## **پردازش زبان‌های طبیعی: انقلابی در Word Embedding با مفهوم Attention**

سلام! امیدوارم حالتون عالی باشه.

در جلسات گذشته درباره **پردازش زبان‌های طبیعی (NLP)** صحبت کردیم و وارد بحث **ورد امبدینگ (Word Embedding)** شدیم. توضیح دادیم که چطور می‌تونیم هر کلمه رو به یک بردار عددی تبدیل کنیم؛ طوری که کلماتی که از نظر معنایی به هم نزدیکن، بردارهای اون‌ها هم در فضای چندبعدی ما نزدیک به هم قرار بگیرن.

### **"Attention Is All You Need": مقاله‌ای که NLP را متحول کرد**

حالا می‌خوایم روی یکی از مهم‌ترین مقالات در تاریخ NLP تمرکز کنیم: مقاله‌ی **"Attention Is All You Need"**. این مقاله یک تحول عظیم در الگوریتم‌های اصلی پردازش زبان‌های طبیعی ایجاد کرد. در واقع، این مقاله سنگ بنای جهش‌های بزرگی شد که بعدها در مدل‌های زبان بزرگ (Large Language Models) و چت‌بات‌هایی مثل ChatGPT دیدیم. به همین خاطر، درک این مقاله و مفهوم "Attention" اهمیت فوق‌العاده‌ای داره.

### **مشکل Word Embedding سنتی: نادیده‌گرفتن مفهوم "توجه" (Attention)**

مفهوم اصلی این مقاله، **"Attention" یا "میزان توجه"** هست. برای اینکه بهتر متوجه بشید، یک مثال می‌زنم:

وقتی ما یک کلمه رو **امبد** می‌کنیم، یعنی بهش یک بردار عددی اختصاص می‌دیم، این بردار معمولاً مستقل از معنای اون کلمه در یک متن خاصه. مثلاً، کلمه‌ی "Light" در انگلیسی می‌تونه به معنای "نور" یا "سبک" باشه. در فارسی هم کلماتی مثل **"شیر"** معانی مختلفی دارن: شیر جنگل، شیر آب، یا شیر نوشیدنی.

در روش‌های سنتی ورد امبدینگ، کلمه‌ی "شیر" (یا "Light") یک بردار ثابت داره، فارغ از اینکه در چه متنی استفاده شده. این یعنی ما توانایی تشخیص معنای دقیق کلمه بر اساس **بافت (Context)** جمله رو ازش می‌گیریم.

**بیایید این جمله رو در نظر بگیریم:**

"بعد از یک روز طولانی در باغ وحش وقتی به خانه برگشتم، یک لیوان **شیر** خوردم و **شیر** آبی که چکه می‌کرد را بستم، اما صدای غرش **شیر** قفس بغلی هنوز در گوشم می‌پیچید."

در این جمله، سه بار کلمه‌ی "شیر" استفاده شده، و هر بار معنی متفاوتی داره. ذهن ما بلافاصله تفاوت این معانی رو درک می‌کنه:

* **شیر (نوشیدنی)**: ارتباط با نوشیدنی، غذا.
* **شیر (فلکه آب)**: ارتباط با لوله‌کشی، آب.
* **شیر (حیوان)**: ارتباط با باغ وحش، غرش، حیوانات.

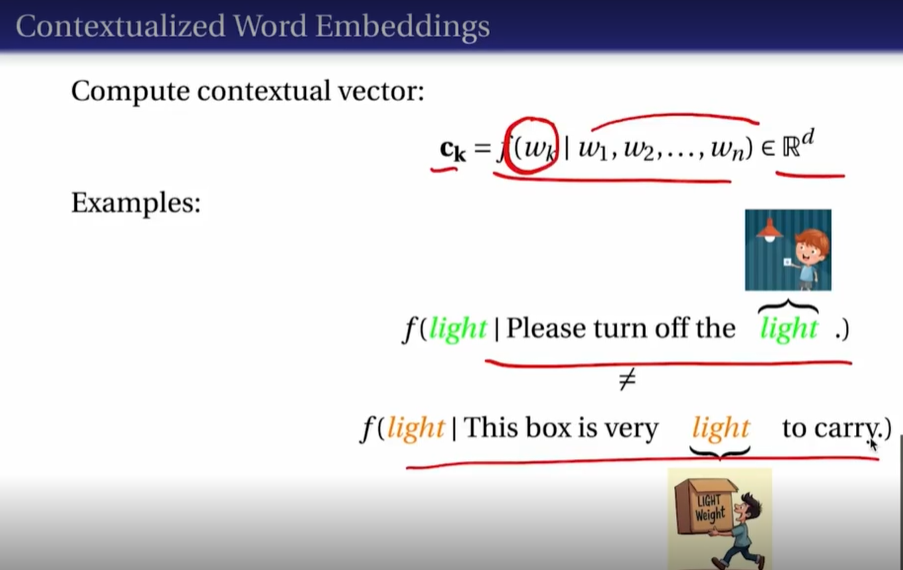
اگه از شما بپرسم **"میزان شباهت 'شیر آب' با 'زرافه' چقدره؟"**، احتمالاً میگید "هیچ شباهتی ندارن!". اما اگه بپرسم **"میزان شباهت 'شیر جنگل' با 'زرافه' چقدره؟"**، بهش امتیاز بالایی می‌دید، چون هر دو حیوانات وحشی هستن.

مشکل ورد امبدینگ‌های سنتی مثل Word2Vec اینه که به کلمه‌ی "شیر" یک بردار کلی می‌ده که همزمان هم با "آب" و "دوش حمام" شباهت داره، هم با "زرافه" و "جنگل"، و هم با "لبنیات" و "پنیر". در صورتی که وقتی ذهن ما به "شیر آب" فکر می‌کنه، دیگه شباهت‌های اون با "شیر جنگل" یا "شیر لبنیات" رو در نظر نمی‌گیره.

### **راه‌حل: Context-Dependent Embedding**

دقیقاً همین نکته بود که مقاله Attention روش تمرکز کرد: ما به یک **امبدینگ وابسته به بافت (Context-Dependent Embedding)** برای هر کلمه نیاز داریم. یعنی برداری از یک کلمه که به **نحوه‌ی استفاده خاص اون کلمه در یک متن خاص** وابسته باشه.

برای مثال، در جمله‌ی "Please turn off the **light**" (لطفاً چراغ را خاموش کن)، کلمه‌ی "light" باید یک امبدینگ متفاوت با "The box was very **light**" (جعبه خیلی سبک بود) داشته باشه.



هدف اینه که وقتی یک دنباله از کلمات (یک جمله) داریم، امبدینگ هر کلمه درون اون جمله، بر اساس کل اون جمله تغییر کنه و مستقل از اون جمله نباشه.

## **بازگشت به گذشته: درک شبکه‌های عصبی بازگشتی (RNNs) پیش از Attention**

همونطور که گفتیم، مقاله **"Attention Is All You Need"** (که اگر اشتباه نکنم در **سال 2017** منتشر شد) یک تحول بزرگ در پردازش زبان طبیعی بود. اما برای درک بهتر این تحول، باید کمی به گذشته برگردیم و با مفهوم **شبکه‌های عصبی بازگشتی (Recurrent Neural Networks - RNNs)** آشنا بشیم.

