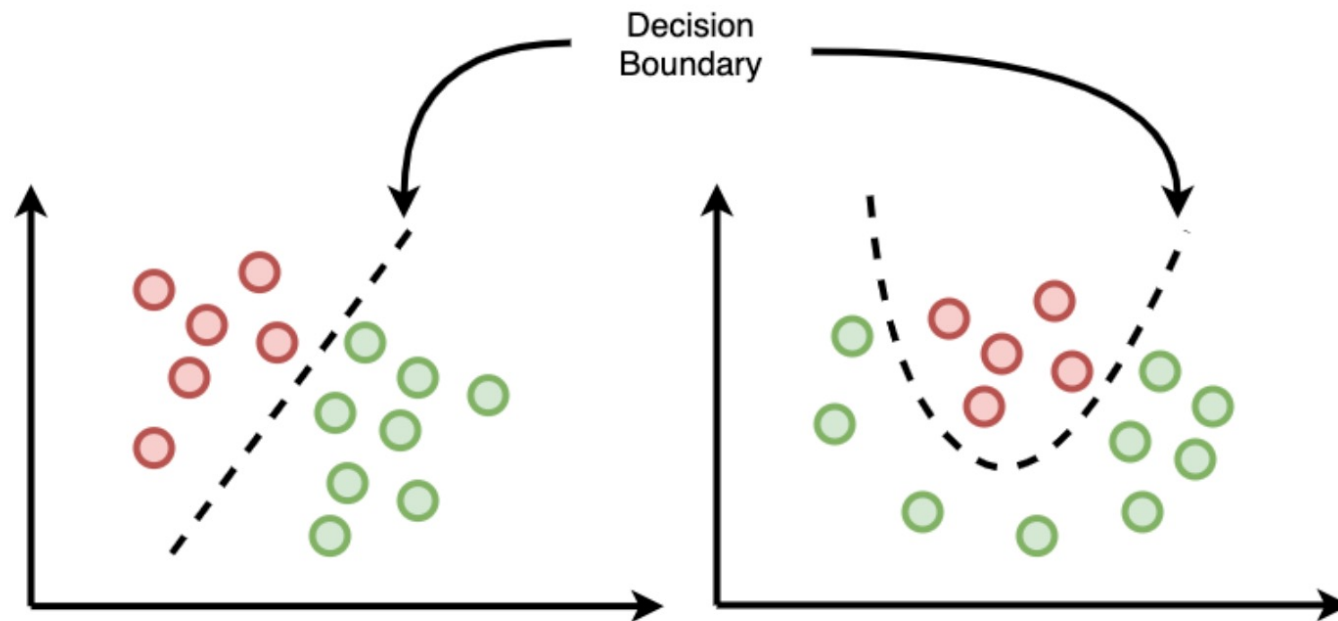
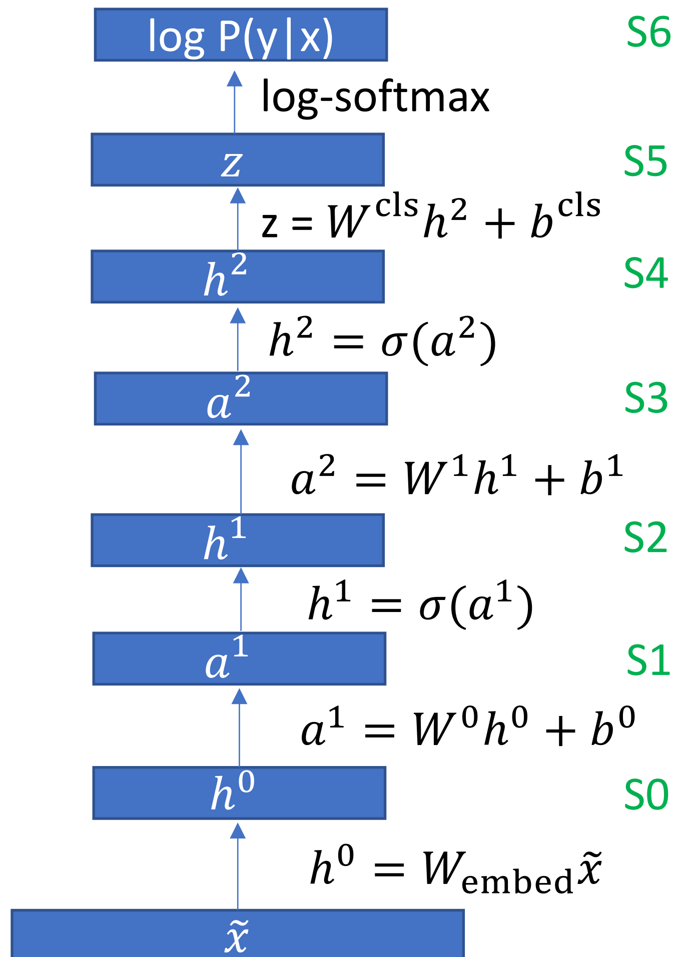


Figure from https://towardsdatascience.com/logistic-regression-and-decision-boundary-eab6e00c1e8

Summary: a NN model defines a series of computation

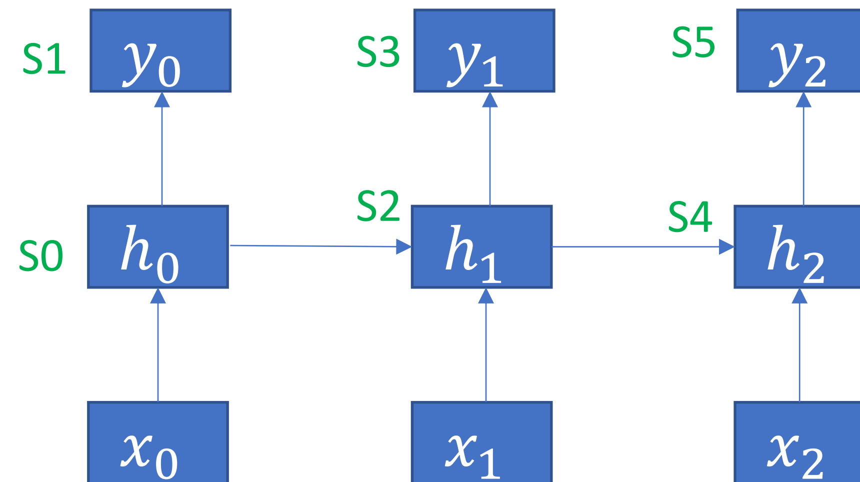


<- We call this a computational graph of a NN model.

