Natural Language Processing
Neural Networks I

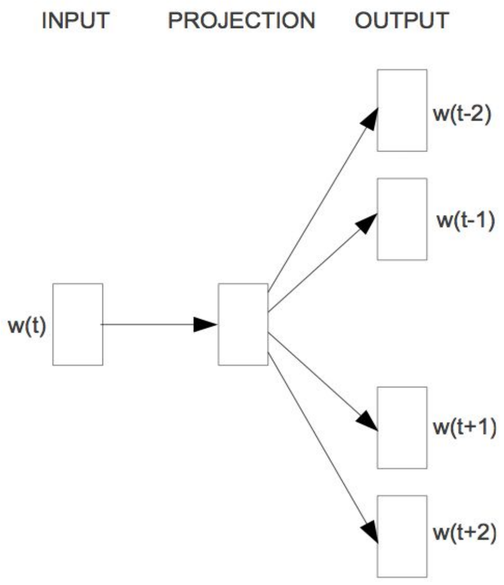
Sofia Serrano
sofias6@cs.washington.edu

Credit to Tianxing He, Yulia Tsvetkov, and Noah Smith for slides

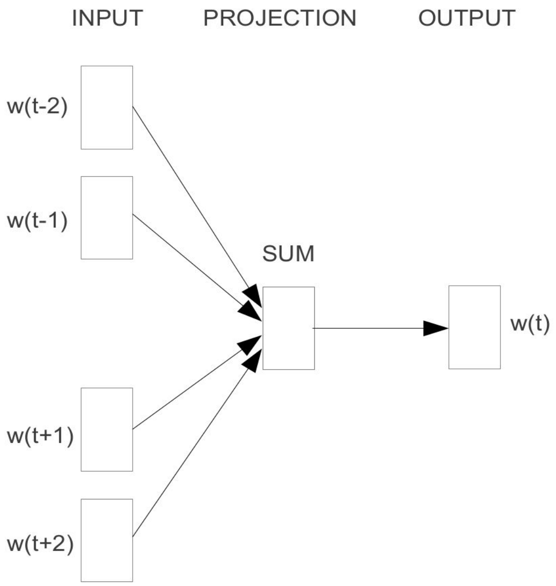
Announcements

- Midterm course eval form (online) is out– please let us know how we're doing!
    - Anonymous, takes at most a few minutes
    - Available through the end of the day today
- A2 is out– start early!! (Please!)

Word2Vec



Skip-gram



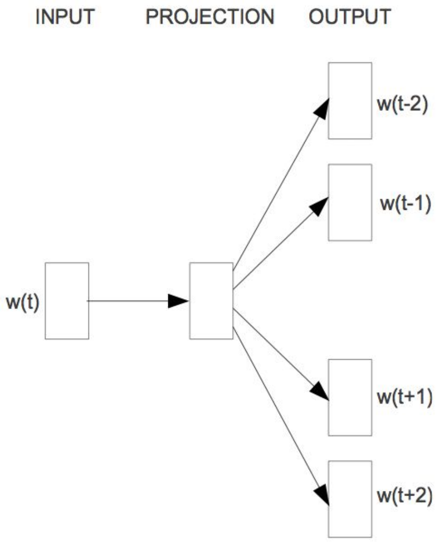
CBOW

- [Mikolov et al.' 13]

Skip-gram Prediction

- Predict vs Count

the cat sat on the mat



Skip-gram

Skip-gram Prediction

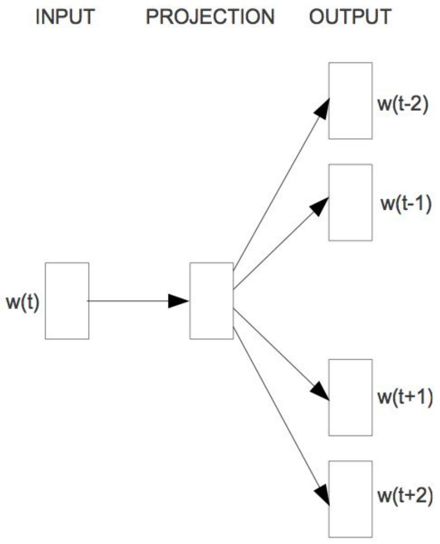
- Predict vs Count

the cat sat on the mat

w_t = the → CLASSIFIER →

w_{t-2} = <start$_{-2}$>
w_{t-1} = <start$_{-1}$>
w_{t+1} = cat
w_{t+2} = sat

context size = 2



Skip-gram

Skip-gram Prediction

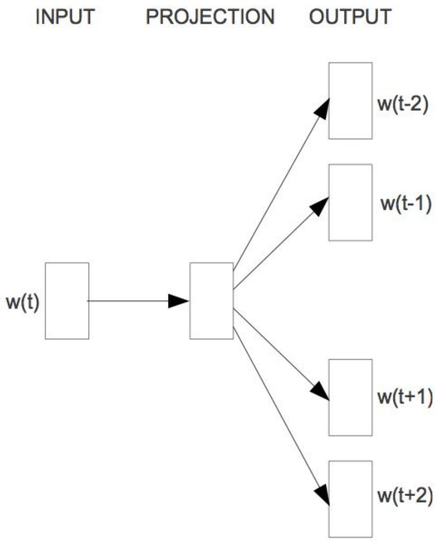
- Predict vs Count

the cat sat on the mat

w_t = cat → **CLASSIFIER** →

w_{t-2} = <start$_{-1}$>
w_{t-1} = the
w_{t+1} = sat
w_{t+2} = on

context size = 2



Skip-gram

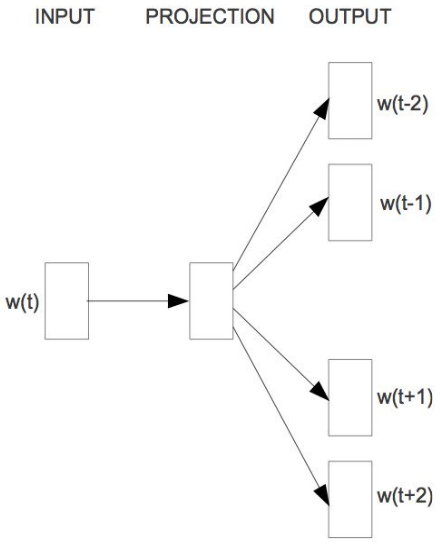
Skip-gram Prediction

- Predict vs Count

the cat sat on the mat

w_t = sat → **CLASSIFIER** → w_{t-2} = the, w_{t-1} = cat, w_{t+1} = on, w_{t+2} = the

