دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

گزارش فنی تمرین سوم

NLP

پدیدآورنده:

محمد امین کیانی

۴۰۰۳۶۱۳۰۵۲

دانشجوی کارشناسی، دانشکده‌ی کامپیوتر، دانشگاه اصفهان، اصفهان،
Aminkianiworkeng@gmail.com

استاد درس: جناب اقای دکتر برادران

نیمسال دوم تحصیلی ۰۴-۱۴۰۳

فهرست مطالب

مستندات

بخش اول: پرسش‌ها

در این بازی: دو واژه‌ی اولیه که از هم "خیلی دور" هستند (فاصله‌شان بیشتر از α) به عنوان «خاص‌واژگان» معرفی می‌شوند. بازیکن واژه‌ای پیشنهاد می‌دهد که باید از لحاظ معنایی به یکی از خاص‌واژگان "نزدیک" باشد. (فاصله کمتر از β) و بازی ادامه پیدا می‌کند تا جایی که همه‌ی واژه‌های معرفی‌شده به‌هم نزدیک باشند. (فاصله بین همه‌شان کمتر از β). یک **روش نگاشت واژگان به فضای تعبیه** (Embedding Space) که در آن بتوان فاصله‌های معنایی بین واژگان را به خوبی اندازه‌گیری کرد و بازی را طبق قوانین پیش برد را پیشنهاد بدهید.

در زبان طبیعی، واژه‌ها مفاهیم انتزاعی هستند. برای اینکه بتوان آن‌ها را با الگوریتم‌ها تحلیل کرد، باید آن‌ها را به بردارهای عددی در فضای برداری نگاشت کرد. این فضا باید ویژگی‌های معنایی را حفظ کند؛ مثلاً:

- واژگان هم‌معنا به هم نزدیک باشند.
- واژگان متضاد از هم دور باشند.
- فرض کنیم دو خاص‌واژه به صورت زیر تعریف شده‌اند:

“dog” ⟶ [۰.۳, ۰.۸]

“cat” ⟶ [۰.۴, ۰.۷]

اگر α = ۰.۲ و β = ۰.۱۵ باشد و بازیکن واژه‌ی “puppy” را پیشنهاد دهد :

اگر فاصله بین puppy و یکی از خاص‌واژگان ۰.۱۵ > باشد، آن را قبول می‌کنیم.

انتخاب فضای تعبیه مناسب:

فضای تعبیه‌ای که در اینجا مناسب است باید خصوصیات زیر را داشته باشد:

الف) حفظ روابط معنایی:

اگر "گربه" و "سگ" از نظر معنایی نزدیک‌اند، بردارهایشان باید فاصله کمی داشته باشند.

ب) قابلیت محاسبه فاصله:

نیاز داریم بین دو واژه فاصله (مثل فاصله اقلیدسی یا کسینوسی) تعریف کنیم.

ج) تعمیم‌پذیری و پوشش وسیع واژگان:

روش انتخابی باید بتواند طیف وسیعی از واژگان زبان را نگاشت کند.

روش پیشنهادی برای نگاشت واژگان:

استفاده از Word۲Vec یا GloVe یا نگاشت E۲ (در تمارین قبلی به طور کامل این روش ها توضیح داده شده است و E2 نیز در ادامه با عنوان روش تک نمود بررسی شده است.)

Word۲Vec:

یک روش معروف مبتنی بر شبکه عصبی است که دو مدل دارد:

- **CBOW (Continuous Bag of Words)**: سعی می‌کند با استفاده از کلمات اطراف، کلمه وسط را پیش‌بینی کند.
- **Skip-gram**: برعکس؛ با استفاده از یک کلمه، سعی می‌کند کلمات اطرافش را پیش‌بینی کند.

ویژگی‌ها:

- کلمات مشابه در فضای برداری نزدیک‌اند.
- امکان استفاده از فاصله کسینوسی یا اقلیدسی برای سنجش نزدیکی معنایی.

GloVe:

- مبتنی بر آمار هم‌رخدادی (co-occurrence) کلمات در کل پیکره متنی.
- مناسب‌تر برای تشخیص روابط جهانی بین کلمات نسبت به Word۲Vec .

از دیگر تفاوت‌های مهم در معماری های گفته شده با معماری داخل تمرین می‌توان به این موارد اشاره کرد:

- One-Hot Encoding نیاز به حافظه زیاد و محاسبات سنگین دارد، در حالی که Word Embedding حافظه و منابع محاسباتی را بهینه‌تر استفاده می‌کند.

- One-Hot Encoding مناسب برای مدل‌های ساده و مسائل کوچک است، ولی Word Embedding برای مدل‌های یادگیری عمیق و پردازش زبان طبیعی در مقیاس بزرگ بسیار ضروری است و مزایای قابل توجهی در درک بهتر معنا و روابط بین کلمات فراهم می‌کند و به همین دلیل در اغلب کاربردهای مدرن NLP به کار می‌رود.

نحوه محاسبه فاصله:

- فاصله اقلیدسی بین دو بردار:

$$\sqrt{\sum_{i=1}^{n}(v_{1i}-v_{2i})^2} = d(v_1, v_2)$$

- فاصله کسینوسی:

$$\frac{v_1 \cdot v_2}{\|v_1\| \cdot \|v_2\|} - 1 = d(v_1, v_2)$$

در کاربردهای معنایی معمولاً فاصله کسینوسی ارجح‌تر است، چون به زاویه بین بردارها توجه می‌کند نه بزرگی آن‌ها.

نگاشت ابتدایی E۱ :

نگاشت E۱(wi) یک نگاشت ساده و پراکنده (sparse) مانند **One-Hot** است. یعنی هر واژه با برداری صفر و یک نمایش داده می‌شود، که فقط در یک بعد مقدار ۱ دارد و بقیه صفر هستند و این روش **معنای واژه‌ها را در نظر نمی‌گیرد**، مثلاً "گربه" و "سگ" کاملاً متفاوت خواهند بود، هرچند از نظر معنایی نزدیک‌اند.

$$E_1(w_i) = [x_1, x_2, \ldots, x_n]$$

نگاشت پیشرفته E۲ با استفاده از شبکه عصبی:

فرایند:

- ابتدا واژه‌ها به صورت One-Hot یعنی نگاشت E۱ به شبکه داده می‌شوند.
- سپس این ورودی از یک لایه پنهان (hidden layer) عبور می‌کند که شامل وزن‌هایی (θ) برای هر نورون است.
- خروجی شبکه نگاشت جدید E۲(wi) است:

$$f(E_1(w_i), \theta) = E_2(w_i)$$

تابع f یک نگاشت خطی است که وزن‌ها را یاد می‌گیرد تا تبدیل بردار پراکنده (One-hot) به بردار فشرده و معنایی با استفاده از **شبکه عصبی ساده** را انجام دهد.

