**به نام خدا**

1. **مبانی تئوری Transformer**

#### **۱.۱ محدودیت‌های RNN/LSTM/GRU**

* **پردازش ترتیبی (غیرموازی):** در RNNها برای پردازش یک دنباله، باید هر عنصر به صورت متوالی پردازش شود. این باعث می‌شود که از قابلیت‌های پردازش موازی در GPUها بهره‌برداری کامل نشود و آموزش مدل‌ها به کندی انجام شود.
* **مشکل حافظه بلندمدت:** LSTM و GRU تا حدی مشکل vanishing gradient را حل کرده‌اند، اما همچنان در یادگیری وابستگی‌های بسیار طولانی‌مدت (بیش از ۱۰۰ توکن) با چالش مواجه هستند.
* **ناتوانی در مدل‌سازی وابستگی‌های طولانی‌مدت:** در پردازش زبان طبیعی، گاهی نیاز داریم کلمه‌ای در ابتدای جمله با کلمه‌ای در انتهای جمله ارتباط برقرار کند که این برای RNNها دشوار است.

#### **۱.۲ معماری Transformer (مقاله 2017 "Attention Is All You Need")**

* **Encoder-Decoder بدون RNN** : Transformer از یک معماری کاملاً مبتنی بر توجه استفاده می‌کند و نیاز به حافظه داخلی مانند hidden state در RNN را حذف می‌کند.
* **:Self-Attention:** این مکانیزم به هر موقعیت در دنباله ورودی اجازه می‌دهد تا با تمام موقعیت‌های دیگر در همان دنباله تعامل داشته باشد. این کار با محاسبه یک weighted sum از مقادیر همه موقعیت‌ها انجام می‌شود.
* **:Positional Encoding:** از آنجا که Transformer از ترتیب ذاتی RNNها برخوردار نیست، اطلاعات موقعیتی را از طریق positional encoding به ورودی اضافه می‌کند.

#### **۱.۳ مکانیزم Attention**

مکانیزم توجه به مدل اجازه می‌دهد تا به بخش‌های مختلف ورودی با درجات اهمیت مختلف توجه کند. فرمول attention به این صورت است:

\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d\_k}}\right)V

* **Q (Query):** نشان‌دهنده چیزی است که به دنبال آن هستیم
* **K (Key):** چیزی که با آن مقایسه می‌شود
* **V (Value):** اطلاعات واقعی که باید جمع‌آوری شود
* **√d\_k:** factor scaling برای جلوگیری از gradients کوچک

### ****پیاده‌سازی Self-Attention****

#### **۲.۱ کد پایه Attention**

**این کد یک لایه Self-Attention (خودتوجهی) را از صفر پیاده‌سازی می‌کند** - یکی از مهم‌ترین مفاهیم در مدل‌های پیشرفته AI مثل ChatGPT. در اینجا به صورت ساده توضیح می‌دهم:

## **هدف کد: درک اینکه مدل‌های زبانی چگونه به کلمات "توجه" می‌کنند**

### 📖 ****صورت مسئله****:

فرض کنید این جمله را داریم:  
**"گربه روی مبل خوابید چون خسته بود"**

مدل باید بفهمد که:

* "خسته بود" به **"گربه"** اشاره دارد (نه به مبل!)
* این درک از **رابطه بین کلمات** با مکانیزم Attention انجام می‌شود

## **مفهوم Self-Attention به زبان ساده**

Imagine you're reading a sentence. Your brain automatically **weights** different words differently based on their importance and context. That's exactly what self-attention does!

## **آنالوژی انسانی:**

### ****مثال در دنیای واقعی****:

* **Query**: "من چه چیزی نیاز دارم بدانم؟" (مثلاً: ضمیر "او" به چه اشاره دارد؟)
* **Key**: "چه اطلاعاتی دارم؟" (همه کلمات جمله)
* **Value**: "اطلاعات واقعی" (معنی هر کلمه)

### ****مثال در جمله****:

"**او** به مغازه رفت چون **گربه** گرسنه بود"

مدل یاد می‌گیرد که:

* "او" بیشترین توجه را به "گربه" داشته باشد
* "گرسنه" توجه به "گربه" و "مغازه" داشته باشد

## **📊 خروجی کد:**

### ****Heatmap توجه****:

* **محور X**: کلمات کلیدی (Key)
* **محور Y**: کلمات پرسشی (Query)
* **رنگ روشن**: توجه بیشتر

### ****تفسیر****:

* کلمه اول به خودش توجه زیادی دارد
* کلمه دوم به کلمه اول توجه دارد
* الگوی توجه یادگرفته شده

## **کاربردهای واقعی:**

1. **ترجمه ماشینی**: فهمیدن اینکه کدام کلمات به هم مرتبط هستند
2. **چت‌بات‌ها**: درک context مکالمه
3. **خلاصه‌سازی**: تشخیص مهم‌ترین بخش‌های متن

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

#---------------------------------------------------------------------------------

# این کد یک لایه  (خودتوجهی) را از صفر پیاده‌سازی می‌کند

# هدف کد: درک اینکه مدل‌های زبانی چگونه به کلمات "توجه" می‌کنند

#---------------------------------------------------------------------------------

# --------------------------------------

# تعریف کلاس SelfAttention

# --------------------------------------

class SelfAttention(tf.keras.layers.Layer):

    # سازنده (init)

    def \_\_init\_\_(self, d\_model):