context size = 2



Skip-gram

Skip-gram Prediction

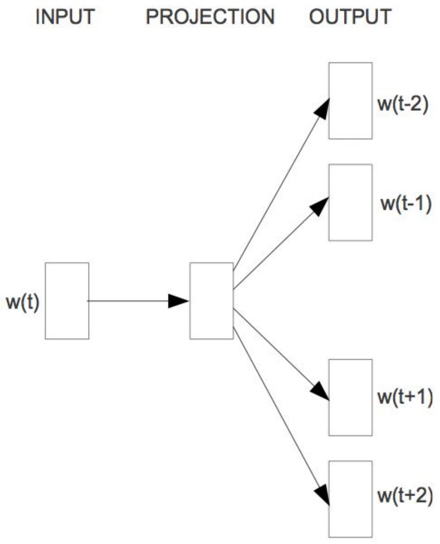
- Predict vs Count

the cat sat <u>on</u> the mat

w_t = on → **CLASSIFIER** →

w_{t-2} = cat
w_{t-1} = sat
w_{t+1} = the
w_{t+2} = mat

context size = 2



Skip-gram

Skip-gram Prediction

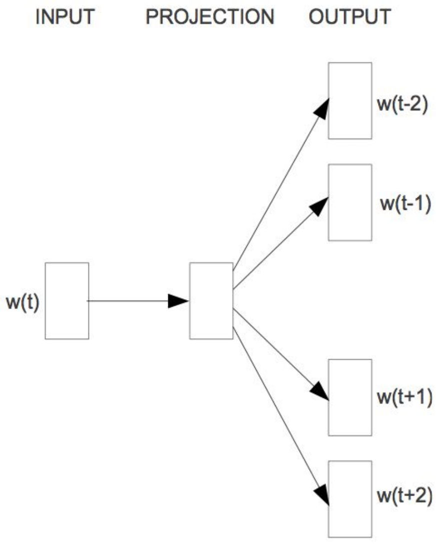
- Predict vs Count

the cat sat on <u>the</u> mat

w_t = the → **CLASSIFIER** →

w_{t-2} = sat
w_{t-1} = on
w_{t+1} = mat
w_{t+2} = <end$_{+1}$>

context size = 2



Skip-gram

Skip-gram Prediction

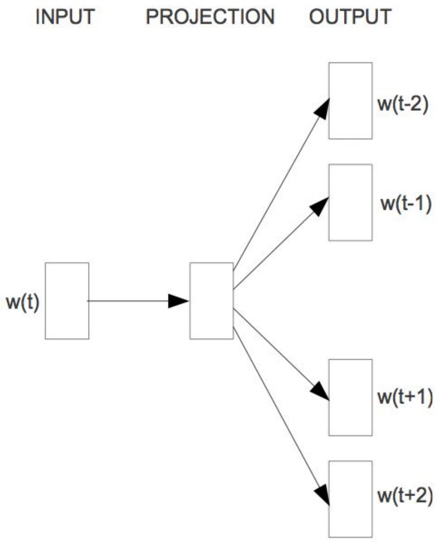
- Predict vs Count

the cat sat on the mat

w_t = mat → CLASSIFIER →

w_{t-2} = on
w_{t-1} = the
w_{t+1} = <end$_{+1}$>
w_{t+2} = <end$_{+2}$>

context size = 2



Skip-gram

Skip-gram Prediction

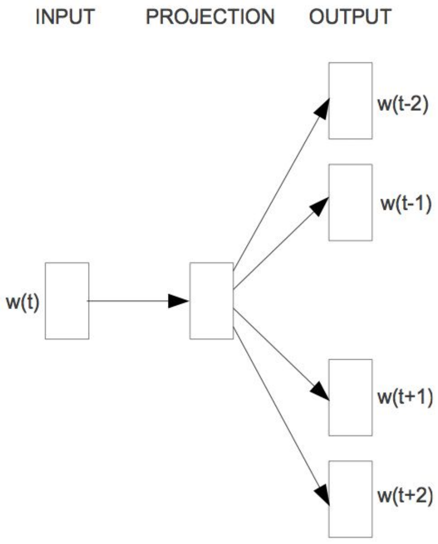
- Predict vs Count

w_t = the → **CLASSIFIER** →

w_{t-2} = sat
w_{t-1} = on
w_{t+1} = mat
w_{t+2} = <end$_{+1}$>

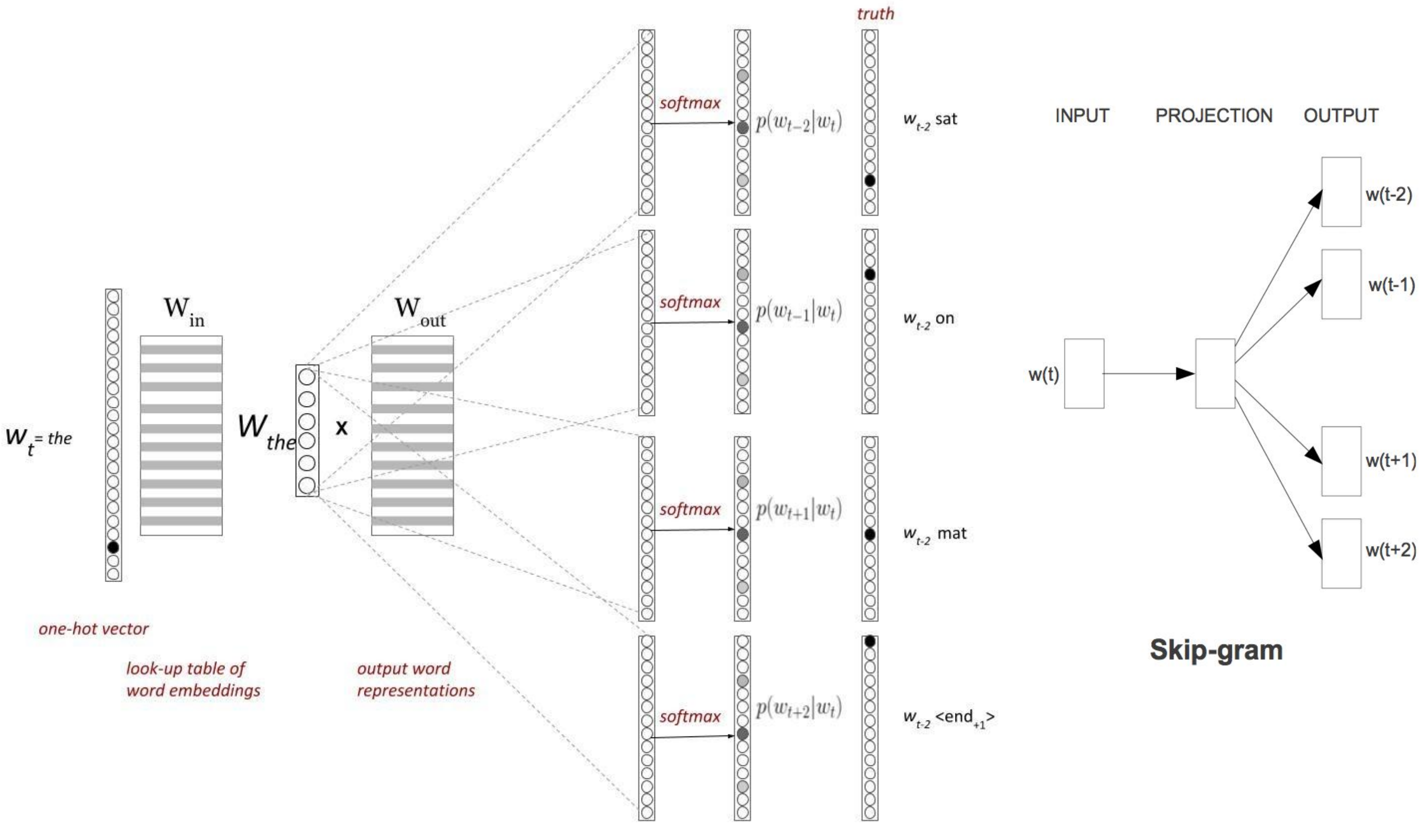
w_t = the → **CLASSIFIER** →

w_{t-2} = <start$_{-2}$>
w_{t-1} = <start$_{-1}$>
w_{t+1} = cat
w_{t+2} = sat