E۱(wi) : نمایش One-hot واژه.

θ\theta : ماتریس وزن‌ها (وزن‌های لایه پنهان).

مزایا:

- واژگان معنایی مشابه بردارهای مشابه دارند.
- فشرده و قابل آموزش است.
- می‌توان فاصله معنایی بین دو واژه را به‌درستی اندازه گرفت.
- نحوه اندازه‌گیری فاصله (Similarity Measure):
- برای بررسی نزدیکی یا دوری دو واژه (برای مقایسه با α و β)، از شباهت کسینوسی استفاده می‌کنیم:
  - $$\frac{E_2(w_i) \cdot E_2(w_j)}{||E_2(w_i)|| \cdot ||E_2(w_j)||} = \text{Cosine similarity}$$
  - Cosine distance=۱−Cosine similarity
  - این روش مستقل از اندازه‌ی بردارهاست و فقط زاویه بین آن‌ها را می‌سنجد که برای تحلیل معنایی ایده‌آل است.

چگونگی محاسبه وزن‌ها در E۲ :

- شبکه عصبی یادگیرنده:

- ساختار شبکه: لایه ورودی(One-hot) ، لایه پنهان (بردار تعبیه)، لایه خروجی (مثلاً پیش‌بینی کلمات بعدی) .
  - ورودی: بردار One-hot با طول |V| که تعداد واژه‌ها در دیکشنری است.
  - لایه مخفی: بردار تعبیه با ابعاد پایین‌تر.
  - خروجی: احتمال واژگان زمینه‌ای.(context words) یا برعکس بسته به روش انتخابی.
  - یک ماتریس با ابعاد d×|V| که |V|تعداد کل واژه‌ها و d تعداد نورون‌های لایه پنهان که هر سطر این ماتریس، بردار تعبیه یک واژه خاص است که تبدیل به یک بردار معنی‌دار می‌شود. این بردارها همان نگاشت نهایی E۲(wi) هستند که در بازی استفاده می‌شوند.
  - **تابع نگاشتf :**
    - یک ضرب ماتریسی ساده: E۲(wi)= E۱(wi)×θ
    - چون E۱(wi) فقط در یک جای خاص ۱ است، نتیجه‌ی این ضرب، دقیقاً سطر مربوط به آن واژه از ماتریس θ است.
  - **تابع تابع هدف(Loss Function) :**
  - مثلاً در Word۲Vec از Negative Samplingیا Softmax Cross Entropyاستفاده می‌شود تا احتمال واژگان زمینه‌ای واقعی بیشتر از سایر واژه‌ها شود.
  - وزن‌ها در حین آموزش با الگوریتم گرادیان نزولی (Gradient Descent) آپدیت می‌شوند و با هر نمونه آموزش (کلمه هدف و زمینه)، شبکه خطا را محاسبه می‌کند. سپس وزن‌های θاصلاح می‌شوند تا نگاشت معنایی بهبود یابد. در نهایت، وزن‌های آموخته‌شده θ شامل بردارهای معنایی واژگان هستند.
- **تابع خطا**: معمولاً از **Cross Entropy**یا **Softmax Loss** برای یادگیری استفاده می‌شود.
- **آموزش** :بر اساس هم‌رخدادی واژگان(context) ، وزن‌ها به‌روزرسانی می‌شوند.

برای این بازی ، **بهترین روش نگاشت**، استفاده از نگاشت E۲ از طریق **مدل‌های بردار معنایی مانند Word۲Vec**یا **GloVe** است. این نگاشت خصوصیات زیر را دارد:

- فاصله‌های معنایی بین واژگان را به‌خوبی مدل می‌کند.
- امکان اندازه‌گیری فاصله بین دو واژه برای بررسی شرایط α و β را فراهم می‌سازد.
- یادگیری وزن‌ها با استفاده از هم‌رخدادی واژگان صورت می‌گیرد.
- قابلیت پیاده‌سازی عملی با ابزارهای NLP مدرن را دارد.
- وزن‌ها در E۲ همان ماتریس θ هستند که در طی آموزش با پیش‌بینی واژگان زمینه بهینه می‌شوند.
- نگاشت E۲ از E۱ یک تبدیل خطی ساده است.
- بردار خروجی E۲ برای هر واژه، نمایش فشرده و معنایی آن واژه است.
- فاصله بین بردارها با شباهت کسینوسی اندازه‌گیری می‌شود تا معیار α و β پیاده‌سازی شوند.

در این بازی واژگانی، نگاشت مناسب برای انتقال واژگان به فضای عددی باید نگاشت E۲ باشد که با استفاده از شبکه عصبی ساده پیاده‌سازی می‌شود. در این روش، ابتدا بردار One-hot نگاشت E۱ از هر واژه گرفته می‌شود. سپس با ضرب این بردار در ماتریس وزن‌هاθ ، بردار تعبیه معنایی E۲ به دست می‌آید. وزن‌های θ در طی آموزش از روی داده‌های زبانی، با پیش‌بینی واژه‌های زمینه‌ای به کمک الگوریتم‌هایی مانند Word۲Vec یاد گرفته می‌شوند. این بردارها برای محاسبه‌ی فاصله بین واژه‌ها با معیارهایی مانند شباهت کسینوسی استفاده می‌شوند تا بتوان قوانین پذیرش یا رد واژگان در بازی شرایط α و β را دقیقاً اعمال کرد.

```
! pip install gensim      # یک کتابخانه متن‌باز و تخصصی برای مدل‌سازی معنایی متون
from gensim.models import Word2Vec
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# دیتاست نمونه (هر جمله یک لیست از واژگان)
sentences = [
    ["dog", "barks", "loud"],
    ["cat", "meows", "softly"],
```

```
    ["dog", "and", "cat", "are", "pets"],
    ["lion", "roars", "in", "jungle"],
    ["puppy", "is", "a", "young", "dog"],
    ["kitten", "is", "a", "young", "cat"], ]
# آموزش مدل Word2Vec
model = Word2Vec(sentences, vector_size=50, window=2, min_count=1, sg=1)
#sg=1 => skip-gram

# گرفتن بردار تعبیه برای یک واژه
vec_dog = model.wv["dog"]
vec_cat = model.wv["cat"]
vec_puppy = model.wv["puppy"]

# محاسبه شباهت کسینوسی بین دو واژه
sim_dog_cat = cosine_similarity([vec_dog], [vec_cat])[0][0]
sim_dog_puppy = cosine_similarity([vec_dog], [vec_puppy])[0][0]

print("Cosine Similarity between 'dog' and 'cat':", sim_dog_cat)
print("Cosine Similarity between 'dog' and 'puppy':", sim_dog_puppy)

# یا از تابع داخلی Gensim:
print("Gensim similarity (dog vs cat):", model.wv.similarity("dog", "cat"))
print("Gensim similarity (dog vs puppy):", model.wv.similarity("dog", "puppy"))
```

Cosine Similarity between 'dog' and 'cat': ۰.۰۴

Cosine Similarity between 'dog' and 'puppy': ۰.۰۵

dog و puppy از نظر معنایی بسیار نزدیک‌اند.

dog و cat نیز نزدیک‌اند ولی کمتر از dog وpuppy .