It defines the dependency of (intermediate) variables.
And it's a directed acyclic graph (DAG).

In order to compute all values, we only need to follow the topological order.

How to get the topologically sorted order Blackboard Example (we will need this for recurrent NN!)



How to get the topologically sorted order (from wiki)

Kahn's algorithm [edit]

Not to be confused with Kuhn's algorithm.

One of these algorithms, first described by Kahn (1962), works by choosing vertices in the same order as the eventual topological sort.[2] First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist in a non-empty acyclic graph. Then:

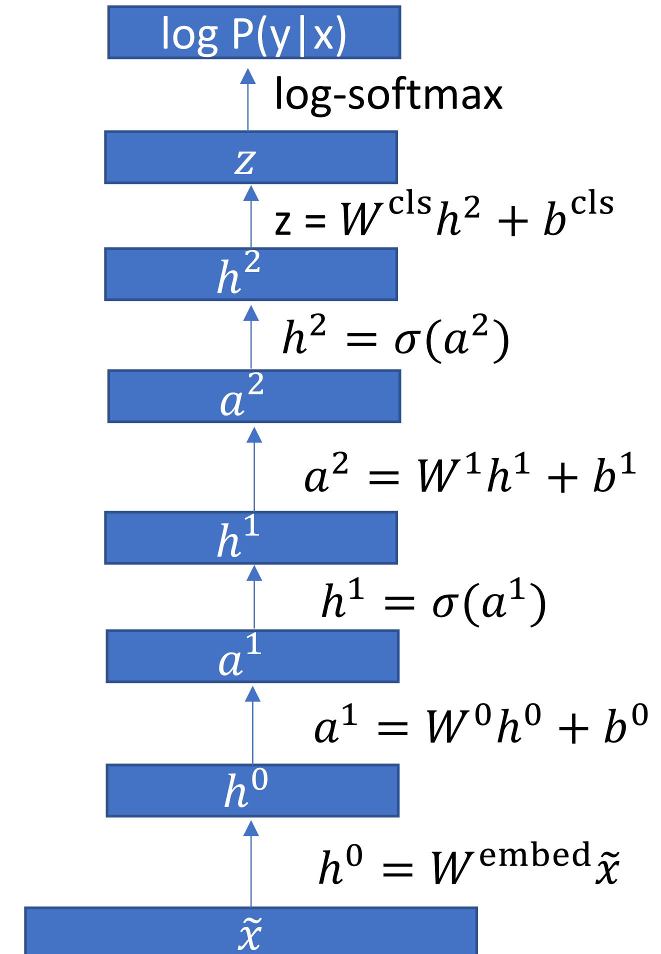
```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

Back-propagation (Goal)

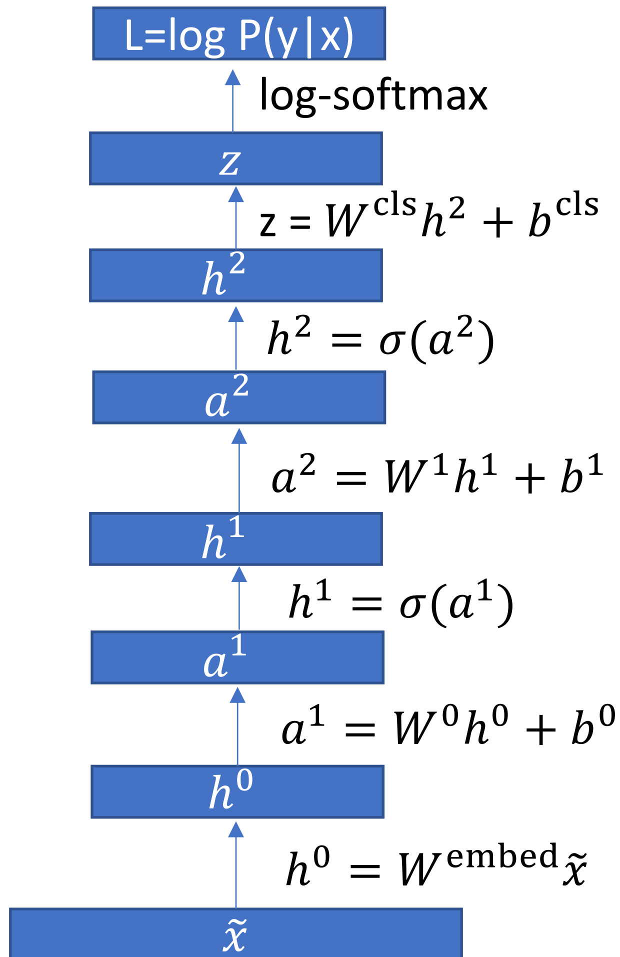
- Now, to do SGD we need to get gradient for the parameters $\theta = \{W^{\text{cls}}, b^{\text{cls}}, W^1, b^1, W^0, b^0, W^{\text{embed}}\}$.