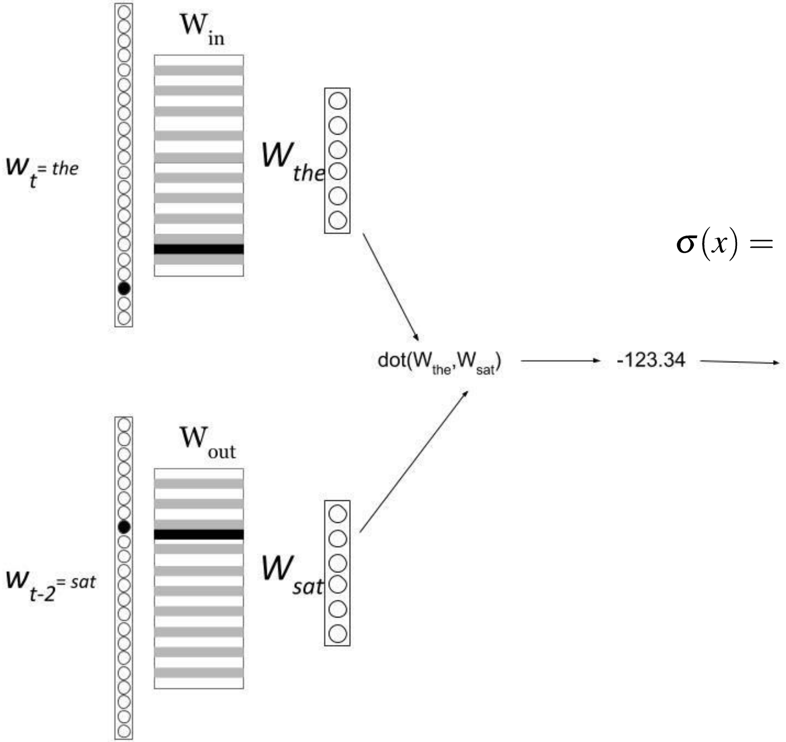


Skip-gram

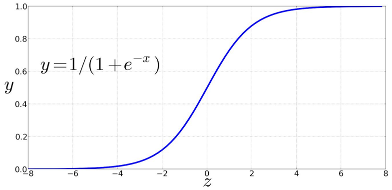
Skip-gram Prediction



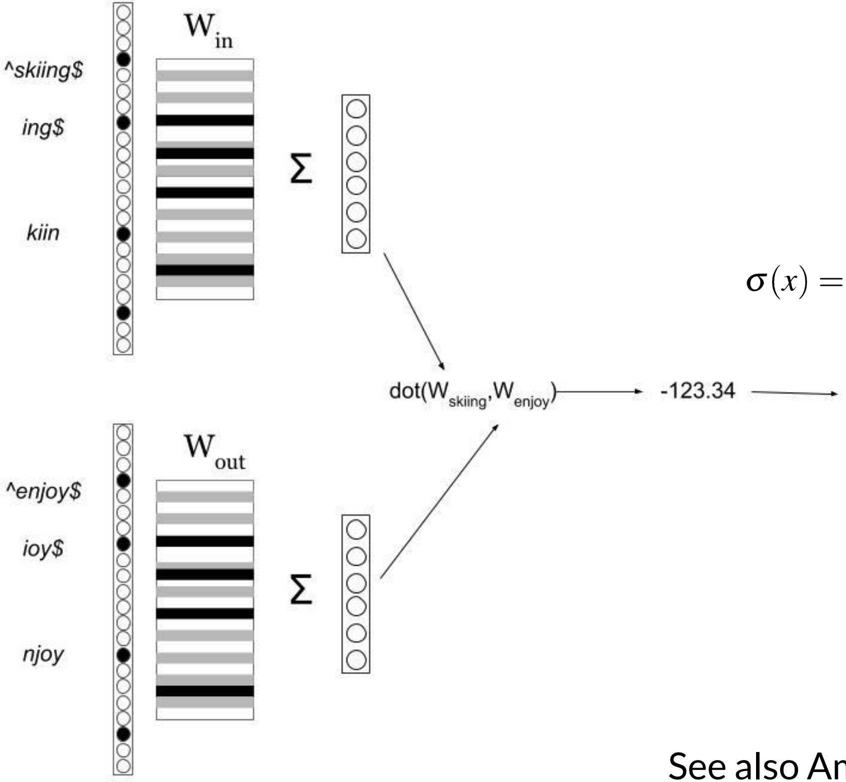
How to compute p(+|t,c)?



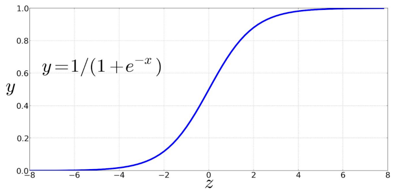
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



FastText



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



See also Amit Chaudhary's [blog post explaining FastText]

Typical traits of these embeddings

Automatically learn some analogies pretty well

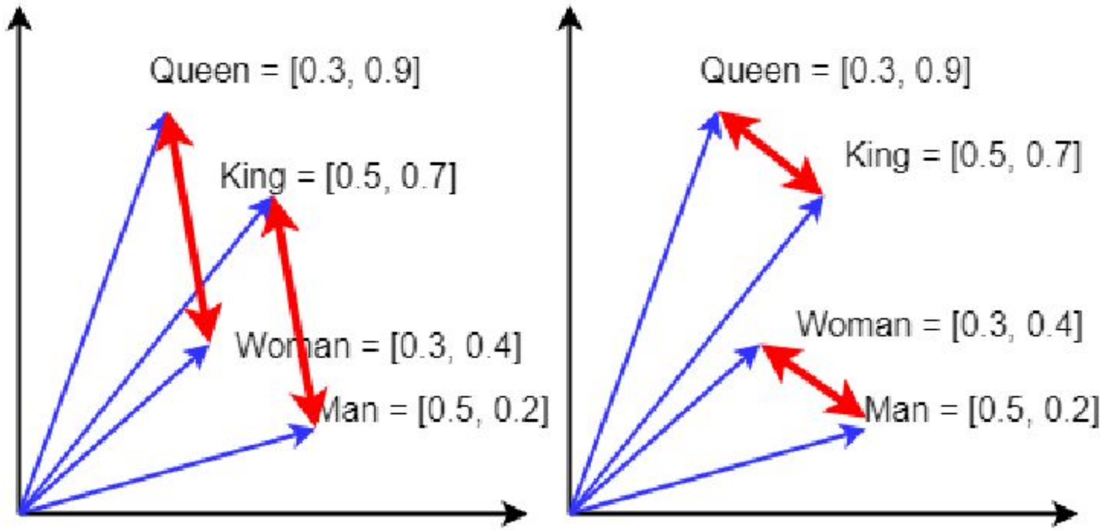


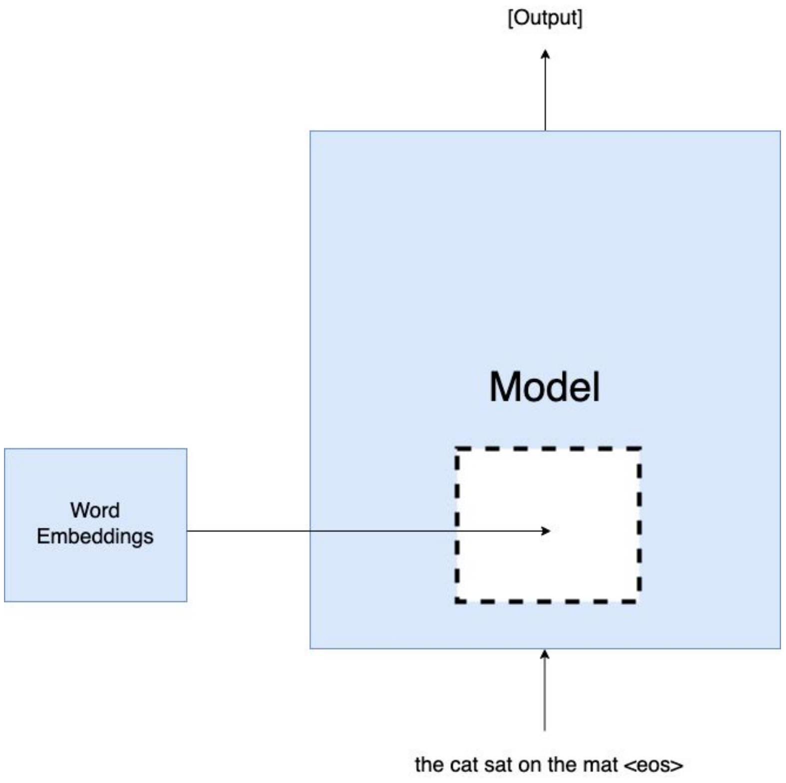
Figure from Sutor et al. MIPR 2019

Takeaways from word representations

What we’ve learned

- The contexts in which a word typically appears (i.e., the tokens that typically appear around it) tell us a lot about that word
- We can use those contexts to automatically learn more powerful representations of words than just a one-hot encoding
- These “word embeddings” can plug in as parameters in models of your choice

Let's talk about more powerful kinds of functions of our input text!



specifically...