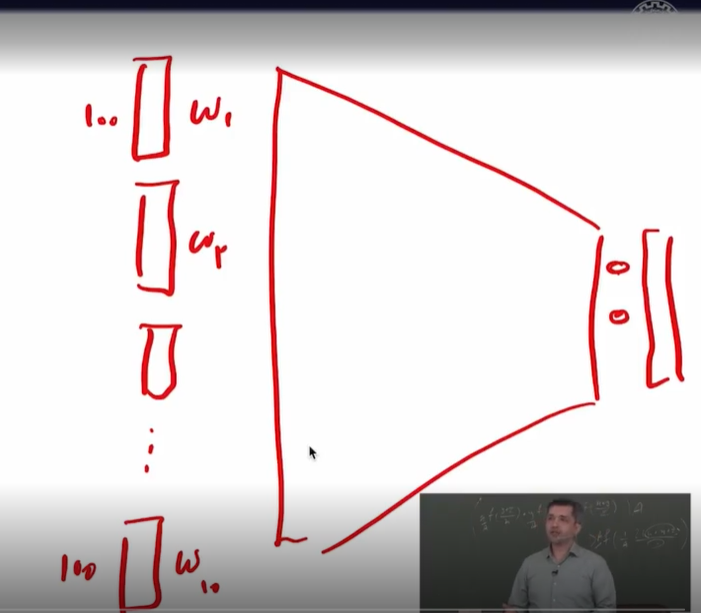
این فصل جدید درسی، به لحاظ تاریخی شما رو به چند دهه قبل، یعنی دوران آغاز به‌کارگیری شبکه‌های عصبی برای پردازش زبان‌های طبیعی، برمی‌گردونه. **RNNها یکی از نخستین معماری‌های شبکه‌های عصبی بودن که برای پردازش توالی‌ها مثل جملات، متن، صوت و هر چیزی که ماهیت "دنباله‌ای" داره، به کار رفتن.**

این فلش‌بک به گذشته رو زدیم تا پرانتزی باز کنیم. چرا؟ چون معماری RNN یا این نوع خاص از شبکه‌های عصبی (که بعداً به **LSTM و GRU** تبدیل شدن)، بعدها خیلی به درد ما خورد تا معماری‌های جدید رو طراحی کنیم و نهایتاً به **"Attention Is All You Need" و ساختار "ترنسفورمر" (Transformer)** برسیم.

### **چالش اصلی در پردازش توالی‌ها: طول متغیر جملات**

خب، **RNNها اولین نوع شبکه‌هایی بودن که توالی‌ها رو برای ما پردازش می‌کردن.** حالا وقتی داریم در مورد یک جمله صحبت می‌کنیم، اولین چالش ما **طول متغیر جمله** هست. یک جمله می‌تونه خیلی کوتاه باشه (مثلاً فقط یک کلمه مثل "سلام") یا می‌تونه به اندازه‌ی یک پاراگراف کامل بلند باشه.

فرض کنید همیشه با عبارات تک‌کلمه‌ای سروکار داشتیم. در این صورت، می‌گفتیم که هر کلمه رو مثلاً در یک فضای صدبُعدی با Word2Vec امبد کردیم. بعد، ۱۰ تا از این بردارهای صدبُعدی رو می‌گذاشتیم (که هر کدوم ۱۰۰ تا نورون رو تشکیل می‌دادن)، و یک معماری شبکه عصبی رو می‌گذاشتیم که نهایتاً مثلاً به ما بگه آیا این جمله بار معنایی مثبت داره یا منفی، یا حتی اون رو ترجمه کنه. (البته برای ترجمه نیاز به خروجی بزرگتری در لایه نهایی داشتیم.)



اما چالش اصلی اینه که **طول جملات مشخص نیست.** این کوتاه و بلندی رو چطور مدیریت کنیم؟

یکی از راه‌حل‌های اولیه این بود که **طولانی‌ترین جمله‌ی موجود در دیتاست رو پیدا کنیم** (مثلاً ۲۰۰ کلمه). بعد، **سایز ورودی شبکه رو به صورت ثابت و فیکس مثلاً ۲۰۰ کلمه در نظر بگیریم.** اگه جمله‌ای ۵ کلمه داشت، ۱۹۵ کلمه‌ی باقی‌مانده رو با **"پدینگ" (Padding)** پر می‌کردیم؛ یعنی ۱۹۵ تا کلمه‌ی "صفر" اضافه می‌کردیم. معمولاً بردار صفر رو برای کلماتی رزرو می‌کردن که نشانه‌ی "پدینگ" بودن و به شبکه می‌فهموندن که جمله تموم شده.

### **مشکلات روش پدینگ و راه‌حل‌های اولیه**

اما می‌تونید حدس بزنید که این ساختار چقدر بده! چرا؟

1. **نادیده‌گرفتن موقعیت کلمه:** فرض کنید کلمه‌ی "زرافه" رو داریم. "زرافه" کلاً یک مفهوم رو منتقل می‌کنه، فرقی نمی‌کنه که در کجای جمله بیاد. اما اگر یک معماری با پدینگ بسازیم، شبکه باید یاد بگیره که "زرافه" چه در ابتدای جمله، چه وسط یا انتهای جمله، باز هم همون "زرافه" است. این شبیه پردازش تصویره که می‌گفتیم اگه تصویر یک توپ رو بالا سمت چپ، بالا سمت راست، یا هر جای دیگه‌ای از تصویر بذاریم، شبکه باید باز هم بگه "توپه". شبکه باید اینقدر کلمه‌ی "زرافه" رو در موقعیت‌های مختلف ببینه تا بفهمه چطور باهاش رفتار کنه. این کار رو خیلی سخت می‌کنه و حجم دیتایی که برای چنین معماری‌ای نیاز داریم، خیلی زیاد میشه.
2. **اتلاف منابع:** استفاده از پدینگ به معنای پردازش مقادیر بی‌معنی و صفر است که منابع محاسباتی را هدر می‌دهد.

راه‌حل جایگزین و بهتر:

می‌تونیم از کانولوشن یک‌بعدی (1D Convolution) استفاده کنیم. کانولوشن‌ها (که در پردازش تصویر بسیار پرکاربرد هستن) ویژگی‌های مکان‌ناور (translation-invariant) دارن. یعنی اگر شیئی در تصویر جابه‌جا بشه، باز هم شبکه قادر به تشخیص اون هست. همین مفهوم رو میشه برای متن و توالی کلمات هم به کار برد.

**کانولوشن یک‌بعدی** می‌تونه موقعیت کلمه رو در جمله نادیده بگیره و صرفاً الگوها و روابط بین کلمات نزدیک رو یاد بگیره. این باعث میشه "زرافه" هر جای جمله که بیاد، همون مفهوم زرافه رو درک کنه. این معماری همونطور که اشاره شد، واقعاً برای پردازش متن و توالی‌ها استفاده میشه و یک مزیت بزرگ داره: **"مکان‌ناوری" (Position Invariance)**.

### **تکامل RNNها: LSTM و GRU**

اما معماری دیگری که برای چند دهه (یا اکستنشن‌هایش) استفاده شد و پایه‌ی بسیاری از مدل‌های NLP بود، همین RNNها بودن. این مدل‌ها به مرور زمان توسعه پیدا کردن و اکستنشن‌های مهمی از اون‌ها مثل **LSTM (Long Short-Term Memory)** و **GRU (Gated Recurrent Unit)** ساخته شدن که مشکلات حافظه‌ی کوتاه‌مدت RNNهای ساده رو حل کردن. این معماری‌ها برای چند دهه بیس پردازش زبان طبیعی با استفاده از شبکه‌های عصبی بودن.

## **ساختار پایه شبکه‌های عصبی بازگشتی (RNNs): مفهوم "حافظه"**

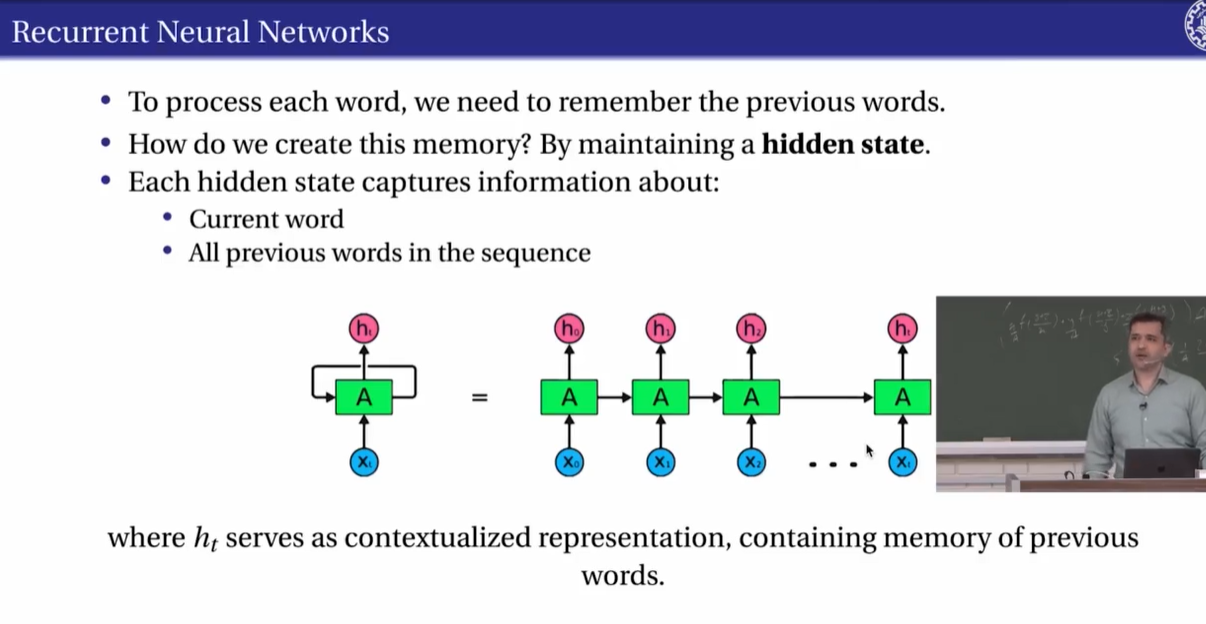
خب، **شبکه‌های عصبی بازگشتی (RNN)** ساختار بسیار ساده‌ای دارن که اساس کارشون بر پایه‌ی مفهوم **"حافظه" (Memory)** هست. ایده‌ی اصلی اینه که شبکه می‌خواد از اطلاعات قبلی که پردازش کرده، برای پردازش اطلاعات فعلی استفاده کنه.