- Remember our loss is $L = -\log P(y|x)$.

- First, we will use the back-propagation algorithm to get the error vectors $\{\frac{\partial L}{\partial z}, \frac{\partial L}{\partial h^2}, \frac{\partial L}{\partial a^2}, \frac{\partial L}{\partial h^1}, \frac{\partial L}{\partial a^1}, \frac{\partial L}{\partial h^0}\}$.
- Note: the error vectors are row vectors.

- Don't be intimated! It's just chain rule! e.g., $\frac{\partial L}{\partial h^2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial h^2}$
- Assume we have computed $\frac{\partial L}{\partial z}$, $\frac{\partial z}{\partial h^2}$ is just the derivative of a simple linear function.



Back-propagation (Derivative of each step)



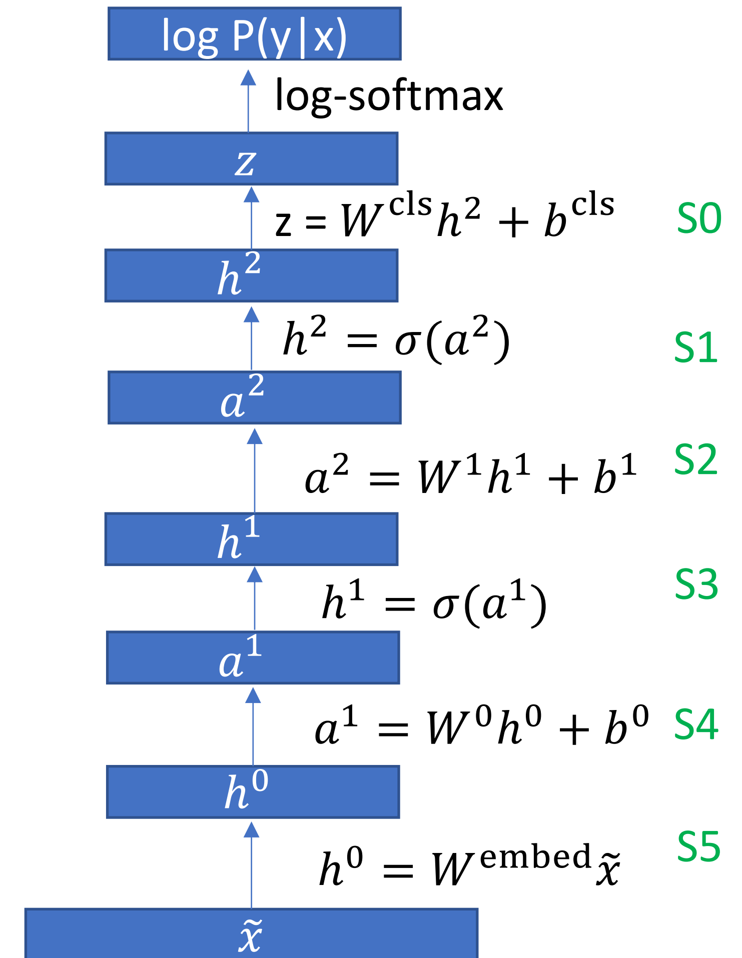
- Remember that NN consists of a series of relatively simple function (e.g., linear transform or sigmoid).
- The Jacobian of each function can be easily derived by applying basic calculus.
- For example:
- *eq1:* $\frac{\partial a^2}{\partial h^1} = \frac{\partial W^1h^1+b^1}{\partial h^1} = W^1$ *<- for today, just remember this one*
- *eq2:* $\frac{\partial \log P(y|x)}{\partial z} = \frac{\partial \text{log-softmax}(z)[y]}{\partial z} = (\text{softmax}(z) - \tilde{y})^T$, where $\tilde{y}$ is the one-hot encoding of ground-truth label y.
- *eq3:* $\frac{\partial h^2}{\partial a^2} = \frac{\partial \sigma(a^2)}{\partial a^2} = \text{diag}[\sigma(a^2)\odot(1-\sigma(a^2))]$,where $\odot$ is element-wise multiplication.

Back-propagation (error vector computation)

- Now, we have everything we need to compute the error vectors. We just follow the reverse topological order.

- S0: compute $\frac{\partial L}{\partial z}$, by eq2 (log-softmax).
- S1: compute $\frac{\partial L}{\partial h^2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial h^2}$, by eq1 (linear).
- S2: compute $\frac{\partial L}{\partial a^2} = \frac{\partial L}{\partial h^2}\frac{\partial h^2}{\partial a^2}$, by eq3 (sigmoid).
- ……
- S6: compute $\frac{\partial L}{\partial h^0} = \frac{\partial L}{\partial a^1}\frac{\partial a^1}{\partial h^0}$

- Every step is a vector (error vector from last step)-matrix (Jacobian) multiplication.