- Feedforward neural networks
- Recurrent neural networks

  (Vanilla RNNs, then and LSTMs and GRUs)

Feedforward neural networks

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.

Brief Review: logistic regression (LR)

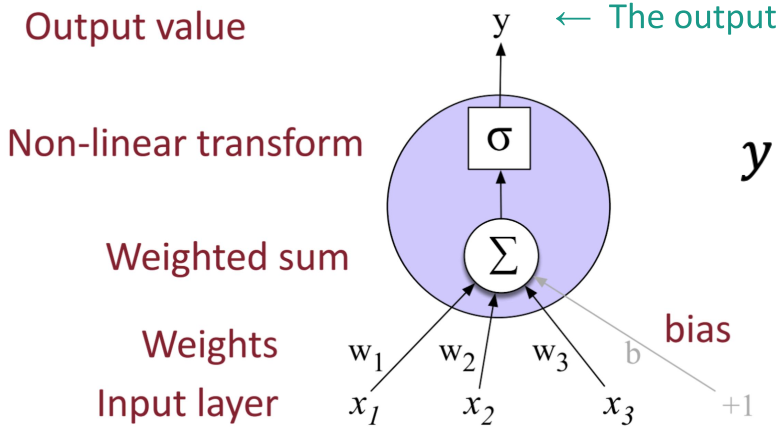
$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

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

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

A neural unit for feature extraction

- In order to do the final prediction, we perhaps 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 !

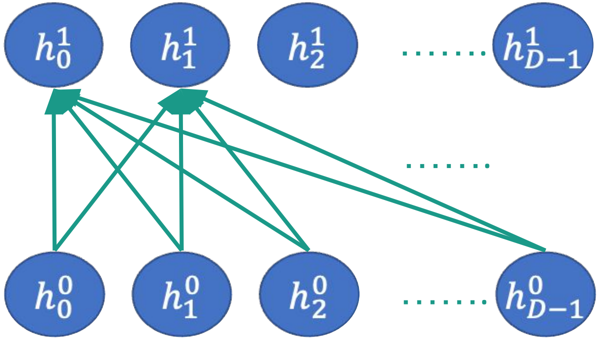


← The output y of this single neuron is a scalar value.

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

One hidden layer of neural network

- A layer of D neurons consists a hidden layer.



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

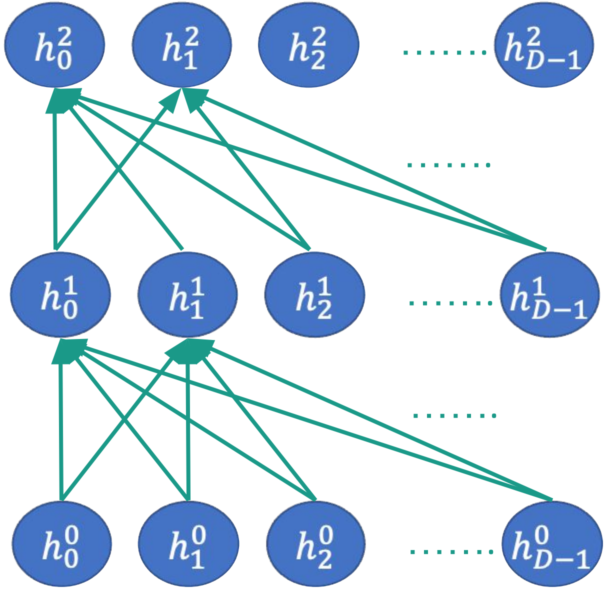
We aggregate the weights into W^0. The i-th row in W^0 corresponds to the weight w in the i-th neuron whose output is h_i^1.

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.)

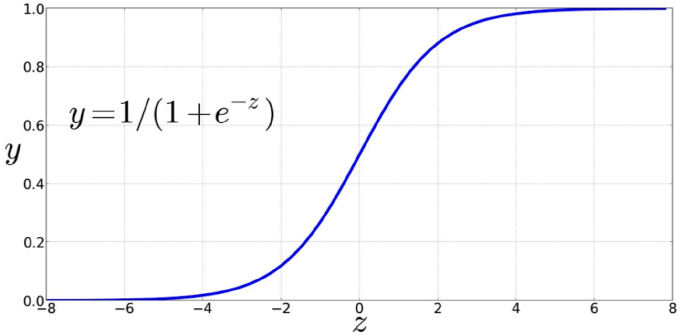
This is called a multi-layer perceptron (MLP) or a feedforward neural network.
It’s the simplest type of neural network. (we will learn about more complicated ones in these two lectures)

Choice of activation function

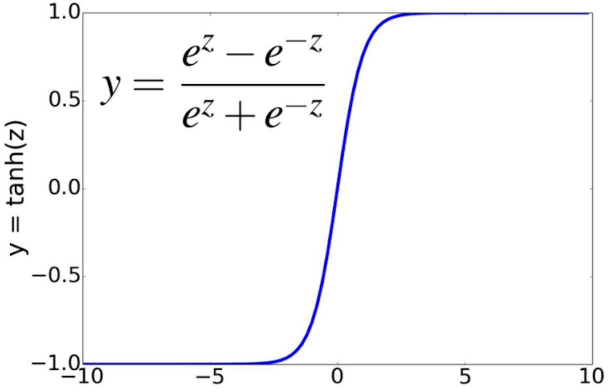
- The sigmoid function σ is one type of activation function.

Sigmoid

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

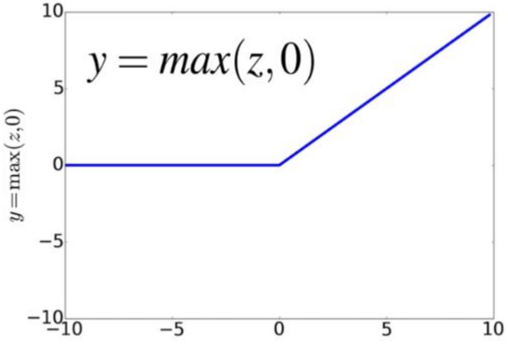


tanh



ReLU

Rectified Linear Unit



Tanh and ReLU have been empirically shown to outperform sigmoid.

