10-301/10-601 Introduction to Machine Learning

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Automatic Differentiation & Transformers

Matt Gormley, Henry Chai, Hoda Heidari

Lecture 19

Mar. 27, 2024

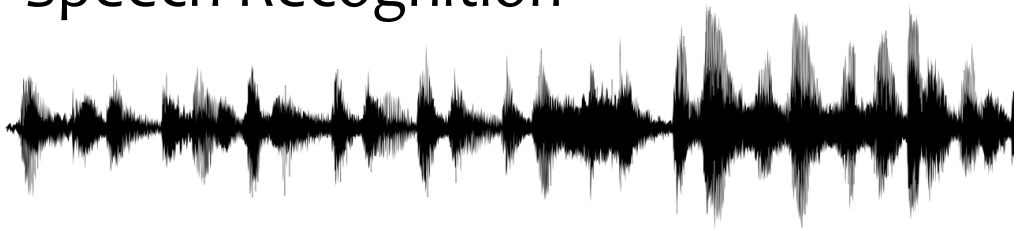
Reminders

- Homework 6: Learning Theory & Generative Models
  - Out: Mon, Mar 18
  - Due: Sun, Mar 24 at 11:59pm
- Exam 2: Thu, Mar 28, 7:00 pm - 9:00 pm

BACKGROUND: HUMAN LANGUAGE TECHNOLOGIES

Human Language Technologies

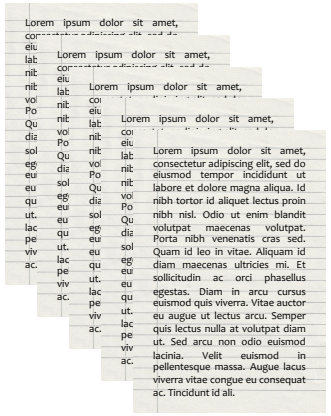
Speech Recognition



Machine Translation

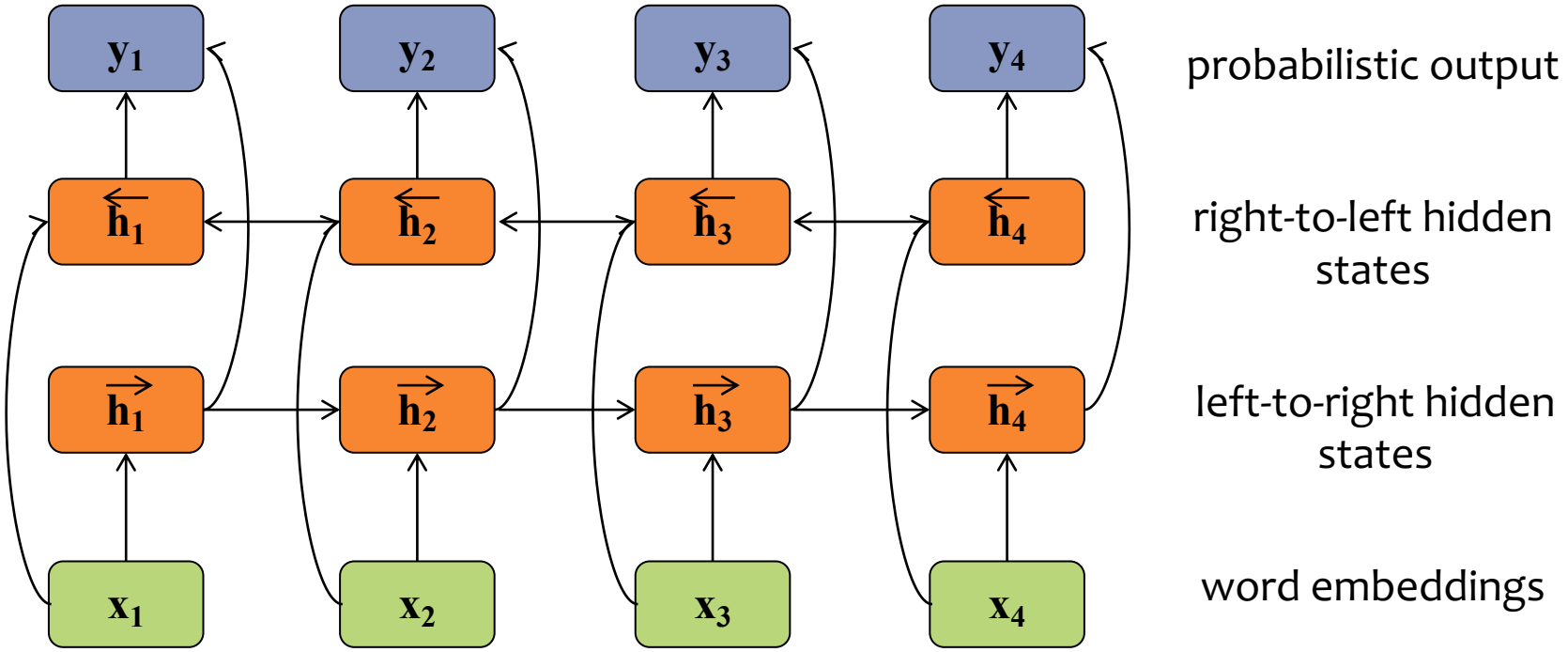
기계 번역은 특히 영어와 한국어와 같은 언어 쌍의 경우 매우 어렵습니다.

Summarization



Bidirectional RNN

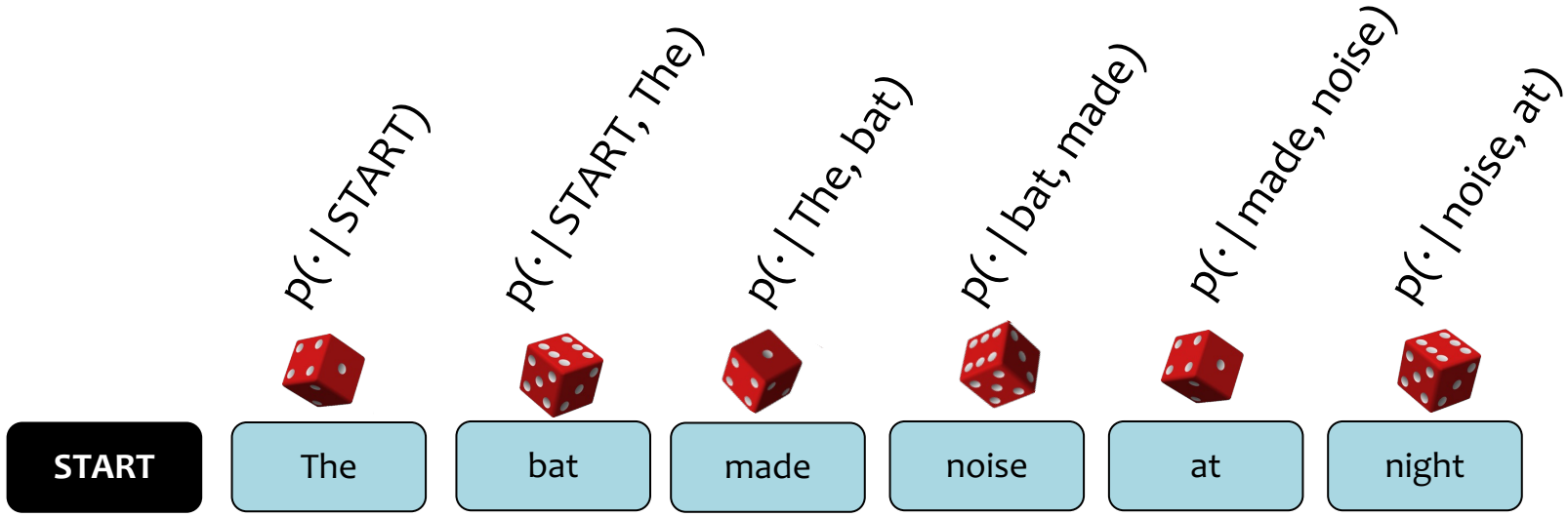
RNNs are a now commonplace backbone in deep learning approaches to natural language processing



BACKGROUND:
N-GRAM LANGUAGE MODELS

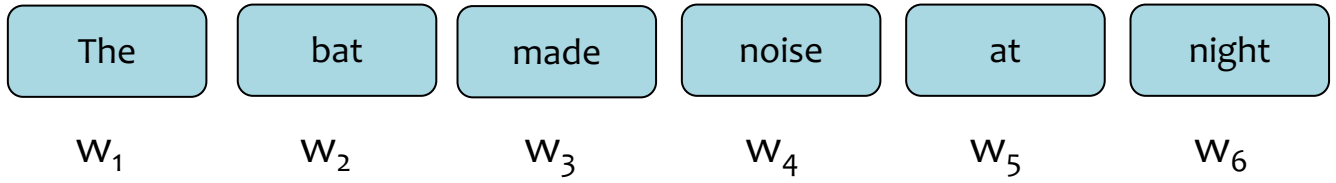
n-Gram Language Model

- *Goal*: Generate realistic looking sentences in a human language
- *Key Idea*: condition on the last n-1 words to sample the n^{th} word



n-Gram Language Model

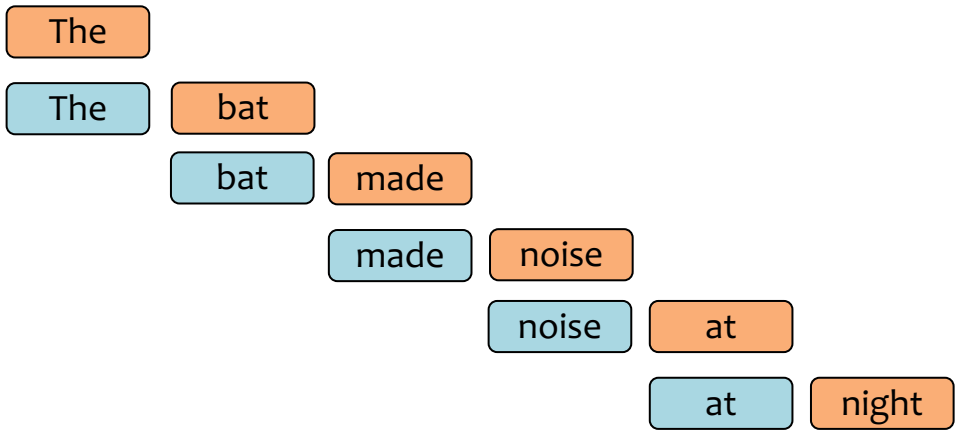
Question: How can we **define** a probability distribution over a sequence of length T?



n-Gram Model (n=2)

$$p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_{t-1})$$

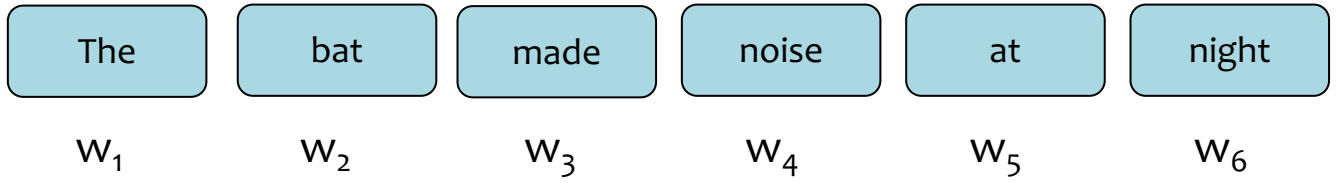
$p(w_1, w_2, w_3, \ldots, w_6) =$



$p(w_1)$
$p(w_2 \mid w_1)$
$p(w_3 \mid w_2)$
$p(w_4 \mid w_3)$
$p(w_5 \mid w_4)$
$p(w_6 \mid w_5)$

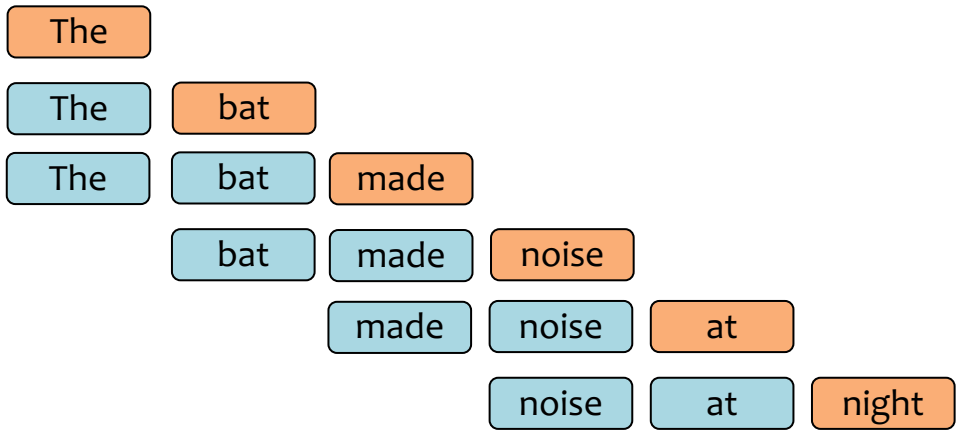
n-Gram Language Model

Question: How can we **define** a probability distribution over a sequence of length T?



n-Gram Model (n=3) $$p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_{t-1}, w_{t-2})$$

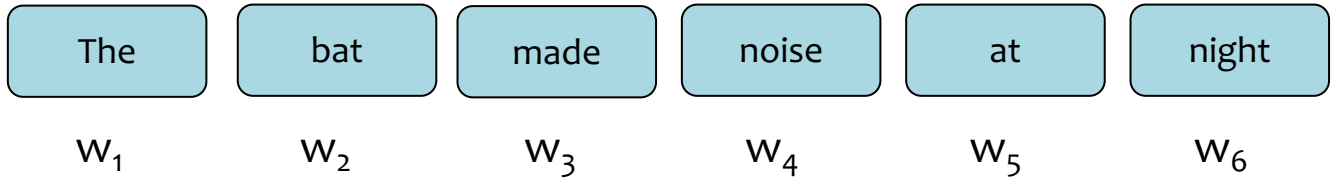
$p(w_1, w_2, w_3, \ldots, w_6) =$



$p(w_1)$
$p(w_2 \mid w_1)$
$p(w_3 \mid w_2, w_1)$
$p(w_4 \mid w_3, w_2)$
$p(w_5 \mid w_4, w_3)$
$p(w_6 \mid w_5, w_4)$

n-Gram Language Model

Question: How can we **define** a probability distribution over a sequence of length T?



n-Gram Model (n=3) $$p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_{t-1}, w_{t-2})$$

$p(w_1, w_2, w_3, \ldots, w_6) =$

$p(w_1)$

The

The

The

Note: This is called a **model** because we made some **assumptions** about how many previous words to condition on (i.e. only n-1 words)

Learning an n-Gram Model

Question: How do we **learn** the probabilities for the n-Gram Model?

$p(w_t \mid w_{t-2} = \text{The}, w_{t-1} = \text{bat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
ate	0.015
…	
flies	0.046
…	
zebra	0.000

$p(w_t \mid w_{t-2} = \text{made}, w_{t-1} = \text{noise})$

w_t	$p(\cdot \mid \cdot, \cdot)$
at	0.020
…	
pollution	0.030
…	
zebra	0.000

$p(w_t \mid w_{t-2} = \text{cows}, w_{t-1} = \text{eat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
corn	0.420
…	
grass	0.510
…	
zebra	0.000

Learning an n-Gram Model

Question: How do we **learn** the probabilities for the n-Gram Model?

Answer: From data! Just **count** n-gram frequencies

…the **cows eat grass**…
…our **cows eat hay** daily…
…factory-farm **cows eat corn**…
…on an organic farm, **cows eat hay** and…
…do your **cows eat grass** or corn?...
…what do **cows eat if** they have…
…**cows eat corn** when there is no…
…which **cows eat which** foods depends…
…if **cows eat grass**…
…when **cows eat corn** their stomachs…
…should we let **cows eat corn**?...

$p(w_t \mid w_{t-2} = \text{cows}, w_{t-1} = \text{eat})$

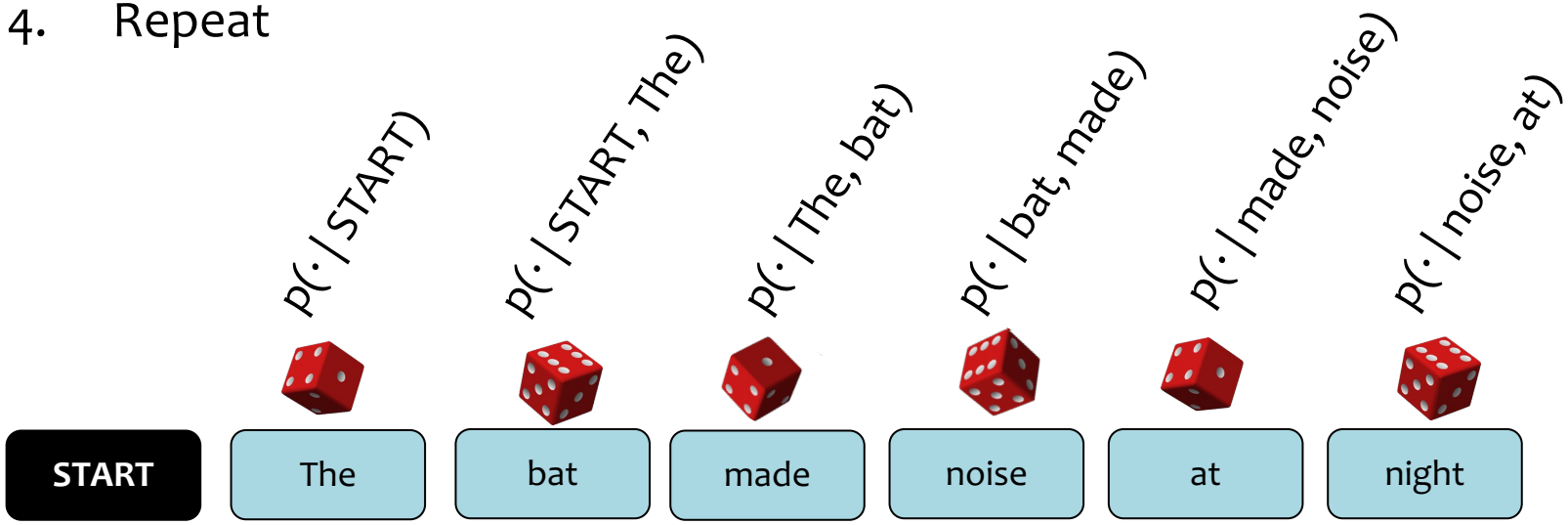
w_t	$p(\cdot \mid \cdot, \cdot)$
corn	4/11
grass	3/11
hay	2/11
if	1/11
which	1/11

Sampling from a Language Model

Question: How do we sample from a Language Model?

Answer:

1. Treat each probability distribution like a (50k-sided) weighted die
2. Pick the die corresponding to $p(w_t \mid w_{t-2}, w_{t-1})$
3. Roll that die and generate whichever word w_t lands face up
4. Repeat



Sampling from a Language Model

Question: How do we sample from a Language Model?

Answer:

1. Treat each probability distribution like a (50k-sided) weighted die
2. Pick the die corresponding to $p(w_t \mid w_{t-2}, w_{t-1})$
3. Roll that die and generate whichever word w_t lands face up
4. Repeat

Training Data (Shakespeaere)
I tell you, friends, most charitable care ave the patricians of you. For your wants, Your suffering in this dearth, you may as well Strike at the heaven with your staves as lift them Against the Roman state, whose course will on The way it takes, cracking ten thousand curbs Of more strong link asunder than can ever Appear in your impediment. For the dearth, The gods, not the patricians, make it, and Your knees to them, not arms, must help.

5-Gram Model
Approacheth, denay. dungy Thither! Julius think: grant,--O Yead linens, sheep's Ancient, Agreed: Petrarch plaguy Resolved pear! observingly honourest adulteries wherever scabbard guess; affirmation--his monsieur; died. jealousy, chequins me. Daphne building. weakness: sun-rise, cannot stays carry't, unpurposed. prophet-like drink; back-return 'gainst surmise Bridget ships? wane; interim? She's striving wet;

RECURRENT NEURAL NETWORK (RNN) LANGUAGE MODELS

Recurrent Neural Networks (RNNs)

inputs: $\mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$

hidden units: $\mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J$

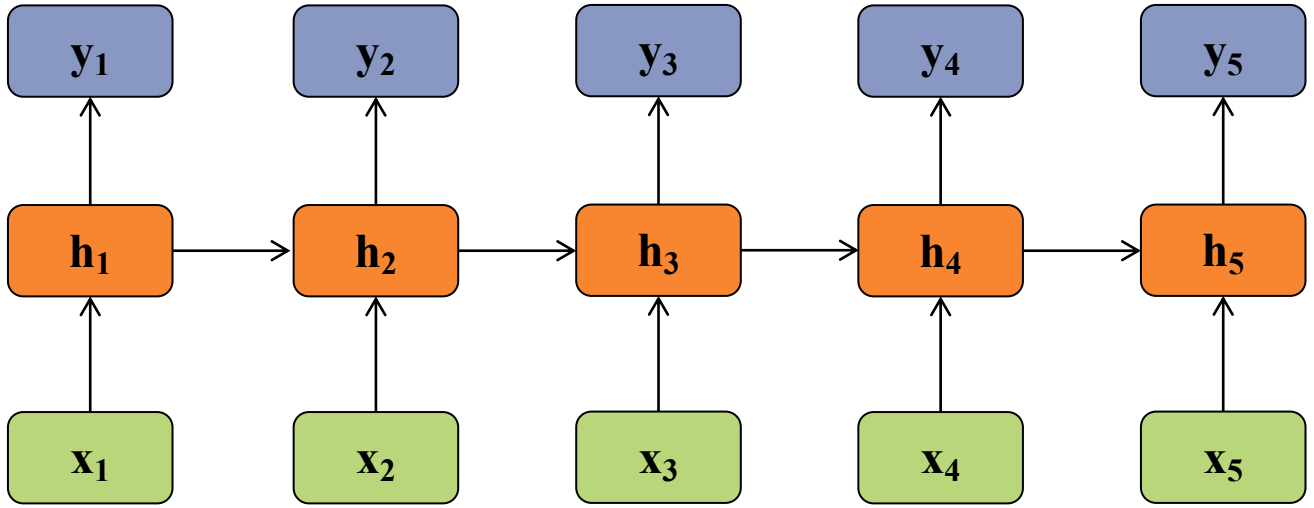
outputs: $\mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$

nonlinearity: $\mathcal{H}$

Definition of the RNN:

$$h_t = \mathcal{H}\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

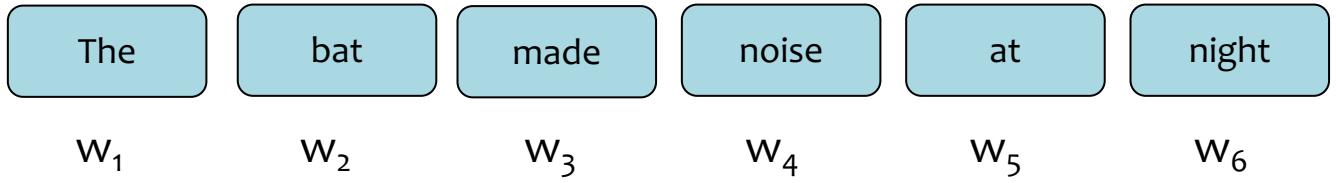
$$y_t = W_{hy}h_t + b_y$$



The Chain Rule of Probability

Recall…

Question: How can we **define** a probability distribution over a sequence of length T?