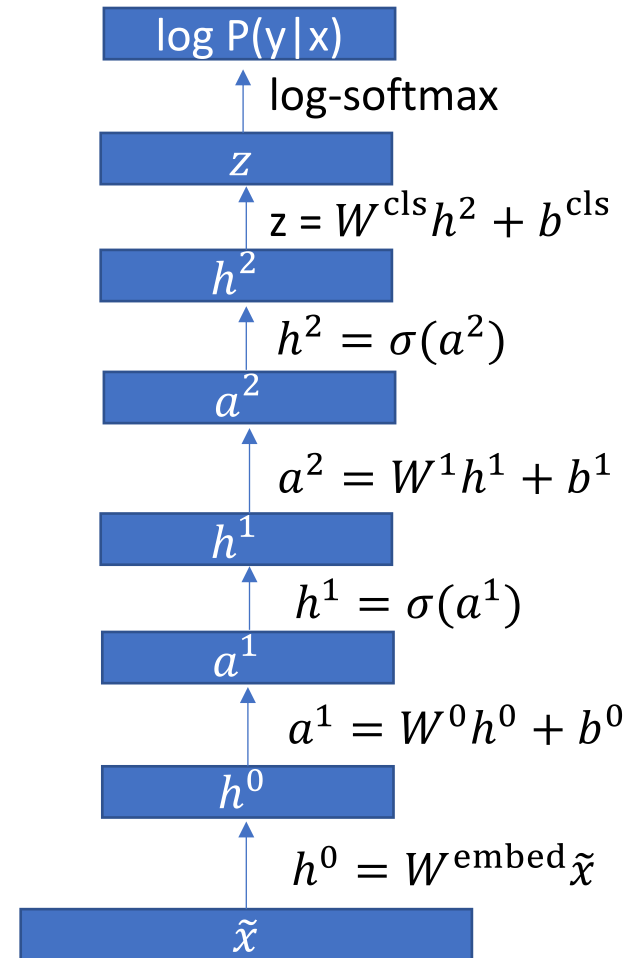
Back-propagation (error vector computation)

- *Remark*: If we expand $\frac{\partial L}{\partial h^0}$:
- $\frac{\partial L}{\partial h^0} = \frac{\partial L}{\partial a^1}\frac{\partial a^1}{\partial h^0} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial h^2}\frac{\partial h^2}{\partial a^2}\frac{\partial a^2}{\partial h^1}\frac{\partial h^1}{\partial a^1}\frac{\partial a^1}{\partial h^0}$

$= \frac{\partial L}{\partial z} W^{\text{cls}} \quad \frac{\partial h^2}{\partial a^2} W^1 \quad \frac{\partial h^1}{\partial a^1} W^0$ <- expand the linear terms

Exercise: check this still works if the hidden layers (h^0, h^1, h^2) have different dimensions.

- As we stack more hidden layers, the error vectors of lower layers involves more matrix multiplications. This gives some intuition why deeper NN are harder to train.
- (we will revisit this in recurrent NN)

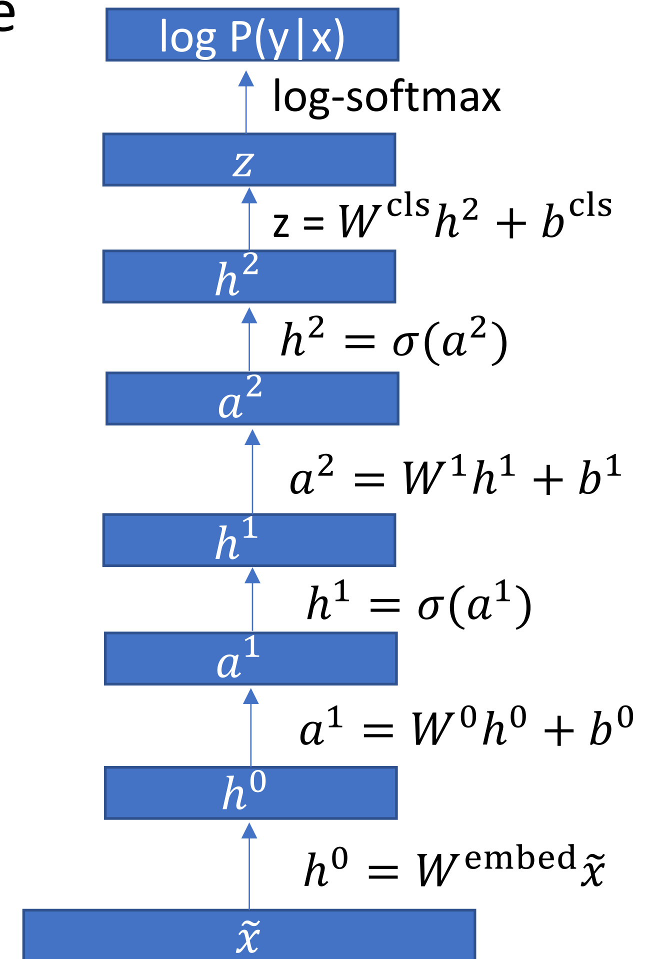


Gradient of each parameter

Grey parts for left for exercise

- Where the error vectors computed, we can apply chain rule again to get the gradient of each parameter.
- For example:
- $\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial W^1} = \left(\frac{\partial L}{\partial a^2}\right)^T (h^1)^T$
- How? Think about each element W_{ij}^1
- $\frac{\partial L}{\partial W_{ij}^1} = \frac{\partial L}{\partial a_i^2}\frac{\partial W_i^1 h^1 + b_i^1}{\partial W_{ij}^1} = \frac{\partial L}{\partial a_i^2} h_j^1$, where the first equation is because W_{ij} only affects a_i.

- We also have:
- $\frac{\partial L}{\partial b^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial b^1} = \frac{\partial L}{\partial a^2}\frac{\partial W^1 h^1 + b^1}{\partial b^1} = \frac{\partial L}{\partial a^2} \cdot I = \frac{\partial L}{\partial a^2}$



The tough part is over!