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

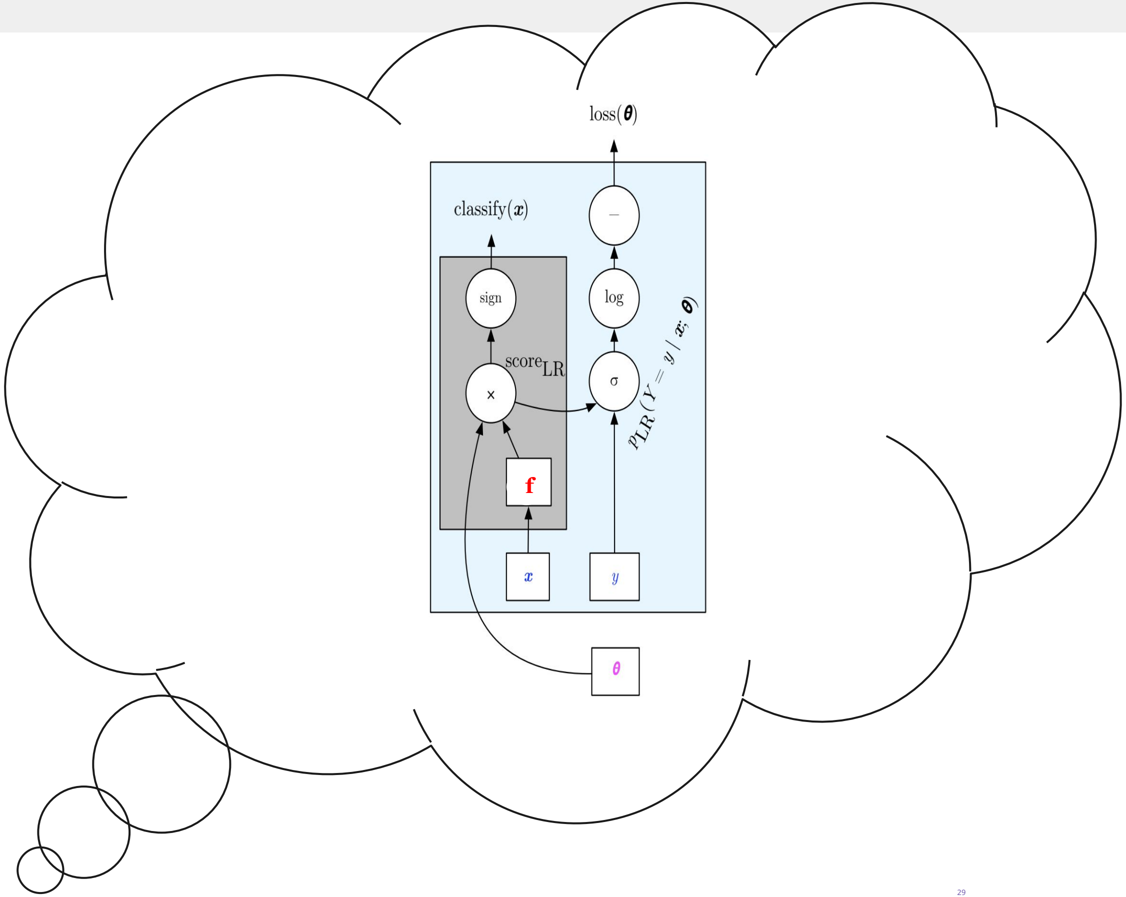
What it's like in pytorch

- 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
```

How do we learn our neural network's parameters?



(Stochastic) Gradient Descent!

(even though our function's probably no longer convex)

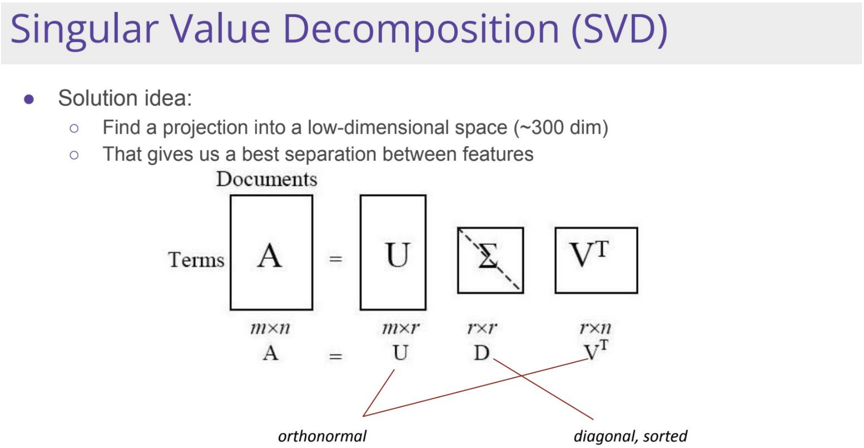
Brief summary

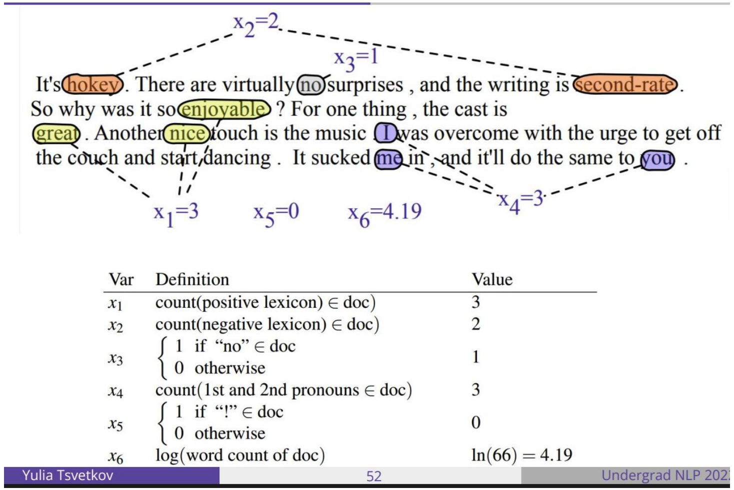
We now know how to compute the forward pass and backward pass of a feedforward NN.

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!

Philosophy (mindset) 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.





Philosophy (mindset) of neural networks for NLP

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

- What we care about now is:
- 1: Using a powerful NN architecture
- 2: Using large amounts of data
- 3: Using a useful learning objective

… but in practice, those word vectors from Wednesday are still really useful.

Keep in mind: we're not in the realm of nice convex functions anymore! Learning is chancier/more difficult!

Initializing the word embedding parameters at the beginning of the model to pretrained word vectors, in practice, is often a *much* better starting point of the parameter space, and makes it much easier for the model to learn a good set of parameters.

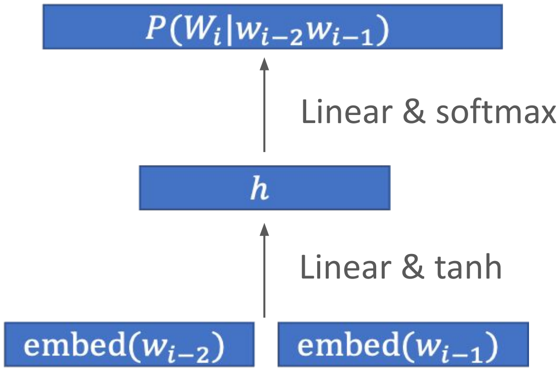
Example: Feedforward trigram language model

- 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 neural network language model



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

- 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 networks

Revisiting our bag-of-words assumption

What if we had a way of computing a (learned) function of input text that *didn't* require that whole input to be compressed into a fixed-length vector?

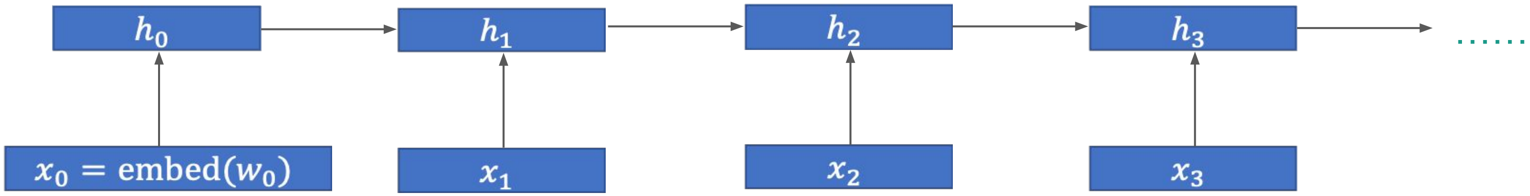
- Would free us from our bag-of-words assumption

How could we structure such a function such that we still only have to learn a fixed number of parameters?