### **نحوه‌ی عملکرد RNN: پردازش کلمه به کلمه**

وقتی شما یک جمله رو به یک RNN می‌دید، کلمه به کلمه بهش داده میشه:

1. ابتدا **کلمه‌ی اول (X0)** به شبکه داده میشه.
2. سپس **کلمه‌ی دوم (X1)** و به همین ترتیب تا **کلمه‌ی i‌اُم (Xi)**.

نکته‌ی مهم اینجاست که وقتی به کلمه‌ی i‌اُم می‌رسید، می‌خواید شبکه یک حافظه از **(i-1) کلمه‌ی قبلی** داشته باشه. سپس، بر اساس این حافظه و کلمه‌ی فعلی، یک خروجی (مثلاً ترجمه‌ی کلمه) تولید کنه.



### 

### **چالش ترجمه‌ی جمله با RNN ساده: ترتیب کلمات**

البته، چنین معماری‌ای برای **ترجمه‌ی کامل یک جمله** خیلی ایده‌آل نیست. چرا؟

چون وقتی شما اولین کلمه رو به شبکه می‌دید، شبکه ازش انتظار خروجی داره. در صورتی که برای ترجمه‌ی یک جمله، ممکنه لازم باشه چندین کلمه رو اول به شبکه بدید تا تازه بتونه اولین کلمه‌ی ترجمه شده رو تولید کنه.

دلیل این موضوع اینه که **ساختار زبان‌ها با هم فرق می‌کنه:**

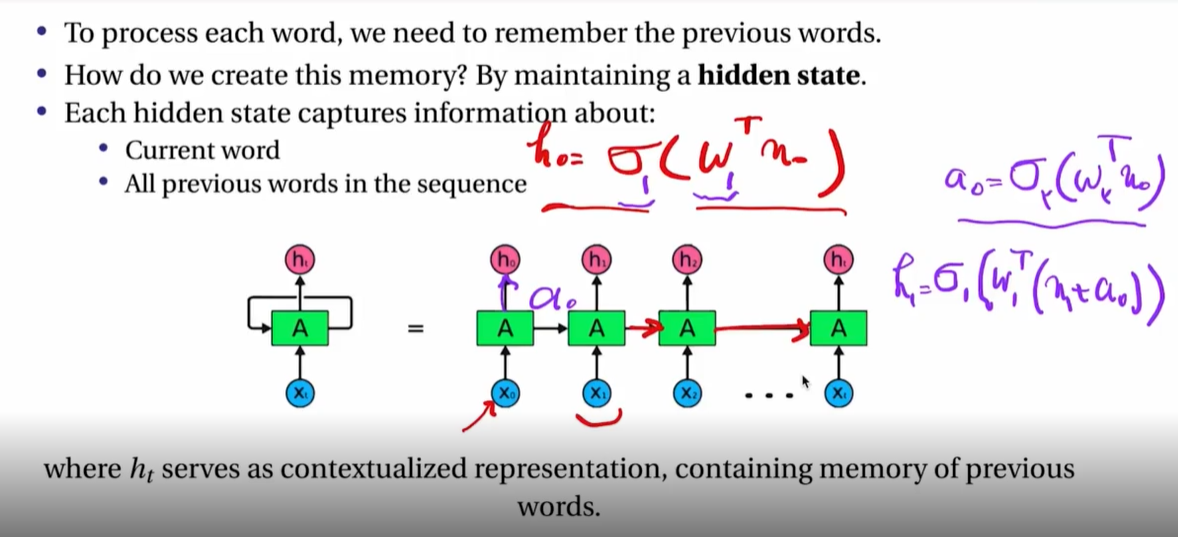
* در یک زبان، صفت ممکنه قبل از موصوف بیاد، در حالی که در زبان دیگر بعد از موصوف. پس، تا موصوف رو نبینی، نمی‌تونی صفت رو ترجمه کنی.
* همینطور، ترتیب فاعل، فعل و مفعول هم در زبان‌های مختلف متفاوته.

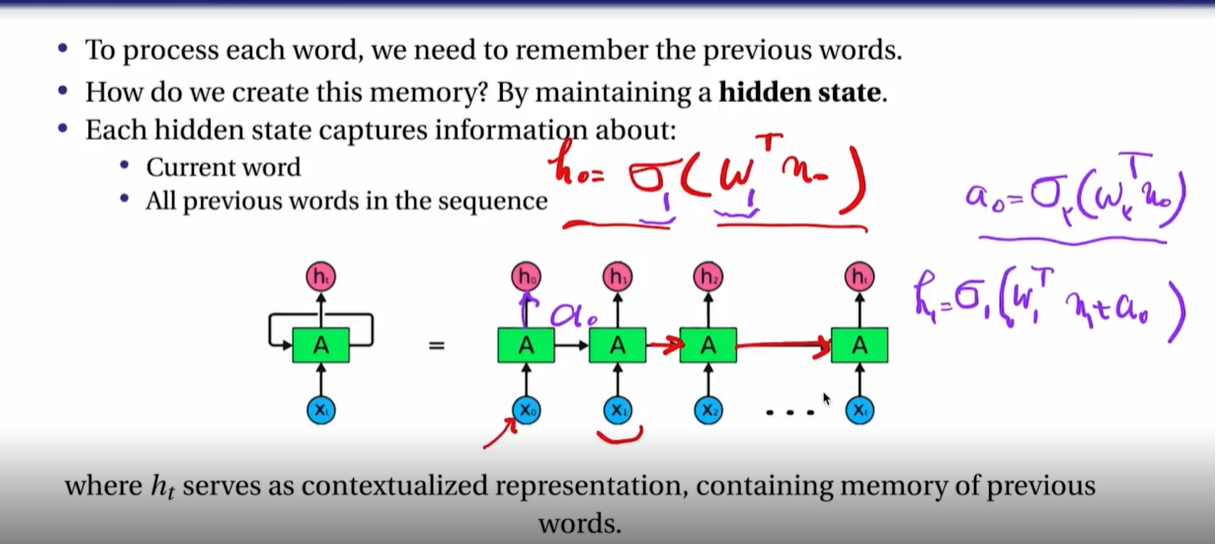
بنابراین، خیلی وقت‌ها برای ترجمه، **باید ترتیب کلمات رو "دوباره مرتب" (Reorder) کنیم.** معماری ساده‌ی RNN که بلافاصله پس از هر کلمه خروجی میده، برای این کار مناسب نیست.