Relax: what it's like in pytorch

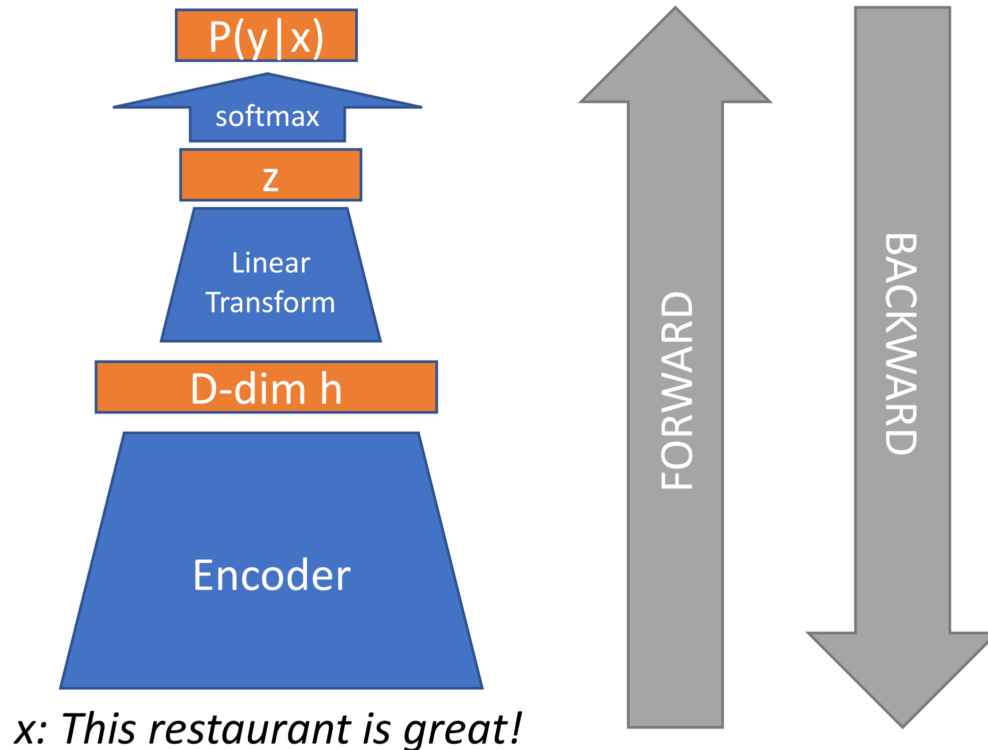
- *#Below is not real code but it's very close:*

- model = sequential(Linear, Sigmoid, Linear, Sigmoid, Linear) *#defines the computation graph*
- z = model(x)
- loss = log-softmax(z, y) *#forward and compute loss*
- loss.backward() *#backward and gradient computation*

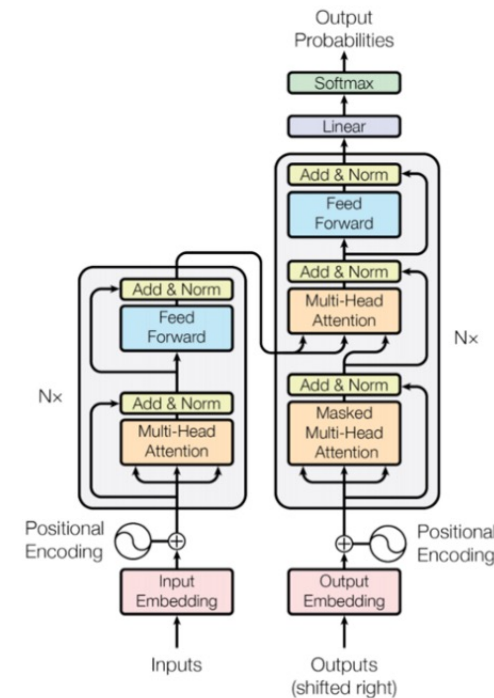
- #print(model[0].weight.gradient)
- optimizer.step() *#do a SGD step*

Brief summary

We now know how to forward and backward a feedforward NN.
Later we will see more complicated recurrent NN, transformers, etc. But as long as we know the structure of the computational graph, it's the same!



x: This restaurant is great!



The graph of a transformer model.

Neural Network Language Model

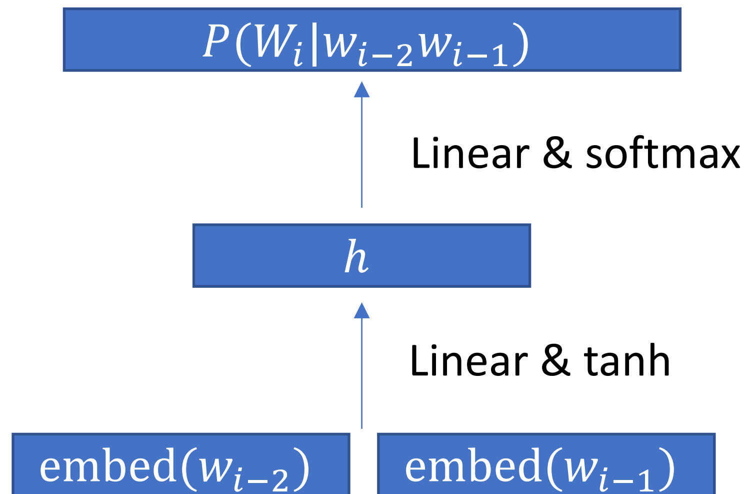
- We will now introduce a series of NNLM.

- Review of the trigram model:

$$q(w_i|w_{i-2}, w_{i-1}) = \frac{\text{count}(w_{i-2}, w_{i-1}, w_i)}{\text{count}(w_{i-2}, w_{i-1})}$$

- Using what we have learnt, how would you build a NN version of the n-gram LM?