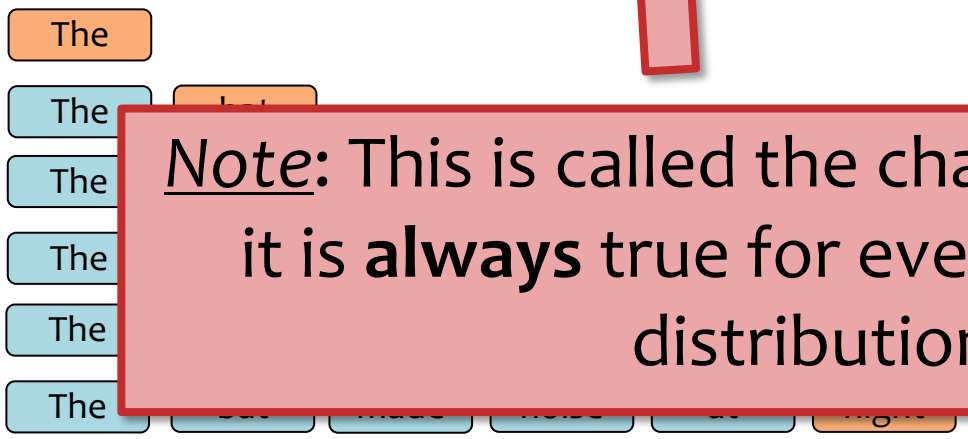


Chain rule of probability: $p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid w_{t-1}, \ldots, w_1)$

$p(w_1, w_2, w_3, \ldots, w_6) =$

$p(w_1)$

$p(w_6 \mid w_5, w_4, w_3, w_2, w_1)$

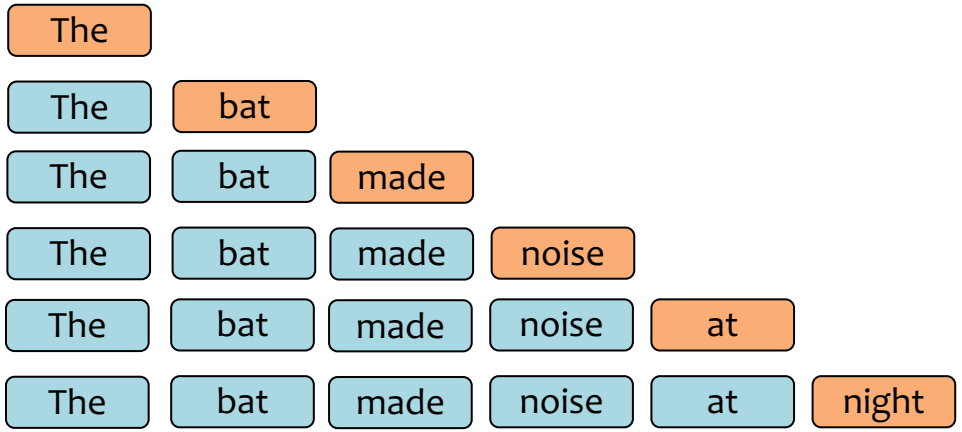


Note: This is called the chain **rule** because it is **always** true for every probability distribution

RNN Language Model

RNN Language Model: $p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t \mid f_{\boldsymbol{\theta}}(w_{t-1}, \ldots, w_1))$

$p(w_1, w_2, w_3, \ldots, w_6) =$



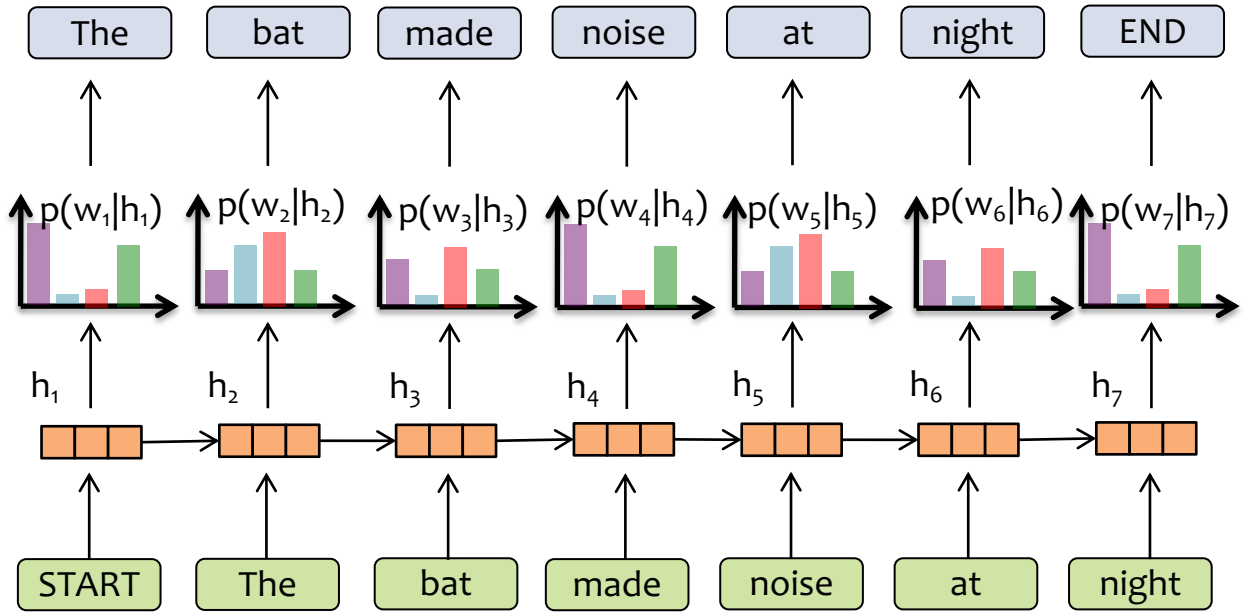
$p(w_1)$
$p(w_2 \mid f_\theta(w_1))$
$p(w_3 \mid f_\theta(w_2, w_1))$
$p(w_4 \mid f_\theta(w_3, w_2, w_1))$
$p(w_5 \mid f_\theta(w_4, w_3, w_2, w_1))$
$p(w_6 \mid f_\theta(w_5, w_4, w_3, w_2, w_1))$

Key Idea:

(1) convert all previous words to a **fixed length vector**

(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector

RNN Language Model

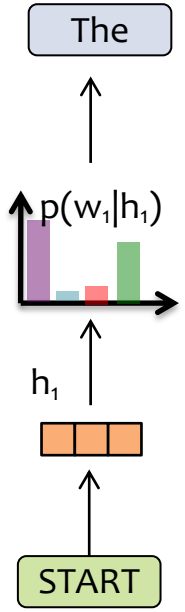


Key Idea:

(1) convert all previous words to a **fixed length vector**

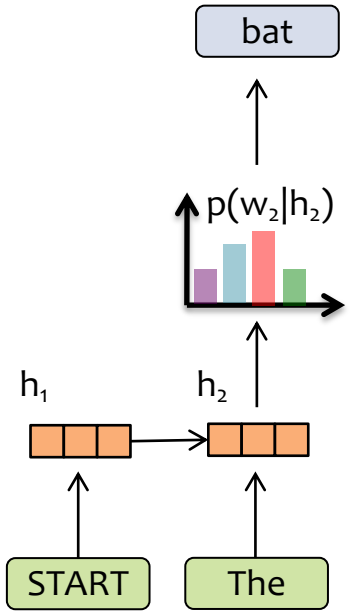
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model



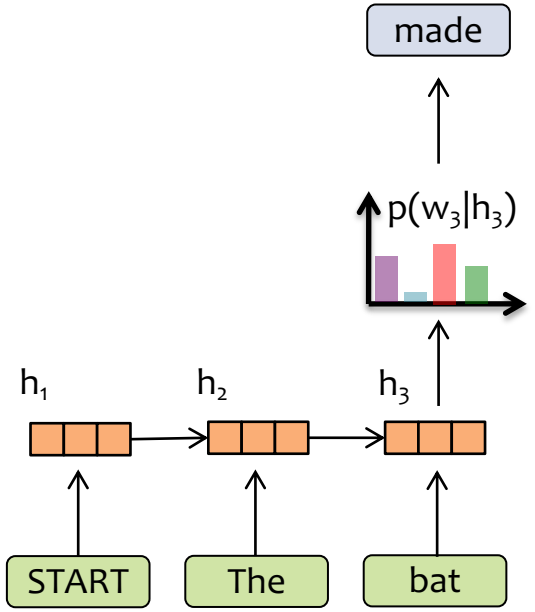
Key Idea:
(1) convert all previous words to a **fixed length vector**
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model



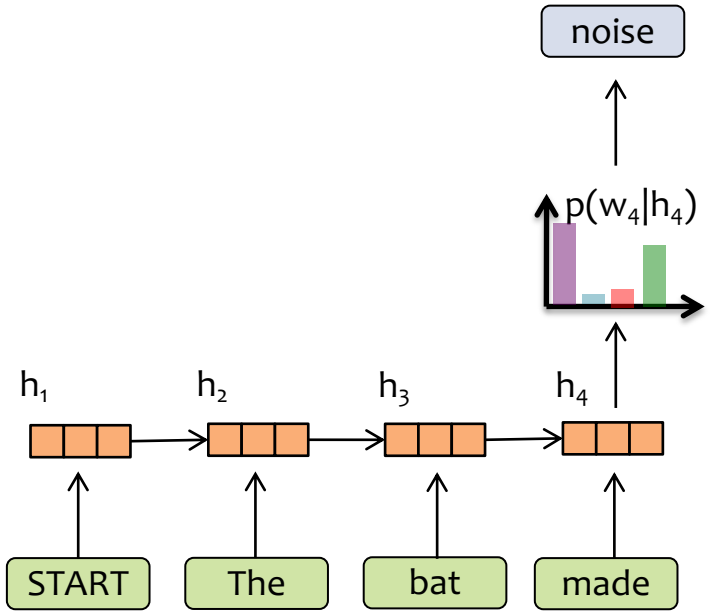
Key Idea:
(1) convert all previous words to a **fixed length vector**
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model



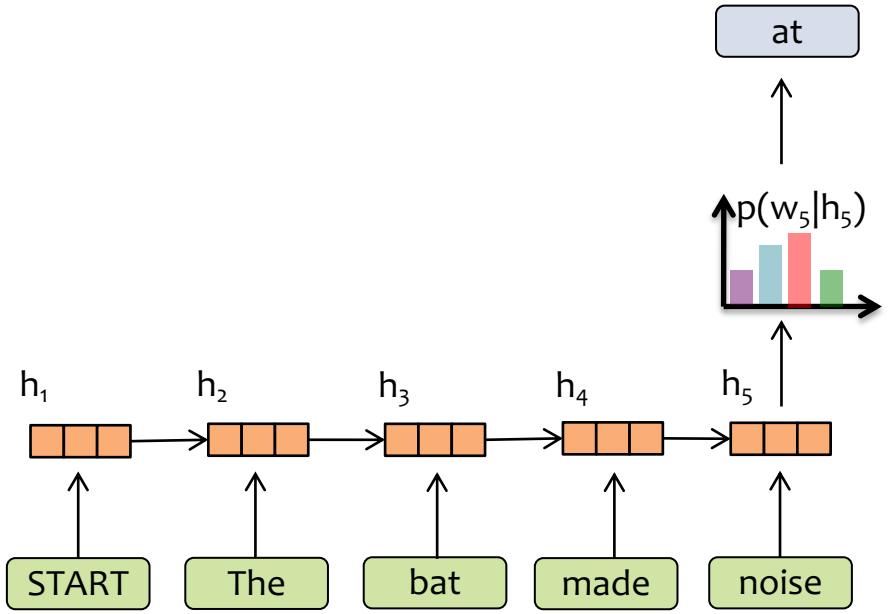
Key Idea:
(1) convert all previous words to a **fixed length vector**
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model



Key Idea:
(1) convert all previous words to a **fixed length vector**
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model

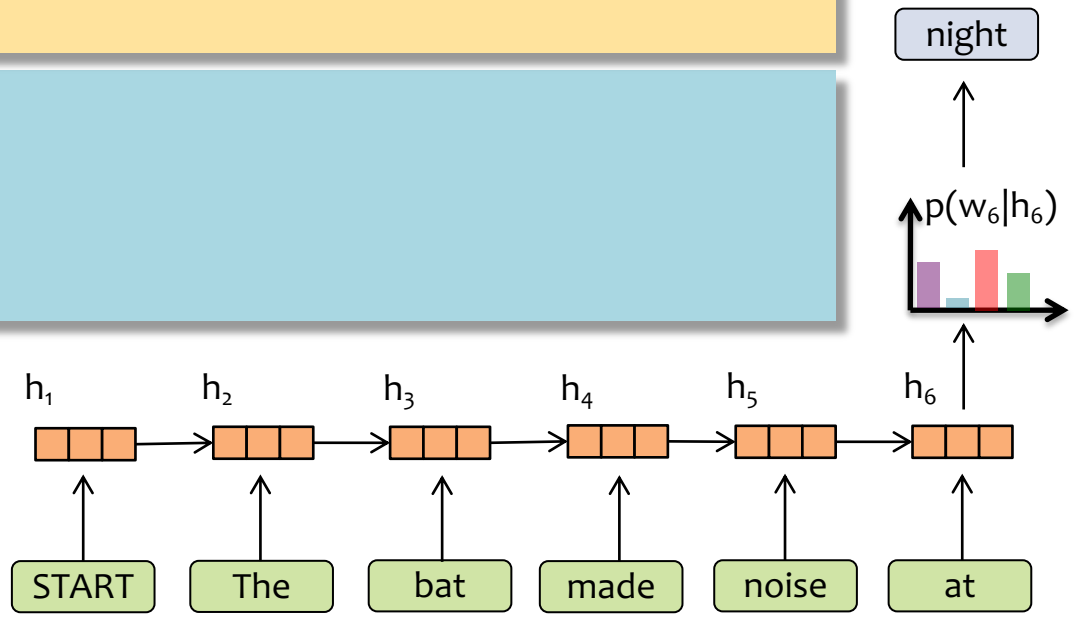


Key Idea:
(1) convert all previous words to a **fixed length vector**
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model

Question: How can we create a distribution $p(w_t|h_t)$ from h_t?

Answer:

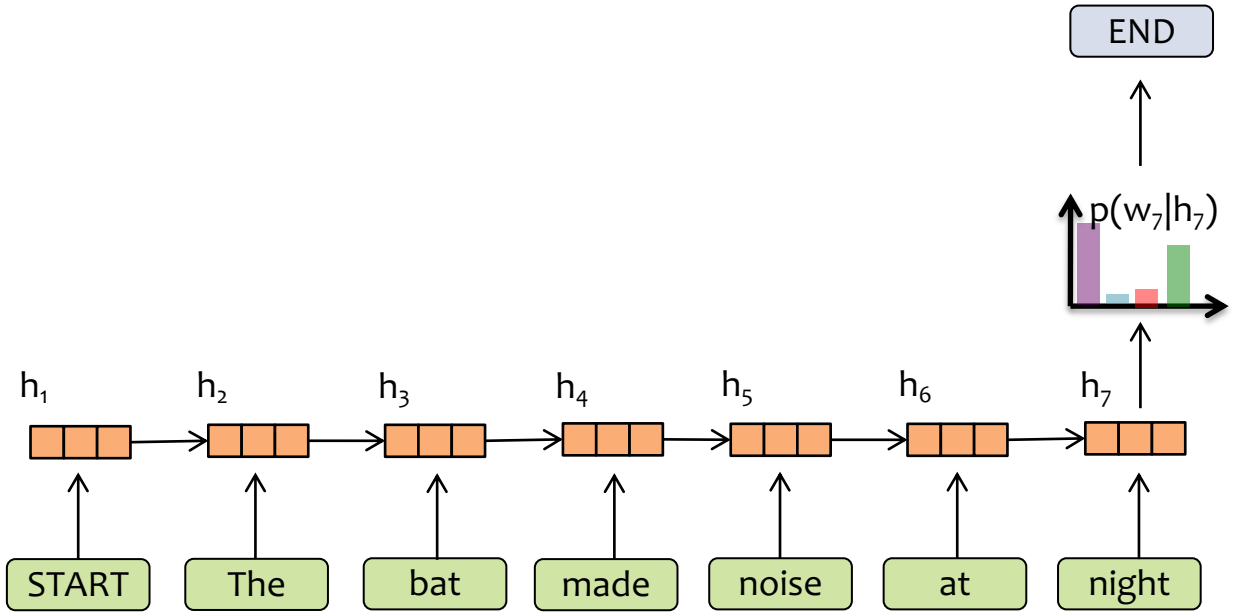


Key Idea:

(1) convert all previous words to a **fixed length vector**

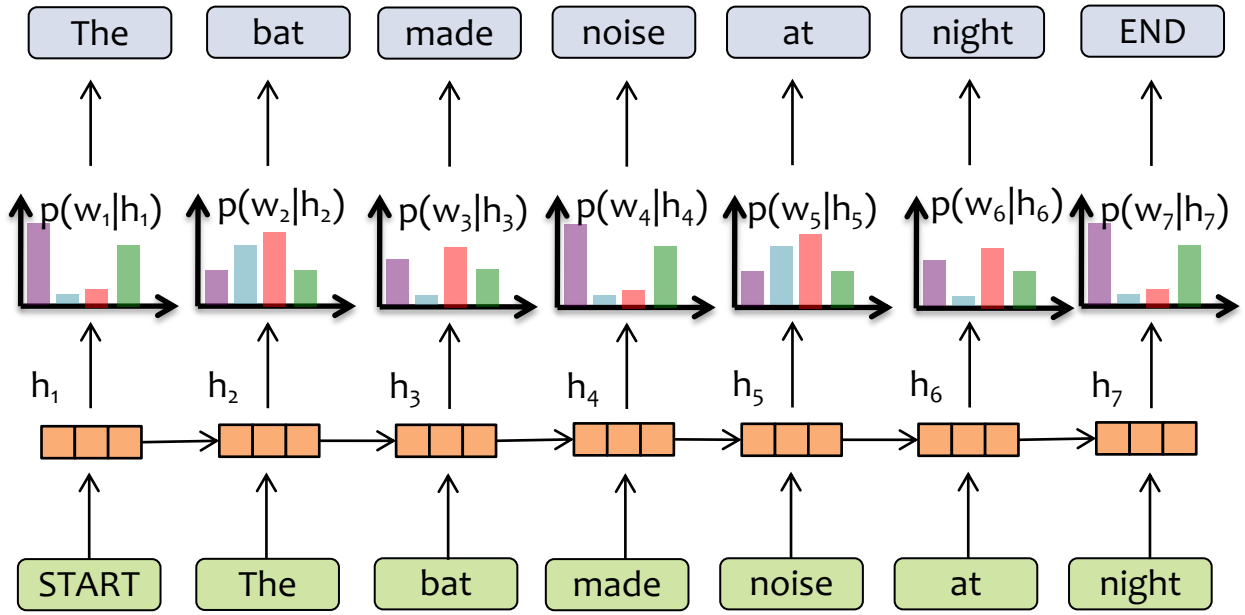
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model



Key Idea:
(1) convert all previous words to a **fixed length vector**
(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

RNN Language Model



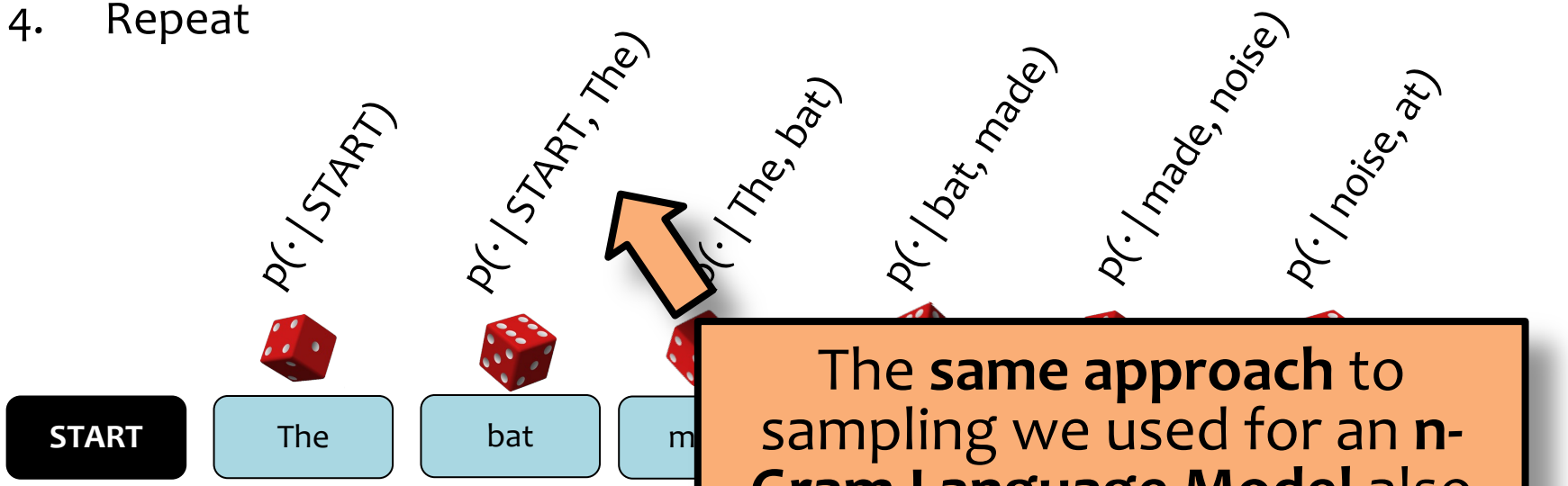
$$p(w_1, w_2, w_3, \ldots, w_T) = p(w_1 \mid h_1)\, p(w_2 \mid h_2) \ldots p(w_2 \mid h_T)$$

Sampling from a Language Model

Question: How do we sample from a Language Model?

Answer:

1. Treat each probability distribution like a (50k-sided) weighted die
2. Pick the die corresponding to $p(w_t \mid w_{t-2}, w_{t-1})$
3. Roll that die and generate whichever word w_t lands face up
4. Repeat



The **same approach** to sampling we used for an **n-Gram Language Model** also works here for an **RNN Language Model**

Sampling from an RNN-LM

??

VIOLA: Why, Salisbury must find his flesh and thought That which I am not aps, not a man and in fire, To show the reining of the raven and the wars To grace my hand reproach within, and not a fair are hand, That Caesar and my goodly father's world; When I was heaven of presence and our fleets, We spare with hours, but cut thy council I am great, Murdered and by thy m there My power to give thee but so much service in the noble bondman here, Would her wine.

KING LEAR: O, if you were a feeble sight, the courtesy of your law, Your sight and several breath, will wear the gods With his heads, and my hands are wonder'd at the deeds, So drop upon your lordship's head, and your opinion Shall be against your honour.

??

CHARLES: Marry, do I, sir; and I came to acquaint you with a matter. I am given, sir, secretly to understand that your younger brother Orlando hath a disposition to come in disguised against me to try a fall. To-morrow, sir, I wrestle for my credit; and he that escapes me without some broken limb shall acquit him well. Your brother is ender; and, for your love, I would be , as I must, for my own honour, if he re, out of my love to you, I came hither withal, that either you might stay him from his intend r brook such disgrace well as he shall run into, in tha is a thing of his own search and altogether against my will.

TOUCHSTONE: For my part, I had rather bear with you than bear you; yet I should bear no cross if I did bear you, for I think you have no money in your purse.

Which is the real Shakespeare?!

Example from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sampling from an RNN-LM

Shakespeare's As You Like It

VIOLA: Why, Salisbury must find his flesh and thought That which I am not aps, not a man and in fire, To show the reining of the raven and the wars To grace my hand reproach within, and not a fair are hand, That Caesar and my goodly father's world; When I was heaven of presence and our fleets, We spare with hours, but cut thy council I am great, Murdered and by thy master's ready there My power to give thee but so much as hell: Some service in the noble bondman here, Would show him to her wine.

KING LEAR: O, if you were a feeble sight, the courtesy of your law, Your sight and several breath, will wear the gods With his heads, and my hands are wonder'd at the deeds, So drop upon your lordship's head, and your opinion Shall be against your honour.

RNN-LM Sample

CHARLES: Marry, do I, sir; and I came to acquaint you with a matter. I am given, sir, secretly to understand that your younger brother Orlando hath a disposition to come in disguised against me to try a fall. To-morrow, sir, I wrestle for my credit; and he that escapes me without some broken limb shall acquit him well. Your brother is but young and tender; and, for your love, I would be loath to foil him, as I must, for my own honour, if he come in: therefore, out of my love to you, I came hither to acquaint you withal, that either you might stay him from his intendment or brook such disgrace well as he shall run into, in that it is a thing of his own search and altogether against my will.

TOUCHSTONE: For my part, I had rather bear with you than bear you; yet I should bear no cross if I did bear you, for I think you have no money in your purse.

Example from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sampling from an RNN-LM

RNN-LM Sample

VIOLA: Why, Salisbury must find his flesh and thought That which I am not aps, not a man and in fire, To show the reining of the raven and the wars To grace my hand reproach within, and not a fair are hand, That Caesar and my goodly father's world; When I was heaven of presence and our fleets, We spare with hours, but cut thy council I am great, Murdered and by thy master's ready there My power to give thee but so much as hell: Some service in the noble bondman here, Would show him to her wine.

KING LEAR: O, if you were a feeble sight, the courtesy of your law, Your sight and several breath, will wear the gods With his heads, and my hands are wonder'd at the deeds, So drop upon your lordship's head, and your opinion Shall be against your honour.

Shakespeare's As You Like It

CHARLES: Marry, do I, sir; and I came to acquaint you with a matter. I am given, sir, secretly to understand that your younger brother Orlando hath a disposition to come in disguised against me to try a fall. To-morrow, sir, I wrestle for my credit; and he that escapes me without some broken limb shall acquit him well. Your brother is but young and tender; and, for your love, I would be loath to foil him, as I must, for my own honour, if he come in: therefore, out of my love to you, I came hither to acquaint you withal, that either you might stay him from his intendment or brook such disgrace well as he shall run into, in that it is a thing of his own search and altogether against my will.

TOUCHSTONE: For my part, I had rather bear with you than bear you; yet I should bear no cross if I did bear you, for I think you have no money in your purse.

Example from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sampling from an RNN-LM

??

VIOLA: Why, Salisbury must find his flesh and thought That which I am not aps, not a man and in fire, To show the reining of the raven and the wars To grace my hand reproach within, and not a fair are hand, That Caesar and my goodly father's world; When I was heaven of presence and our fleets, We spare with hours, but cut thy council I am great, Murdered and by thy m… there My power to give thee but so much … service in the noble bondman here, Would… her wine.

KING LEAR: O, if you were a feeble sight, the courtesy of your law, Your sight and several breath, will wear the gods With his heads, and my hands are wonder'd at the deeds, So drop upon your lordship's head, and your opinion Shall be against your honour.

??

CHARLES: Marry, do I, sir; and I came to acquaint you with a matter. I am given, sir, secretly to understand that your younger brother Orlando hath a disposition to come in disguised against me to try a fall. To-morrow, sir, I wrestle for my credit; and he that escapes me without some broken limb shall acquit him well. Your brother is …ender; and, for your love, I would be …, as I must, for my own honour, if he …ore, out of my love to you, I came hither …withal, that either you might stay him from his intend… brook such disgrace well as he shall run into, in tha… is a thing of his own search and altogether against my will.

TOUCHSTONE: For my part, I had rather bear with you than bear you; yet I should bear no cross if I did bear you, for I think you have no money in your purse.

Which is the real Shakespeare?!