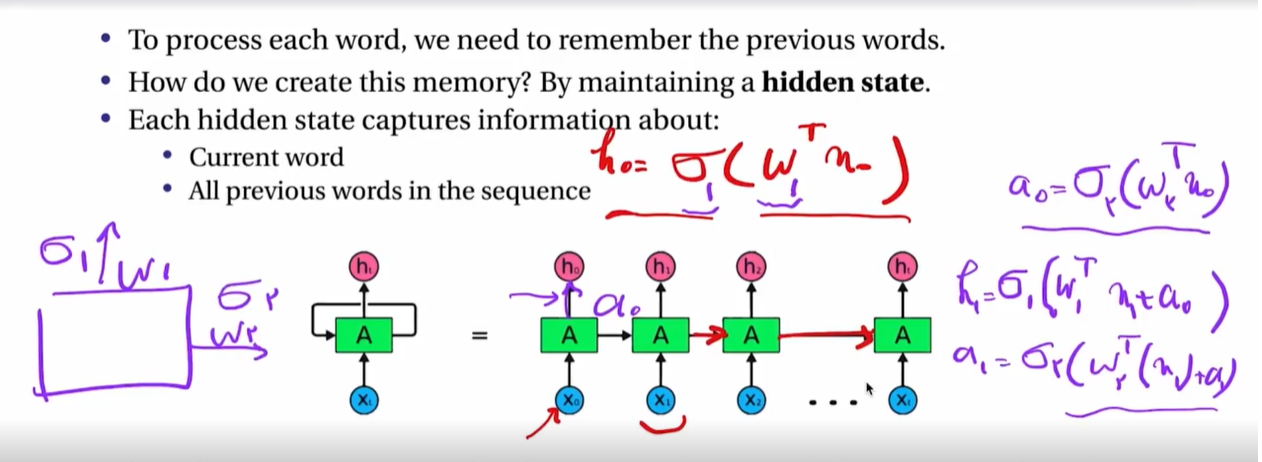
### **مکانیزم "حافظه" در RNN**

حالا فرض کنید می‌خوایم این معماری رو برای ترجمه استفاده کنیم (با فرض اینکه مشکلات ترتیب کلمات رو فعلاً نادیده می‌گیریم).

1. **ورودی اولیه (X0):**
   * ما **X0** (اولین کلمه به صورت بردار امبدینگ) رو به شبکه می‌دیم.
   * این ورودی در یک **ماتریس وزن (Weight Matrix)** ضرب میشه و از یک **تابع فعال‌سازی (Activation Function)** (مثل سیگموئید یا ReLU) عبور می‌کنه. اسم این تابع فعال‌سازی رو **σ** می‌ذاریم.
   * نتیجه، حالت پنهان اولیه (Hidden State - H0) رو می‌سازه:  
     H0​=σ(W1T​X0​)  
     (اینجا W1​ ماتریس وزن مربوط به خروجی و σ تابع فعال‌سازی است.)
2. **ایجاد "حافظه" (State) برای مرحله بعدی:**
   * علاوه بر محاسبه H0​، ما باید یک **"حافظه"** یا **"حالت" (State)** رو هم برای مرحله‌ی بعدی محاسبه کنیم. این حالت رو مثلاً **A0​** می‌نامیم.
   * A0​ هم بر اساس X0​ محاسبه میشه، اما با یک ماتریس وزن جداگانه (مثلاً W2​):  
     A0​=σ(W2T​X0​)  
     این A0​ همان فلش افقی است که در دیاگرام‌های RNN دیده می‌شود و اطلاعات را به مرحله‌ی بعدی منتقل می‌کند.
3. **پردازش کلمه بعدی (X1) با استفاده از حافظه:**
   * حالا برای محاسبه H1​ (خروجی کلمه‌ی اول)، ما از **X1** و همچنین **حافظه‌ی قبلی (A0​)** استفاده می‌کنیم.
   * یکی از روش‌های ساده اینه که X1​ و A0​ رو با هم جمع کنیم و بعد در ماتریس وزن W1​ ضرب کنیم:  
     H1​=σ(W1T​(X1​+A0​))
   * برای محاسبه A1​ (حافظه برای مرحله‌ی بعدی) هم به همین ترتیب عمل می‌کنیم:  
     A1​=σ(W2T​(X1​+A0​))







### **نکته‌ی مهم درباره‌ی وزن‌ها در RNN**

* **وزن‌های مشترک:** در یک RNN، **ماتریس‌های وزن (W1​ و W2​) در تمام مراحل زمانی (برای تمام کلمات جمله) یکسان هستن.** یعنی W1​ و W2​ برای کلمه‌ی اول، کلمه‌ی دهم، و هر کلمه‌ی دیگری در جمله، همان W1​ و W2​ اولیه هستن.
* **تعداد وزن‌ها:** در کل، ما فقط دو مجموعه بردار وزن داریم:
  + یک مجموعه (W1​) برای محاسبه خروجی.
  + یک مجموعه (W2​) برای محاسبه اون بردار "حافظه" یا "حالت" که قراره به کلمات بعدی پاس داده بشه.
* **آپدیت وزن‌ها:** هنگام **پس‌انتشار (Backpropagation)**، این مقادیر وزن (W1​ و W2​) بر اساس **کل کلمات جمله** و گرادیان‌های مربوط به آن‌ها آپدیت میشن. یعنی لاسم (Loss) کلی رو نسبت به W1​ و W2​ محاسبه می‌کنیم (مثلاً میانگین یا مجموع گرادیان‌ها روی تمام کلمات جمله) و سپس در یک گام آموزش، وزن‌ها رو به‌روز می‌کنیم.

این ساختار شباهت زیادی به شبکه‌های عصبی کانولوشنی (CNN) داره، که در اون‌ها هم یک "کرنل" (Kernel) یا مجموعه‌ای از وزن‌ها در تمام نقاط تصویر تکرار (کانوالو) می‌شد تا ویژگی‌ها رو استخراج کنه. در RNN هم همینطوره، یک مجموعه‌ی ثابت از وزن‌ها روی تمام کلمات جمله اعمال میشه.

### **تفاوت‌های "حالت پنهان" و "خروجی"**

در مثالی که زدیم، ما دو نوع خروجی داشتیم: یکی **خروجی نهایی (Ht​)** که مثلاً ترجمه یک کلمه بود، و یکی **حافظه داخلی (At​)** که به مرحله بعدی پاس داده می‌شد.

در ساده‌ترین شکل RNN، گاهی اوقات **حالت پنهان (ht​) و خروجی (yt​) یکسان در نظر گرفته میشن، یا اینکه خروجی به سادگی یک تبدیل خطی از حالت پنهان است.** یعنی انگار:

* yt​=ht​ (در ساده‌ترین حالت)
* یا yt​=Why​ht​ (اگر تنها یک تبدیل خطی باشد)

این بدان معناست که **همان برداری که به عنوان "حافظه" به مرحله بعدی منتقل می‌شود (ht​)، به عنوان خروجی نیز در نظر گرفته می‌شود.**

اما همونطور که اشاره کردم، این رویکرد برای تمام کاربردها مناسب نیست. مثلاً در ترجمه:

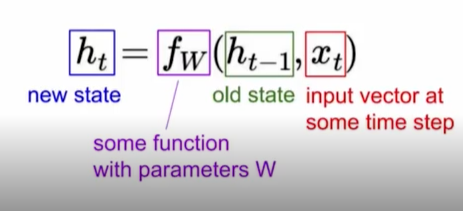
* شما می‌خواهید کلمه "lion" (ورودی) را به "شیر" (خروجی) ترجمه کنید.
* اما این "شیر" که خروجی است، لزوماً همون "حافظه"ای نیست که شامل تمام اطلاعات بافتی از کلمات قبلیه. "حافظه" باید شامل اطلاعات عمیق‌تری باشه که به شبکه کمک می‌کنه کلمه‌ی بعدی رو درست پردازش کنه، نه صرفاً ترجمه‌ی کلمه‌ی فعلی.

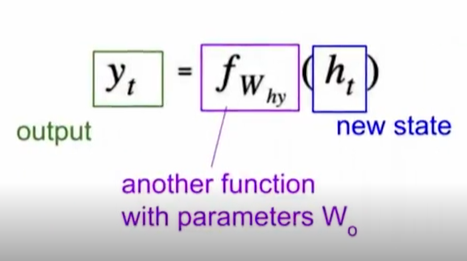
**به بیان دیگر:**

* **حالت پنهان (ht​)**: این همون "استیت" (State) هست که بین مراحل زمانی (بین کلمات) پاس داده میشه و اطلاعات بافتی رو در خودش نگه می‌داره.
* **خروجی (yt​)**: این نتیجه‌ای هست که شبکه در هر مرحله تولید می‌کنه (مثلاً ترجمه‌ی یک کلمه، یا پیش‌بینی کلمه‌ی بعدی).