در بازی مورد نظر:

- اگر فاصله کسینوسی بین word_new و یکی از خاص‌واژه‌ها کمتر از β بود ⟵ پذیرش
- اگر فاصله بین خاص‌واژه‌ها بیشتر از α بود ⟵ شروع بازی مجاز است.

در ابتدای بازی، دو واژه از واژگان مدل انتخاب می‌شوند و به عنوان special_words معرفی می‌شوند.

یعنی شرط d >= ALPHA کاملاً رعایت شده. حتی شرط معقول 0.8 =>هم اضافه شده تا خیلی دور هم نباشند.

فقط اگر فاصله‌ی کسینوسی با حداقل یک واژه موجود β >باشد، واژه پذیرفته می‌شود.

تابع all_within_beta()بررسی می‌کند که همه‌ی واژگان پذیرفته‌شده به هم نزدیک باشند یعنی فاصله بین همه β > . باید واژه‌هایی پیشنهاد داد که: **با همه‌ی واژگان قبلی** فاصله‌شون β > باشه مثلا اگر ۴ تا کلمه داخل لیست هست، واژه جدید باید با هر ۴تا «نزدیک» باشد.

اگر اولین واژه‌ی بازیکن به لیست پذیرفته‌شده اضافه شود، و بلافاصله all_within_beta() برقرار باشد، بازی همان‌جا به پایان می‌رسد.

چرخه‌ی بازی (while True) ادامه پیدا می‌کند و واژگان توسط بازیکن پیشنهاد می‌شوند، تا زمانی که شرط نهایی برقرار شود یا حداکثر تلاش رد شده باشد که دومی ناشی از واژه‌هایی که وارد شده پذیرفته نمی‌شوند یا اگر هم پذیرفته بشوند، در نهایت **مجموعه‌ی واژه‌های پذیرفته‌شده** به‌قدری پراکنده اند که شرط فاصله بین همه β > برقرار نمی‌شود.

بخش	در نسخه اصلی (با Word2Vec)	در این نسخه (دستی)
نگاشت E1 (One-hot)	خودکار درWord2Vec	دستی تعریف شده با `np.zeros()`

E2 نگاشت (بردار فشرده)	آموزش‌دیده توسط Gensim	در ماتریس One-hot با ضرب وزنθ
ماتریس وزن θ	خودکار و مخفی در مدل	`np.random.randn()` ساخته و آموزش داده شده
آموزش مدل (یادگیری وزن)	negative sampling و skip-gram با Word2Vec	دستی با سافت مکس و گرادیان نزولی
فاصله کسینوسی	`sklearn.metrics.pairwise.cosine_similarity`	`np.dot` و `np.linalg.norm`
پیشنهاد هوشمند واژه بعدی	Word2Vec similarity	دستی با محاسبه شباهت کسینوسی و میانگین فاصله
اجرای بازی	با مدل آماده	E2 با مدل آموزش‌دیده‌ی دستی

```
def E1(word):
    vec = np.zeros(vocab_size)
    vec[word2idx[word]] = 1
    return vec
```

این یعنی هر کلمه یک بردار با صفرهاست که فقط در جای خودش ۱ دارد.

```
def E2(word):
    return E1(word).dot(theta)
```

یعنی ضرب One-hot در θ که برداری مثل [۰.۲, ۰.۶, -۱.۳] می‌دهد.

```
for target, context in positive_pairs:
…
    y_pred = x.dot(theta).dot(theta.T)  # softmax
…
```

```
theta -= lr * grad گرادیان نزولی #
```

وزن‌ها را خودمان با مشتق و اصلاح ماتریس‌ها آموزش دادیم تا بردارهای مشابه به هم نزدیک شوند.

```
def cosine_similarity_manual(v1, v2):
    return np.dot(v1, v2) / (norm(v1)*norm(v2))
```

مطابق فرمول ریاضی به صورت دستی.

کل کد به فرم دستی دقیقاً همان کاری است که پشت صحنه Word2Vec انجام می‌شود.

بخش دوم: برنامه نویسی

۱- مجموعه داده:

توضیح	بخش سوال
نصب شد	نصب `portalocker`
انجام شد	ایمپورت `torchtext.datasets`
با لیست تبدیل شد	بارگذاری `train_iter` و`test_iter`
پنج داده‌ی اولیه نمایش داده شد	چاپ نمونه‌های اولیه
تصحیح و استانداردسازی شد	توجه به نقل‌قول‌های خراب پی‌دی‌اف

- Portalocker یک کتابخانه برای قفل‌گذاری فایل‌ها در Python است. این کتابخانه کمک می‌کند تا از دسترسی همزمان به یک فایل توسط چندین پروسه یا رشته جلوگیری کنید. این کار به ویژه در برنامه‌های چندرشته‌ای یا چندپروسه‌ای (multiprocessing) مهم است، زیرا می‌تواند از بروز مشکلاتی مانند تداخل داده‌ها جلوگیری کند. پکیج torchtext برای بارگیری دیتاست‌ها مثل IMDB ، هنگام خواندن فایل‌ها از portalocker برای **قفل‌گذاری همزمان** (**thread-safe access**) استفاده می‌کند.یعنی وقتی چند پردازه یا نخ می‌خوان همزمان فایل دیتاست رو بخونن یا بنویسن، از تداخل جلوگیری می‌کند پس مستقیم از portalocker استفاده نمی‌کنیم، ولی چون torchtext به آن وابسته‌است، **باید نصب کنیم تا دیتاست IMDB درست بارگذاری بشود.**

```
نصب ضروریات #
```

```
!pip install torchtext --upgrade
!pip install "portalocker>=2.0.0"

# پاکسازی نسخه‌های قبلی ()
!pip uninstall -y torch torchtext

# نصب نسخه‌های سازگار
!pip install torch==2.0.1+cpu torchtext==0.15.2 -f
https://download.pytorch.org/whl/torch_stable.html
from torchtext.datasets import IMDB

# تبدیل به لیست کامل قابل ایندکس
train_iter = list(IMDB(split='train'))
test_iter = list(IMDB(split='test'))

# حالا می‌تونی ایندکس‌گذاری و نمونه‌گیری داشته باشی
for i, (label, line) in enumerate(train_iter[:5]):
    print(f" Label: {label}\n Text: {line[:150]}...\n{'-'*50}")
from torchtext.datasets import IMDB
import random

train_iter = list(IMDB(split='train'))
test_iter = list(IMDB(split='test'))

# کاهش حجم: فقط نیمی از آموزش و تست نگه داریم
random.seed(42)
train_iter = random.sample(train_iter, len(train_iter) // 2)
test_iter = random.sample(test_iter, len(test_iter) // 2)

print(f"تعداد داده‌های آموزش: {len(train_iter)}")
print(f"تعداد داده‌های آزمون: {len(test_iter)}")
```

۲- پیش پردازش:

۱. حروف کوچک(lowercase)

۲. حذف HTML tags مثل

۳. حذف stopwords کلمات پرتکرار بی‌معنا مثل"the", "and"

۴. توکن‌سازی (تبدیل متن به لیست واژه‌ها)

۵. حذف واژه‌هایی که فقط یک‌بار در کل دیتاست ظاهر شدند.