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 $w_0 \dots . w_{t-1}$.
- 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

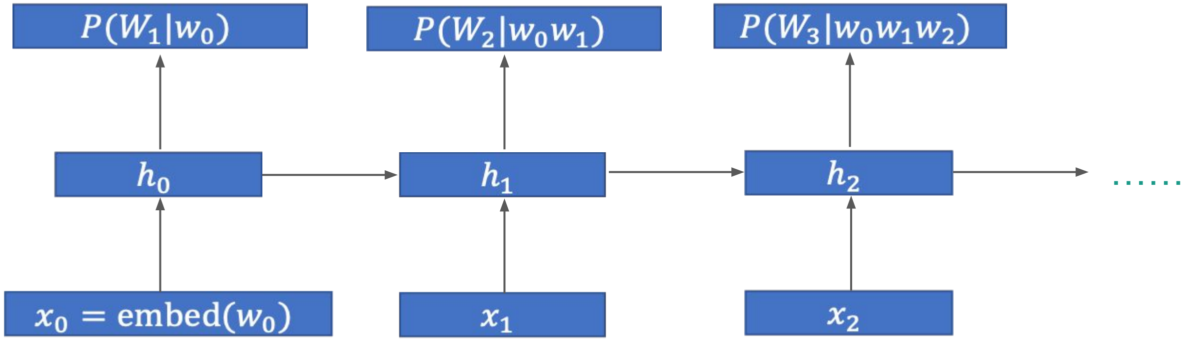
- Complete formulation:

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

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

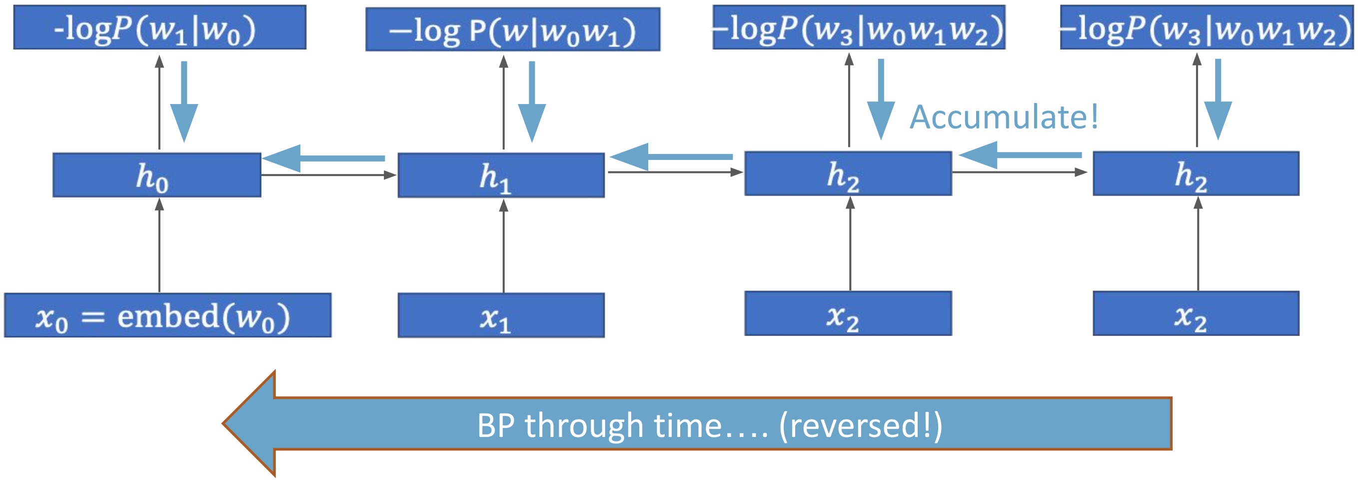
$$\text{L(w)}=\sum_i -\log P(w_i|w_{0..i-1})$$

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



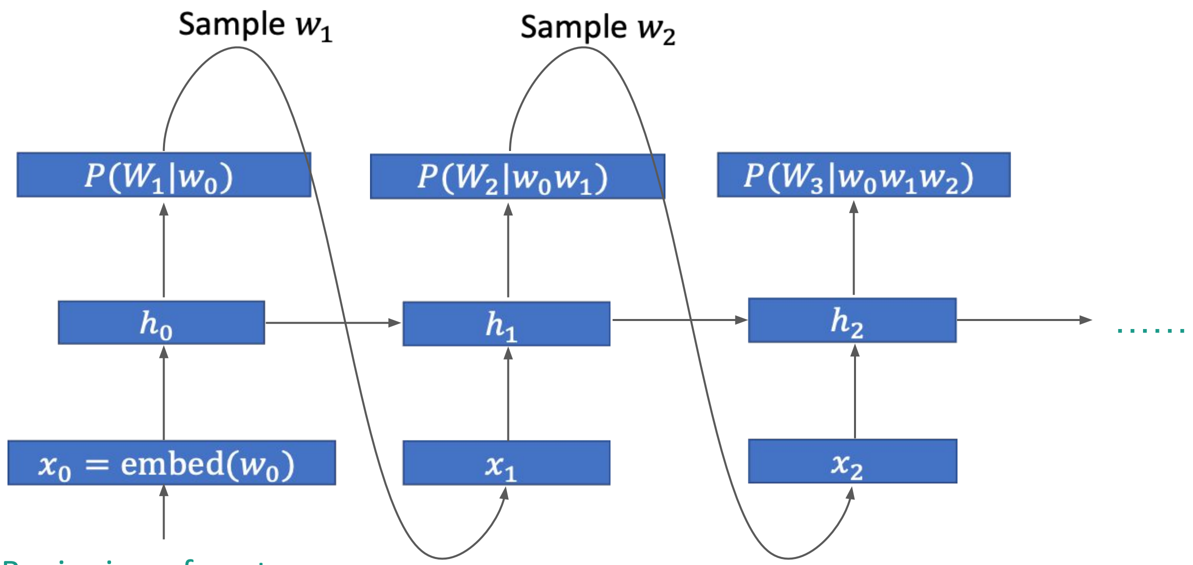
Backpropagation through time (BPTT)

- To do BP, again follow the reverse topological order.
- The error vector of h_t is an accumulation of errors from time t and future time steps!



Generation with an RNN language model

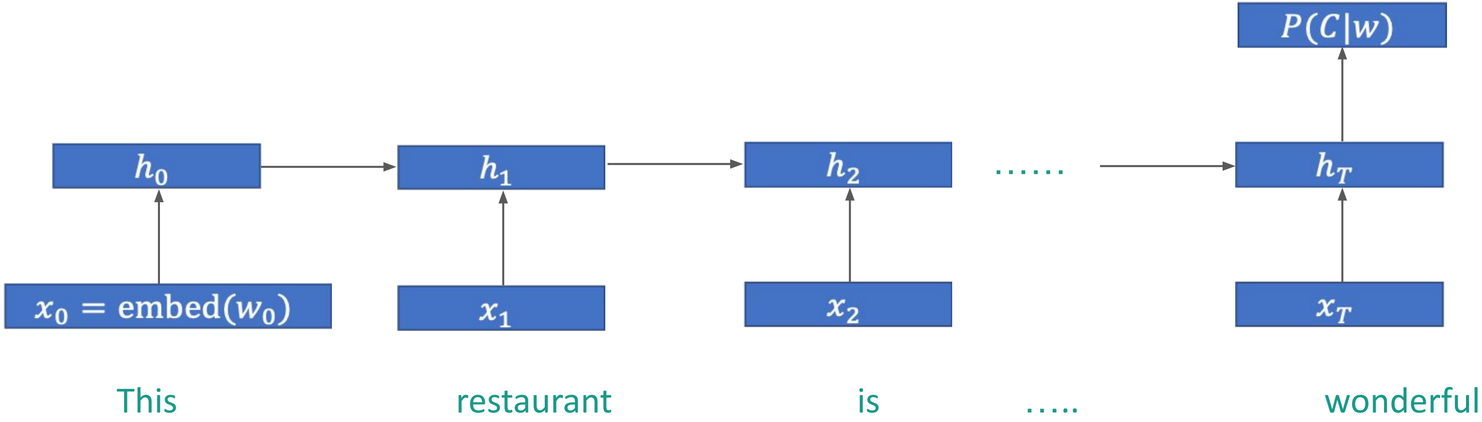
- We can do text generation with a trained RNNLM:
- At each time step t, we sample w_t from $P(W_t| \ldots)$, and feed it to the next timestep!
- LM with this kind of generation process is called autoregressive LM.



A Beginning-of-sentence (BOS) token

RNN for text classification

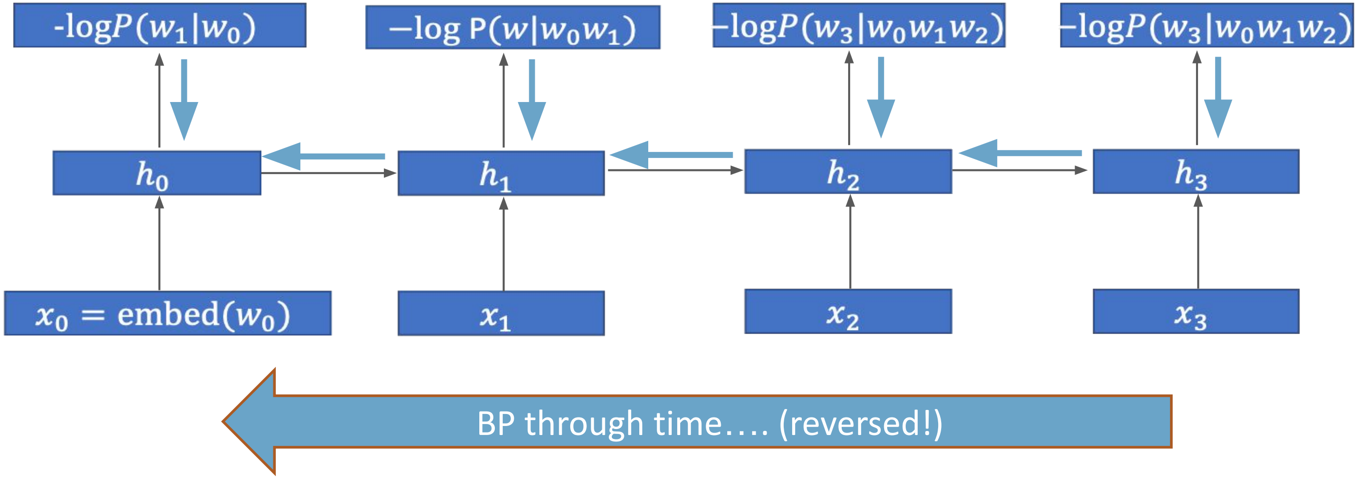
- The last hidden state $h_t$can be regarded as an encoding of the whole sentence, on which you can add a linear classifier head.



Gradient exploding and gradient vanishing

- In BPTT, we could meet two serious problems. They are called gradient exploding (error vector become too large) and gradient vanishing (error vector become too small).

- Gradient exploding is more serious because it makes training impossible.



Intuition: Gradient exploding and gradient vanishing

We make two crude simplifications: Simpilify: $h_t = W_{hh}h_{t-1} + W_{ih}x_t$

And only considering L_t

$$\begin{tikzcd}
 & & & & L_t \arrow[d] \\
h_1 & \cdots\cdots \arrow[l, "W_{hh}"'] & h_{t-2} & h_{t-1} \arrow[l, "W_{hh}"'] & h_t \arrow[l, "W_{hh}"']
\end{tikzcd}$$

Simpilify: $h_t = W_{hh}h_{t-1} + W_{ih}x_t$, we get the following during backprop:

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial h_t} {W_{hh}^T}^{t-1} \otimes h_1 + \frac{\partial L_t}{\partial h_t} {W_{hh}^T}^{t-2} \otimes h_2 + \cdots + \frac{\partial L_t}{\partial h_t} \otimes h_t$$