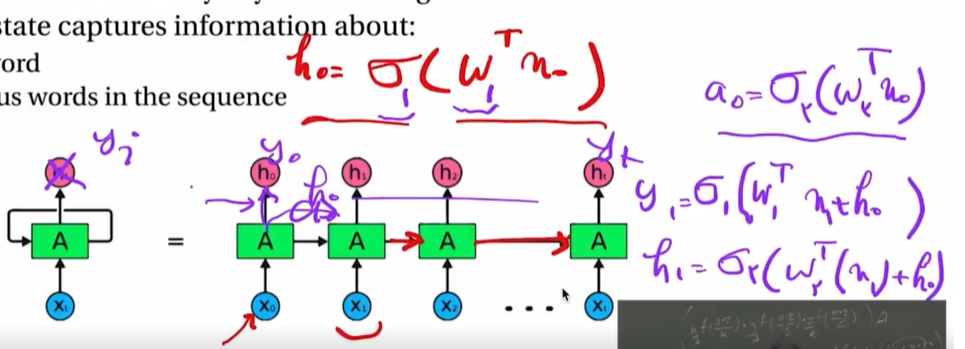
پس، حالت پنهان جدید (ht​) یک تابعی از حالت پنهان قبلی (ht−1​) و ورودی جدید (xt​) است. خروجی (yt​) هم یک تابع دیگری از این حالت پنهان فعلی (ht​) است.

در نهایت، با این مدل ما در واقع "استیت"ها (h0​,h1​,…,hT​) رو پاس می‌دیم و خروجی‌ها (y0​,y1​,…,yT​) رو محاسبه می‌کنیم. این نوتیشن به شما کمک می‌کنه ارتباط بین حالت‌های پنهان (حافظه داخلی) و خروجی‌های قابل مشاهده رو درک کنید.

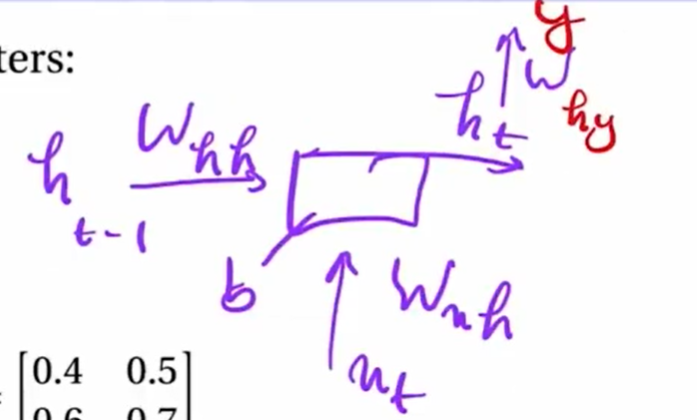




با نوتیشن جدید اصلاح میکنم:







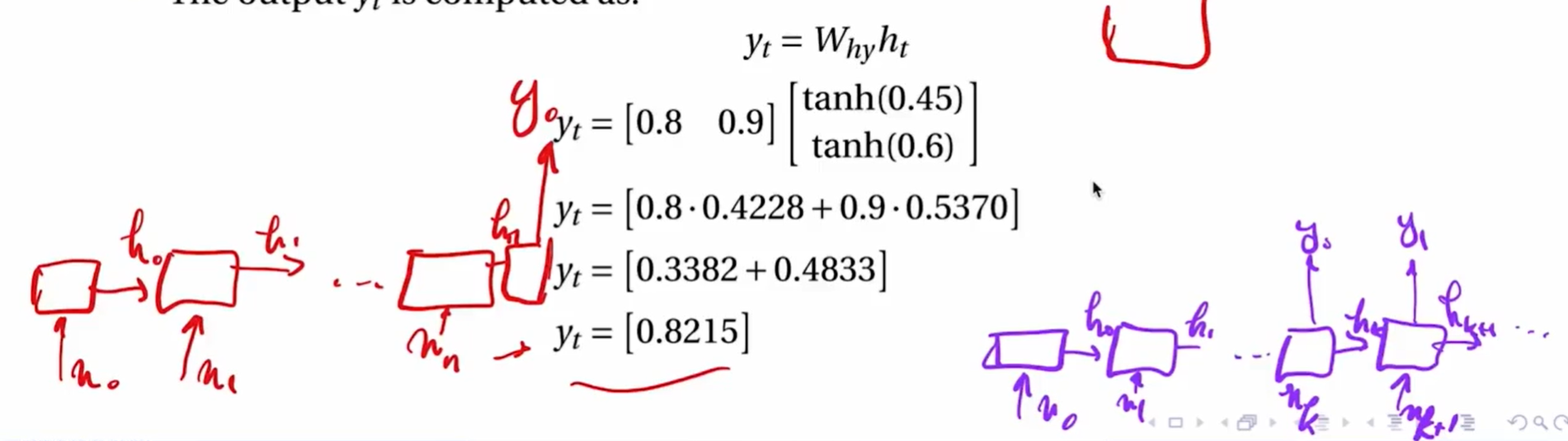
## **کاربردهای مختلف RNN و چالش‌های آن**

همانطور که تا اینجا متوجه شدید، ساختار پایه **شبکه‌های عصبی بازگشتی (RNN)** حول محور مفهوم "حافظه" کار می‌کند. حالا بیایید ببینیم این ساختار چگونه برای کاربردهای مختلف تنظیم می‌شود و چه چالش‌هایی دارد.

### **سناریوهای مختلف خروجی در RNN**

شکل خروجی در RNN می‌تواند بسته به نوع مسئله متفاوت باشد:

1. یک خروجی نهایی برای کل توالی (Many-to-One):  
   در این حالت، شبکه در هر مرحله، ورودی (کلمه) را دریافت کرده و حالت پنهان خود را به‌روز می‌کند، اما تا زمانی که تمام کلمات توالی (جمله) به پایان نرسیده، هیچ خروجی‌ای تولید نمی‌کند. تنها پس از پردازش آخرین کلمه، یک تک خروجی نهایی تولید می‌شود. \_ معماری سمت چپ در تصویر زیر
   * **مثال:** فرض کنید هدف این است که شبکه بگوید کل این جمله بار معنایی **مثبت** یا **منفی** دارد. یا اینکه **احساسات** نهفته در جمله (خشم، هیجان) چیست، یا آیا جمله **توهین‌آمیز** است یا خیر. در این سناریوها، RNN تمام جمله را پردازش می‌کند تا به یک نتیجه‌گیری کلی برسد.
2. خروجی با تأخیر (Encoder-Decoder with Delay):  
   در این معماری، شبکه ابتدا چندین کلمه اولیه (مثلاً K کلمه) را بدون تولید خروجی پردازش می‌کند. سپس، از کلمه‌ی $(K+1)$اُم به بعد، شروع به تولید خروجی برای هر کلمه می‌کند. - معماری سمت راست در تصویر زیر
   * **مثال:** این نوع معماری برای **ترجمه** بسیار کاربردی است. همانطور که پیش‌تر گفتیم، ساختار زبان‌ها متفاوت است و برای ترجمه‌ی دقیق، ممکن است لازم باشد شبکه تا یک نقطه‌ی مشخصی از جمله را ببیند (یعنی K کلمه را دریافت کند) تا بتواند ترجمه‌ی اولین کلمه را شروع کند. این تأخیر به شبکه اجازه می‌دهد تا اطلاعات بافتی کافی را برای ترجمه صحیح کلمات بعدی جمع‌آوری کند.