آیا حذف کردن واژگانی که تنها در یک نظر اتفاق افتاده‌اند می‌تواند به پروسه‌ی دسته‌بندی کمک کند؟

بله، در اغلب موارد حذف این واژه‌ها مفید است. با توجه به مراحل بعدی این بخش نیز دلایل زیر را داریم:

۱ .کاهش نویز(Noise Reduction)

واژگانی که فقط در یک نظر ظاهر شده‌اند معمولاً شامل:

- اشتباه تایپی
- اسم‌های خاص بی‌ربط
- اصطلاحات منحصر به فرد فردی

هستند و مدل دسته‌بند نمی‌تواند از آن‌ها الگوی قابل تعمیم بسازد.

۲ .کاهش ابعاد فضای برداری(Dimensionality Reduction)

در نگاشت‌هایی مثل:

- **Bag of Words (BOW)**
- **One-hot encoding**

وجود واژگان نادر باعث بزرگ شدن فضای ویژگی می‌شود، اما اطلاعات مفیدی به مدل اضافه نمی‌کنند.

حذف آن‌ها باعث:

- افزایش سرعت آموزش
- کاهش حافظه مصرفی
- بهبود تعمیم‌پذیری مدل

می‌شود.

۳ .پیش‌نیازی برای الگوریتم‌های بعدی مثلBOW ، KMeans، شبکه‌های عصبی

در مراحل بعدی تمرین:

- می‌خواییم از **BOW و خوشه‌بندی KMeans** استفاده کنیم.
- اگر واژگان نادر حذف نشوند، بردارها به‌شدت پراکنده (sparse) و بی‌ارزش می‌شوند.
- در الگوریتم‌هایی مثل KMeans ، ویژگی‌های نادر می‌توانند باعث خوشه‌بندی اشتباه بشوند.

۴ .مطالعات تجربی و تجربیات صنعتی

در اکثر پروژه‌های صنعتی NLP مثلاً متن‌کاوی، تحلیل احساسات، توصیه‌گرها:

حذف واژه‌هایی با فراوانی پایین یکی از مراحل ثابت و استاندارد پیش‌پردازش است.

نکته: اگر از مدل‌هایی مثل Word2Vec یا BERT استفاده کنیم، **لزومی به حذف واژه‌های نادر نیست** چون embeddingآن‌ها از اطلاعات هم‌رخدادی یاد گرفته می‌شود. اما چون این تمرین روی BOW و نگاشت دستی تمرکز دارد، **حذف این واژه‌ها کاملاً بجاست.**

```
import re
import nltk
from nltk.corpus import stopwords
from collections import Counter

# دانلود منابع nltk
nltk.download('punkt')
nltk.download('stopwords')
# پاکسازی کامل کش‌های خراب
nltk.download('popular')
nltk.download('all')

stop_words = set(stopwords.words('english'))

#  تابع پیش‌پردازش برای یک جمله
def preprocess(text):
    # 1. تبدیل به حروف کوچک
    text = text.lower()
    # 2. حذف تگ‌های HTML
    text = re.sub(r"<.*?>", " ", text)
    # 3. حذف کاراکترهای غیرمتنی
    text = re.sub(r"[^a-z\s]", " ", text)
    # 4. توکن‌سازی
    tokens = nltk.word_tokenize(text)
    # 5. حذف stopwords
```

```
    tokens = [w for w in tokens if w not in stop_words]
    return tokens

# پردازش کل داده‌های آموزش و تست
train_tokens = [(label, preprocess(text)) for label, text in train_iter]
test_tokens = [(label, preprocess(text)) for label, text in test_iter]

# نمایش نمونه پیش‌پردازش‌شده
for i in range(3):
    print(f" Label: {train_tokens[i][0]}")
    print(f" Tokens: {train_tokens[i][1][:10]}...\n")
```

فقط از **train** برای شمارش فراوانی استفاده می‌کنیم تا از نشت اطلاعات (data leakage) جلوگیری شود.

```
from collections import Counter

# train_tokens = [(label, tokens)]  ← خروجی پیش‌پردازش اولیه روی داده‌های آموزش
# test_tokens = [(label, tokens)]   ← خروجی پیش‌پردازش اولیه روی داده‌های تست

# شمارش فراوانی تمام واژه‌ها در آموزش
all_tokens = [token for _, tokens in train_tokens for token in tokens]
token_freq = Counter(all_tokens)

# تعریف تابعی برای حذف واژه‌های با فراوانی کم (مثلاً فقط یک بار ظاهر شده‌اند)
def remove_rare_words(tokens, min_freq=2):
    return [token for token in tokens if token_freq[token] >= min_freq]

# اعمال حذف روی train و test
train_tokens_filtered = [(label, remove_rare_words(tokens)) for label, tokens in train_tokens]
test_tokens_filtered = [(label, remove_rare_words(tokens)) for label, tokens in test_tokens]

# نمایش چند نمونه‌ی بعد از فیلتر برای بررسی
for i in range(3):
    print(f" Label: {train_tokens_filtered[i][0]}")
    print(f" Tokens (filtered): {train_tokens_filtered[i][1][:10]}...\n")
```

۳- ساخت بردار BOW برای هر جمله با استفاده از نگاشت E1 :

- نگاشت E1

- برای واژه wi: بردار One-hot فقط در موقعیت واژه مقدار ۱ دارد، بقیه صفر است.

$$E_1(w_i) = [x_n | x_n = 1 \; for \; n = i \; and \; x_n = 0 \; for \; n \neq i, \;\; n = 1, ..., N] \;\; for \; w_i \in Dict$$