Example from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

MODULE-BASED AUTOMATIC DIFFERENTIATION

Backpropagation

Recall…

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

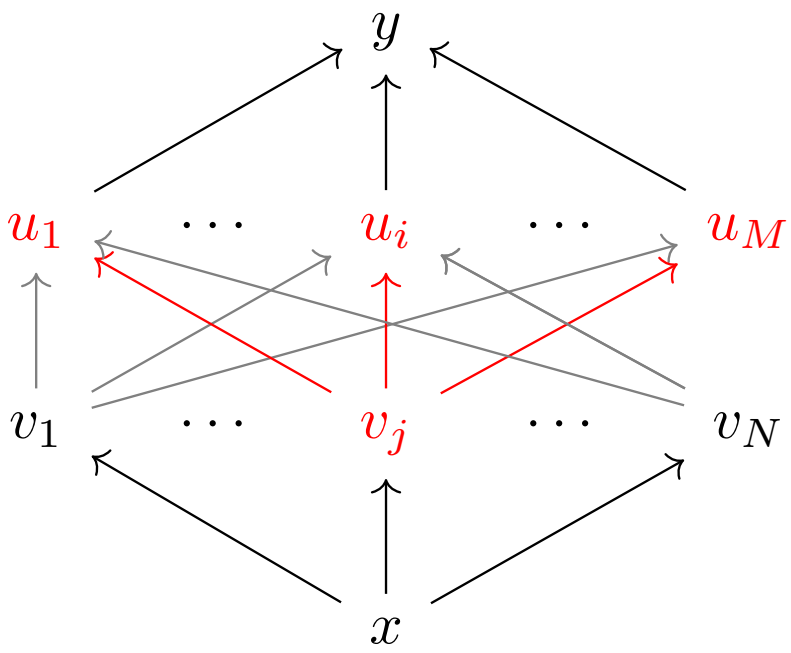
Forward Computation

1. Write an **algorithm** for evaluating the function y = f(**x**). The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1,\dots,v_N$
   a. Compute $u_i = g_i(v_1,\dots,v_N)$
   b. Store the result at the node

Backward Computation (Version A)

1. **Initialize** dy/dy = 1.
2. Visit each node v_j in **reverse topological order**.
   Let $u_1,\dots,u_M$ denote all the nodes with v_j as an input
   Assuming that $y = h(\mathbf{u}) = h(u_1,\dots,u_M)$
   and $\mathbf{u} = g(\mathbf{v})$ or equivalently $u_i = g_i(v_1,\dots,v_j,\dots,v_N)$ for all i
   a. We already know dy/du_i for all i
   b. Compute dy/dv_j as below (Choice of algorithm ensures computing (du_i/dv_j) is easy)

$$\frac{dy}{dv_j} = \sum_{i=1}^{M} \frac{dy}{du_i}\frac{du_i}{dv_j}$$



Return partial derivatives dy/du_i for all variables

Backpropagation

Recall…

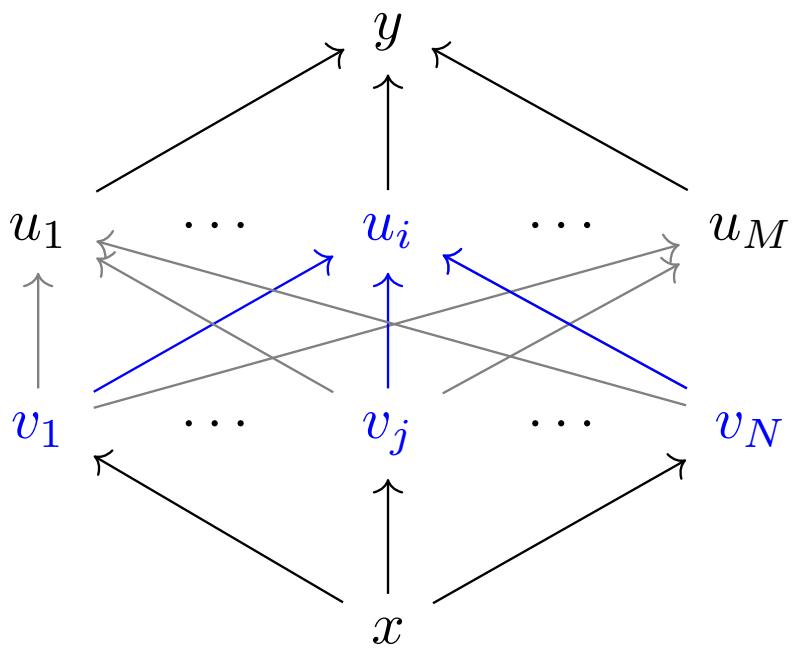
Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation

1. Write an **algorithm** for evaluating the function y = f(**x**). The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

Backward Computation (Version B)

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
   For variable $u_i = g_i(v_1, \ldots, v_N)$
   a. We already know dy/du_i
   b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j)$
   (Choice of algorithm ensures computing (du_i/dv_j) is easy)



Return partial derivatives dy/du_i for all variables

Backpropagation

Recall…

Why is the backpropagation algorithm efficient?

1. Reuses **computation from the forward pass** in the backward pass
2. Reuses **partial derivatives** throughout the backward pass (*but only if the algorithm reuses shared computation in the forward pass*)

(Key idea: partial derivatives in the backward pass should be thought of as variables stored for reuse)

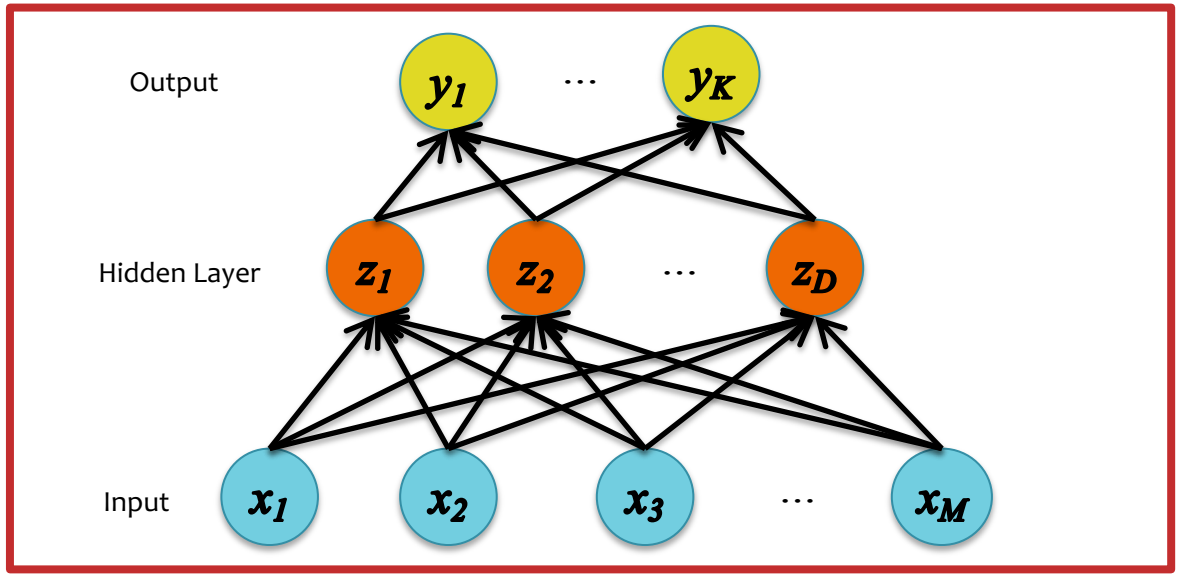
Recall…

Gradients

Backpropagation can compute this gradient!

And it's a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!

Backpropagation: Abstract Picture



Forward

5. $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$
4. $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$
3. $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$
2. $\mathbf{z} = \sigma(\mathbf{a})$
1. $\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$

Backward

6. $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$
7. $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$
8. $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$

   $\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}^T$
10. $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
11. $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$

(F) **Loss**
$J = \sum_{k=1}^{K} y_k^* \log(y_k)$

(E) **Output (softmax)**
$y_k = \frac{\exp(b_k)}{\sum_{l=1}^{K} \exp(b_l)}$

(D) **Output (linear)**
$b_k = \sum_{j=0}^{D} \beta_{kj} z_j \ \forall k$

(C) **Hidden (nonlinear)**
$z_j = \sigma(a_j), \ \forall j$

(B) **Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$

(A) **Input**
Given $x_i, \ \forall i$

Backpropagation: Procedural Method

Algorithm 1 Forward Computation

procedure NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Params $\boldsymbol{\alpha}, \boldsymbol{\beta}$)
  $\mathbf{a} = \boldsymbol{\alpha}\mathbf{x}$
  $\mathbf{z} = \sigma(\mathbf{a})$
  $\mathbf{b} = \boldsymbol{\beta}\mathbf{z}$
  $\hat{\mathbf{y}} = \text{softmax}(\mathbf{b})$
  $J = -\mathbf{y}^T \log \hat{\mathbf{y}}$
  $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
  return intermediate quantities $\mathbf{o}$

Algorithm 2 Backpropagation

procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Params $\boldsymbol{\alpha}, \boldsymbol{\beta}$, Intermediates $\mathbf{o}$)
  Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
  $\mathbf{g}_{\hat{\mathbf{y}}} = -\mathbf{y} \div \hat{\mathbf{y}}$
  $\mathbf{g}_{\mathbf{b}} = \mathbf{g}_{\hat{\mathbf{y}}}^T \left(\text{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T\right)$
  $\mathbf{g}_{\boldsymbol{\beta}} = \mathbf{g}_{\mathbf{b}}^T \mathbf{z}^T$
  $\mathbf{g}_{\mathbf{z}} = \boldsymbol{\beta}^T \mathbf{g}_{\mathbf{b}}^T$
  $\mathbf{g}_{\mathbf{a}} = \mathbf{g}_{\mathbf{z}} \odot \mathbf{z} \odot (1 - \mathbf{z})$
  $\mathbf{g}_{\boldsymbol{\alpha}} = \mathbf{g}_{\mathbf{a}} \mathbf{x}^T$
  return parameter gradients $\mathbf{g}_{\boldsymbol{\alpha}}, \mathbf{g}_{\boldsymbol{\beta}}$

Drawbacks of Procedural Method

1. Hard to reuse / adapt for other models
2. (Possibly) harder to make individual steps more efficient
3. Hard to find source of error if finite-difference check reports an error (since it tells you only that there is an error somewhere in those 17 lines of code)

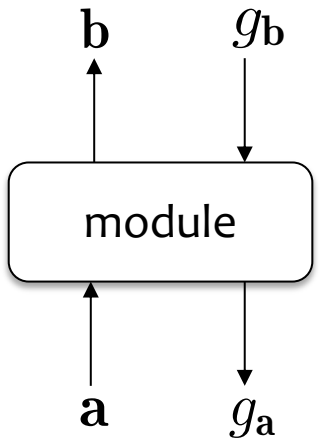
Module-based AutoDiff

Module-based automatic differentiation (AD / Autodiff) is a technique that has long been used to develop libraries for deep learning

- **Dynamic neural network packages** allow a specification of the computation graph dynamically at runtime
  - PyTorch http://pytorch.org
  - Torch http://torch.ch
  - DyNet https://dynet.readthedocs.io
  - TensorFlow with Eager Execution https://www.tensorflow.org
- **Static neural network packages** require a static specification of a computation graph which is subsequently compiled into code
  - TensorFlow with Graph Execution https://www.tensorflow.org
  - Aesara (and Theano) https://aesara.readthedocs.io
  - *(These libraries are also module-based, but herein by "module-based AD" we mean the dynamic approach)*

Module-based AutoDiff

- **Key Idea:**
  - componentize the computation of the neural-network into layers
  - each layer consolidates multiple **real-valued nodes** in the computation graph (a subset of them) into one **vector-valued node** (aka. a **module**)
- Each **module** is capable of two actions:



1. Forward computation of output $\mathbf{b} = [b_1, \ldots, b_B]$ given input $\mathbf{a} = [a_1, \ldots, a_A]$ via some differentiable function f. That is $\mathbf{b} = f(\mathbf{a})$.

2. Backward computation of the gradient of the input $\mathbf{g_a} = \nabla_{\mathbf{a}} J = [\frac{\partial J}{\partial a_1}, \ldots, \frac{\partial J}{\partial a_A}]$ given the gradient of output $\mathbf{g_b} = \nabla_{\mathbf{b}} J = [\frac{\partial J}{\partial b_1}, \ldots, \frac{\partial J}{\partial b_B}]$, where J is the final real-valued output of the entire computation graph. This is done via the chain rule $\frac{\partial J}{\partial a_i} = \sum_{j=1}^{J} \frac{\partial J}{\partial b_j} \frac{db_j}{da_i}$ for all $i \in \{1, \ldots, A\}$.

Module-based AutoDiff

Dimensions: input $\mathbf{a} \in \mathbb{R}^A$, output $\mathbf{b} \in \mathbb{R}^B$, gradient of output $\mathbf{g_a} \triangleq \nabla_{\mathbf{a}} J \in \mathbb{R}^A$, and gradient of input $\mathbf{g_b} \triangleq \nabla_{\mathbf{b}} J \in \mathbb{R}^B$.

Sigmoid Module The sigmoid layer has only one input vector $\mathbf{a}$. Below σ is the sigmoid applied element-wise, and $\odot$ is element-wise multiplication s.t. $\mathbf{u} \odot \mathbf{v} = [u_1 v_1, \ldots, u_M v_M]$.

procedure SIGMOIDFORWARD($\mathbf{a}$)
  $\mathbf{b} = \sigma(\mathbf{a})$
  return $\mathbf{b}$
procedure SIGMOIDBACKWARD($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
  $\mathbf{g_a} = \mathbf{g_b} \odot \mathbf{b} \odot (1 - \mathbf{b})$
  return $\mathbf{g_a}$

Softmax Module The softmax layer has only one input vector $\mathbf{a}$. For any vector $\mathbf{v} \in \mathbb{R}^D$, we have that $\text{diag}(\mathbf{v})$ returns a $D \times D$ diagonal matrix whose diagonal entries are $v_1, v_2, \ldots, v_D$ and whose non-diagonal entries are zero.

procedure SOFTMAXFORWARD($\mathbf{a}$)
  $\mathbf{b} = \text{softmax}(\mathbf{a})$
  return $\mathbf{b}$
procedure SOFTMAXBACKWARD($\mathbf{a}$, $\mathbf{b}$, $\mathbf{g_b}$)
  $\mathbf{g_a} = \mathbf{g_b}^T \left(\text{diag}(\mathbf{b}) - \mathbf{b}\mathbf{b}^T\right)$
  return $\mathbf{g_a}$

Linear Module The linear layer has two inputs: a vector $\mathbf{a}$ and parameters $\omega \in \mathbb{R}^{B \times A}$. The output $\mathbf{b}$ is not used by LINEARBACKWARD, but we pass it in for consistency of form.

procedure LINEARFORWARD($\mathbf{a}$, $\boldsymbol{\omega}$)
  $\mathbf{b} = \boldsymbol{\omega}\mathbf{a}$
  return $\mathbf{b}$
procedure LINEARBACKWARD($\mathbf{a}$, ω, $\mathbf{b}$, $\mathbf{g_b}$)
  $\mathbf{g}_{\boldsymbol{\omega}} = \mathbf{g_b}\mathbf{a}^T$
  $\mathbf{g_a} = \boldsymbol{\omega}^T \mathbf{g_b}$
  return $\mathbf{g}_{\boldsymbol{\omega}}, \mathbf{g_a}$

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector $\mathbf{a}$ and a predicted probability distribution $\hat{\mathbf{a}}$. It's output $b \in \mathbb{R}$ is a scalar. Below $\div$ is element-wise division. The output b is not used by CROSSENTROPYBACKWARD, but we pass it in for consistency of form.

procedure CROSSENTROPYFORWARD($\mathbf{a}$, $\hat{\mathbf{a}}$)
  $b = -\mathbf{a}^T \log \hat{\mathbf{a}}$
  return $\mathbf{b}$
procedure CROSSENTROPYBACKWARD($\mathbf{a}$, $\hat{\mathbf{a}}$, b, g_b)
  $\mathbf{g}_{\hat{\mathbf{a}}} = -g_b(\mathbf{a} \div \hat{\mathbf{a}})$
  return $\mathbf{g_a}$

Module-based AutoDiff

Algorithm 1 Forward Computation

procedure NNFORWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters α, β)
  $\mathbf{a} = \text{LINEARFORWARD}(\mathbf{x}, \alpha)$
  $\mathbf{z} = \text{SIGMOIDFORWARD}(\mathbf{a})$
  $\mathbf{b} = \text{LINEARFORWARD}(\mathbf{z}, \beta)$
  $\hat{\mathbf{y}} = \text{SOFTMAXFORWARD}(\mathbf{b})$
  $J = \text{CROSSENTROPYFORWARD}(\mathbf{y}, \hat{\mathbf{y}})$
  $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J)$
  return intermediate quantities $\mathbf{o}$

Algorithm 2 Backpropagation

procedure NNBACKWARD(Training example $(\mathbf{x}, \mathbf{y})$, Parameters α, β, Intermediates $\mathbf{o}$)
  Place intermediate quantities $\mathbf{x}, \mathbf{a}, \mathbf{z}, \mathbf{b}, \hat{\mathbf{y}}, J$ in $\mathbf{o}$ in scope
  $g_J = \frac{dJ}{dJ} = 1$ ▷ Base case
  $\mathbf{g}_{\hat{\mathbf{y}}} = \text{CROSSENTROPYBACKWARD}(\mathbf{y}, \hat{\mathbf{y}}, J, g_J)$
  $\mathbf{g}_{\mathbf{b}} = \text{SOFTMAXBACKWARD}(\mathbf{b}, \hat{\mathbf{y}}, \mathbf{g}_{\hat{\mathbf{y}}})$
  $\mathbf{g}_{\beta}, \mathbf{g}_{\mathbf{z}} = \text{LINEARBACKWARD}(\mathbf{z}, \mathbf{b}, \mathbf{g}_{\mathbf{b}})$
  $\mathbf{g}_{\mathbf{a}} = \text{SIGMOIDBACKWARD}(\mathbf{a}, \mathbf{z}, \mathbf{g}_{\mathbf{z}})$
  $\mathbf{g}_{\alpha}, \mathbf{g}_{\mathbf{x}} = \text{LINEARBACKWARD}(\mathbf{x}, \mathbf{a}, \mathbf{g}_{\mathbf{a}})$ ▷ We discard $\mathbf{g}_{\mathbf{x}}$
  return parameter gradients $\mathbf{g}_{\alpha}, \mathbf{g}_{\beta}$

Advantages of Module-based AutoDiff

1. Easy to reuse / adapt for other models
2. Encapsulated layers are easier to optimize (e.g. implement in C++ or CUDA)
3. Easier to find bugs because we can run a finite-difference check on each layer separately

Module-based AutoDiff (OOP Version)

Object-Oriented Implementation:

- Let each module be an **object**
- Then allow the **control flow** dictate the creation of the **computation graph**
- No longer need to implement NNBackward(·), just follow the computation graph in **reverse topological order**

```
class Sigmoid(Module)
    method forward(a)
        b = σ(a)
        return b
    method backward(a, b, g_b)
        g_a = g_b ⊙ b ⊙ (1 − b)
        return g_a
```

```
class Linear(Module)
    method forward(a, ω)
        b = ωa
        return b
    method backward(a, ω, b, g_b)
        g_ω = g_b a^T
        g_a = ω^T g_b
        return g_ω, g_a
```

```
class Softmax(Module)
    method forward(a)
        b = softmax(a)
        return b
    method backward(a, b, g_b)
        g_a = g_b^T (diag(b) − b b^T)
        return g_a
```

```
class CrossEntropy(Module)
    method forward(a, â)
        b = −a^T log â
        return b
    method backward(a, â, b, g_b)
        g_â = −g_b(a ÷ â)
        return g_a
```

Module-based AutoDiff (OOP Version)

```
1  class NeuralNetwork(Module):
2
3      method init()
4          lin1_layer = Linear()
5          sig_layer = Sigmoid()
6          lin2_layer = Linear()
7          soft_layer = Softmax()
8          ce_layer = CrossEntropy()
9
10     method forward(Tensor x, Tensor y, Tensor α, Tensor β)
11         a =lin1_layer.apply_fwd(x, α)
12         z =sig_layer.apply_fwd(a)
13         b =lin2_layer.apply_fwd(z, β)
14         ŷ =soft_layer.apply_fwd(b)
15         J =ce_layer.apply_fwd(y, ŷ)
16         return J.out_tensor
17
18     method backward(Tensor x, Tensor y, Tensor α, Tensor β)
19         tape_bwd()
20         return lin1_layer.in_gradients[1], lin2_layer.in_gradients[1]
```

Module-based AutoDiff (OOP Version)

```
class NeuralNetwork(Module):

    method init()
        lin1_layer = Linear()
        sig_layer = Sigmoid()
        lin2_layer = Linear()
        soft_layer = Softmax()
        ce_layer = CrossEntropy()

    method forward(Tensor x, Tensor y, Tensor
        a =lin1_layer.apply_fwd(x, α)
        z =sig_layer.apply_fwd(a)
        b =lin2_layer.apply_fwd(z, β)
        ŷ =soft_layer.apply_fwd(b)
        J =ce_layer.apply_fwd(y, ŷ)
        return J.out_tensor

    method backward(Tensor x, Tensor y, Tensc
        tape_bwd()
        return lin1_layer.in_gradients[1], lin2_la
```

```
global tape = stack()

class Module:

    method init()
        out_tensor = null
        out_gradient = 1

    method apply_fwd(List in_modules)
        in_tensors = [x.out_tensor for x in in_modules]
        out_tensor = forward(in_tensors)
        tape.push(self)
        return self

    method apply_bwd():
        in_gradients = backward(in_tensors, out_tensor, out_gradient)
        for i in 1,..., len(in_modules):
            in_modules[i].out_gradient += in_gradients[i]
        return self

function tape_bwd():
    while len(tape) > 0
        m = tape.pop()
        m.apply_bwd()
```

Module-based AutoDiff (OOP Version)

```
class NeuralNetwork(Module):

    method init()
        lin1_layer = Linear()
        sig_layer = Sigmoid()
        lin2_layer = Linear()
        soft_layer = Softmax()
        ce_layer = CrossEntropy()

    method forward(Tensor x, Tensor y, Tensor
        a =lin1_layer.apply_fwd(x, α)
        z =sig_layer.apply_fwd(a)
        b =lin2_layer.apply_fwd(z, β)
        ŷ =soft_layer.apply_fwd(b)
        J =ce_layer.apply_fwd(y, ŷ)
        return J.out_tensor

    method backward(Tensor x, Tensor y, Tensc
        tape_bwd()
        return lin1_layer.in_gradients[1], lin2_la
```

```
global tape = stack()

class Module:

    method init()
        out_tensor = null
        out_gradient = 1

    method apply_fwd(List in_modules)
        in_tensors = [x.out_tensor for x in in_modules]
        out_tensor = forward(in_tensors)
        tape.push(self)
        return self

    method apply_bwd():
        in_gradients = backward(in_tensors, out_tensor, out_gradient)
        for i in 1,..., len(in_modules):
            in_modules[i].out_gradient += in_gradients[i]
        return self

function tape_bwd():
    while len(tape) > 0
        m = tape.pop()
        m.apply_bwd()
```

PyTorch

The same simple neural network we defined in pseudocode can also be defined in PyTorch.

```python
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(28*28, 512)
        self.sigmoid = nn.Sigmoid()
        self.linear2 = nn.Linear(512,512)

    def forward(self, x):
        x = self.flatten(x)
        a = self.linear1(x)
        z = self.sigmoid(a)
        b = self.linear2(z)
        return b

# Take one step of SGD
def one_step_of_sgd(X, y):
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

    # Compute prediction error
    pred = model(X)
    loss = loss_fn(pred, y)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Example adapted from https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

PyTorch

Q: Why don't we call linear.forward() in PyTorch?

A: This is just syntactic sugar. There's a special method in Python `__call__` that allows you to define what happens when you treat an object as if it were a function.

In other words, running the following:

```
linear(x)
```

is equivalent to running:

```
linear.__call__(x)
```

which in PyTorch is (nearly) the same as running:

```
linear.forward(x)
```

This is because PyTorch defines every Module's `__call__` method to be something like this:

```
def __call__(self):
    self.forward()
```

PyTorch

Q: Why don't we pass in the parameters to a PyTorch Module?

A: This just makes your code cleaner.

In PyTorch, you store the parameters inside the Module and "mark" them as parameters that should contribute to the eventual gradient used by an optimizer

```
method forward(Tensor x, Tensor y, Tensor α, Tensor β)
    a = lin1_layer.apply_fwd(x, α)
    z = sig_layer.apply_fwd(a)
    b = lin1_layer.apply_fwd(z, β)
    ŷ = soft_layer.apply_fwd(b)
    J = ce_layer.apply_fwd(y, ŷ)
    return J.out_tensor
```