### **نقاط ضعف اصلی RNNها**

با وجود نوآوری‌هایشان، RNNهای پایه چندین نقطه ضعف اساسی دارند که باعث محدودیت عملکرد آن‌ها در مسائل پیچیده‌تر می‌شود:

1. وابستگی‌های بلندمدت (Long-Term Dependencies):  
   این بزرگترین مشکل RNNهاست. اگر دو کلمه که از لحاظ معنایی به هم مرتبط هستند، فاصله‌ی زیادی در جمله داشته باشند، RNN در به خاطر سپردن ارتباط بین آن‌ها با مشکل مواجه می‌شود.
   * **دلیل:** در هر مرحله، ما فقط یک "حالت پنهان" (state) را به مرحله‌ی بعدی پاس می‌دهیم. این حالت پنهان پیوسته در یک ماتریس وزن ضرب شده و از توابع فعال‌سازی (مثل تانژانت هایپربولیک) عبور می‌کند. توابع فعال‌سازی اغلب مقادیر را بین -1 و 1 فشرده می‌کنند (یا بین 0 و 1 در مورد سیگموئید). این عملیات‌های ضرب مکرر و فشرده‌سازی باعث می‌شود که **اثر اطلاعات اولیه به تدریج "محو" (Vanishing) شود.**
   * **مثال:** در جمله‌ای مثل: "The **jacket** was red. It was made of leather."، کلمه‌ی "It" به "jacket" اشاره دارد. اگر بین "jacket" و "It" کلمات زیادی وجود داشته باشد، اطلاعات مربوط به "jacket" در حالت پنهان تدریجاً کمرنگ شده و در نهایت RNN نمی‌تواند به درستی بفهمد که "It" به چه چیزی اشاره می‌کند. این مشکل به **"مشکل گرادیان محوشونده" (Vanishing Gradient Problem)** معروف است.
2. مشکل گرادیان محوشونده/انفجاری (Vanishing/Exploding Gradients):  
   این مشکل مستقیماً با فرآیند آموزش شبکه‌های عصبی (پس‌انتشار گرادیان) مرتبط است:
   * **گرادیان محوشونده (Vanishing Gradient):** همانطور که در بالا توضیح داده شد، گرادیان‌ها در طول زمان به سمت صفر میل می‌کنند و اطلاعات از دست می‌روند. این باعث می‌شود وزن‌های اولیه‌ی شبکه به سختی به‌روز شوند.
   * **گرادیان انفجاری (Exploding Gradient):** در مقابل، اگر وزن‌ها یا مقادیر فعال‌سازی بزرگ باشند، گرادیان‌ها می‌توانند به سرعت رشد کرده و بسیار بزرگ شوند. این باعث می‌شود مدل ناپایدار شده و آموزش آن غیرممکن شود.
3. پردازش ترتیبی و کندی (Sequential Processing & Slow Speed):  
   معماری RNN ذاتاً ترتیبی (Sequential) است. هر کلمه‌ی فعلی برای پردازش نیاز به حالت پنهان کلمه‌ی قبلی دارد. این بدان معناست که نمی‌توانید پردازش را به صورت موازی (Parallelization) انجام دهید.
   * **مثال:** اگر بخواهید یک دایرةالمعارف بزرگ را با RNN پردازش کنید، باید کلمه به کلمه پیش بروید. این فرآیند به شدت زمان‌بر است، زیرا هر گام محاسباتی به گام قبلی وابسته است. این عدم قابلیت موازی‌سازی، یکی از بزرگترین موانع برای آموزش مدل‌های RNN بر روی دیتاست‌های عظیم بود.

**در نتیجه:**

تمام این مشکلات و محدودیت‌ها در **RNNهای پایه و حتی مشتقات پیشرفته‌تر آن‌ها مانند LSTM (Long Short-Term Memory) و GRU (Gated Recurrent Unit)** وجود داشتند. این‌ها مقدمه‌ای بود تا متوجه شویم که تاریخچه‌ی پردازش زبان‌های طبیعی برای چندین دهه، تا قبل از ظهور ساختار **Attention** و **ترنسفورمرها**، تحت سلطه‌ی RNN و مشتقات آن بود.

## **انقلاب Attention و ترنسفورمرها: راه‌حل مشکلات RNN**

همانطور که دیدیم، شبکه‌های عصبی بازگشتی (RNN) با وجود نوآوری‌هایشان، دارای نقاط ضعف اساسی مانند **عدم قابلیت موازی‌سازی، فراموشی اطلاعات بلندمدت، و مشکلات گرادیان محوشونده/انفجاری** بودند.

مقاله **"Attention Is All You Need"** آمد و کل جامعه علمی را به لرزه درآورد، چرا که پاسخی هوشمندانه به تمام این چالش‌ها ارائه داد. این مقاله، **مکانیزم Attention** و ساختار **ترنسفورمر (Transformer)** را معرفی کرد که بر پایه **Self-Attention** بنا شده است.

### **مزایای کلیدی Attention و Transformer بر RNN**

مکانیزم **Self-Attention** که در معماری ترنسفورمر به کار رفته، می‌خواست تمام مشکلات RNN را یکجا حل کند:

1. قابلیت موازی‌سازی کامل (Full Parallelization):  
   ساختار ترنسفورمر کاملاً موازی (Parallel) است. یعنی شما می‌توانید کل جمله را یکجا به شبکه بدهید و همه کلمات را به صورت همزمان پردازش کند. این قابلیت، امکان استفاده بهینه از GPUها (واحد پردازش گرافیکی) را برای پردازش زبان طبیعی فراهم می‌کند که سرعت آموزش و پردازش مدل‌ها را به طرز چشمگیری افزایش می‌دهد. این یکی از بزرگترین مزیت‌های ترنسفورمرها بود.
2. حل مشکل وابستگی‌های بلندمدت:  
   در ترنسفورمرها، دوری و نزدیکی کلمات دیگر باعث فراموشی ارتباط بین آن‌ها نمی‌شود. شبکه می‌تواند به طور مستقیم به هر کلمه‌ای در توالی نگاه کند و وابستگی‌های دوردست را هم به خوبی درک کند.
3. امبدینگ وابسته به بافت (Context-Dependent Embedding):  
   این ساختار دقیقاً به همان چالشی پاسخ می‌دهد که در ابتدای جلسه به آن اشاره کردیم: هر کلمه، متناسب با بافت (Context) جمله‌ای که در آن قرار دارد، یک بردار امبدینگ منحصربه‌فرد دریافت می‌کند.

### **ایده اصلی پشت Attention**

ایده اصلی **Attention** شبیه به عملکرد مغز انسان است. وقتی شما در حال پردازش یک جمله هستید و به یک کلمه‌ی خاص می‌رسید، مغز شما به صورت خودکار به کلمات دیگری از همان پاراگراف یا جمله که برای درک بهتر کلمه‌ی فعلی اهمیت دارند، **توجه (Attention)** بیشتری می‌کند.

**مثال: کلمه "شیر" در جمله "در جنگل شیری را دیدم."**

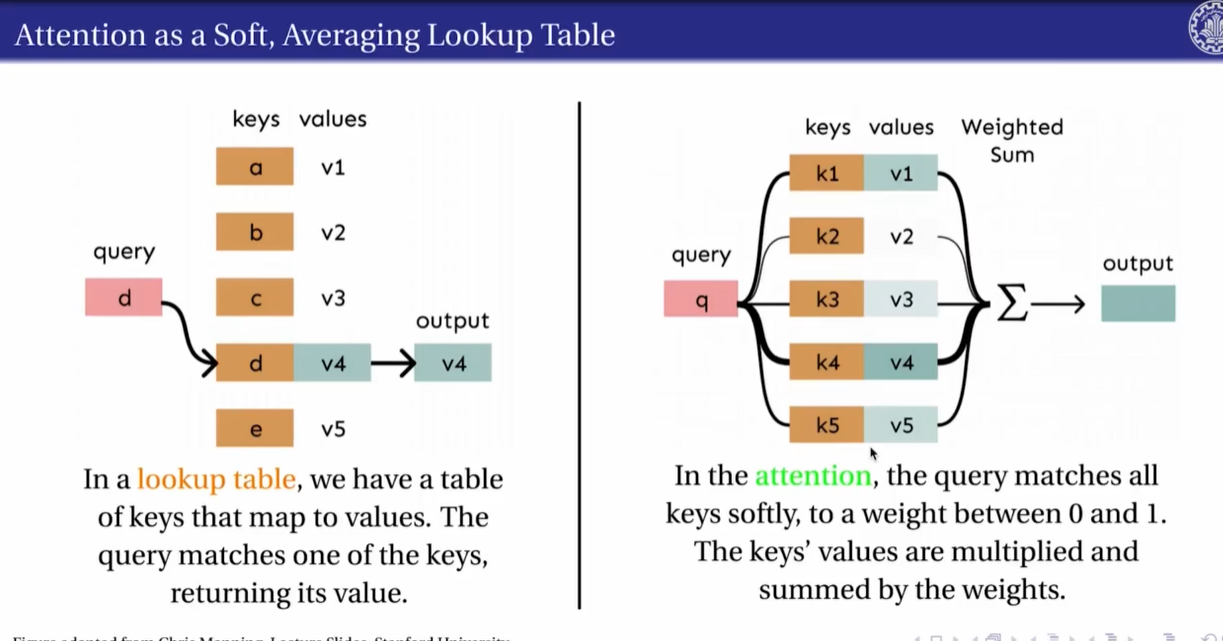
وقتی به کلمه‌ی **"شیر"** می‌رسید و در ادامه‌ی جمله کلمه‌ی **"جنگل"** را می‌بینید (یا حتی اگر کلمه‌ی "جنگل" کیلومترها جلوتر یا قبل‌تر آمده باشد و شما مشغول تعریف خاطرات آن روزتان باشید)، مغز شما به طور خودکار وزن بیشتری به کلمه‌ی **"جنگل"** می‌دهد تا معنای **"شیر (حیوان)"** را درک کند.