- برای جمله si: بردارBag of Words ، حاصل جمع بردارهای One-hot واژگان جمله است. این بردار:
  - طول = اندازه واژه‌نامه
  - مقدار هر درایه = تعداد تکرار آن واژه در جمله

$$BOW(s_i) = E_1(w_1) + ... + E_1(w_N) \; for \; s_i \in Dataset$$

- حال هر جمله به بردار عددی تبدیل شده و آماده‌ی:
  - خوشه‌بندی (KMeans)
  - یا تغذیه به شبکه عصبی

```
# ساخت دیکشنری واژگان نهایی
all_tokens = [token for _, tokens in train_tokens_filtered for token in
tokens]
vocab = sorted(set(all_tokens))
word2idx = {word: idx for idx, word in enumerate(vocab)}
vocab_size = len(vocab)

import numpy as np

def E1(word):
    vec = np.zeros(vocab_size)
    vec[word2idx[word]] = 1
    return vec

def bow_vector(tokens):
    vec = np.zeros(vocab_size)
    for word in tokens:
        if word in word2idx:  # اطمینان از وجود در دیکشنری
            vec += E1(word)
    return vec

train_bow = [(label, bow_vector(tokens)) for label, tokens in
train_tokens_filtered]
```

```
test_bow = [(label, bow_vector(tokens)) for label, tokens in
test_tokens_filtered]

# نمایش یکی از بردارهای BOW به همراه لیبل
label, vec = train_bow[0]
print(f" Label: {label}")
print(f" BOW vector shape: {vec.shape}")
print(f" Non-zero entries: {np.count_nonzero(vec)}")
print(f" Sample vector (first 20 dims): {vec[:20]}")
```

بخش خروجی	معنی
`Label: 1`	**(positive review)** این جمله دارای برچسب **مثبت**
`BOW vector shape: (34801,)`	بردار کیسه کلمات این جمله دارای ۳٤ هزار ویژگی است
`Non-zero entries: 47`	جمله دارای ٤٧ واژه از وکب که در بردار کیسه کلمات مقدار ۱ یا بیشتر دارند.
`Sample vector (first 20 dims)`	فقط ۲۰ مقدار اول بردار را نشان داده و این واژه‌ها در جمله حضور نداشتن (عادی است)

- `shape`: نشان می‌دهد که نگاشت تک‌نمودی برای کل واژه‌نامه ساخته شده است.

- تعداد `non-zero` منطقی است (جمله معمولاً ۲۰ تا ۶۰ واژه مهم دارد).

- مقدارهای `0` در اول بردار طبیعیست چون ترتیب واژگان در واژه‌نامه ما بر اساس sort بوده است.

مورد	توضیح
BOW	جمع بردارهای One-hot برای واژه‌های یک جمله است (نماینده کل جمله)
vec2word / E2(w_i)	نگاشت **هر واژه** به یک بردار متراکم (dense) با استفاده از **شبکه عصبی** آموزش‌دیده

- ساخت نگاشت E۱
- ساخت BOW با جمع E۱ ها
- استفاده از BOW برای خوشه‌بندی
- ساخت نگاشت vec۲word = E۲ برای واژگان (با شبکه عصبی)
- استفاده از نگاشت E۲ برای دسته‌بندی و شباهت‌سنجی بین واژه‌ها

۴- خوشه‌بندی:

چرا خوشه‌بندی فقط روی داده آموزش انجام شد؟

برای جلوگیری از نشت اطلاعات (data leakage). تست باید برای ارزیابی باقی بماند، نه آموزش. داده‌های آزمون قرار است برای ارزیابی مدل نهایی استفاده شوند و اگر داده‌های آزمون وارد فرایند خوشه‌بندی شوند، باعث نشت اطلاعات می‌شود .در یادگیری ماشین، همیشه باید تمام عملیات آماری یا پیش‌پردازشی که به داده مربوط می‌شود، فقط روی train set انجام شوند.

چطور بهترین k را پیدا کردیم؟

با silhouette score :

- برای هر k از ۲ تا ۱۰، مدل KMeans اجرا شد.
- silhouette_score(X, labels)محاسبه شد.
- هر چه score بیشتر باشد، خوشه‌بندی بهتر است.
- **بهترین k =** بیشترین امتیاز
- برای هر نمونه، شباهت با خوشه‌ی خودش (a) و با نزدیک‌ترین خوشه‌ی دیگر (b) اندازه‌گیری می‌شود.
- فرمول امتیاز هر نمونه:

$$\frac{b-a}{\max(a,b)} = s$$

مقدار نهایی بین -۱ تا ۱ است؛ هرچه بالاتر بهتر است. امتیاز silhouette: 0.387 یعنی این عدد ۰٫۳۸ نشان می‌دهد که خوشه‌بندی متوسطی بوده‌است.

- نزدیک به **1** خوشه‌بندی خوب
- نزدیک به **0** خوشه‌ها هم‌پوشانی دارن
- **0 <** خوشه‌بندی ضعیف و اشتباه

چرا از PCA استفاده کردیم؟

- برای کاهش ابعاد BOW از ~۳۰۰۰۰ بعد به ۲ بعد، جهت رسم و تجسم خوشه‌ها.

چرا از کاهش یکنواخت نمونه‌ها در هر خوشه استفاده کردیم؟

- چون بعضی خوشه‌ها ممکن است خیلی بزرگ یا خیلی کوچک باشند. برای اینکه مدل بعدی (دسته‌بند) unbalanced نشود، از هر خوشه m_i نمونه با یکنواختی انتخاب کردیم.

$$New\ total\ number\ of\ sentences\ in\ the\ dataset: \quad M = \sum_{i=1}^{k} m_i$$

Mi = تعداد جملات انتخاب‌شده از هر خوشه

روش ۱ (ثابت برای همه خوشه‌ها – ساده و یکنواخت):

max_per_cluster = 5 گرفتن ۵ نمونه از هر کدام، باعث **برابری غیرمنطقی** و تنوع خوشه‌ها را نادیده می‌گیرد.

روش ۲ (نسبت به تعداد اعضای هر خوشه):

```
mi = int(len(clustered_sentences[i]) * 0.1)
```

پس از اجرا Colab کرش کرد : (Your session crashed after using all available RAM.) چرا؟

بردارهای BOW که ساختیم خیلی بزرگ‌اند (مثلاً طول هر بردار ~۳۵,۰۰۰)، و تعداد جمله‌ها هم چند هزار تاست. پس:

```
# X.shape = (12500, 35000) یعنی حدود ۴۰۰+ میلیون
```

و این یعنی حدود 1.4گیگ رم فقط برای ماتریسX و الگوریتم‌هایی مثلKMeans ، PCA یا silhouette روی این ماتریس اجرا می‌شوند، چند برابر RAM مصرف می‌کنند!