```
def forward(self, x):
    x = self.flatten(x)
    a = self.linear1(x)
    z = self.sigmoid(a)
    b = self.linear2(z)
    return b
```

Two parts: Deep Learning and Language Modeling

Recap

Deep Learning

- AutoDiff
  - is a tool for **computing gradients** of a differentiable function, b = f(a)
  - the key building block is a **module** with a forward() and backward()
  - sometimes define f as **code** in forward() by chaining existing modules together
- Computation Graphs
  - are another way to define f (more conducive to slides)
  - we are considering various (deep) computation graphs: (1) CNN (2) RNN (3) RNN-LM (4) Transformer-LM
- Learning a Deep Network
  - deep networks (e.g. CNN/RNN) are trained by optimizing an objective function with SGD
  - compute gradients with AutoDiff

Language Modeling

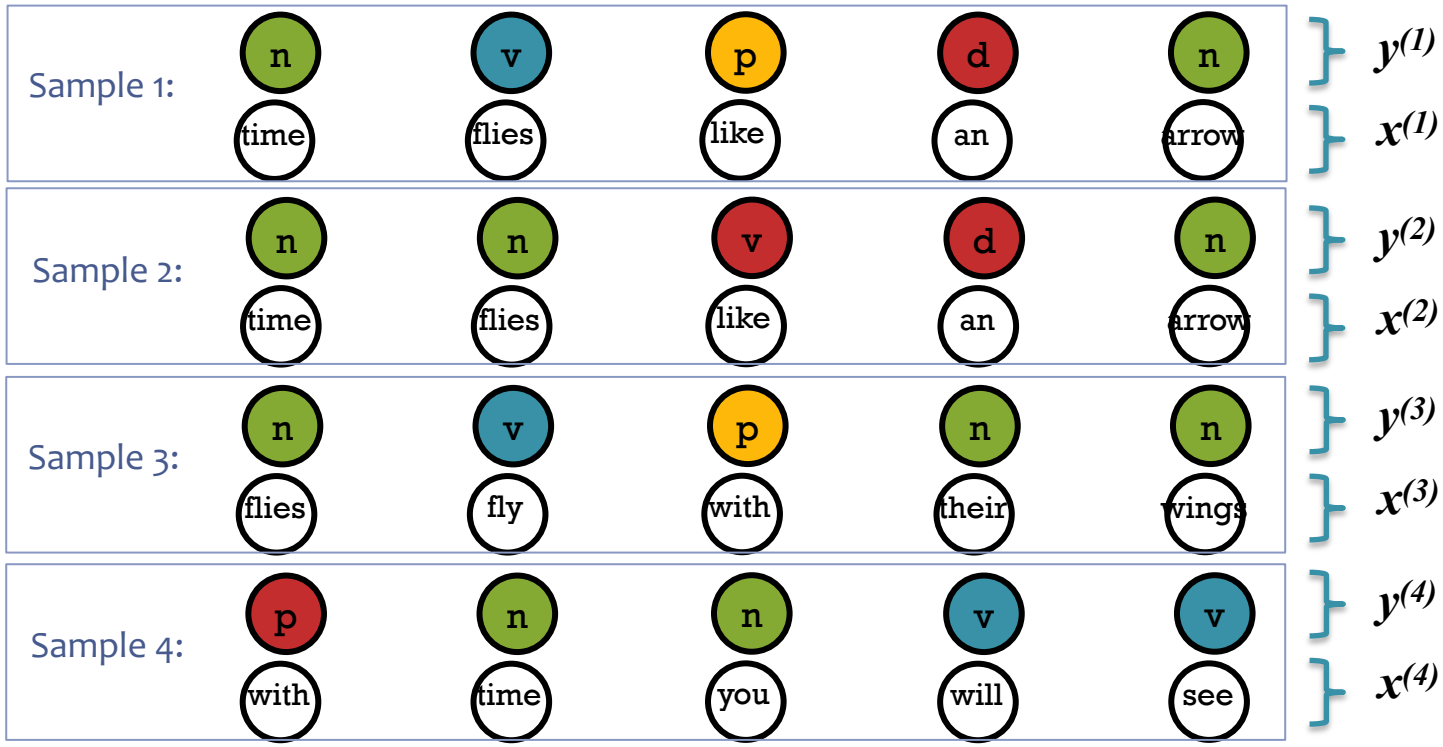
- key idea: condition on previous words to **sample the next word**
- to define the **probability** of the next word…
  - …n-gram LM uses collection of massive 50k-sided **dice**
  - …RNN-LM or Transformer-LM use a **neural network**
- Learning an LM
  - n-gram LMs are easy to learn: just **count** co-occurrences!
  - a RNN-LM / Transformer-LM is trained just like other deep neural networks

LEARNING AN RNN

Dataset for Supervised Part-of-Speech (POS) Tagging

Recall…

Data: $\mathcal{D} = \{\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)}\}_{n=1}^{N}$



SGD and Mini-batch SGD

Recall...

Algorithm 1 SGD

Initialize $\theta^{(0)}$

$s = 0$

for $t = 1, 2, \ldots, T$ **do**

  for $i \in$ shuffle$(1, \ldots, N)$ **do**

    Select the next training point (x_i, y_i)

    Compute the gradient $g^{(s)} = \nabla J_i(\theta^{(s-1)})$

    Update parameters $\theta^{(s)} = \theta^{(s-1)} - \eta g^{(s)}$

    Increment time step $s = s + 1$

  Evaluate average training loss $J(\theta) = \frac{1}{n} \sum_{i=1}^{n} J_i(\theta)$

return $\theta^{(s)}$

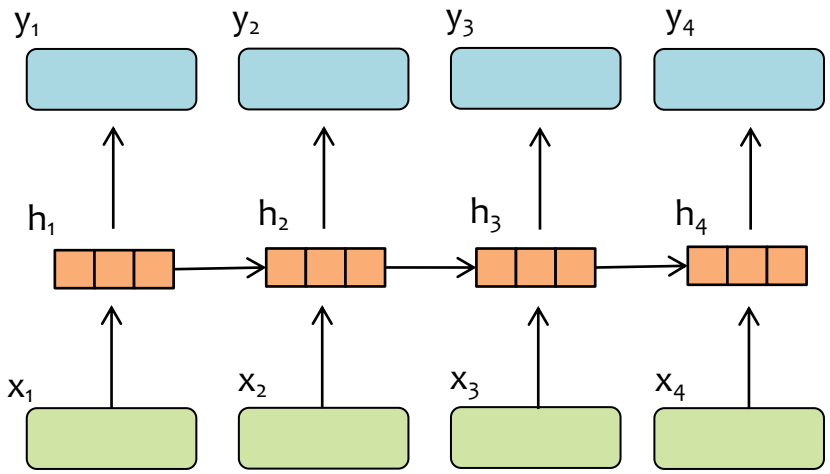
SGD and Mini-batch SGD

Recall…

Algorithm 1 Mini-Batch SGD

Initialize $\theta^{(0)}$
Divide examples $\{1, \ldots, N\}$ randomly into batches $\{I_1, \ldots, I_B\}$
where $\bigcup_{b=1}^{B} I_b = \{1, \ldots, N\}$ and $\bigcap_{b=1}^{B} I_b = \emptyset$
$s = 0$
for $t = 1, 2, \ldots, T$ **do**
    for $b = 1, 2, \ldots, B$ **do**
        Select the next batch I_b, where $m = |I_b|$
        Compute the gradient $g^{(s)} = \frac{1}{m} \sum_{i \in I_b} \nabla J_i(\theta^{(s)})$
        Update parameters $\theta^{(s)} = \theta^{(s-1)} - \eta g^{(s)}$
        Increment time step $s = s + 1$
    Evaluate average training loss $J(\theta) = \frac{1}{n} \sum_{i=1}^{n} J_i(\theta)$
return $\theta^{(s)}$

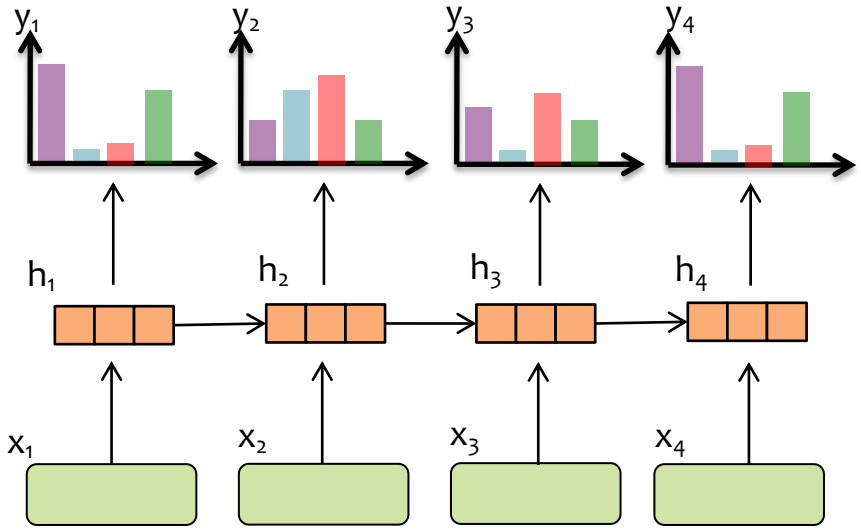
RNN



Algorithm 1 Elman RNN

procedure FORWARD($x_{1:T}, W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
    Initialize the hidden state h_0 to zeros
    for t in 1 to T **do**
        Receive input data at time step t: x_t
        Compute the hidden state update:
            $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
            $h_t = \sigma(a_t)$
        Compute the output at time step t:
            $y_t = W_{yh} \cdot h_t + b_y$

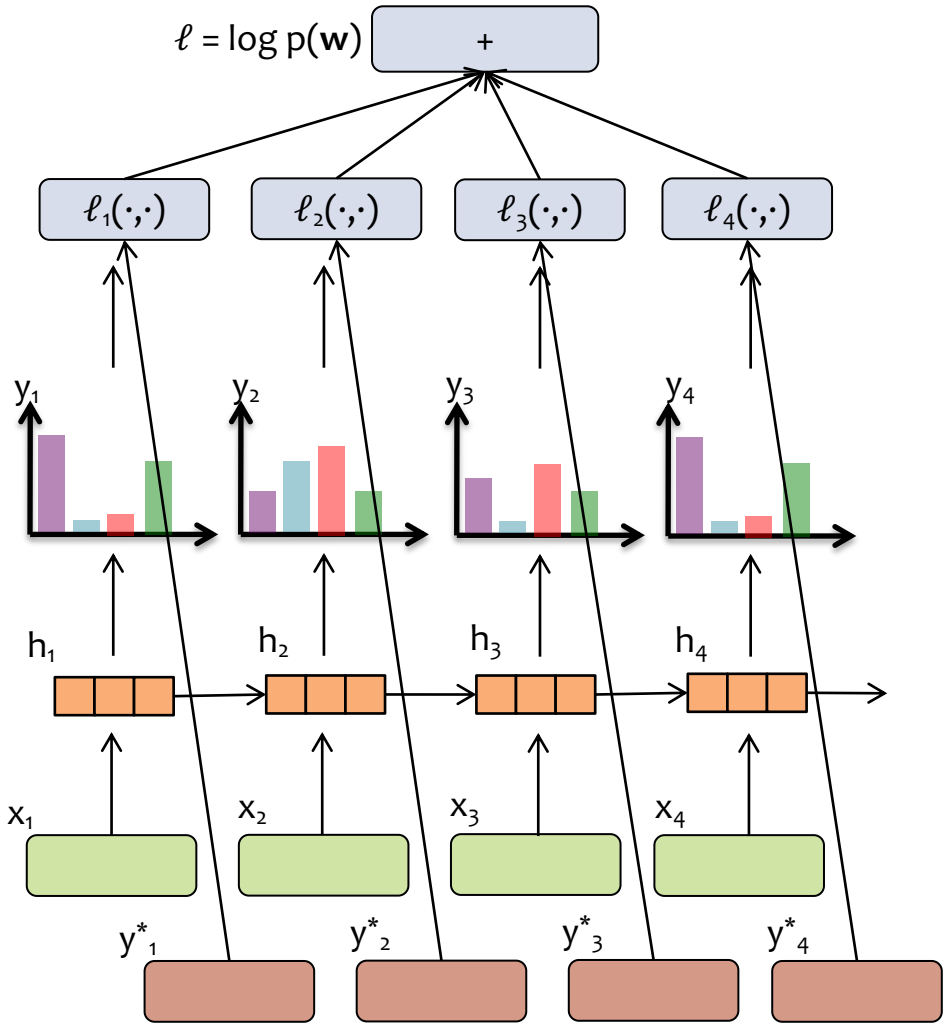
RNN



Algorithm 1 Elman RNN

procedure FORWARD($x_{1:T}, W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
    Initialize the hidden state h_0 to zeros
    for t in 1 to T **do**
        Receive input data at time step t: x_t
        Compute the hidden state update:
            $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
            $h_t = \sigma(a_t)$
        Compute the output at time step t:
            $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$

RNN + Loss



Algorithm 1 Elman RNN + Loss

procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
  Initialize the hidden state h_0 to zeros
  for t in 1 to T **do**
    Receive input data at time step t: x_t
    Compute the hidden state update:
      $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
      $h_t = \sigma(a_t)$
    Compute the output at time step t:
      $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
    Compute the cross-entropy loss at time step t:
      $\ell_t = -\sum_{k=1}^{K} (y_t^*)_k \log((y_t)_k)$
  Compute the total loss:
    $\ell = \sum_{t=1}^{T} \ell_t$

LEARNING AN RNN-LM

Learning a Language Model

Question: How do we **learn** the probabilities for the n-Gram Model?

Answer: From data! Just **count** n-gram frequencies

…the **cows eat grass**…
…our **cows eat hay** daily…
…factory-farm **cows eat corn**…
…on an organic farm, **cows eat hay** and…
…do your **cows eat grass** or corn?...
…what do **cows eat if** they have…
…**cows eat corn** when there is no…