Further approximation, think everything as a scalar…

$W_{hh} < 1$: Gradient Vanishing -> LSTM …

$W_{hh} > 1$: Gradient Exploding -> Gradient Clipping

Gradient clipping for the exploding problem

It's simple!

Assume we want to set the maximum norm of gradient to be γ

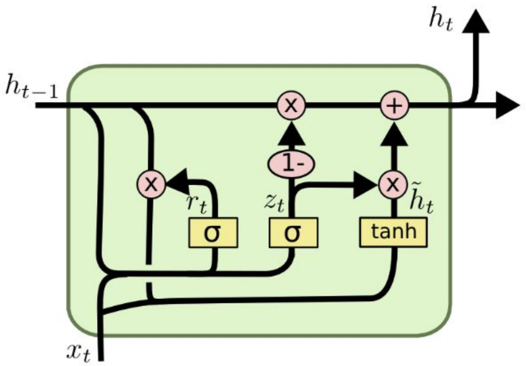
$$\text{clip}(\nabla L) = \min\left\{1, \frac{\gamma}{\left\|\nabla L\right\|_2}\right\} \nabla L.$$

In practice, γ is a hyper-parameter, and is usually set to be 1 or 0.5.

LSTMs and GRUs
(Long Short-Term Memory and Gated Recurrent Units)

LSTM or GRU for gradient vanishing

- Historical note: The LSTM (long-short term memory) network was first used in (Sundermeyer et.al. 2012), dealing with the g-vanishing problem.
- Then, GRU (gated recurrent unit) is proposed as a simplification of LSTM.
- We will discuss GRU because it's simpler and has the same core idea.



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

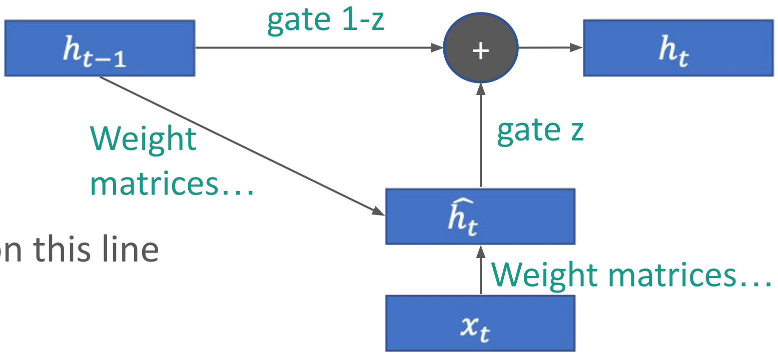
Christopher Olah's blog post on Understanding LSTM Networks is great btw

Gated recurrent unit for gradient vanishing

GRU is by itself, a small neural network, input: x_t,h_{t-1} , output: h_t

$$
\begin{aligned}
z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\
\hat{h}_t &= \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\
h_t &= z_t \odot \hat{h}_t + (1 - z_t) \odot h_{t-1}
\end{aligned}
$$

← Let's just focus on this line



Variables

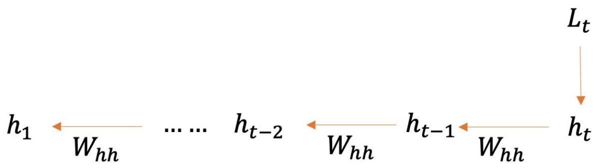
- x_t : input vector
- h_t : output vector
- $\hat{h}_t$: candidate activation vector
- z_t : update gate vector
- r_t : reset gate vector
- W, U and b: parameter matrices and vector

Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling

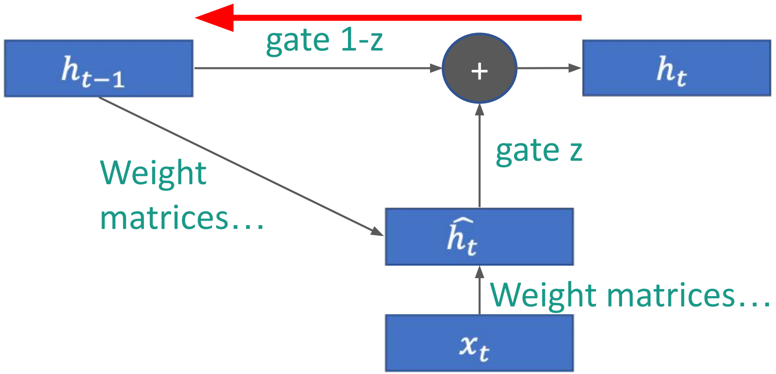
Junyoung Chung   Caglar Gulcehre   KyungHyun Cho   Yoshua Bengio
Université de Montréal   Université de Montréal
CIFAR Senior Fellow

Gated recurrent unit for gradient vanishing

- Think about back-propagation from h_t to h_{t-1}.