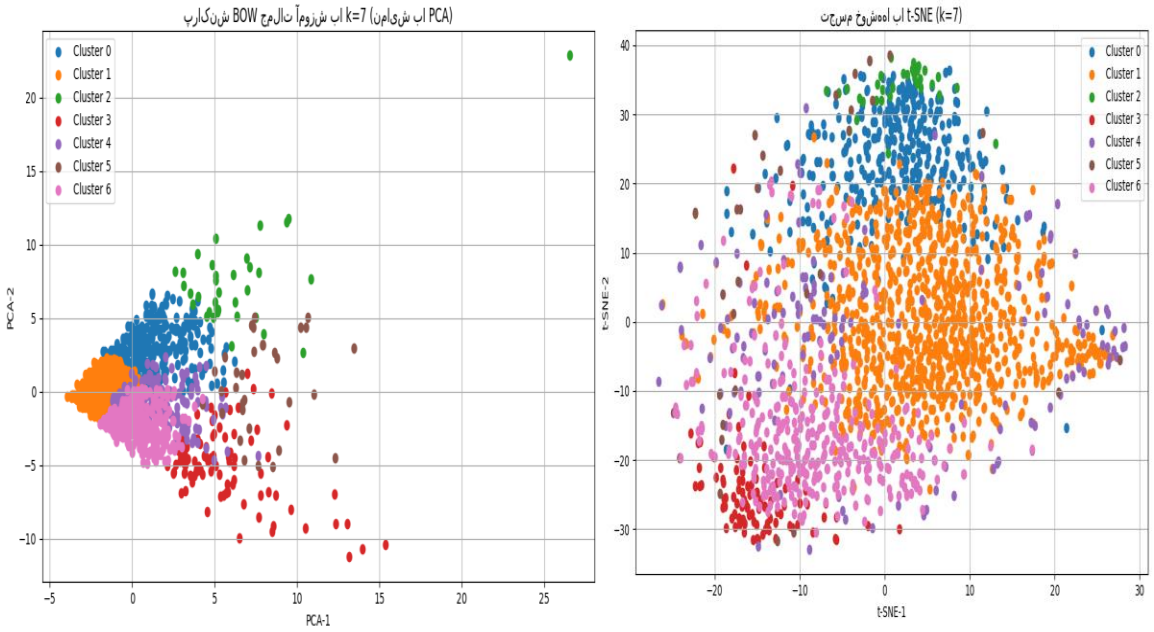
RAMکنترل شده با TruncatedSVD زیرا :

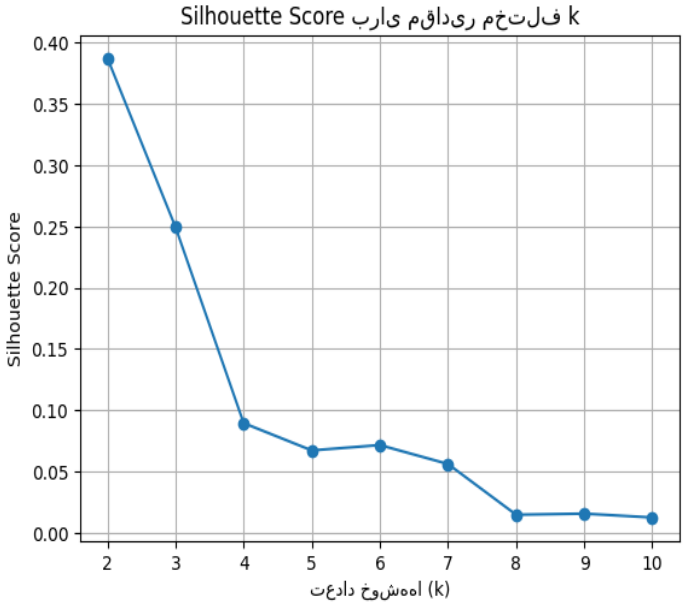
- این روش ابعاد BOW را به ابعاد پایین‌تر فشرده می‌کند، بدون اینکه اطلاعات کلیدی از بین برود.
- مشابه PCA هست ولی مخصوص داده‌های sparse و بزرگ مثلBOW .

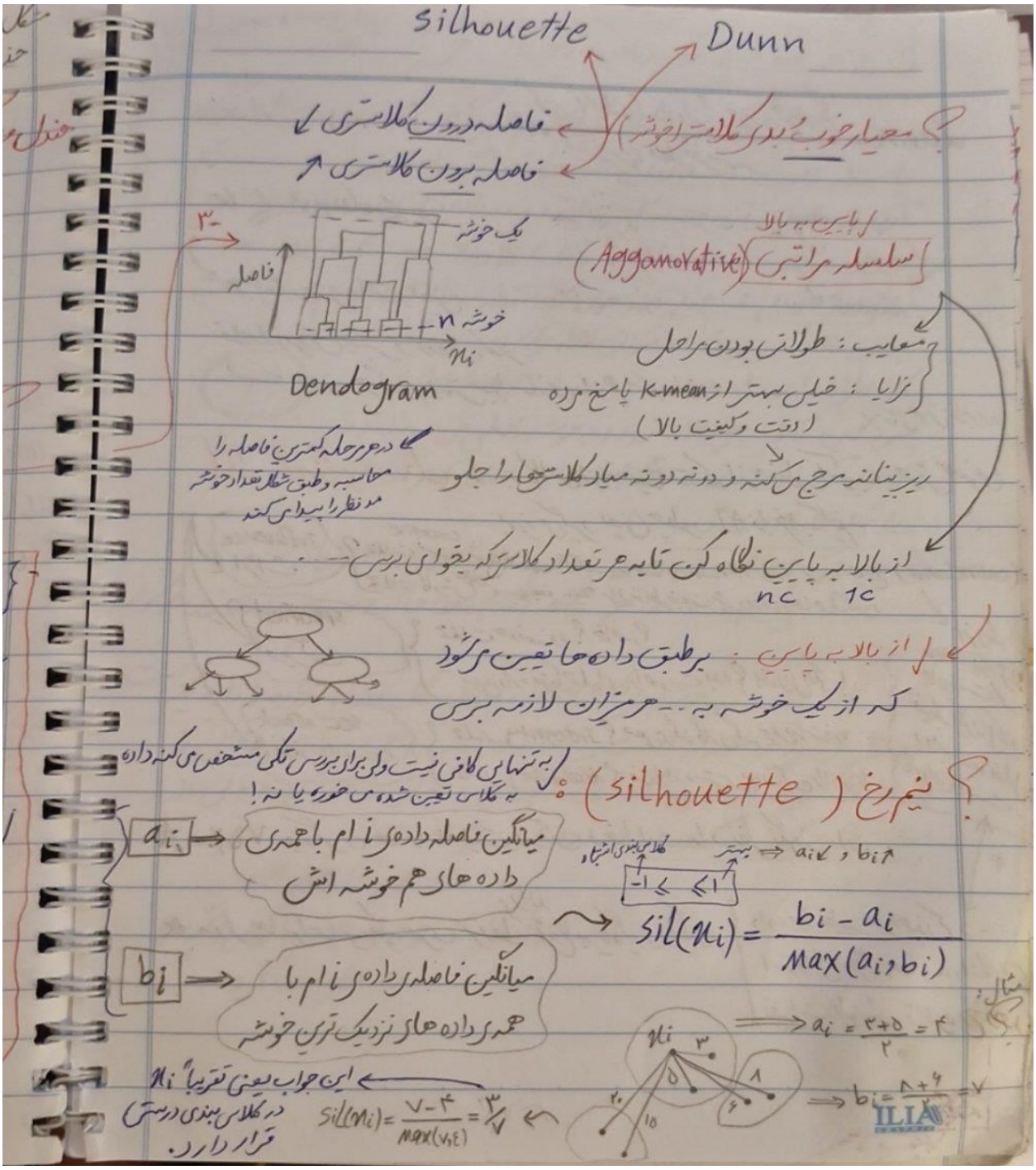
- svd = TruncatedSVD(n_components=100) یعنی به جای ۳۰,۰۰۰ ویژگی، فقط ۱۰۰ ویژگی داریم!