        # d\_model: بعد بردارهای داخلی (مثلاً ۶۴ بعد)

        super(SelfAttention, self).\_\_init\_\_()

        self.d\_model = d\_model

        # سه لایه Dense برای تولید:

        # Query (wq): چیزی که می‌خواهم بدانم

        self.wq = tf.keras.layers.Dense(d\_model)

        # Key (wk): چیزی که دارم

        self.wk = tf.keras.layers.Dense(d\_model)

        # Value (wv): اطلاعات واقعی

        self.wv = tf.keras.layers.Dense(d\_model)

    # --------------------------------------

    # عملیات اصلی (call)

    # --------------------------------------

    # ورودی: یک جمله به صورت بردار (مثلاً ۵ کلمه، هر کلمه ۶۴ بعد)

    def call(self, inputs):

        # مثال در دنیای واقعی

          # Query : من چه چیزی نیاز دارم بدانم؟" (مثلاً: ضمیر "او" به چه اشاره دارد؟)

          # Key: "چه اطلاعاتی دارم؟" (همه کلمات جمله)

          # Value: "اطلاعات واقعی" (معنی هر کلمه)

        Q = self.wq(inputs) # تولید Query

        K = self.wk(inputs) # تولید Key

        V = self.wv(inputs) # تولید Value

        # محاسبه توجه

          # مرحله ۱: محاسبه شباهت بین هر جفت کلمه (ضرب Q و K)

          # مرحله ۲: نرمال‌سازی برای پایداری عددی

          # مرحله ۳: تبدیل به احتمال با softmax

          # مرحله ۴: ترکیب weighted با مقادیر واقعی (V)

        scores = tf.matmul(Q, K, transpose\_b=True)

        scores = scores / tf.math.sqrt(tf.cast(self.d\_model, tf.float32))

        attention\_weights = tf.nn.softmax(scores, axis=-1)

        # خروجی: همان جمله اما با "توجه" به روابط بین کلمات

        return tf.matmul(attention\_weights, V)

# --------------------------------------

# تست اصلی با ابعاد صحیح

# --------------------------------------

# ساخت یک ورودی نمونه (Dummy Input)

  # tf.random.normal: تولید اعداد تصادفی از توزیع نرمال

  # مثال واقعی: مثل اینه که یک جمله ۵ کلمه‌ای داریم و هر کلمه با ۶۴ عدد نمایش داده شد

  # (1, 5, 64): شکل (shape) ورودی → ۱ جمله، ۵ کلمه، هر کلمه ۶۴ ویژگی

sample\_input = tf.random.normal((1, 5, 64))

# ایجاد لایه Self-Attention

  # 64: پارامتر d\_model → باید با بعد آخر ورودی یکسان باشد (۶۴)

attention\_layer = SelfAttention(64)

# اعمال لایه روی ورودی

  # ورودی را به لایه Self-Attention می‌دهیم

  # لایه محاسبات توجه را انجام می‌دهد

  # خروجی جدیدی تولید می‌کند

output = attention\_layer(sample\_input)

print("Input shape:", sample\_input.shape)

print("Output shape:", output.shape)

#### **۲.۲ تجسم مکانیزم Attention**

# --------------------------------------

# تست دوم با ابعاد متناسب

# --------------------------------------

# ساخت داده تست ساده

  # می‌سازیم یک جمله مصنوعی با ۳ کلمه

  # هر کلمه یک بردار ۲ بعدی داره (مثلاً [1.0, 0.5])

  # مثل اینه: ["گربه", "خواب", "نرم"]

test\_input = tf.constant([[[1.0, 0.5], [0.5, 1.0], [0.2, 0.8]]], dtype=tf.float32)

# تکثیر برای batch

  # مدل‌های واقعی همزمان چند جمله پردازش می‌کنن (batch)

  # اینجا ۲ تا جمله یکسان می‌سازیم

test\_input = tf.tile(test\_input, [2, 1, 1])  # شکل: (2, 3, 2)

# محاسبه توجه

  # محاسبه می‌کنه هر کلمه چقدر به کلمات دیگه شباهت داره

attention\_layer\_2 = SelfAttention(d\_model=2)  # d\_model متناسب با ورودی

Q = attention\_layer\_2.wq(test\_input)

K = attention\_layer\_2.wk(test\_input)

scores = tf.matmul(Q, K, transpose\_b=True)

# تبدیل به احتمال

  # اعداد رو به احتمال تبدیل می‌کنه (بین ۰ تا ۱)

  # جمع هر سطر = ۱ (مثل درصد توجه)

attention\_weights = tf.nn.softmax(scores, axis=-1)

# --------------------------------------

# توضیح نمودار Heatmap:

# --------------------------------------

# خواندن نمودار:

  # محور عمودی (Query): کلمه‌ای که سؤال داره ("چه کلمه‌ای به من توجه کنه؟")

  # محور افقی (Key): کلمه‌ای که جواب میده ("من چقدر مهمم؟")

  # رنگ‌ها: میزان توجه (زرد = توجه زیاد، بنفش = توجه کم)

# Heatmap توجه:

 # محور X: کلمات کلیدی (Key)

 # محور Y: کلمات پرسشی (Query)

 # رنگ روشن: توجه بیشتر

plt.figure(figsize=(10, 5))

plt.imshow(attention\_weights[0].numpy(), cmap='viridis')

plt.colorbar()

plt.title('Attention Weights')

plt.xlabel('Key Position')

plt.ylabel('Query Position')

plt.show()

plt.show()