…which **cows eat which** foods depends…
…if **cows eat grass**…
…when **cows eat corn** their stomachs…
…should we let **cows eat corn**?...

$p(w_t \mid w_{t-2} = \text{cows}, w_{t-1} = \text{eat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
corn	4/11
grass	3/11
hay	2/11
if	1/11
which	1/11

MLE for n-gram LM

- This counting method gives us the **maximum likelihood estimate** of the n-gram LM parameters
- We can derive it in the usual way:
  - **Write the likelihood** of the sentences under the n-gram LM
  - **Set the gradient to zero** and impose the constraint that the probabilities sum-to-one
  - **Solve** for the MLE

Learning a Language Model

MLE for Deep Neural LM

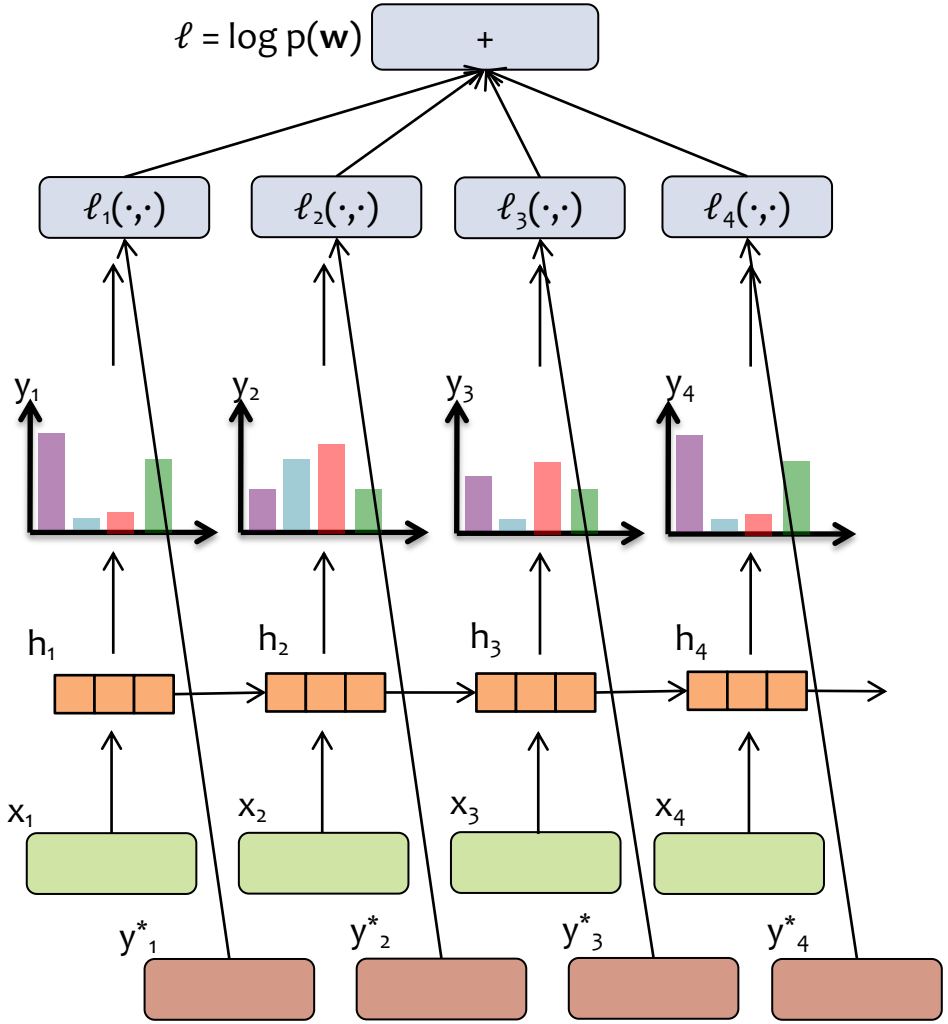
- We can also use maximum likelihood estimation to learn the parameters of an RNN-LM or Transformer-LM too!
- But **not in closed form** – instead we follow a different recipe:
  - Write the **likelihood** of the sentences under the Deep Neural LM model
  - Compute the **gradient** of the (batch) likelihood w.r.t. the parameters **by AutoDiff**
  - Follow the negative gradient using **Mini-batch SGD** (or your favorite optimizer)

MLE for n-gram LM

- This counting method gives us the **maximum likelihood estimate** of the n-gram LM parameters
- We can derive it in the usual way:
  - **Write the likelihood** of the sentences under the n-gram LM
  - **Set the gradient to zero** and impose the constraint that the probabilities sum-to-one
  - **Solve** for the MLE

RNN + Loss

How can we use this to compute the loss for an RNN-LM?



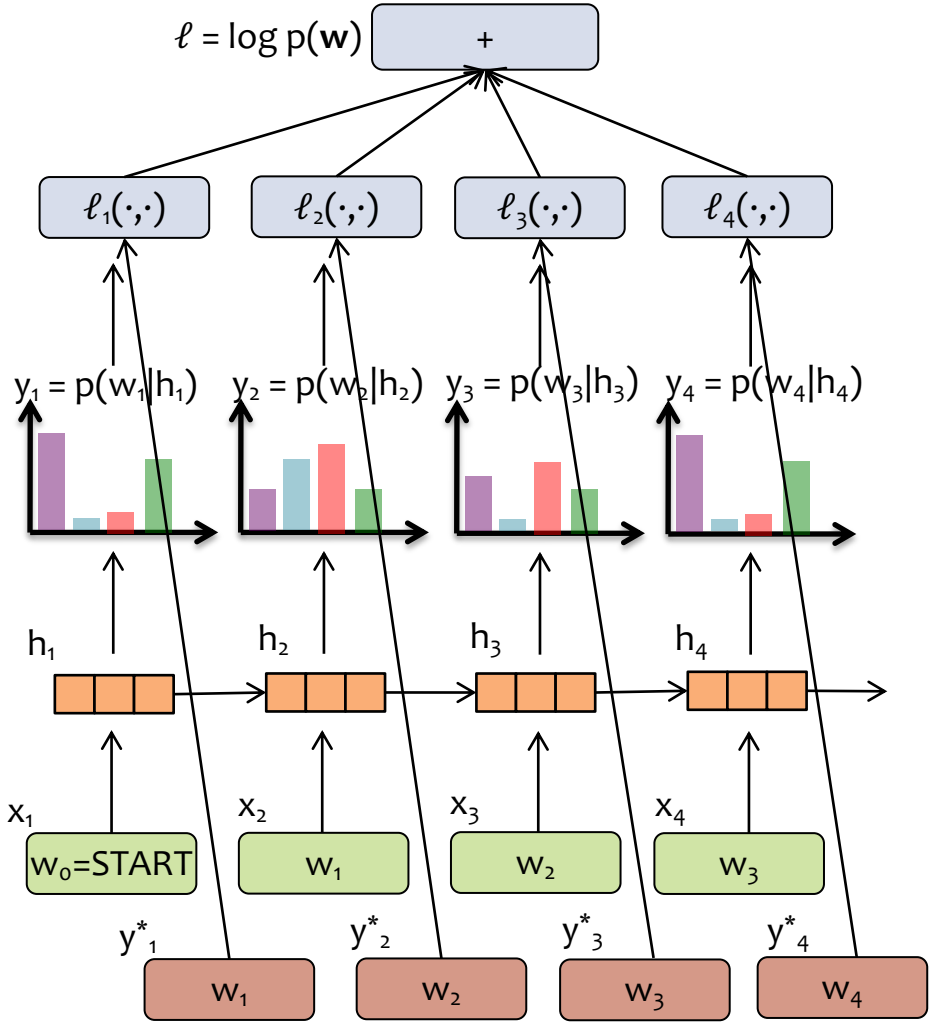
Algorithm 1 Elman RNN + Loss

procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
  Initialize the hidden state h_0 to zeros
  for t in 1 to T **do**
    Receive input data at time step t: x_t
    Compute the hidden state update:
      $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
      $h_t = \sigma(a_t)$
    Compute the output at time step t:
      $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
    Compute the cross-entropy loss at time step t:
      $\ell_t = -\sum_{k=1}^{K} (y_t^*)_k \log((y_t)_k)$
  Compute the total loss:
    $\ell = \sum_{t=1}^{T} \ell_t$

RNN-LM + Loss

How can we use this to compute the loss for an RNN-LM?

$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$
$= \log p(w_1 \mid h_1) + \ldots + \log p(w_2 \mid h_T)$



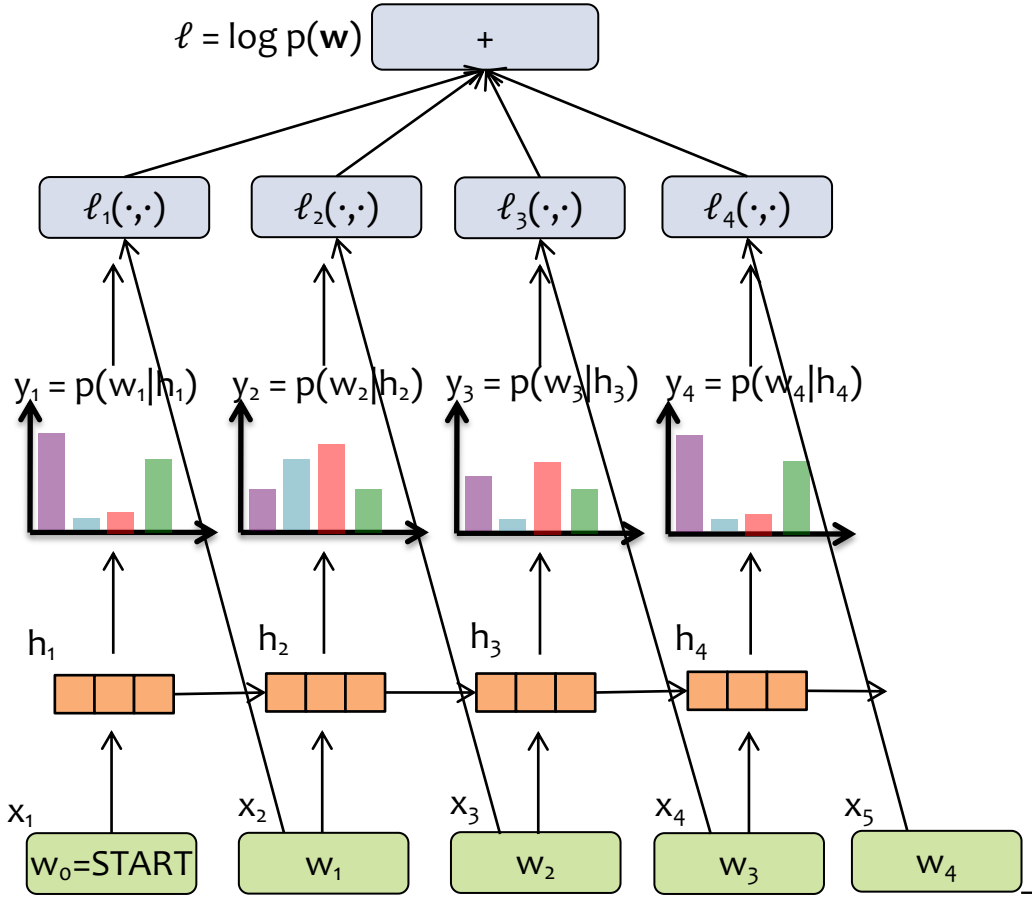
Algorithm 1 Elman RNN + Loss

procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
  Initialize the hidden state h_0 to zeros
  for t in 1 to T **do**
    Receive input data at time step t: x_t
    Compute the hidden state update:
      $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
      $h_t = \sigma(a_t)$
    Compute the output at time step t:
      $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
    Compute the cross-entropy loss at time step t:
      $\ell_t = -\sum_{k=1}^{K} (y_t^*)_k \log((y_t)_k)$
  Compute the total loss:
    $\ell = \sum_{t=1}^{T} \ell_t$

RNN-LM + Loss

How can we use this to compute the loss for an RNN-LM?

$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$
$= \log p(w_1 \mid h_1) + \ldots + \log p(w_2 \mid h_T)$



Algorithm 1 Elman RNN + Loss

procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
  Initialize the hidden state h_0 to zeros
  for t in 1 to T **do**
    Receive input data at time step t: x_t
    Compute the hidden state update:
      $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
      $h_t = \sigma(a_t)$
    Compute the output at time step t:
      $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
    Compute the cross-entropy loss at time step t:
      $\ell_t = -\sum_{k=1}^{K} (y_t^*)_k \log((y_t)_k)$
  Compute the total loss:
    $\ell = \sum_{t=1}^{T} \ell_t$

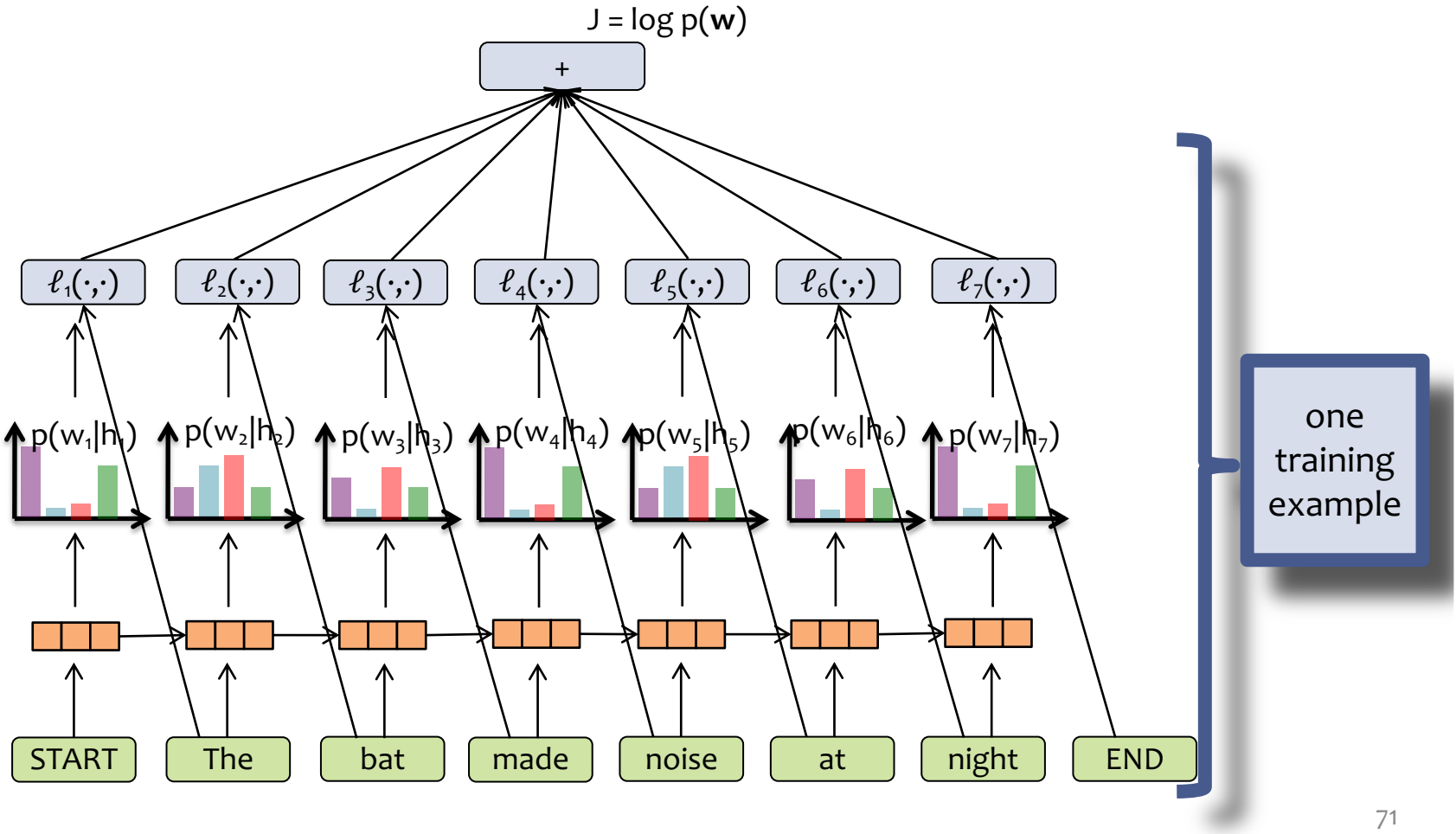
Learning an RNN-LM

- Each training example is a sequence (e.g. sentence), so we have training data D = $\{\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots, \mathbf{w}^{(N)}\}$
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:

$$J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$$

- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \ldots + \log p(w_2 \mid h_T)$$



LARGE LANGUAGE MODELS

How large are LLMs?

Comparison of some recent **large language models** (LLMs)

Model	Creators	Year of release	Training Data (# tokens)	Model Size (# parameters)
GPT-2	OpenAI	2019	~10 billion (40Gb)	1.5 billion
GPT-3 (cf. ChatGPT)	OpenAI	2020	300 billion	175 billion
PaLM	Google	2022	780 billion	540 billion
Chinchilla	DeepMind	2022	1.4 trillion	70 billion
LaMDA (cf. Bard)	Google	2022	1.56 trillion	137 billion
LLaMA	Meta	2023	1.4 trillion	65 billion
GPT-4	OpenAI	2023	?	?