silhouette score برای k=2 بهتر بود زیرا:

- وقتی k کوچک‌تر است، داده‌ها ممکن است بهتر در دو دسته بزرگ (مثلاً مثبت/منفی) تقسیم شوند. در داده‌هایی مثل IMDB (نظرات مثبت و منفی)، داده ذاتاً **دو بخشی** هست(positive vs. negative) .
- k=7 باعث <u>overclustering</u> است (یعنی خوشه‌های بیش‌ازحد ریز که معنی خاصی ندارند).








```
# تابع محاسبه فاصله اقلیدسی
def euclidean_dist(p1, p2):
    return np.sqrt(np.sum((p1 - p2) ** 2))

# پیاده‌سازی دستی معیار سیلوئت
def silhouette_score_manual(X, labels):
    n = len(X)
```

```
    clusters = defaultdict(list)
    for idx, label in enumerate(labels):
        clusters[label].append(idx)

    silhouette_vals = []
    for i in range(n):
        xi = X[i]
        cluster_i = labels[i]
        same_cluster = [j for j in clusters[cluster_i] if j != i]
        a_i = np.mean([euclidean_dist(xi, X[j]) for j in same_cluster]) if
same_cluster else 0

        b_i_list = []
        for other_cid, indices in clusters.items():
            if other_cid == cluster_i:
                continue
            dist = np.mean([euclidean_dist(xi, X[j]) for j in indices])
            b_i_list.append(dist)
        b_i = min(b_i_list) if b_i_list else 0

        s_i = (b_i - a_i) / max(a_i, b_i) if max(a_i, b_i) != 0 else 0
        silhouette_vals.append(s_i)

    return np.mean(silhouette_vals)

# با سیلوئت دستی k یافتن بهترین
silhouette_scores = []
k_range = range(2, 11)
for test_k in k_range:
    km = KMeans(n_clusters=test_k, random_state=42).fit(X_reduced)
    score = silhouette_score_manual(X_reduced, km.labels_)
    silhouette_scores.append((test_k, score))
    print(f"k={test_k}, Silhouette Score={round(score, 4)}")

best_k, best_score = max(silhouette_scores, key=lambda x: x[1])
print(f"\nبهترین k: {best_k}, امتیاز سیلوئت: {round(best_score, 4)}")
```

کد بالا الگوریتم محاسبه ی دستی یافتن بهترین تعداد خوشه برای این مسئله است که در ادامه نسخه‌ی آماده‌ی آن نیز استفاده شد:

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
```

```
from collections import defaultdict
import random

train_bow = train_bow[:2000]

#  استخراج BOWاز ها train_bow
labels, vectors = zip(*train_bow)  # train_bow = [(label, bow_vec), ...]
X = np.array(vectors)  # ماتریس ویژگی‌ها

from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n_components=100, random_state=42)
X_reduced = svd.fit_transform(X)


#   KMeans با k=7
k = 7
kmeans = KMeans(n_clusters=k, random_state=42)
clusters = kmeans.fit_predict(X_reduced)

#   خوشه‌بندی فقط روی داده‌های آموزش انجام
#  تست را وارد کنیم، اطلاعات از آینده نشت پیدا می‌کند (data leakage)
# هدف از خوشه‌بندی، تحلیل یا انتخاب داده برای آموزش مدل بعدی است، نه تست.

#  تعیین بهترین k با silhouette score
silhouette_scores = []
k_range = range(2, 11)
for test_k in k_range:
    km = KMeans(n_clusters=test_k, random_state=42).fit(X_reduced)
    score = silhouette_score(X_reduced, km.labels_)
    silhouette_scores.append((test_k, score))

# نمایش silhouette score
best_k, best_score = max(silhouette_scores, key=lambda x: x[1])

#  تجسم خوشه‌ها با PCA
pca = PCA(n_components=2)
X_2d = pca.fit_transform(X_reduced)

plt.figure(figsize=(10, 6))
for cluster_id in range(k):
    plt.scatter(X_2d[clusters == cluster_id, 0],
                X_2d[clusters == cluster_id, 1],
                label=f"Cluster {cluster_id}")
plt.title("پراکنش BOW جملات آموزش با k=7 (نمایش با PCA)")
```

```
plt.xlabel("PCA-1")
plt.ylabel("PCA-2")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=1000)
X_tsne = tsne.fit_transform(X_reduced)

plt.figure(figsize=(10, 6))
for cluster_id in range(k):
    plt.scatter(X_tsne[clusters == cluster_id, 0],
                X_tsne[clusters == cluster_id, 1],
                label=f"Cluster {cluster_id}")
plt.title("تجسم خوشه‌ها با t-SNE (k=7)")
plt.xlabel("t-SNE-1")
plt.ylabel("t-SNE-2")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()


#  کاهش حجم دادگان: انتخاب یکنواخت از هر خوشه
# تعداد m_i برای هر خوشه = min(5, تعداد اعضای خوشه)
clustered_sentences = defaultdict(list)
for i, vec in enumerate(X_reduced):
    clustered_sentences[clusters[i]].append((labels[i], vec))

def select_uniform(cluster_data, max_per_cluster=5):
    if len(cluster_data) <= max_per_cluster:
        return cluster_data
    return random.sample(cluster_data, max_per_cluster)

reduced_dataset = []
mi_list = []
for cid in clustered_sentences:
    selected = select_uniform(clustered_sentences[cid], max_per_cluster=5)
    reduced_dataset.extend(selected)
    mi_list.append(len(selected))