- There will be multiple paths, and the errors will be summed up. But in the red path, it does not involve any weight matrix! It's just $(1-z) \odot h_{t-1}$.
- This path alleviates gradient vanishing.



The RNN case for reference.

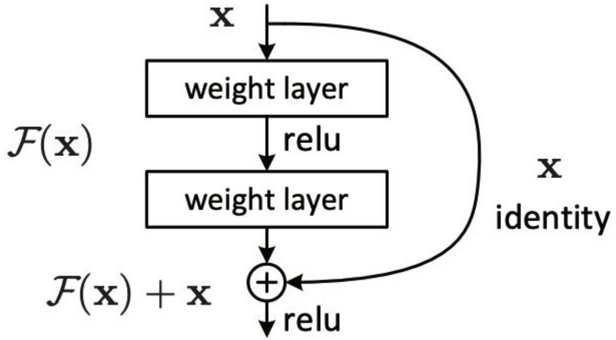


Residual connection in deep feedforward NN

- (Diverge topic a bit) Similar idea can be used to help us build deeper networks.

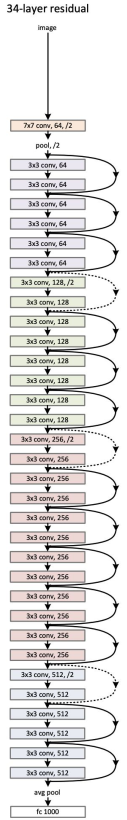
- Adding a direct link between hidden layers:
- $h_{l+1} = h_l + F(h_l)$
- F may include linear transform,ReLU, gating, etc.

- We will revisit this residual connection in transformers!



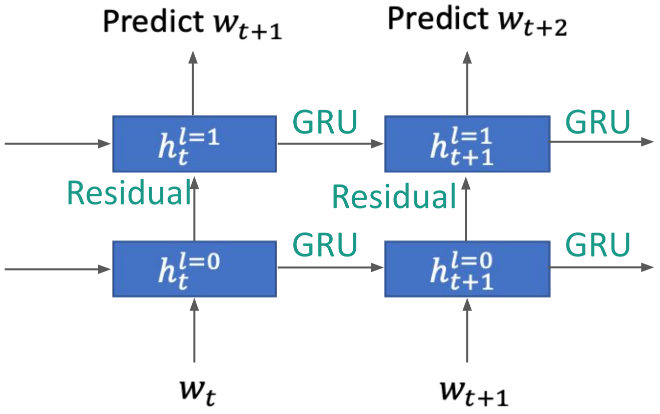
Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com



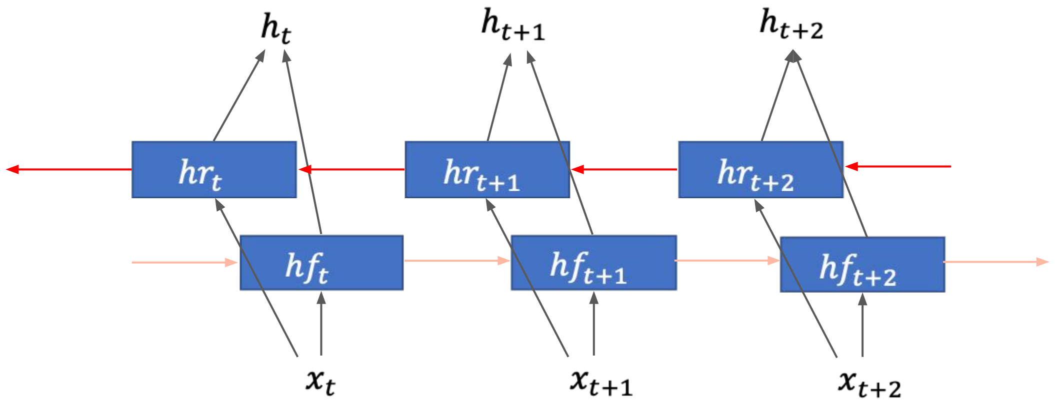
Philosophy: Combining NN modules

- We have now learnt several neural modules (rnn, lstm/gru, etc.), which are by themselves, a small neural network. We can combine different modules together to form a large neural model.
- For example, we build a AR-LM by stacking several GRU layers, and linking them with a residual link:



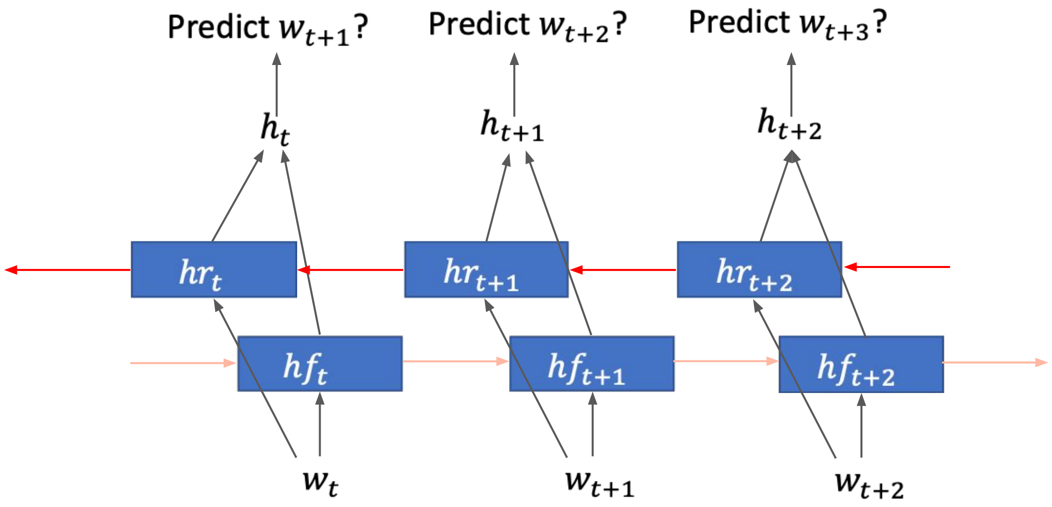
Bi-directional RNN

- In uni-directional RNN, h_t has context from the "left".
- For some applications (e.g., part-of-speech tagging), it would be useful if h_t has bi-directional context.
- We can achieve this by adding a layer of RNN with reversed direction.
- Exercise: what's the topological order of this graph (it's still a DAG!)?



Bi-directional RNN for language modeling?

- Exercise: When we switch from a uni-rnn to a bi-rnn, and we don't change anything else, can we still do language modelling?
- Answer: No! In a language model, we can not utilize information from the future!

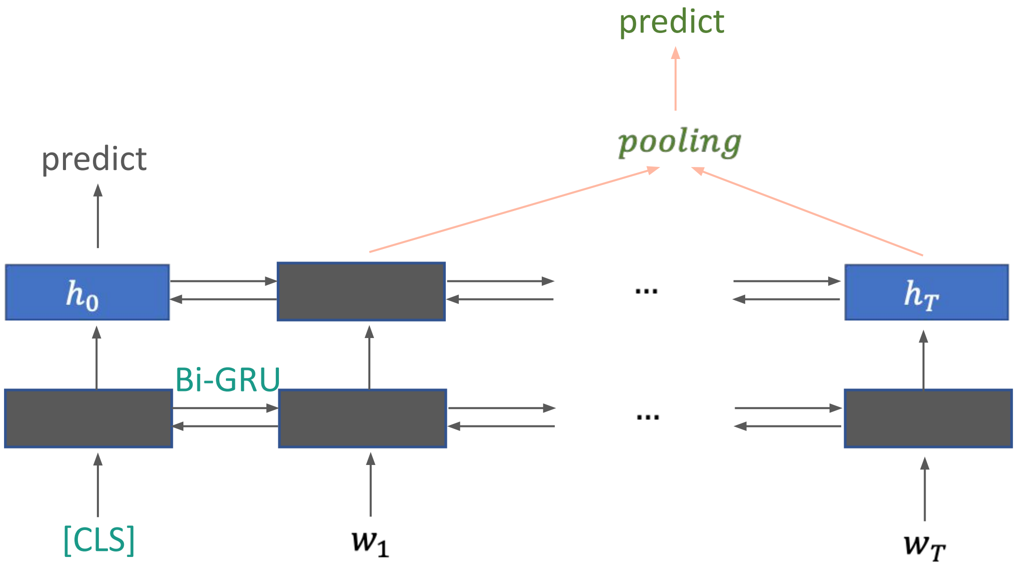


Bi-directional RNN for encoding a sequence as a fixed-length vector?

There are several ways to get a fixed-length sequence encoding from a bi-rnn:

Way1: add a special token to the input.

Way2: do a max-pooling or mean-pooling of the hidden states.



Next class

Sequence labeling