A feedforward NN LM



$$L = \sum_{(w_{i-2},w_{i-1},w_i)\in \text{data}} -\log P\,(w_i|w_{i-2}w_{i-1})$$

- Note a big difference with the sentiment classifier is that the output class number is now |V|, making the model slow. Proposed remedies: *class-based LM* or *noise contrastive estimation*.

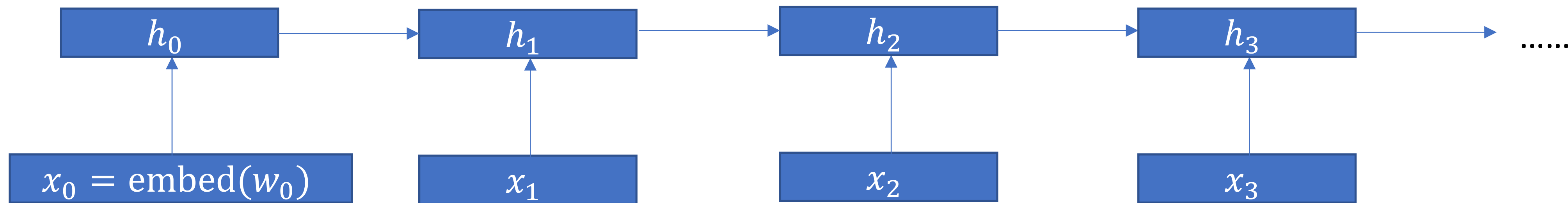
A Neural Probabilistic Language Model

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA

Recurrent neural network language model

- The (F)NNLM only encodes a very limited context (n-gram).

- RNN defines an efficient flow of computation to encode the whole history $w_0 \ldots . w_{t-1}$.
- The RNN maintains a hidden state h_t which is updated at each time step.



$$h_t = \sigma(W_{ih}x_t + W_{hh}h_{t-1} + b)$$