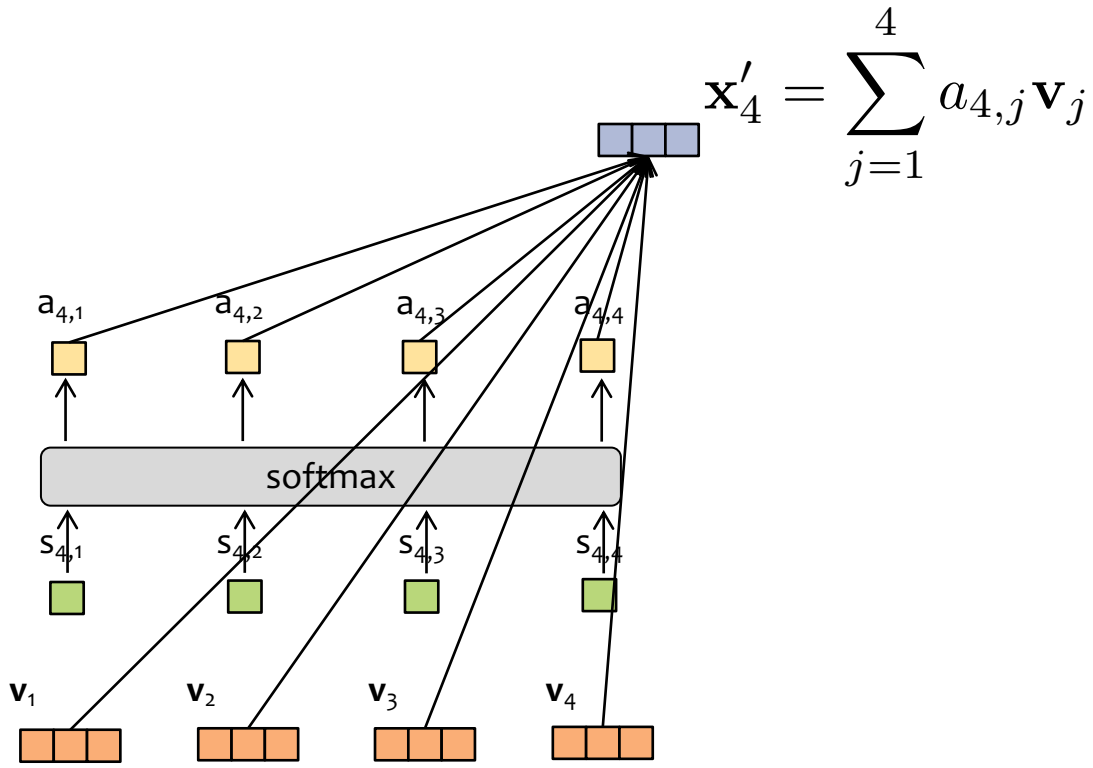
What is ChatGPT?

- ChatGPT is a large (in the sense of having many parameters) language model, fine-tuned to be a dialogue agent
- The base language model is GPT-3.5 which was trained on a large quantity of text

Transformer Language Models

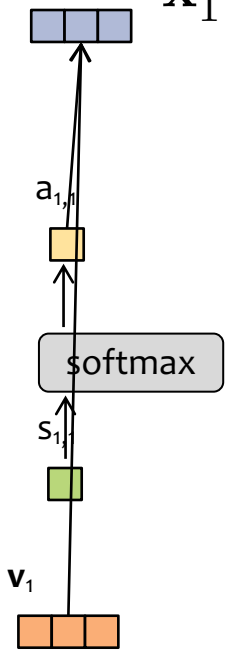
MODEL: GPT

Attention


$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$
$a_{4,1}$
$a_{4,2}$
$a_{4,3}$
$a_{4,4}$
softmax
$s_{4,1}$
$s_{4,2}$
$s_{4,3}$
$s_{4,4}$
$\mathbf{v}_1$
$\mathbf{v}_2$
$\mathbf{v}_3$
$\mathbf{v}_4$

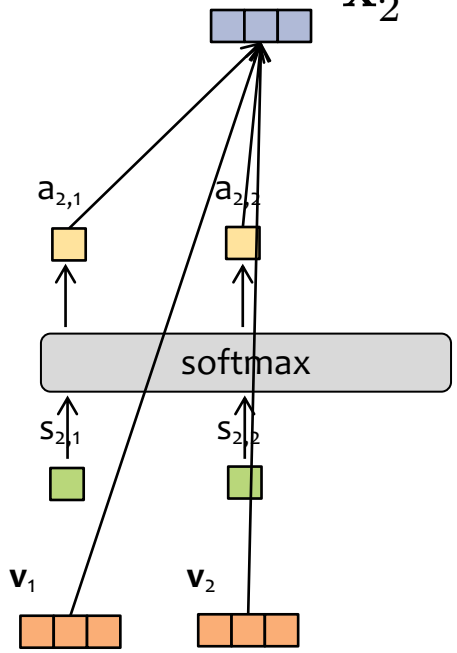
Attention

$$\mathbf{x}'_1 = \sum_{j=1}^{1} a_{1,j}\mathbf{v}_j$$

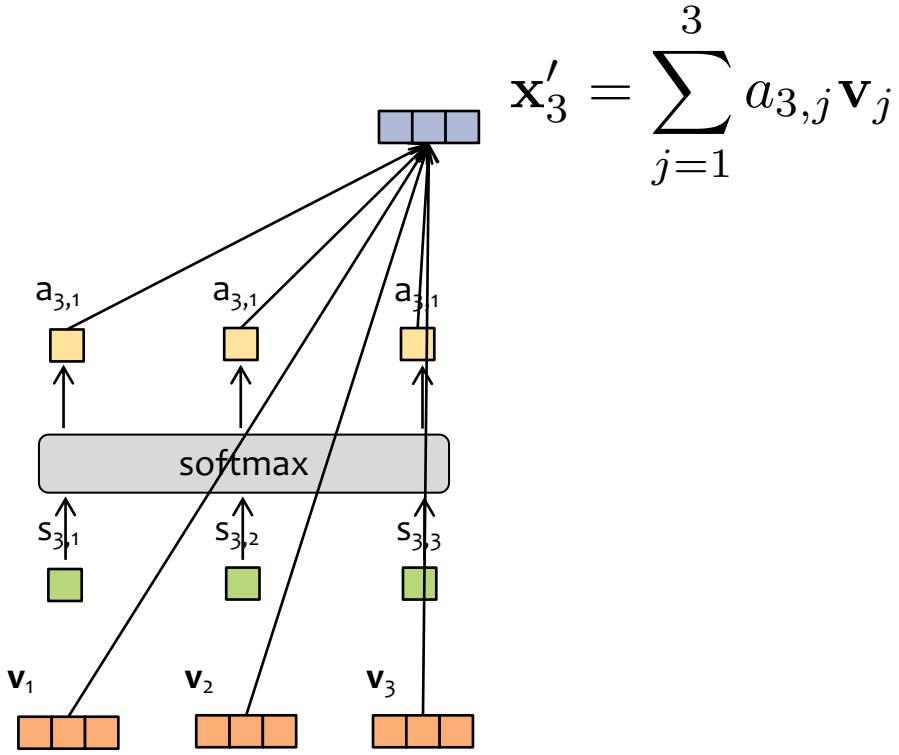


Attention

$$\mathbf{x}'_2 = \sum_{j=1}^{2} a_{2,j}\mathbf{v}_j$$

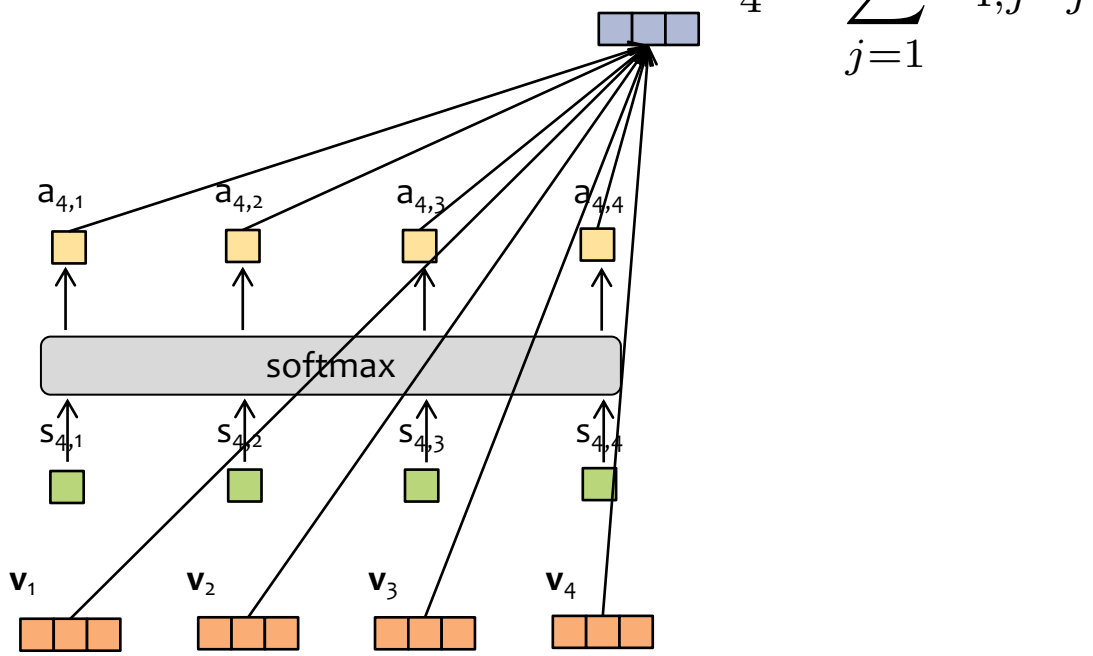


Attention

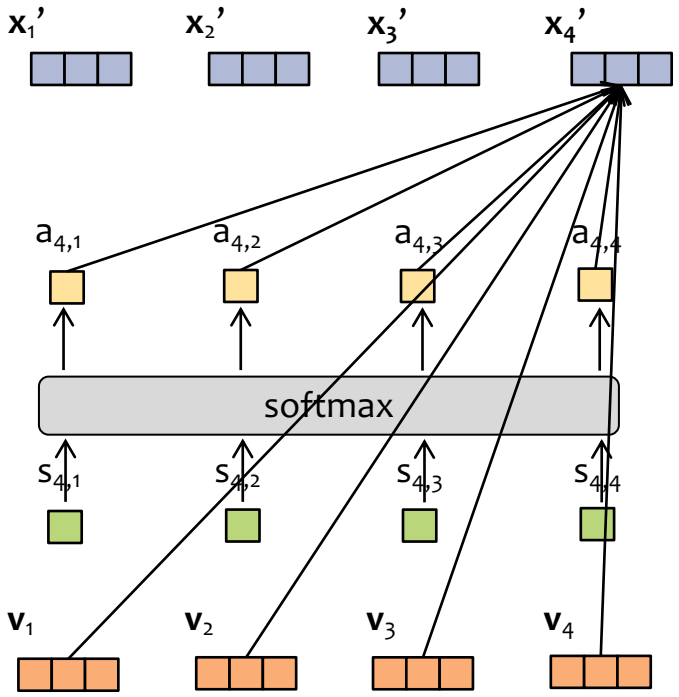

$\mathbf{x}'_3 = \sum_{j=1}^{3} a_{3,j}\mathbf{v}_j$
a3,1
a3,1
a3,1
softmax
s3,1
s3,2
s3,3
v1
v2
v3

Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



Attention



$$\mathbf{x}'_t = \sum_{j=1}^{t} a_{t,j}\mathbf{v}_j$$

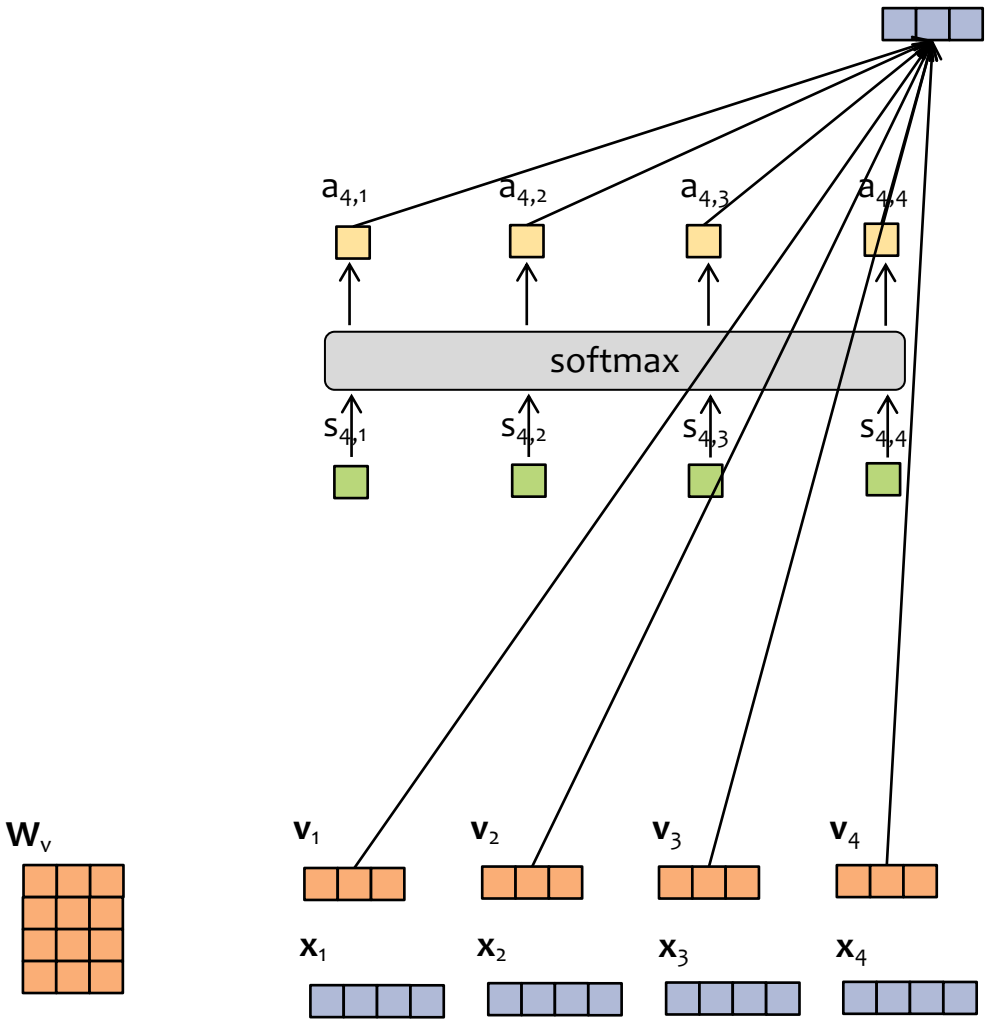
attention weights

scores

values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

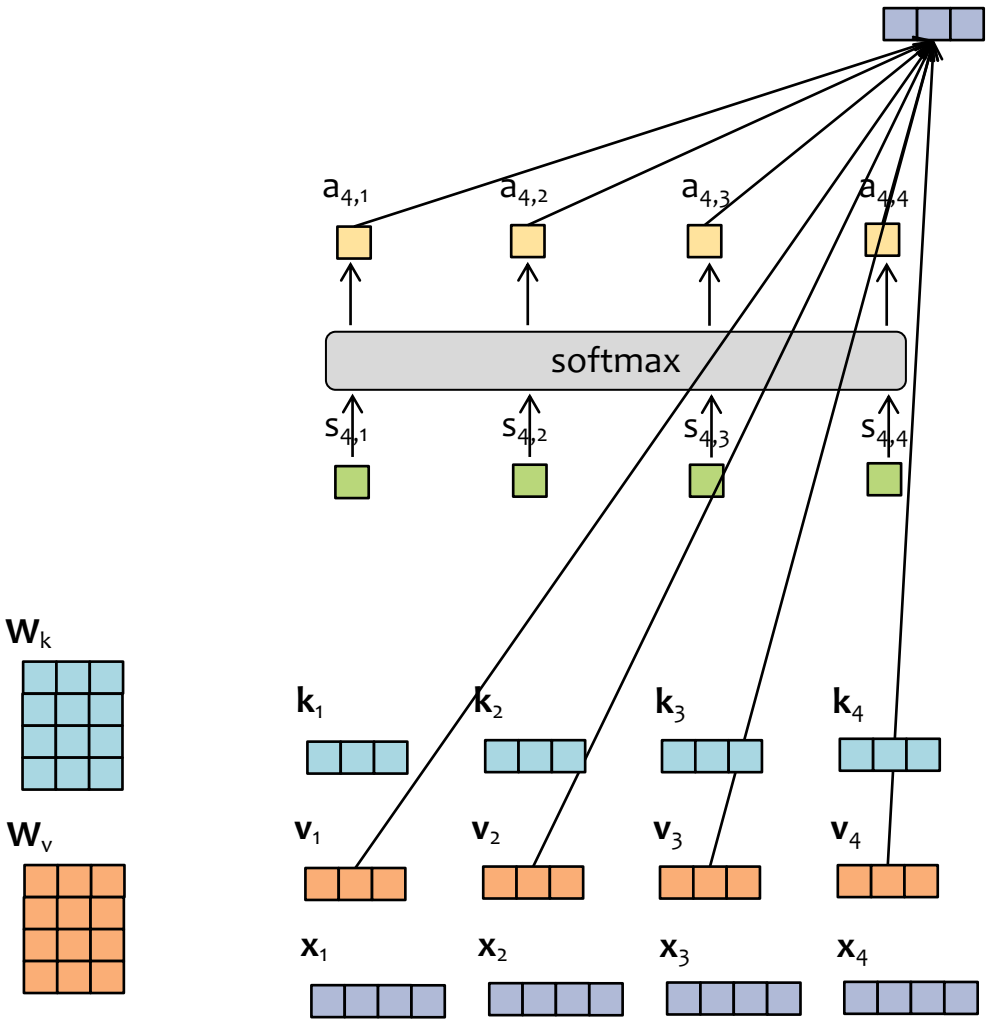


$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$$

values

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

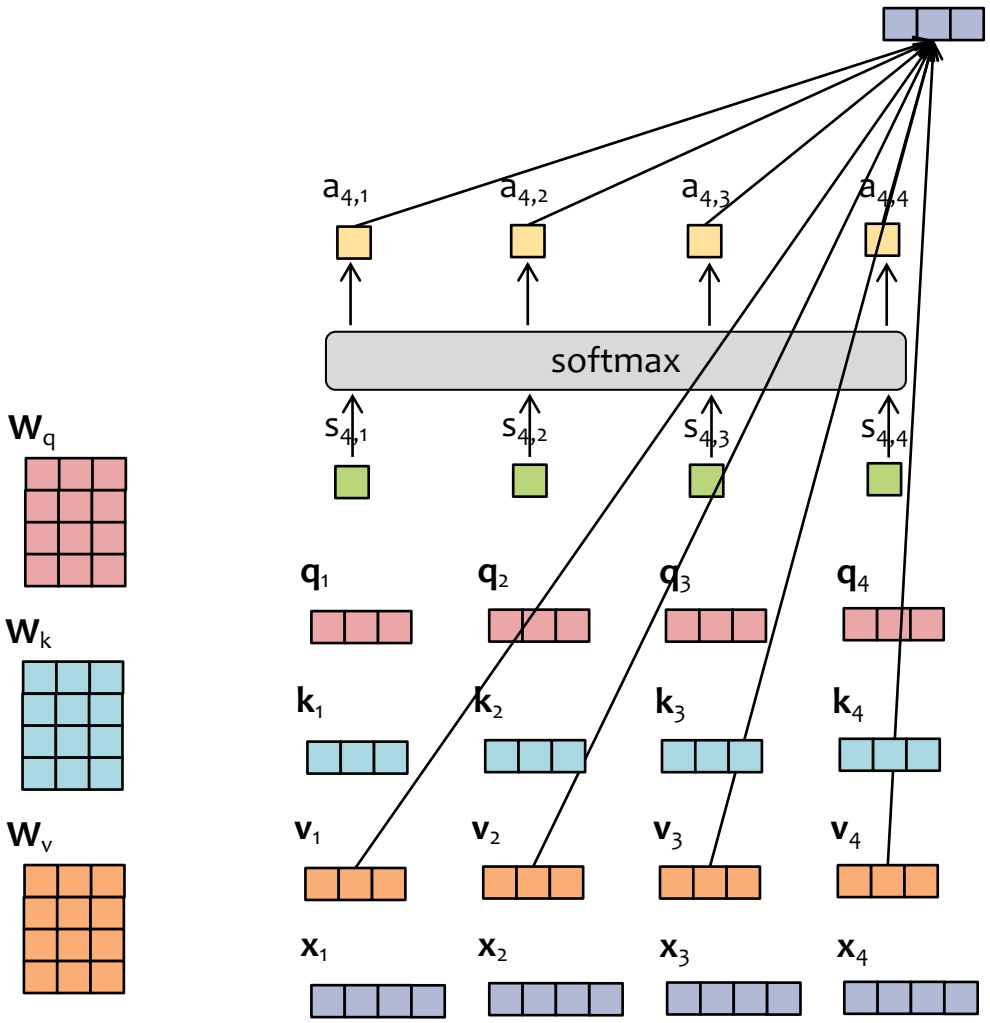


$$\mathbf{k}_j = \mathbf{W}_k^T\mathbf{x}_j \qquad \text{keys}$$

$$\mathbf{v}_j = \mathbf{W}_v^T\mathbf{x}_j \qquad \text{values}$$

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



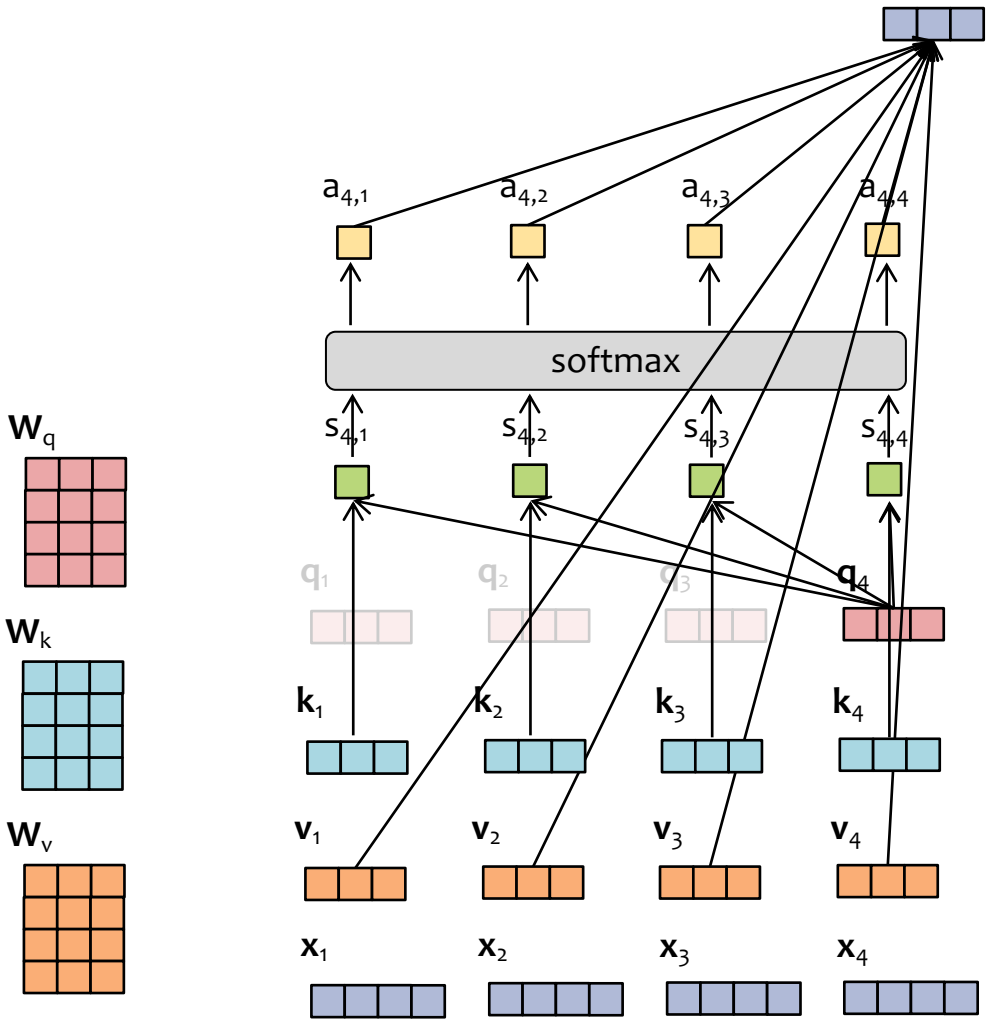
$$\mathbf{q}_j = \mathbf{W}_q^T\mathbf{x}_j \quad \text{queries}$$

$$\mathbf{k}_j = \mathbf{W}_k^T\mathbf{x}_j \quad \text{keys}$$

$$\mathbf{v}_j = \mathbf{W}_v^T\mathbf{x}_j \quad \text{values}$$

Scaled Dot-Product Attention

$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$



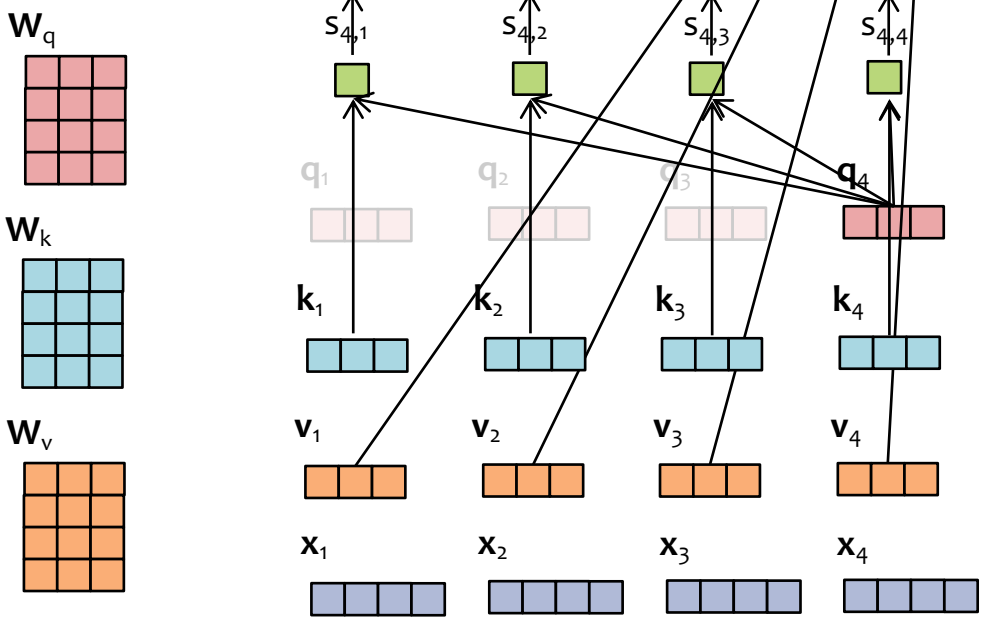
$$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k} \quad \text{scores}$$

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j \quad \text{queries}$$

$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j \quad \text{keys}$$

$$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j \quad \text{values}$$

Scaled Dot-Product Attention



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

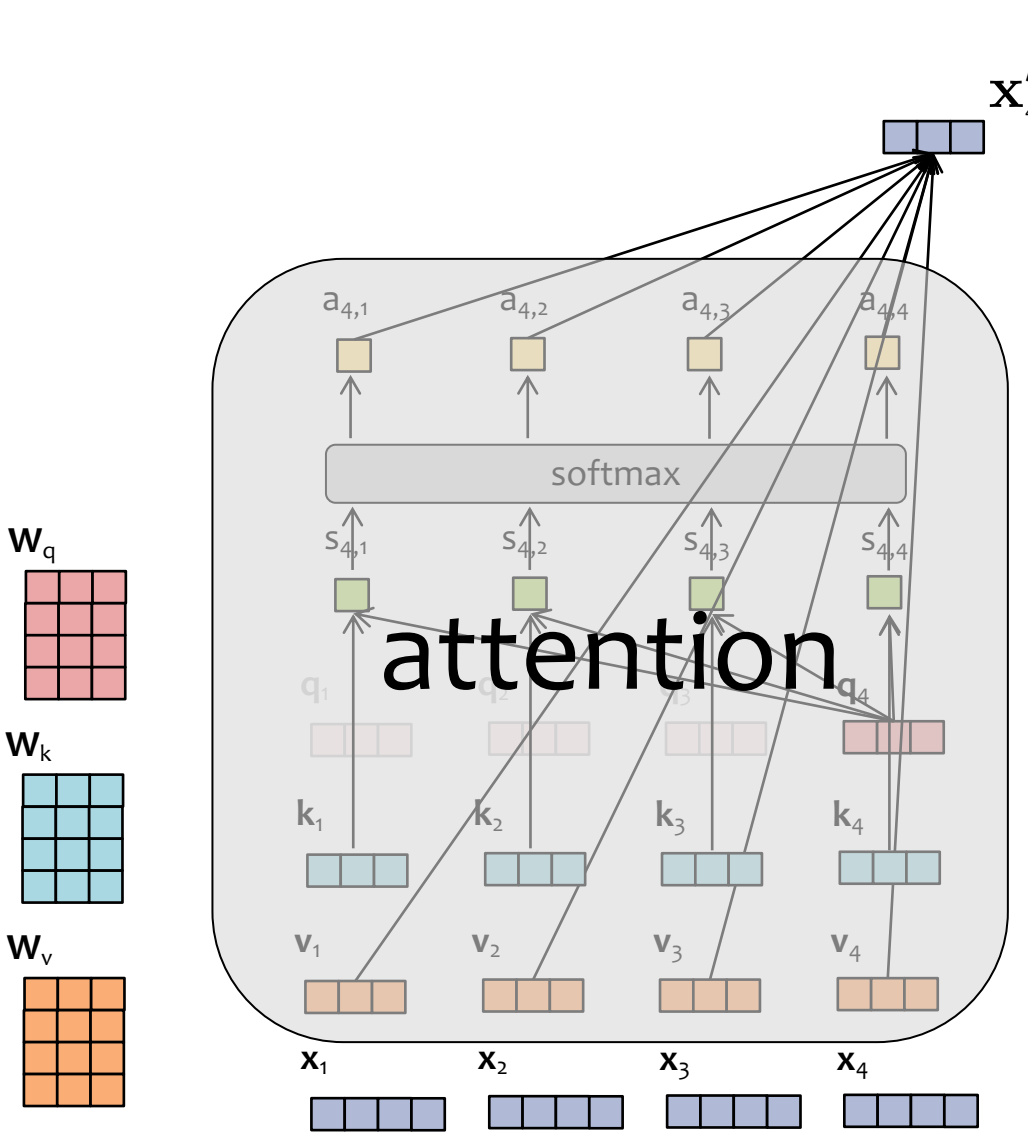
$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Scaled Dot-Product Attention



$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

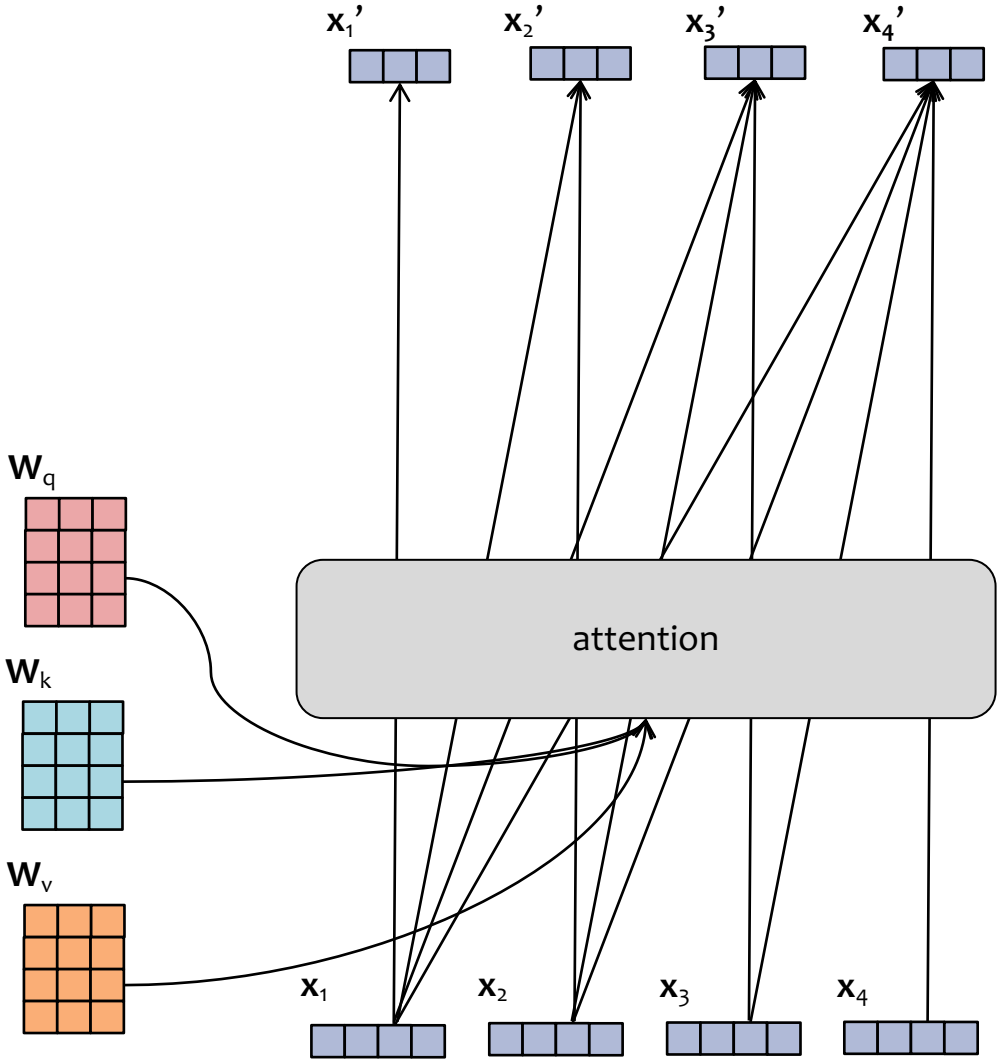
$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Scaled Dot-Product Attention



$$\mathbf{x}'_t = \sum_{j=1}^{t} a_{t,j}\mathbf{v}_j$$

$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t)$ attention weights

$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

Animation of 3D Convolution

Recall…

http://cs231n.github.io/convolutional-networks/

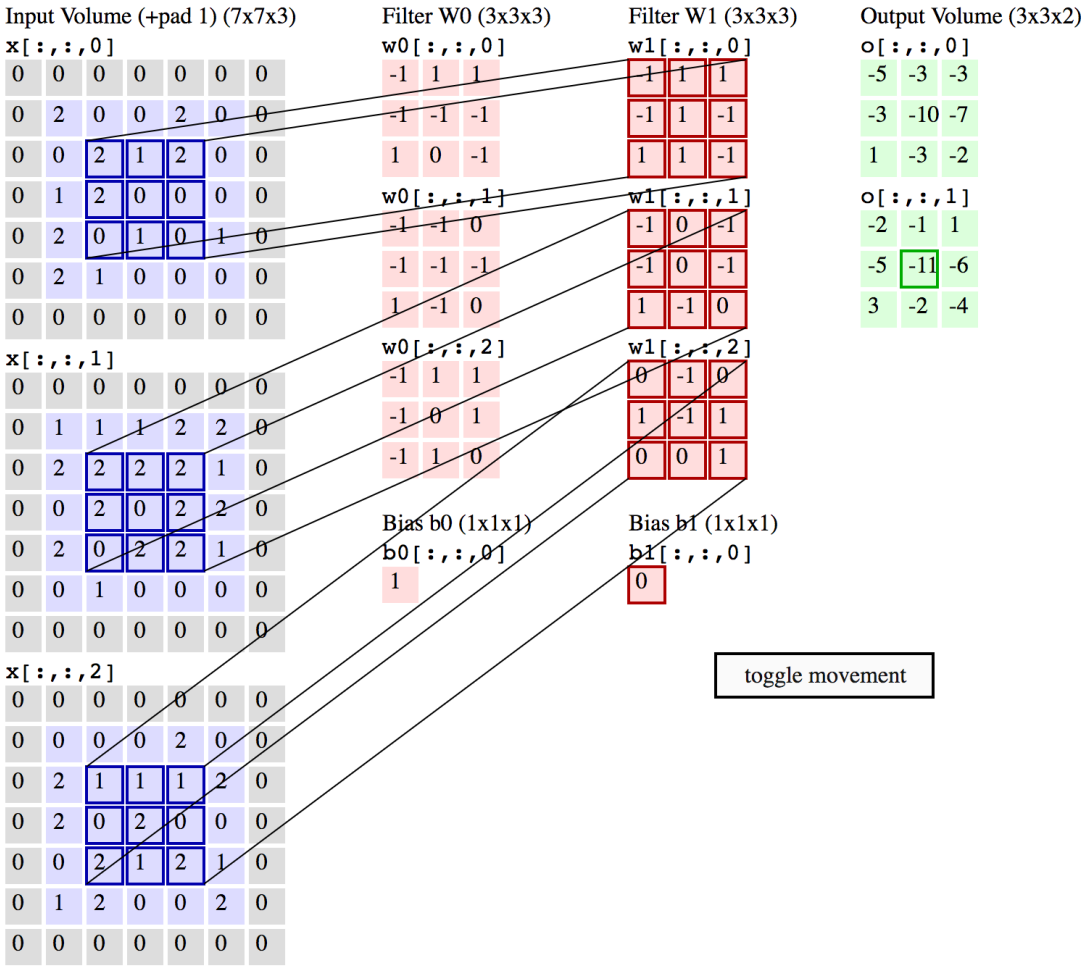
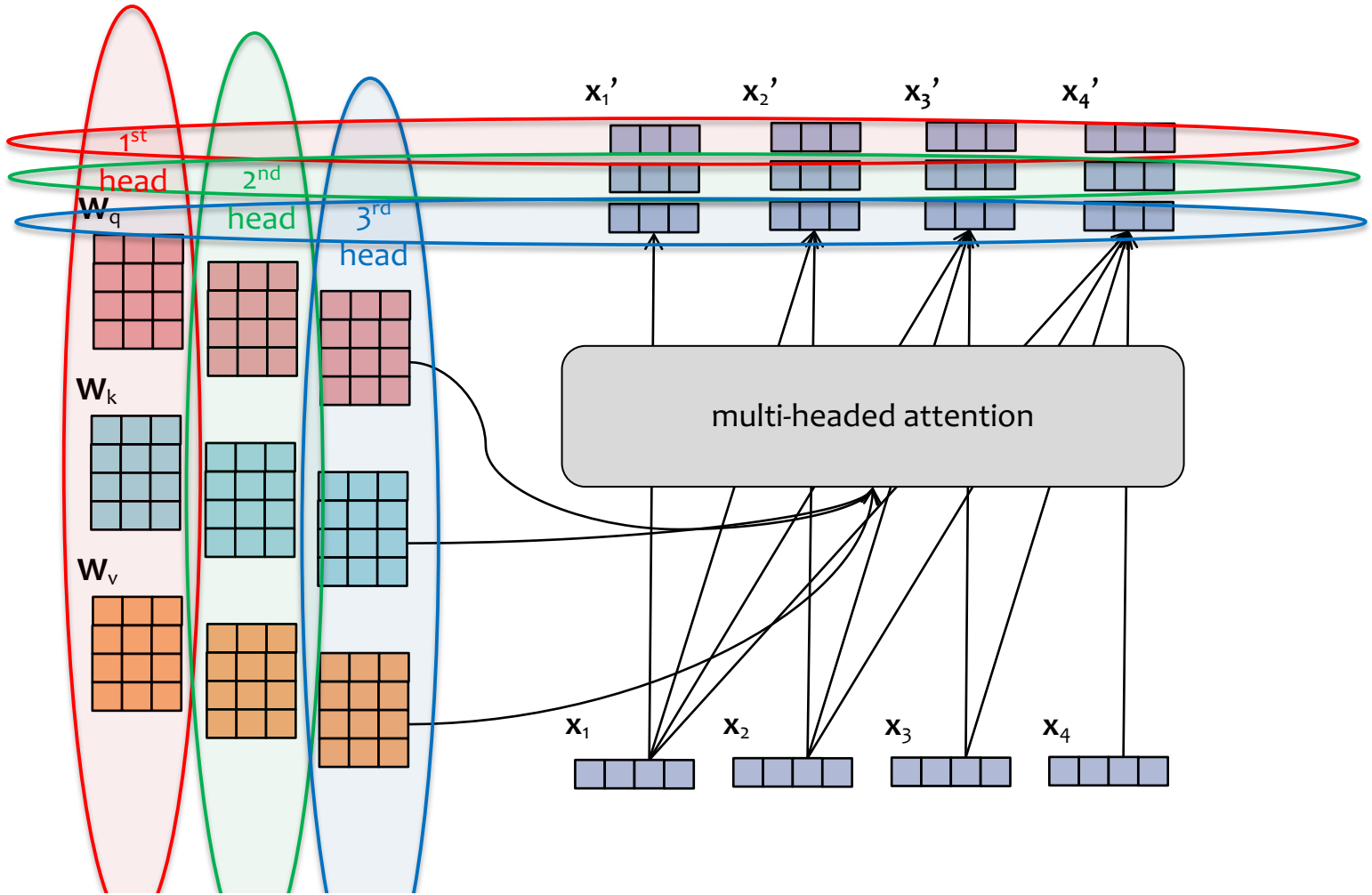


Figure from Fei-Fei Li & Andrej Karpathy & Justin Johnson (CS231N)

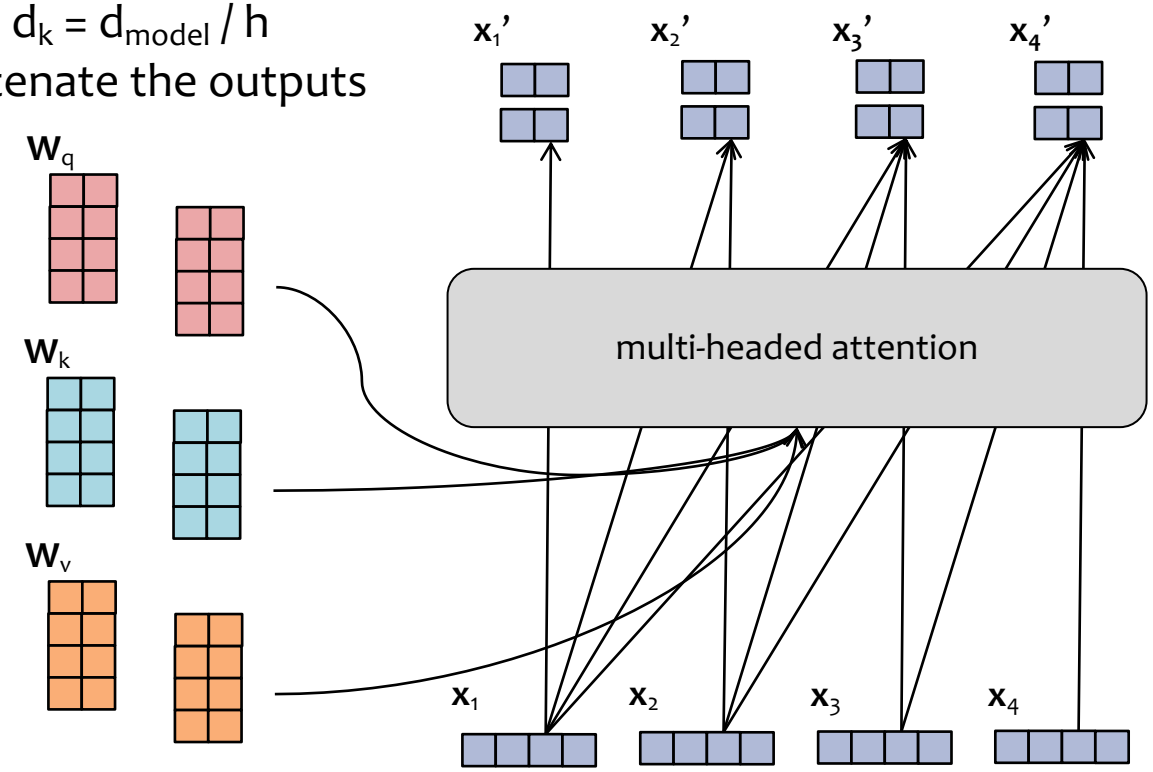
Multi-headed Attention



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

Multi-headed Attention

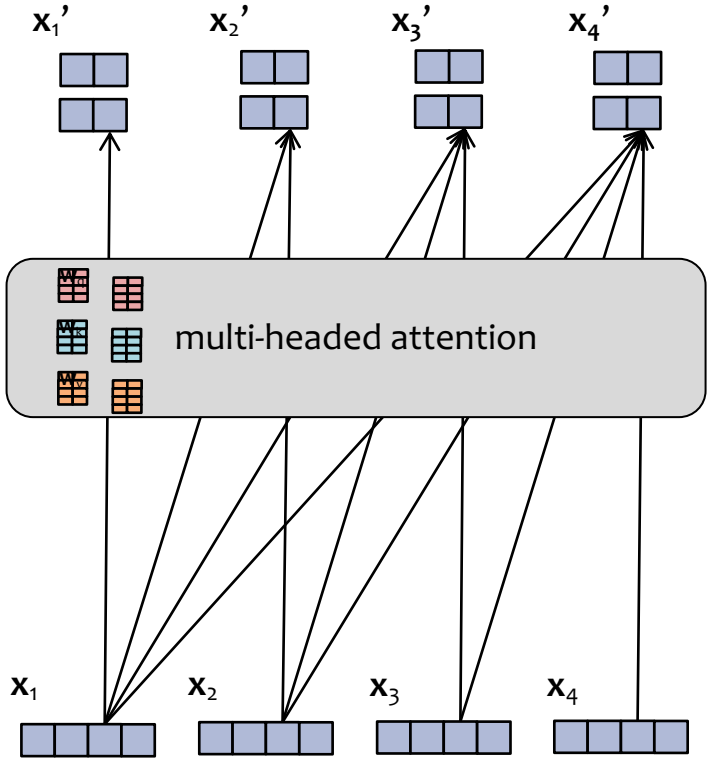
- To ensure the dimension of the **input** embedding $\mathbf{x}_t$ is the same as the **output** embedding $\mathbf{x}_t$', Transformers usually choose the embedding sizes and number of heads appropriately:
  - d_{model} = dim. of inputs
  - d_k = dim. of each output
  - h = # of heads
  - Choose $d_k = d_{model} / h$
- Then concatenate the outputs



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

Multi-headed Attention

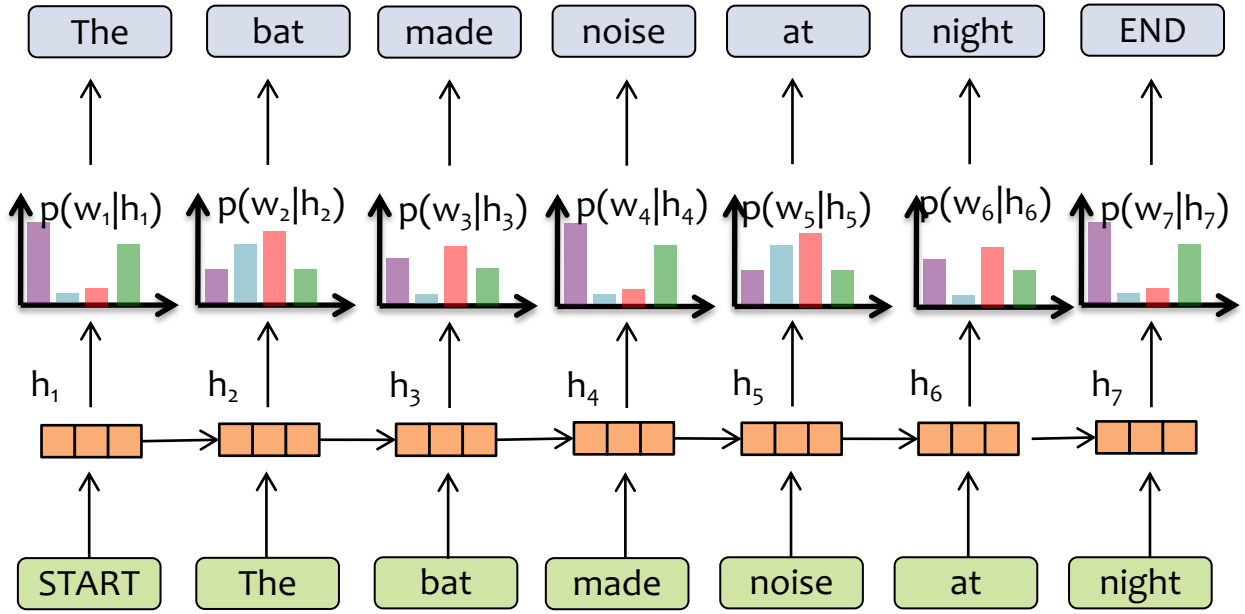
- To ensure the dimension of the **input** embedding $\mathbf{x}_t$ is the same as the **output** embedding $\mathbf{x}_t$', Transformers usually choose the embedding sizes and number of heads appropriately:
  - d_{model} = dim. of inputs
  - d_k = dim. of each output
  - h = # of heads
  - Choose $d_k = d_{model} / h$
- Then concatenate the outputs



- Just as we can have **multiple channels** in a **convolution** layer, we can use **multiple heads** in an **attention** layer
- Each head gets **its own parameters**
- We can **concatenate** all the outputs to get a single vector for each time step

RNN Language Model

Recall…



Key Idea:

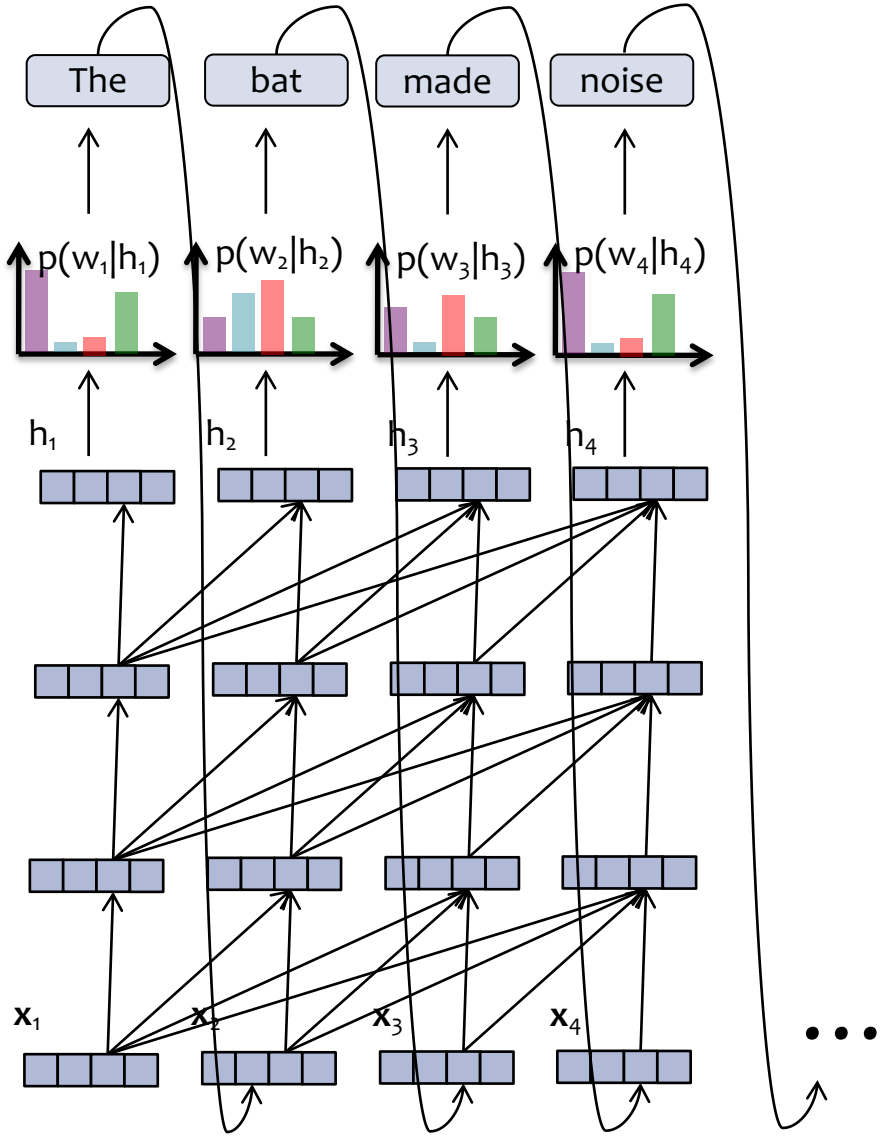
(1) convert all previous words to a **fixed length vector**

(2) define distribution $p(w_t \mid f_\theta(w_{t-1}, \ldots, w_1))$ that conditions on the vector $\mathbf{h}_t = f_\theta(w_{t-1}, \ldots, w_1)$

Transformer Language Model

Important!