* اگر به جای "جنگل"، عبارت "یخچال" یا "لیوان" آمده بود، مغز شما به صورت خودکار به آن کلمات وزن بیشتری می‌داد تا معنای "شیر (لبنیات)" یا "شیر (فلکه آب)" را بفهمد.

این دقیقاً همان ساختاری است که مغز ما برای درک معنای کلمات یاد می‌گیرد.

### **Self-Attention: مکانیزم Query, Key, Value**

حالا بیایید ببینیم Self-Attention چگونه این کار را انجام می‌دهد:



در معماری Self-Attention، ما مفاهیم "کوئری" (Query)، "کلید" (Key) و "ولیو" (Value) را داریم که از دنیای پایگاه داده قرض گرفته شده‌اند:

* **Query (پرس و جو):** نمایانگر کلمه‌ای است که می‌خواهیم معنای آن را در بافت جمله درک کنیم. (مثلاً بردار امبدینگ کلمه‌ی "شیر")
* **Key (کلید):** نمایانگر سایر کلمات در جمله است که ممکن است برای درک معنای کلمه‌ی Query مرتبط باشند. (مثلاً بردارهای امبدینگ "جنگل"، "لیوان"، "یخچال" و سایر کلمات جمله)
* **Value (مقدار):** این‌ها همان بردارهای اصلی کلمات هستند که قرار است با توجه به امتیازات Attention، ترکیب شوند تا یک امبدینگ جدید و بافتی برای کلمه‌ی Query ایجاد کنند.

تفاوت با دیتابیس سنتی:

در یک دیتابیس سنتی، شما یک کوئری را با کلیدها مقایسه می‌کنید و دقیقاً یک کلید مطابق را پیدا کرده و ولیوی مرتبط با آن را برمی‌گردانید.

اما در Attention (و به‌ویژه Self-Attention):

1. **مقایسه با تمام کلیدها:** ما **کوئری** را با **تک‌تک کلیدها** (یعنی تمام کلمات دیگر در جمله) مقایسه می‌کنیم.
2. **محاسبه امتیاز شباهت:** برای هر مقایسه (کوئری با یک کلید خاص)، یک **امتیاز شباهت (Similarity Score)** محاسبه می‌کنیم. این شباهت معمولاً با **ضرب برداری (Dot Product)** کوئری و کلید به‌دست می‌آید. (اگر بردارها عمود بر هم باشند، حاصل صفر است؛ اگر نزدیک به هم باشند، حاصل بزرگتر است.)
3. **نرمال‌سازی امتیازات با Softmax:** این امتیازات خام را از یک تابع **Softmax** عبور می‌دهیم. Softmax این امتیازات را به اعدادی بین **صفر و یک** تبدیل می‌کند که **مجموع آن‌ها یک** است. این اعداد، همان **وزن‌های Attention** یا **میزان توجه** به هر کلمه هستند.
4. **جمع وزن‌دار ولیوها:** در نهایت، هر **ولیو** (بردار کلمه‌ی اصلی) را در وزن Attention مربوط به خودش ضرب می‌کنیم و سپس **همه‌ی آن‌ها را با هم جمع می‌کنیم (جمع وزن‌دار)**. نتیجه این جمع، یک بردار جدید است که **امبدینگ بافتی** کلمه‌ی Query ما را نشان می‌دهد.

چرا "Self-Attention"؟

این مکانیزم را "Self-Attention" می‌نامیم، چون کوئری، کلیدها و ولیوها همگی از همان جمله‌ی ورودی (یعنی از همان کلمات خود جمله) مشتق شده‌اند.

**مثال "شیر" و "جنگل" دوباره:**

* هنگام پردازش کلمه‌ی "شیر" (به عنوان کوئری)، بردار امبدینگ "شیر" با بردار امبدینگ "جنگل" (به عنوان کلید) ضرب می‌شود. چون این دو کلمه اغلب در کنار هم می‌آیند، حاصل ضرب (امتیاز شباهت) بالایی خواهد بود.
* وقتی این امتیازات وارد Softmax می‌شوند، کلمه‌ی "جنگل" یک وزن Attention بسیار بالا دریافت می‌کند.
* در نهایت، این وزن بالا در ولیو "جنگل" ضرب شده و باعث می‌شود که در امبدینگ نهایی "شیر"، **تأثیر "جنگل" بسیار پررنگ شود** و مفاهیم دیگر "شیر" (مثل شیر آب یا شیر لبنیات) عملاً نادیده گرفته شوند. این همان چیزی است که ما از یک امبدینگ وابسته به بافت انتظار داریم.

درک کلی:

این فرآیند، یعنی محاسبه امتیاز شباهت بین هر کلمه (به عنوان کوئری) با تمام کلمات دیگر در جمله (به عنوان کلیدها) و سپس جمع وزن‌دار ولیوهای آن‌ها برای تولید یک امبدینگ جدید، به صورت موازی برای تمام کلمات جمله انجام می‌شود. اگر جمله‌ی شما N کلمه داشته باشد، ما N تا بردار اتنشن محاسبه می‌کنیم (هر کدام به سایز N)، به این معنی که در نهایت N2 مقدار اتنشن بین هر دو کلمه‌ای در جمله محاسبه می‌شود.

اصل **Attention** همین بود که توضیح دادم. از این مقاله، یک مفهوم مهم دیگر به نام **"ترنسفورمر"** نیز استخراج می‌شود.