- Important: The parameters $\{W_{ih}, W_{hh}\}$ are shared across timesteps (hence the name recurrent).

Recurrent neural network language model

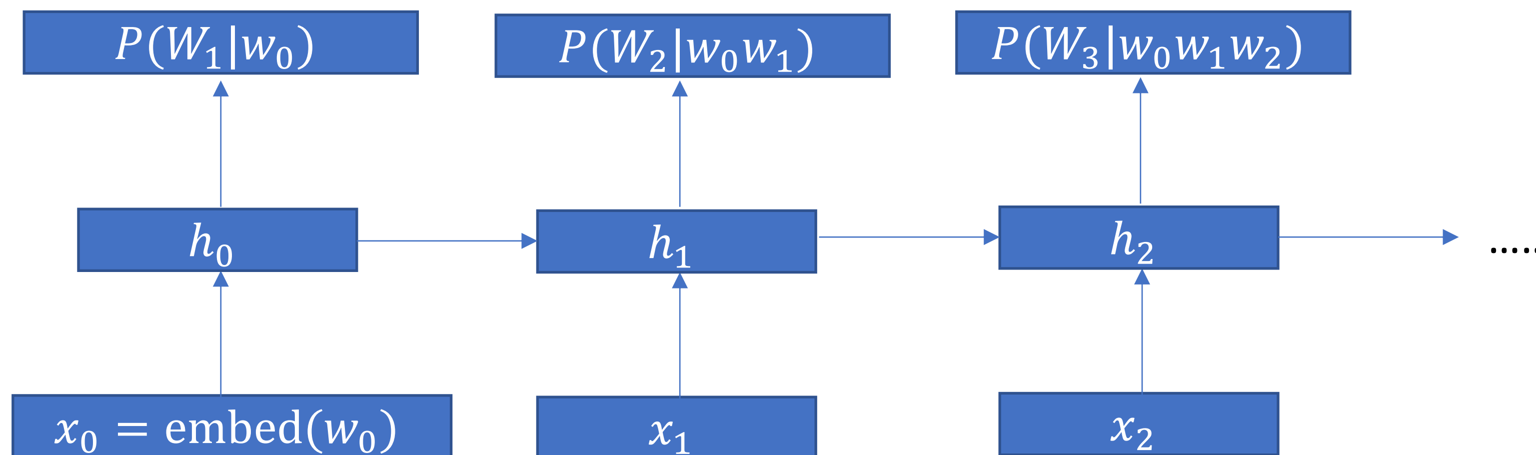
- Complete formulation:

$$h_t = \sigma(W_{ih} x_t + W_{hh} h_{t-1} + b_h)$$

$$y_t = \text{softmax}(W_{ho} h_t + b_o)$$

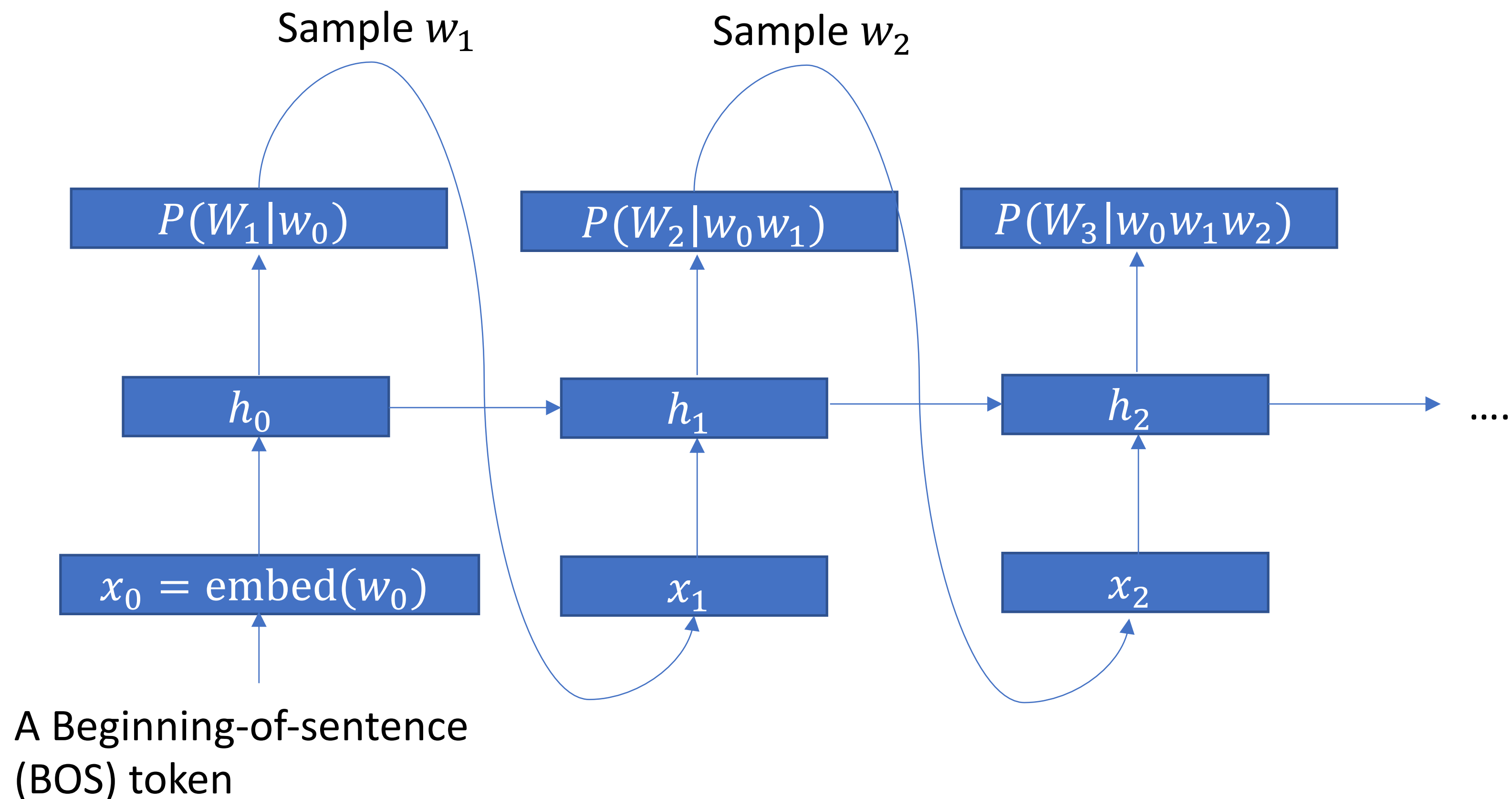
$$L(w) = \sum_i -\log P(w_i | w_{0..i-1})$$

- It's efficient: During training, we just feed the sequence (sentence) once into the RNN, and we get the output (loss) on every timestep.



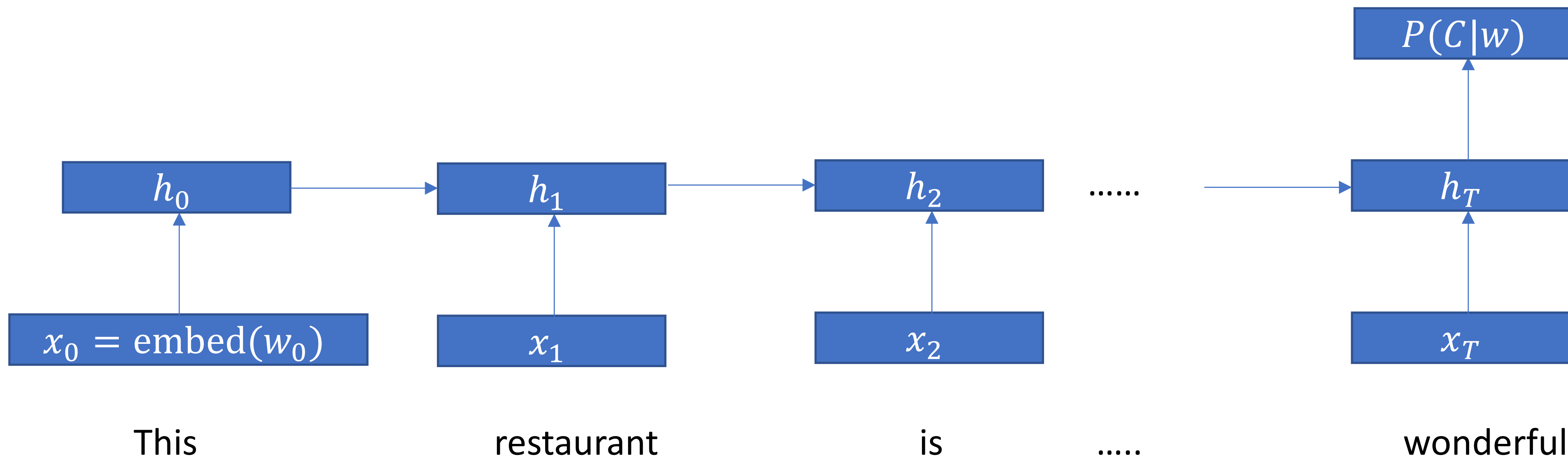
Generation with RNNLM

- We can do text generation with a trained RNNLM:
- At each time step t, we sample w_t from $P(W_t| \dots)$, and feed it to the next timestep!
- LM with this kind of generation process is called autoregressive LM.