- RNN computation graph grows **linearly** with the number of input tokens
- Transformer-LM computation graph grows **quadratically** with the number of input tokens



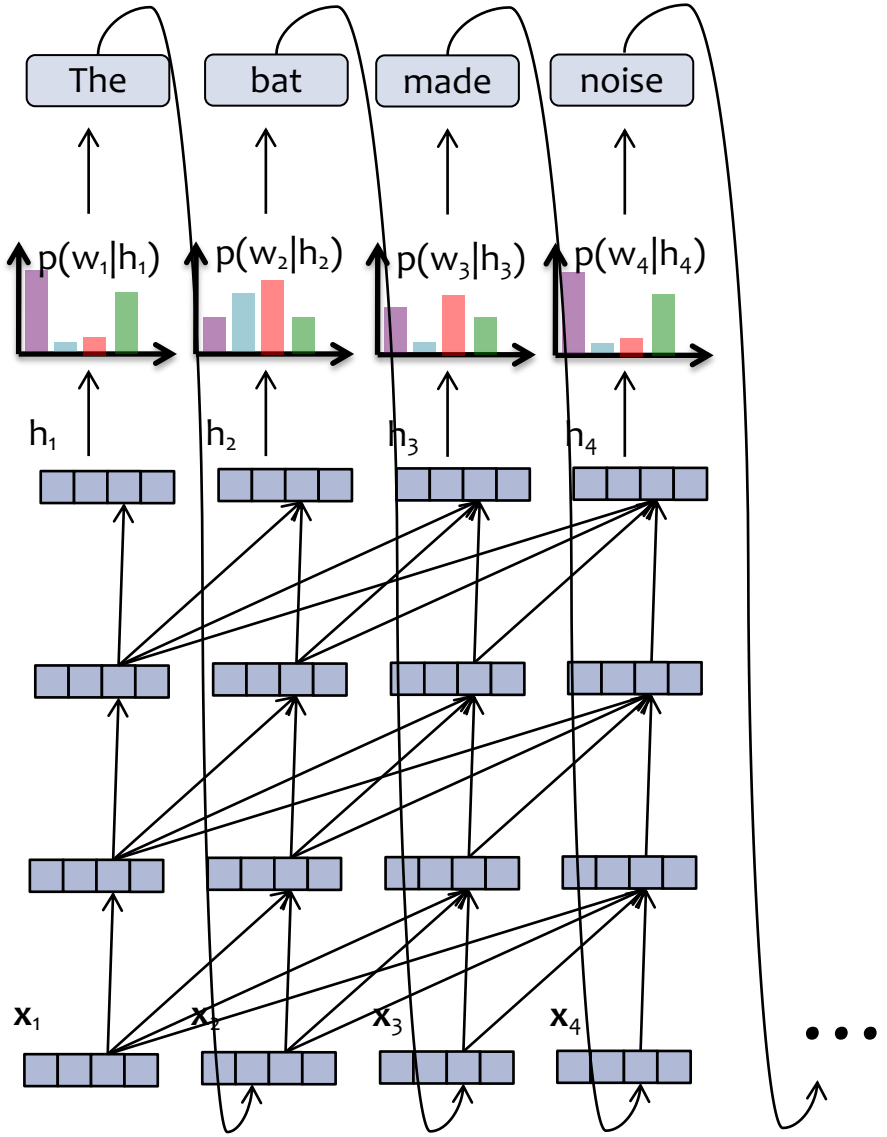
Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM!

Transformer Language Model

Important!

- RNN computation graph grows **linearly** with the number of input tokens
- Transformer-LM computation graph grows **quadratically** with the number of input tokens



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM!

Layer Normalization

- *The Problem:* **internal covariate shift** occurs during training of a deep network when a small change in the low layers amplifies into a large change in the high layers
- *One Solution:* **Layer normalization** normalizes each layer and learns elementwise gain/bias
- Such normalization allows for higher learning rates (for **faster convergence**) without issues of diverging gradients

Given input $\mathbf{a} \in \mathbb{R}^K$, LayerNorm computes output $\mathbf{b} \in \mathbb{R}^K$:

$$\mathbf{b} = \boldsymbol{\gamma} \odot \frac{\mathbf{a} - \mu}{\sigma} \oplus \boldsymbol{\beta}$$

where we have mean $\mu = \frac{1}{K}\sum_{k=1}^{K} a_k$,
standard deviation $\sigma = \sqrt{\frac{1}{K}\sum_{k=1}^{K}(a_k - \mu)^2}$,
and parameters $\boldsymbol{\gamma} \in \mathbb{R}^K$, $\boldsymbol{\beta} \in \mathbb{R}^K$.
$\odot$ and $\oplus$ denote elementwise multiplication and addition.

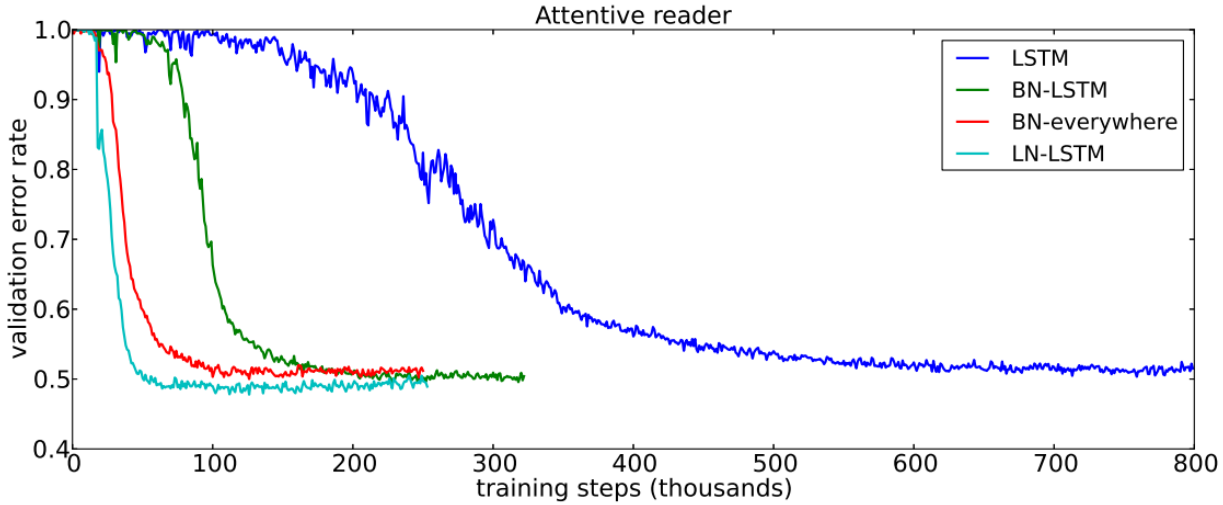
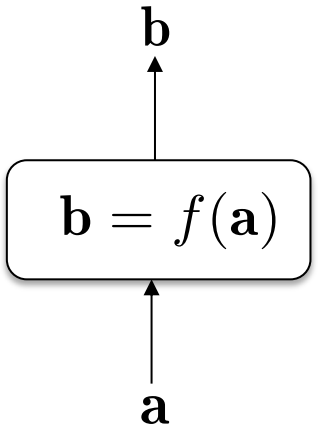


Figure from https://arxiv.org/pdf/1607.06450.pdf

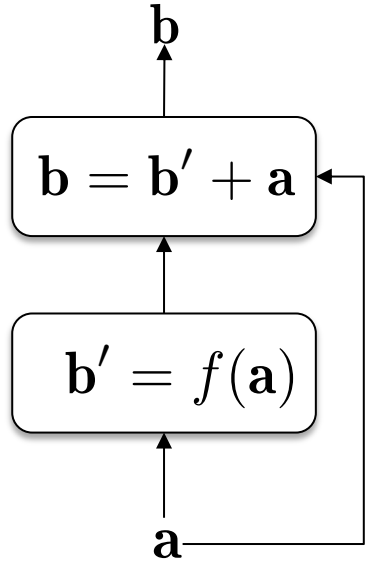
Residual Connections

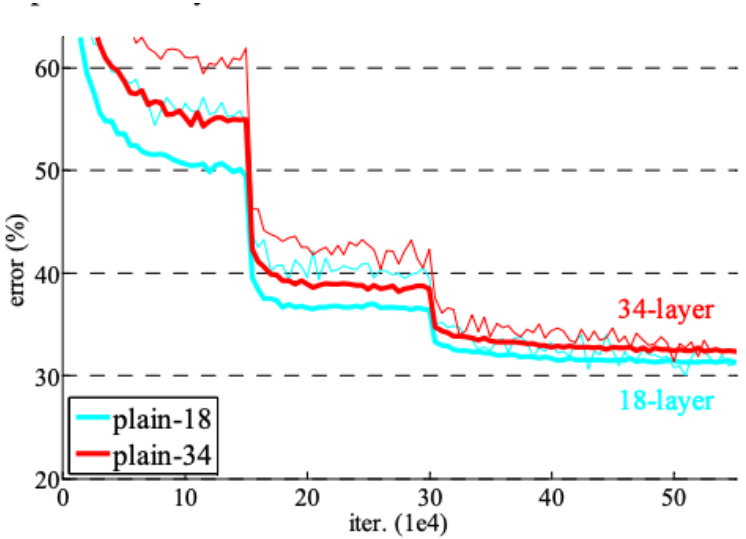
- *The Problem:* as network depth grows very large, a **performance degradation** occurs that is not explained by overfitting (i.e. train / test error both worsen)
- *One Solution:* **Residual connections** pass a copy of the input alongside another function so that information can flow more directly
- These residual connections allow for **effective training of very deep networks** that perform better than their shallower (though still deep) counterparts

Plain Connection



Residual Connection





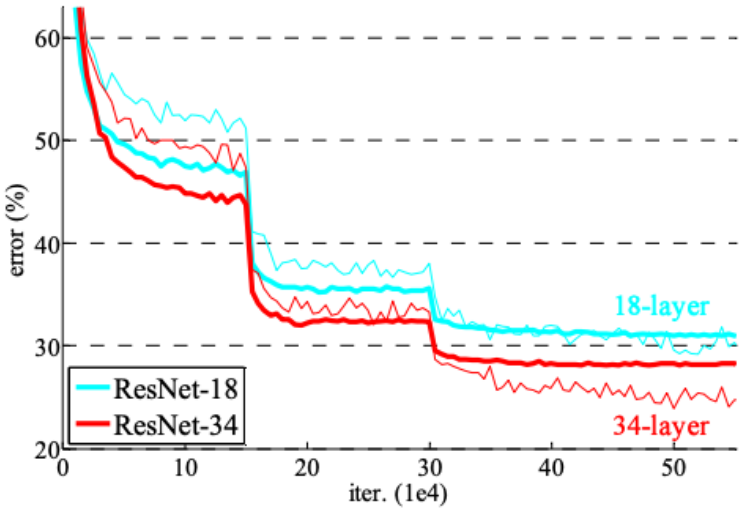
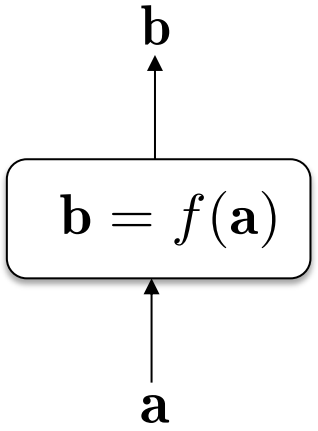


Figure from https://arxiv.org/pdf/1512.03385.pdf

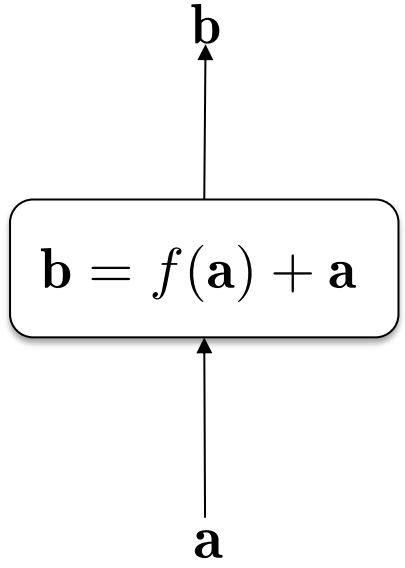
Residual Connections

- *The Problem:* as network depth grows very large, a **performance degradation** occurs that is not explained by overfitting (i.e. train / test error both worsen)
- *One Solution:* **Residual connections** pass a copy of the input alongside another function so that information can flow more directly
- These residual connections allow for **effective training of very deep networks** that perform better than their shallower (though still deep) counterparts

Plain Connection



Residual Connection

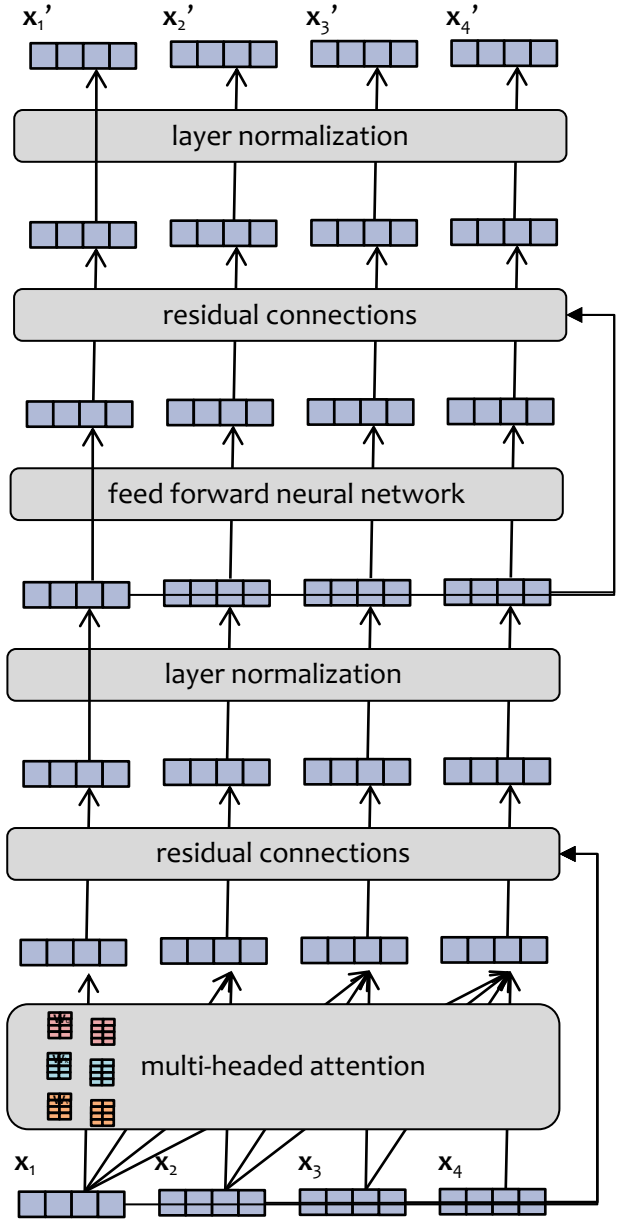


Why are residual connections helpful?

Instead of f(a) having to learn a full transformation of a, f(a) only needs to learn an additive modification of a (i.e. the residual).

Figure from https://arxiv.org/pdf/1512.03385.pdf

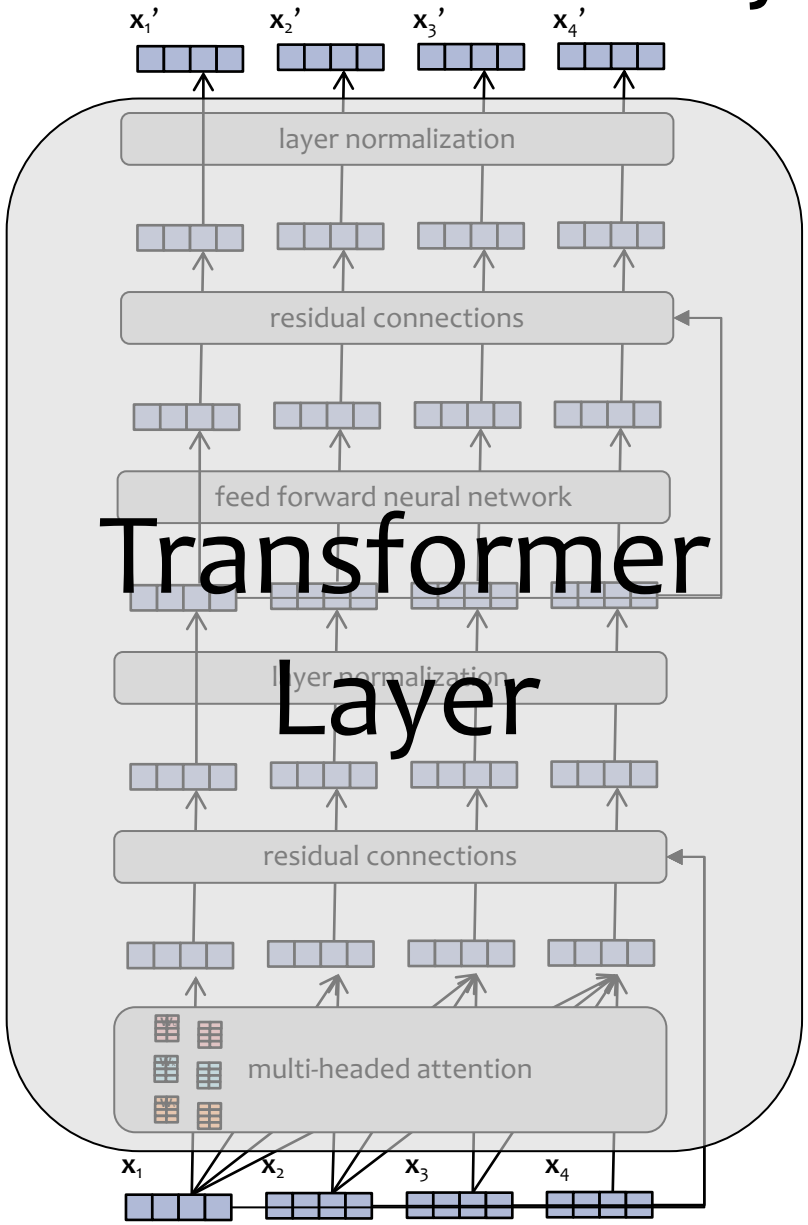
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

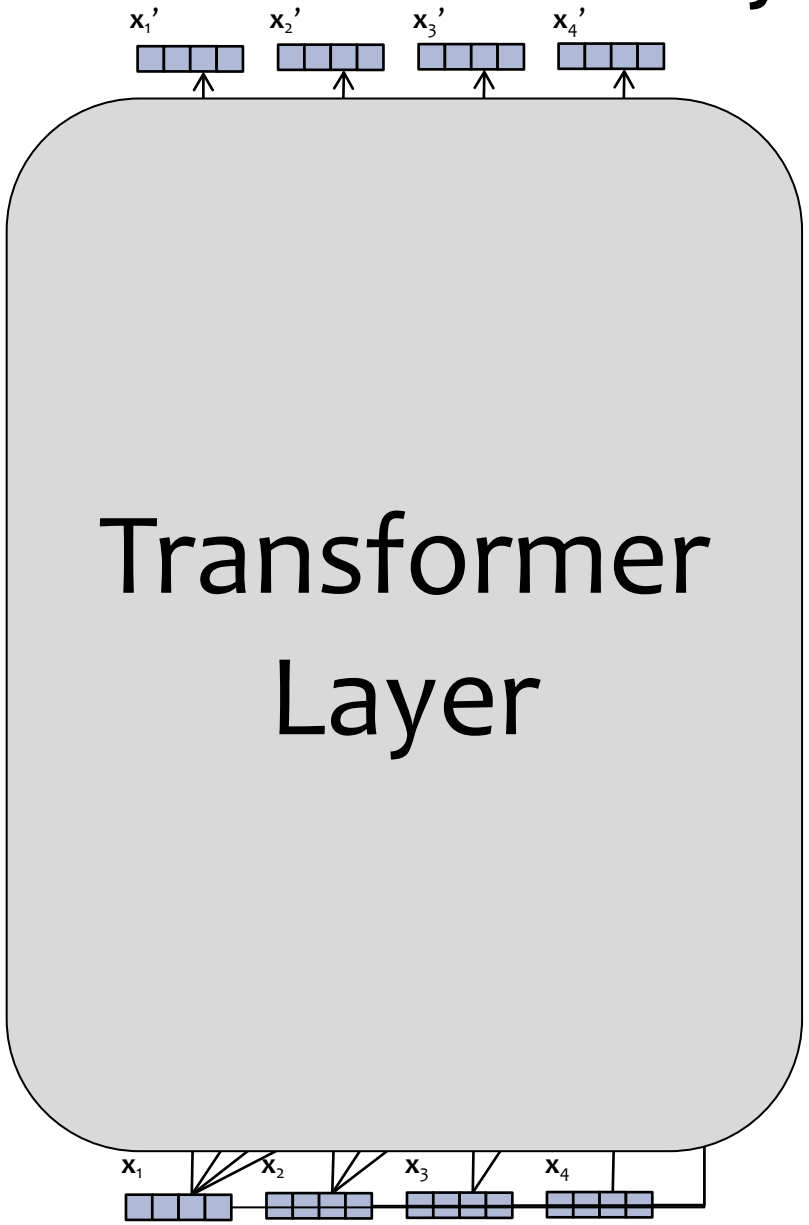
Transformer Layer



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Transformer Layer



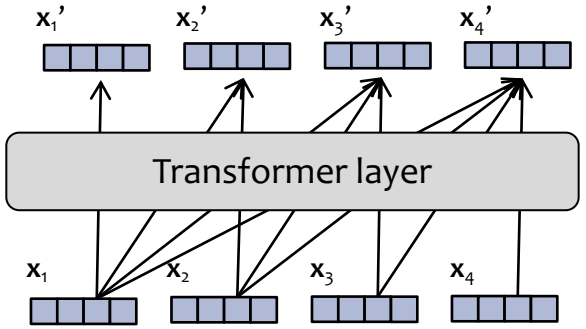
Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

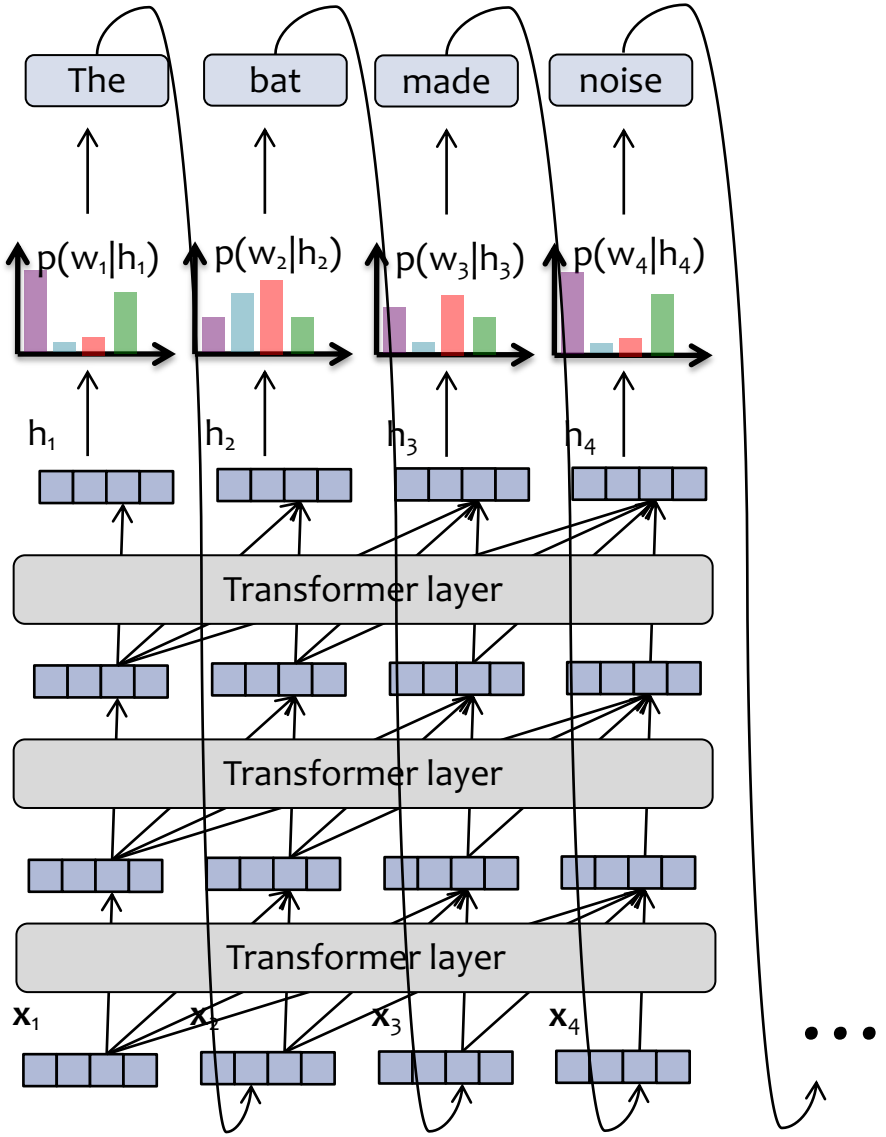
Transformer Layer

Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections



Transformer Language Model



Each layer of a Transformer LM consists of several **sublayers:**

1. attention
2. feed-forward neural network
3. layer normalization
4. residual connections

Each hidden vector looks back at the hidden vectors of the **current and previous timesteps in the previous layer.**

The language model part is just like an RNN-LM.

In-Class Poll

Question:

Suppose we have the following input embeddings and attention weights:

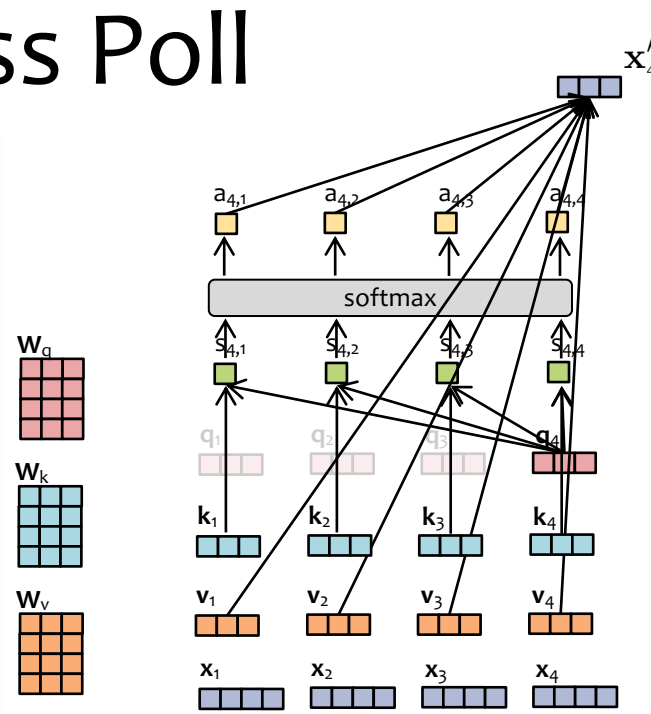
- $x_1 = [1,0,0,0]$ $a_{4,1} = 0.1$
- $x_2 = [0,1,0,0]$ $a_{4,2} = 0.2$
- $x_3 = [0,0,2,0]$ $a_{4,3} = 0.6$
- $x_4 = [0,0,0,1]$ $a_{4,4} = 0.1$

And $W_v = I$. Then we can compute x_4'.

Now suppose we swap the embeddings x_2 and x_3 such that

- $x_2 = [0,0,2,0]$
- $x_3 = [0,1,0,0]$

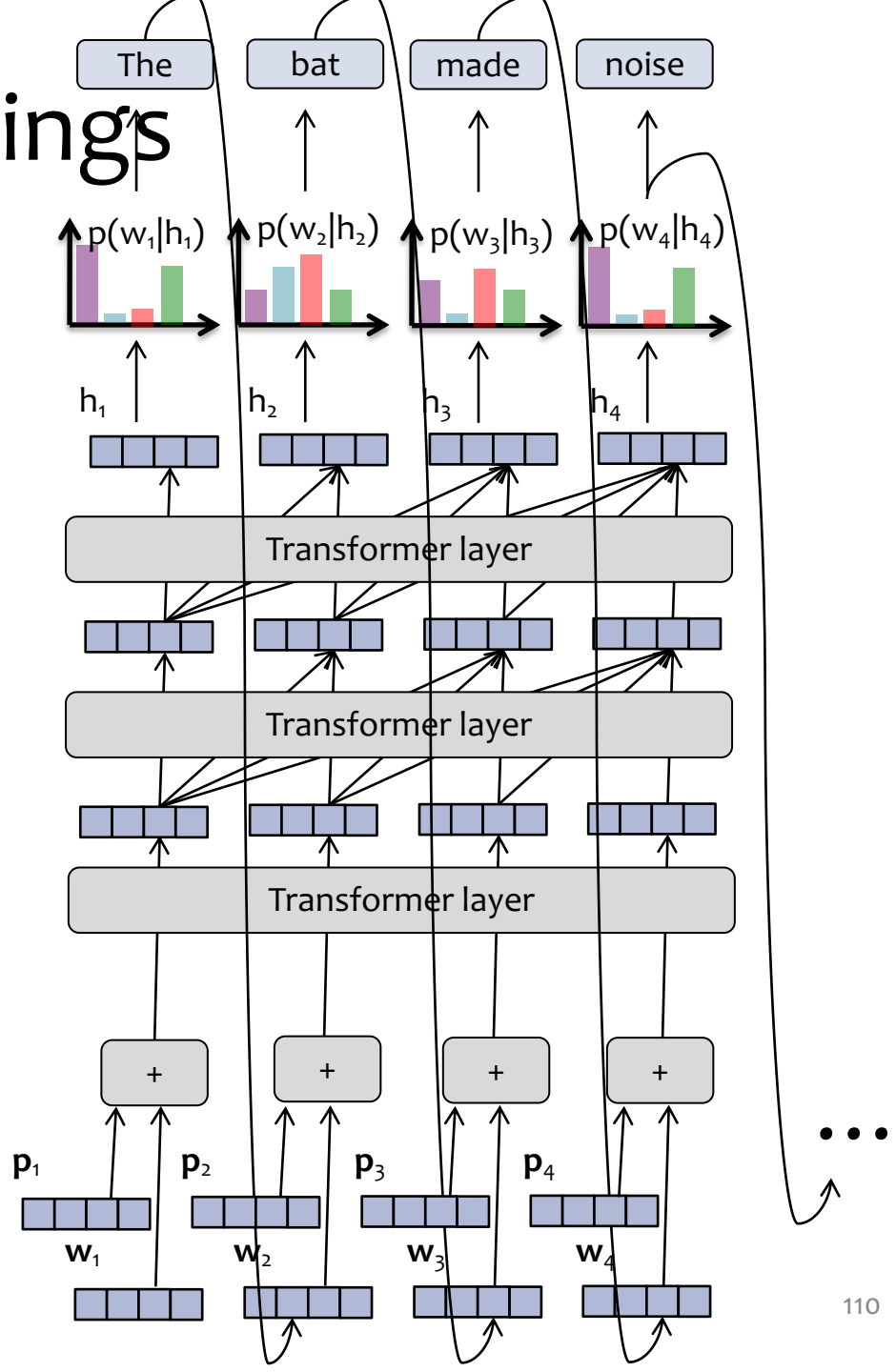
What is the new value of x_4'?



Answer:

Position Embeddings

- The Problem: Because attention is position invariant, we **need** a way to learn about positions
- The Solution: Use (or learn) a collection of position specific embeddings: $\mathbf{p}_t$ represents what it means to be in position t. And add this to the word embedding $\mathbf{w}_t$.