## **مکانیزم Attention: جزئیات ریاضی و Scaled Dot-Product Attention**

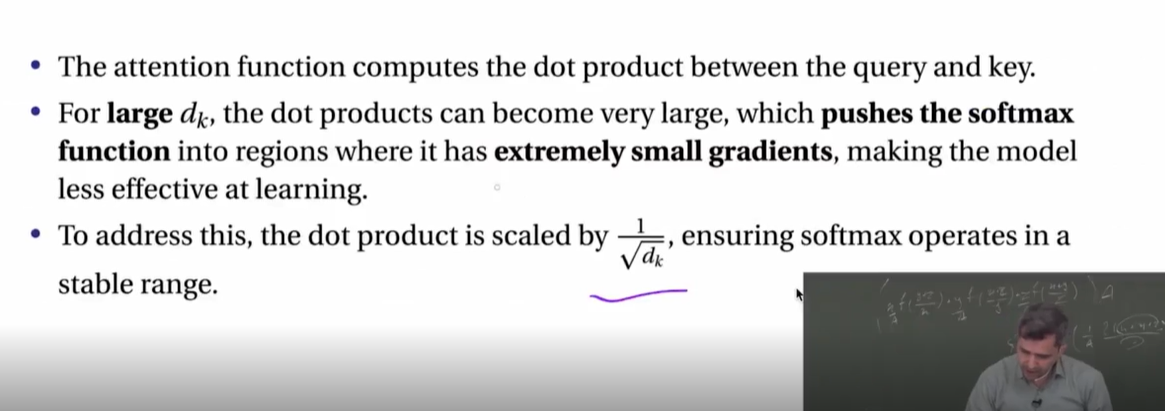
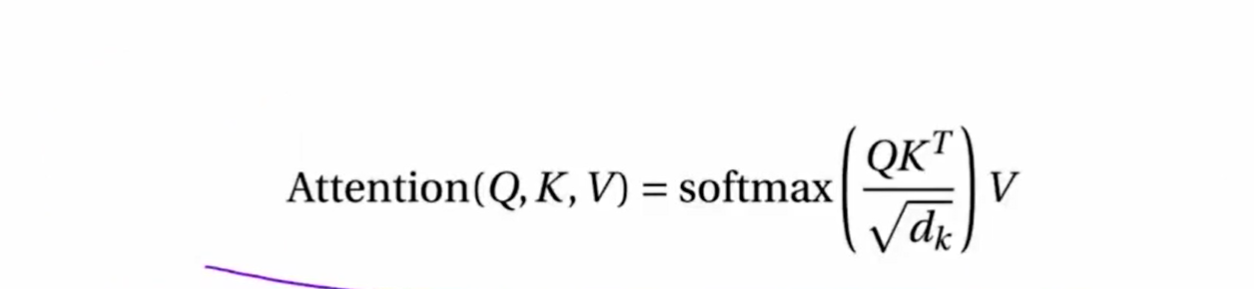
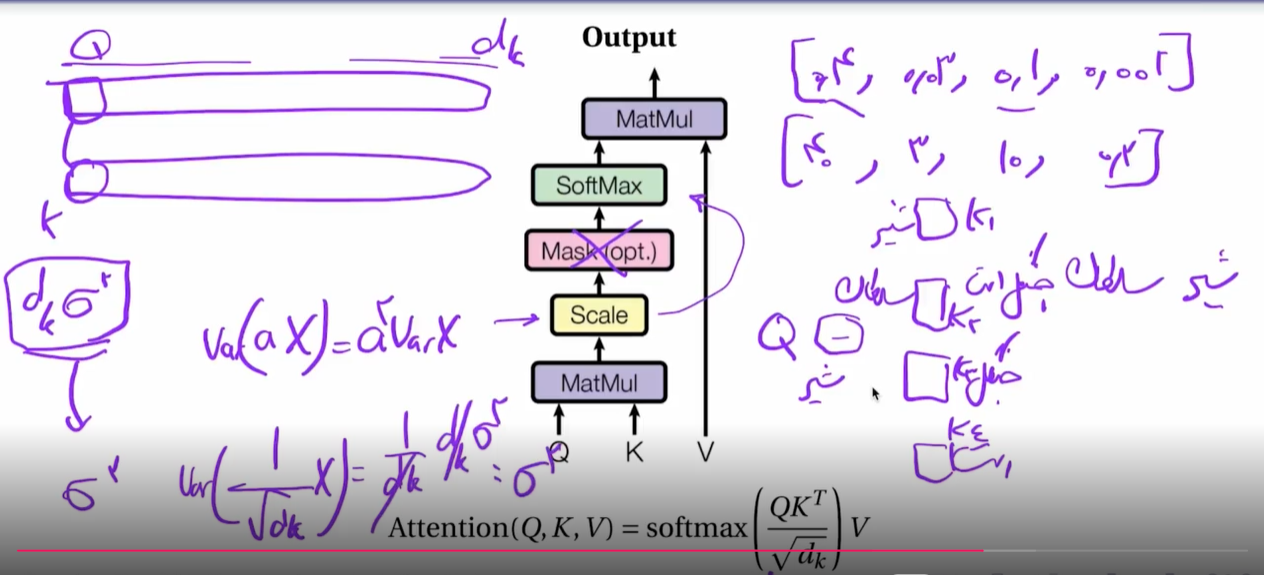
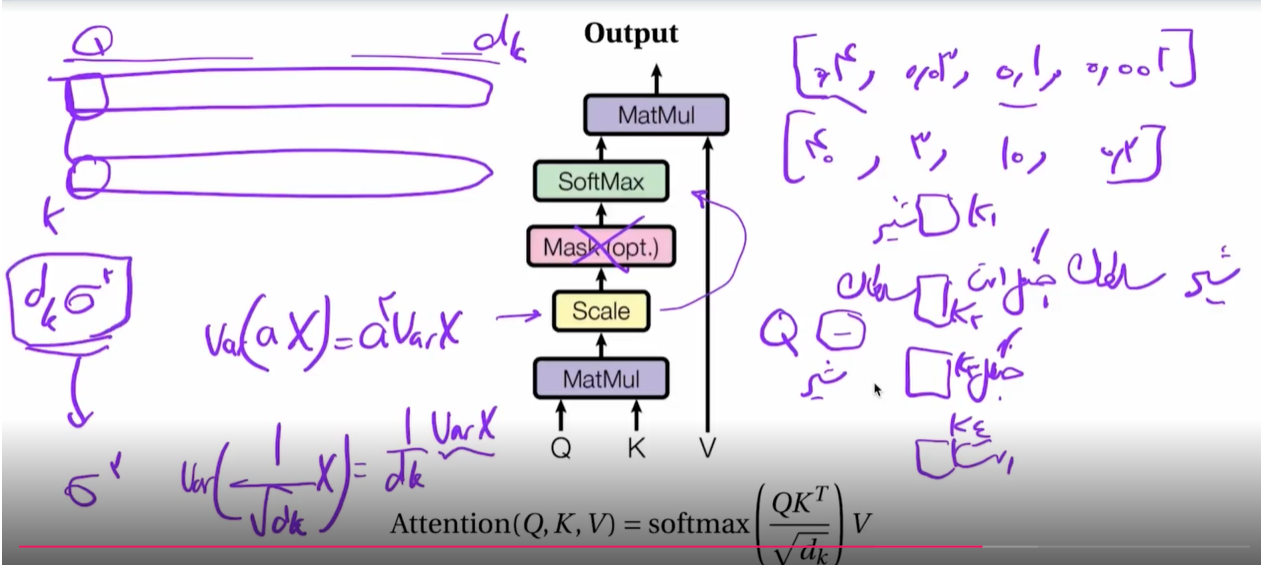
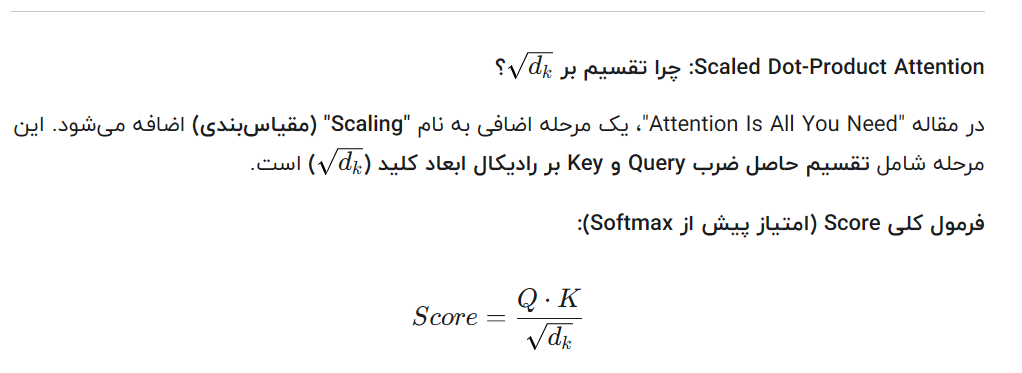
بسیار عالی! تا اینجا با مفهوم کلی Attention آشنا شدید. حالا می‌خواهیم به جزئیات ریاضی آن بپردازیم و ببینیم چگونه این "توجه" محاسبه می‌شود.

در قلب مکانیزم Attention، عملیات ساده‌ای وجود دارد: **ضرب داخلی (Dot Product)** یک **Query (پرس‌وجو)** با هر **Key (کلید)**.

### **فرمول اصلی: Query × Key**

وقتی یک Query و یک Key داریم، آن‌ها را به سادگی در هم ضرب می‌کنیم. مثلاً اگر Q بردار Query و K بردار Key باشد، امتیاز شباهت اولیه از Q⋅K (ضرب داخلی) به‌دست می‌آید.

که هر کدوم همون بردارای امبدینگ word2wec هستند



داره از گرادیان های خیلی کوچک جلوگیری میکنه