RNN for text classification

- The last hidden state h_t can be regarded as an encoding of the whole sentence, on which you can add a linear classifier head.



(Brief) Word2vec

- The Word2vec project shows that if *we just want the word embeddings*, it can be trained in a very efficient way.

- Its training adopts the principle of *distributional hypothesis*.

"The meaning of a word is its use in the language"

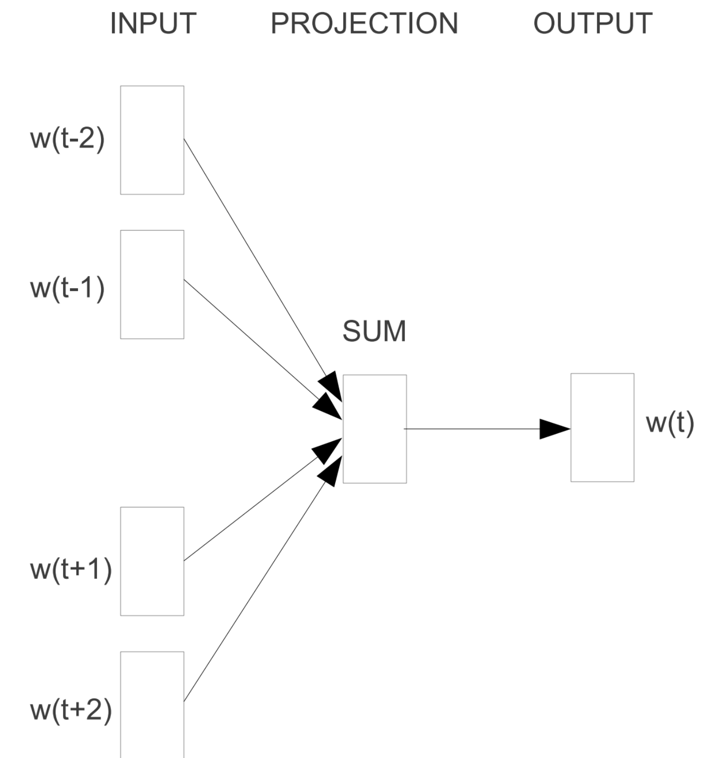
[Wittgenstein PI 43]

"You shall know a word by the company it keeps"

[Firth 1957]

If A and B have almost identical environments we say that they are synonyms.

[Harris 1954]



CBOW

Efficient Estimation of Word Representations in Vector Space

Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

Word2Vec: The vector arithmetic

- We found the trained embeddings have amazing arithmetic properties.

- For example:
- emb(king)-emb(man)+emb(woman)=emb(queen)!

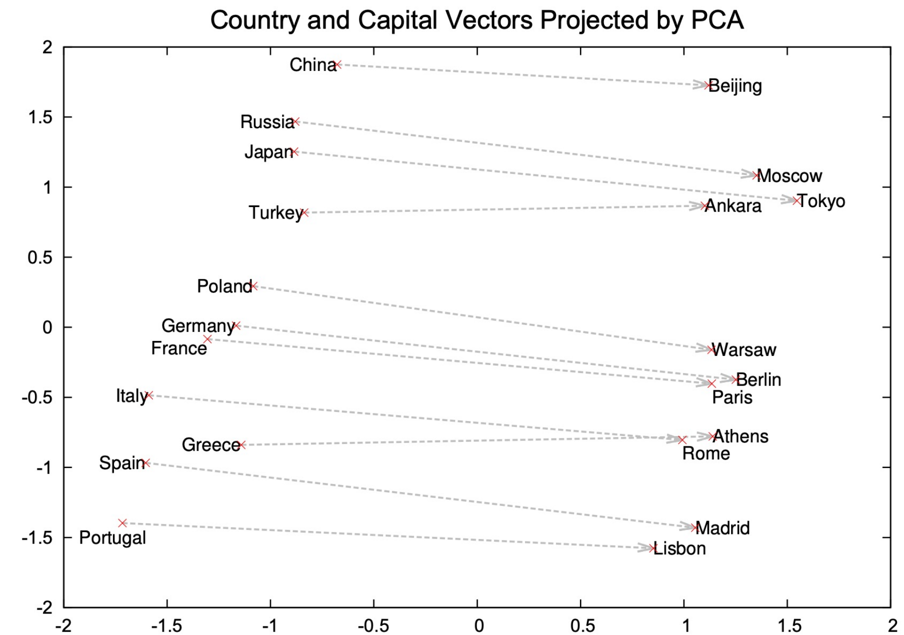


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

Distributed Representations of Words and Phrases and their Compositionality

Tomas Mikolov
Google Inc.
Mountain View
mikolov@google.com

Ilya Sutskever
Google Inc.
Mountain View
ilyasu@google.com

Kai Chen
Google Inc.
Mountain View
kai@google.com

Greg Corrado
Google Inc.
Mountain View
gcorrado@google.com

Jeffrey Dean
Google Inc.
Mountain View
jeff@google.com

Word2Vec for initialization

- The training of word2vec can be done very efficiently on large unsupervised data (due to speed-up techniques like negative sampling).

- A good strategy: First pretrain a set of good word embeddings with a large corpora. Then use it to initialize the embedding layer of your NN model. And finally finetune it on labeled data (e.g., for classification).

Recommended reading

- Definitely play with some pytorch (official) tutorials. (you don't need GPU to do that)

- The deep learning book Chapter 6
- https://www.deeplearningbook.org/contents/mlp.html

- The word2vec paper:
- https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

Thanks!

- Next lecture, we will continue to LSTM, attention, transformers, etc.