  The **key idea** is that every word that appears in position t uses the same position embedding $\mathbf{p}_t$
- There are a number of varieties of position embeddings:
  - Some are fixed (based on sine and cosine), whereas others are learned (like word embeddings)
  - Some are absolute (as described above) but we can also use relative position embeddings (i.e. relative to the position of the query vector)

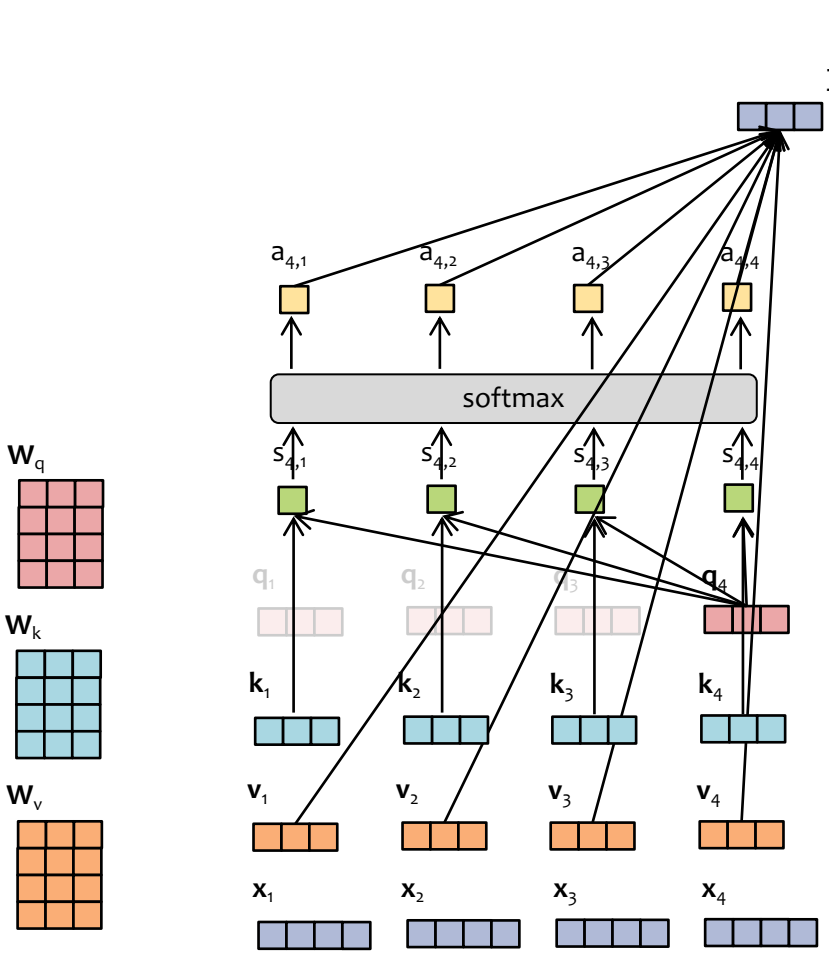


GPT-3

- GPT stands for Generative Pre-trained Transformer
- GPT is just a Transformer LM, but with a huge number of parameters

Model	# layers	dimension of states	dimension of inner states	# attention heads	# params
GPT (2018)	12	768	3072	12	117M
GPT-2 (2019)	48	1600	--	--	1542M
GPT-3 (2020)	96	12288	4*12288	96	175000M

Matrix Version of Scaled Dot-Product Attention



$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$ attention weights

$s_{4,j} = \mathbf{k}_j^T \mathbf{q}_4 / \sqrt{d_k}$ scores

$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j$ queries

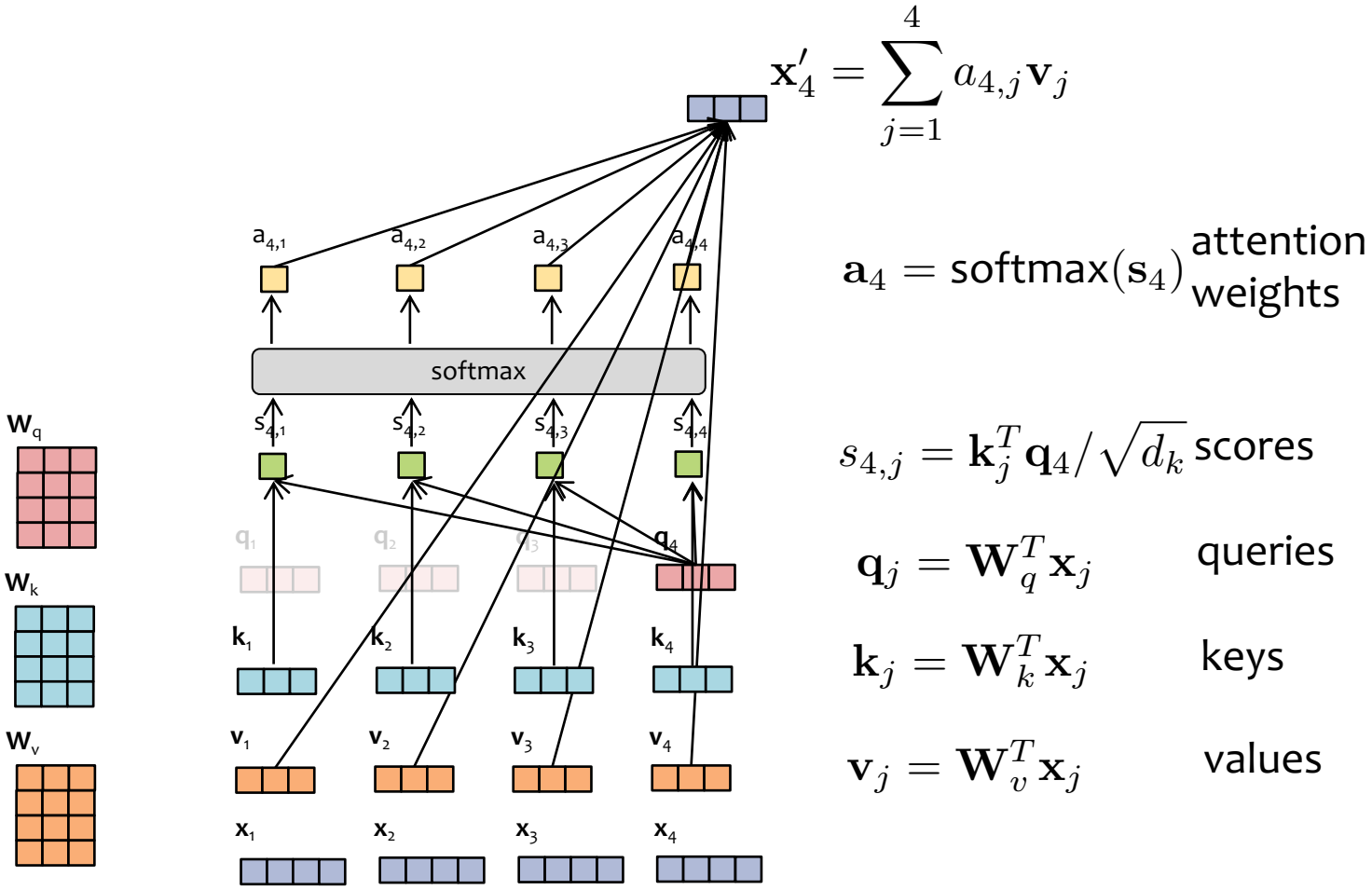
$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j$ keys

$\mathbf{v}_j = \mathbf{W}_v^T \mathbf{x}_j$ values

- For speed, we compute all the queries at once using matrix operations
- First we pack the queries, keys, values into matrices:
  - $Q = [q_1,\ldots,q_N]^T$
  - $K = [k_1,\ldots,k_N]^T$
  - $V = [v_1,\ldots,v_N]^T$
- Then we compute all the queries at once:

$$\text{Attn}(\mathbf{x}_{1:N}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right)$$

Matrix Version of Scaled Dot-Product Attention



In practice, the attention weights are computed for all time steps T, then we mask out (by setting to –inf) all the inputs to the softmax that are for the timesteps to the right of the query.

- For speed, we compute all the queries at once using matrix operations
- First we pack the queries, keys, values into matrices:
  - $Q = [q_1,\ldots,q_N]^T$
  - $K = [k_1,\ldots,k_N]^T$
  - $V = [v_1,\ldots,v_N]^T$
- Then we compute all the queries at once:

$$\text{Attn}(\mathbf{x}_{1:N}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}V\right)$$

LEARNING A TRANSFORMER LM

Learning a Transformer LM

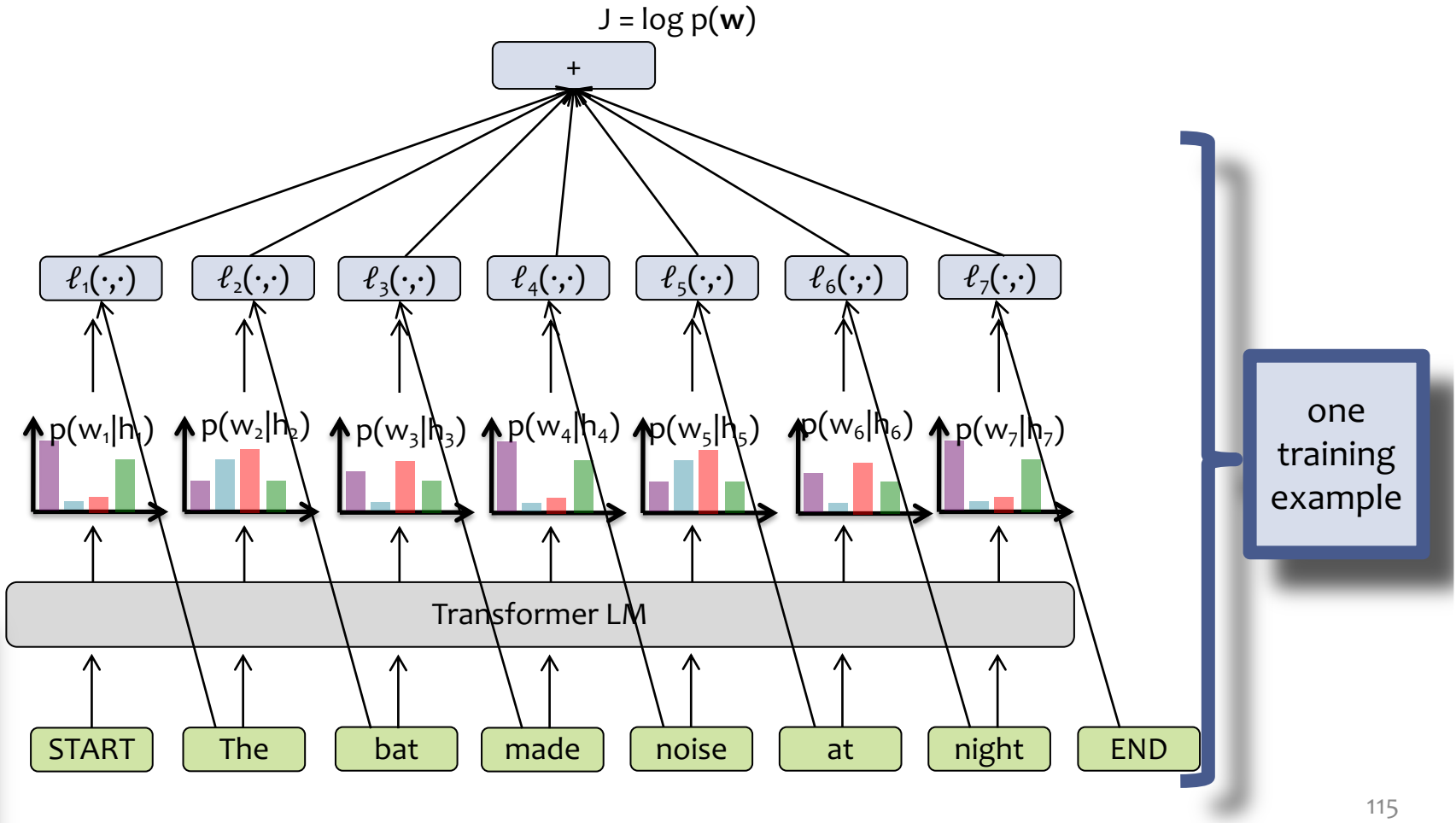
- Each training example is a sequence (e.g. sentence), so we have training data $D = \{\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots, \mathbf{w}^{(N)}\}$
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:

$$J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$$

- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

Training a Transformer-LM is the same, except we swap in a different deep language model.

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \ldots + \log p(w_2 \mid h_T)$$



Language Modeling

An aside:

- State-of-the-art language models currently tend to rely on **transformer networks** (e.g. GPT-3)
- RNN-LMs comprised most of the early neural LMs that **led to** current SOTA architectures

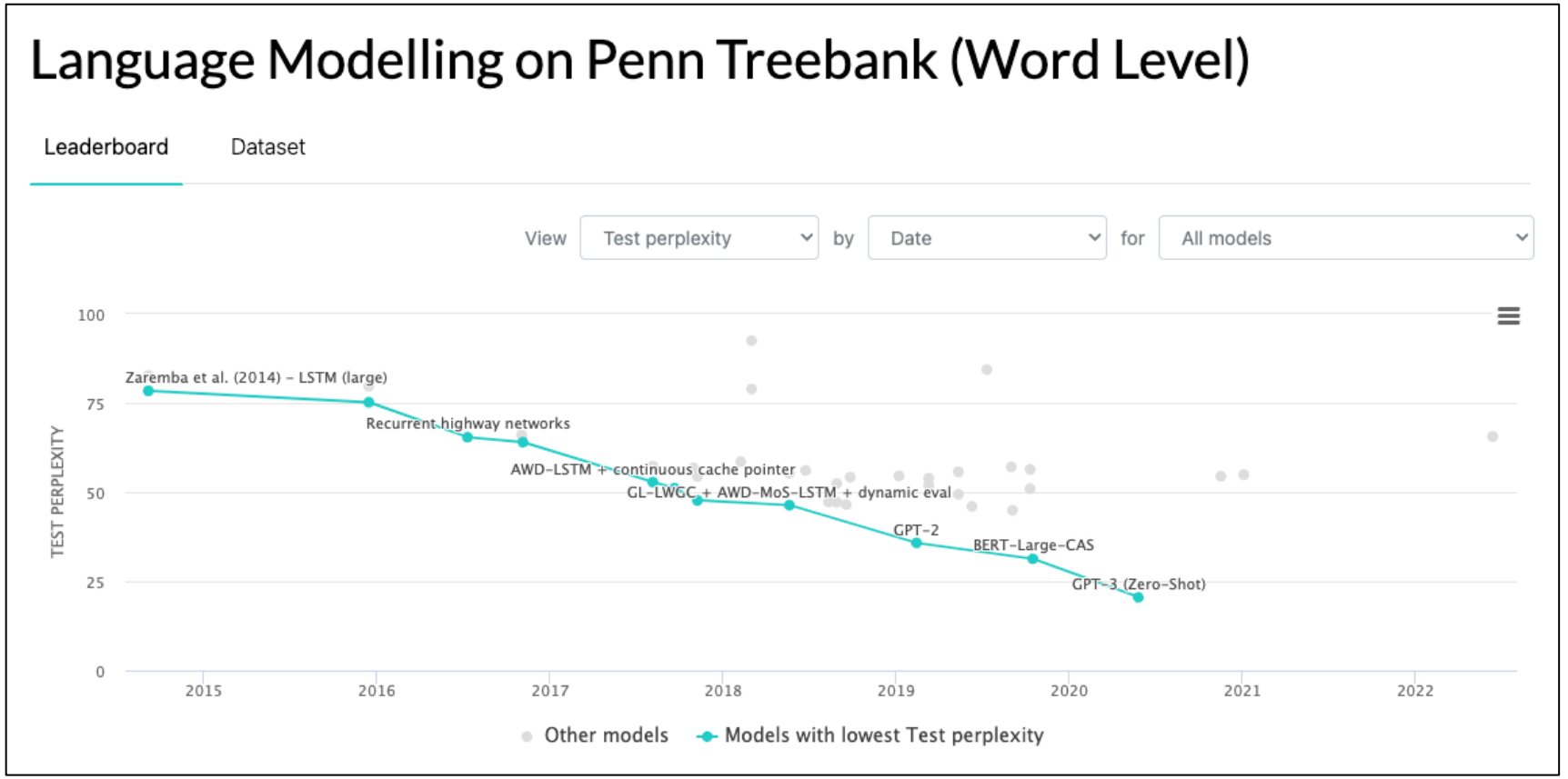


Figure from https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word

Why does efficiency matter?

Case Study: GPT-3

- # of training tokens = 500 billion
- # of parameters = 175 billion
- # of cycles = 50 petaflop/s-days (each of which are 8.64e+19 flops)

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2.2: Datasets used to train GPT-3. "Weight in training mix" refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

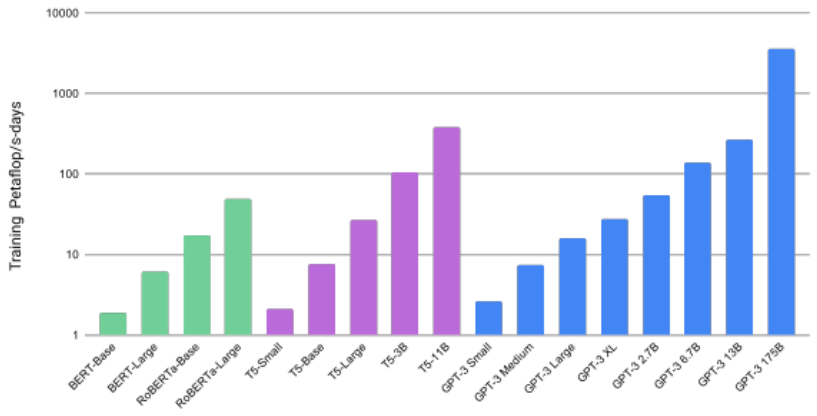


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

Figure from https://arxiv.org/pdf/2005.14165.

Two parts: Deep Learning and Language Modeling

Recap

Deep Learning

- AutoDiff
  - is a tool for **computing gradients** of a differentiable function, b = f(a)
  - the key building block is a **module** with a forward() and backward()
  - sometimes define f as **code** in forward() by chaining existing modules together
- Computation Graphs
  - are another way to define f (more conducive to slides)
  - we are considering various (deep) computation graphs: (1) CNN (2) RNN (3) RNN-LM (4) Transformer-LM
- Learning a Deep Network
  - deep networks (e.g. CNN/RNN) are trained by optimizing an objective function with SGD
  - compute gradients with AutoDiff

Language Modeling

- key idea: condition on previous words to **sample the next word**
- to define the **probability** of the next word…
  - …n-gram LM uses collection of massive 50k-sided **dice**
  - …RNN-LM or Transformer-LM use a **neural network**
- Learning an LM
  - n-gram LMs are easy to learn: just **count** co-occurrences!
  - a RNN-LM / Transformer-LM is trained just like other deep neural networks