# خروجی نهایی
```

```
summary = {
    "کل جملات اولیه": len(X_reduced),
    "تعداد خوشه‌ها (k)": k,
    "تعداد جمله‌ها در هر خوشه (mi)": mi_list,
    "کل جملات پس از کاهش": sum(mi_list),
    "بهترین k بر اساس silhouette": best_k,
    "امتیاز silhouette": round(best_score, 4),
}

summary
```

درنهایت از کا برابر با ۲ استفاده شد و برای کاهش حجم دادگان، از هر خوشه به تعداد متناسب با اندازه آن (m_i) جمله انتخاب شد تا ساختار واقعی خوشه‌ها حفظ گردد. کل داده نهایی به مقدار زیر محدود شد:

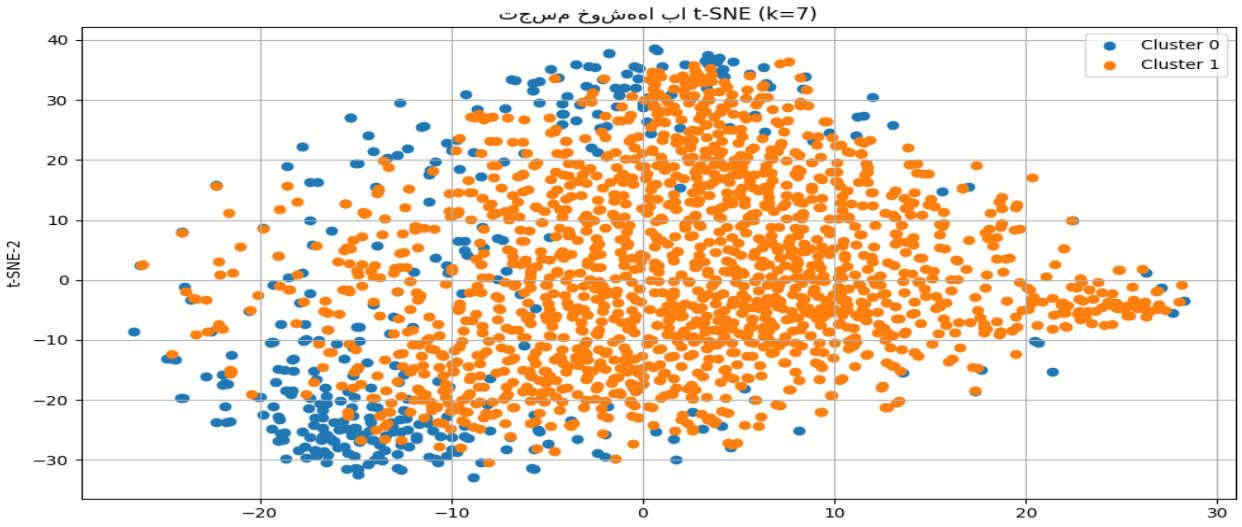
M = len(X_reduced) // ۳ = کل جمله‌های نهایی که می‌خواهیم

$$\left\lfloor M \times \frac{|C_i|}{N} \right\rfloor = m_i$$

Ci= اندازه خوشه‌ی i ام / N = مجموع کل جمله‌ها یعنی len(X_reduced)

Total reduced sentences selected: ۶۶۵

m_i per cluster: {۱: ۵۴۵, ۰: ۱۲۰}



۵- نگاشت (E2) vec2word با مدل Word2Vec گوگل و دسته‌بندی با CNN یاRNN :

نسخه اول(فرضیات):

- خوشه‌بندی با k = 7
- از هر خوشه ۵ جمله ←حجم کاهش‌یافته

نسخه دوم(بهترین‌ها) :

- خوشه‌بندی با k = 2بهترین از نظرsilhouette
- انتخاب جمله‌ها به‌صورت درصدی (تناسبی) از هر خوشه

انتخاب مدل دسته‌بندی:

CNN 1D روی بردارهای توالی:

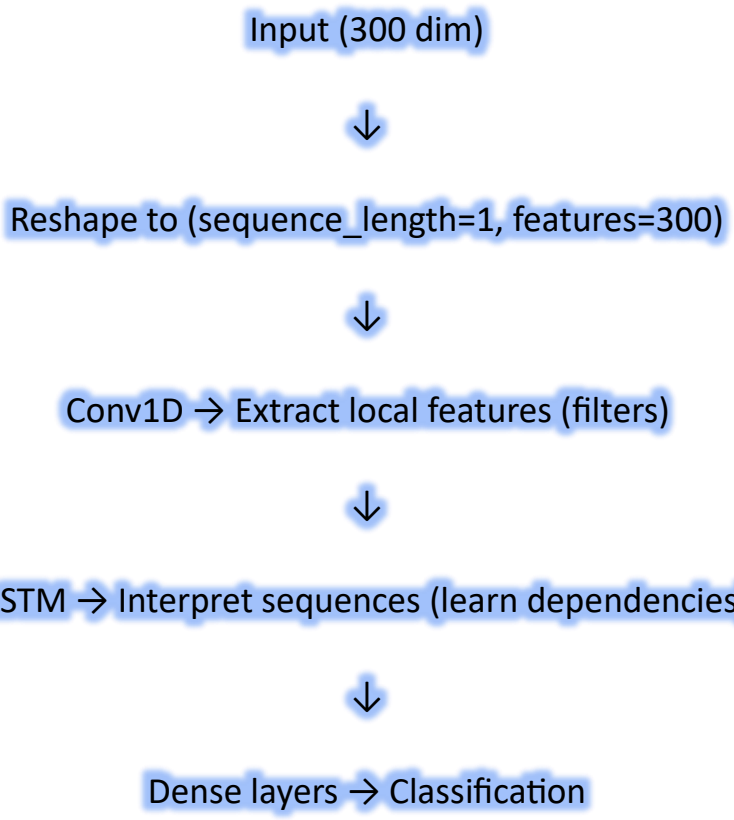
برای جملات padded با طول ثابت

RNNیاLSTM :

مناسب‌تر برای تحلیل ترتیب واژه‌ها

شبکه	قدرت اصلی
CNN	استخراج ویژگی‌های محلی وn-gram ها مثلاً الگوهای کلمه‌ای مثل"not good"
RNN (LSTM)	درک **ترتیب واژه‌ها** و **وابستگی زمانی** در جملات

با ترکیب CNN و LSTM ما هم **الگوهای محلی** و هم **وابستگی طولانی‌مدت** رو پوشش می‌دهیم(اختیاری): ابتدا لایه‌ی کانولوشن روی بردارهای توزیع‌شده اعمال و سپس خروجی به LSTM داده شده تا ترتیب و مفهوم کلی جمله را درک کند.



در این مدل ابتدا با Conv1Dویژگی‌های محلی از توالی بردارهای معنایی استخراج می‌شود، سپس این ویژگی‌ها وارد LSTMمی‌شوند تا ترتیب و ساختار معنایی جمله به‌خوبی تحلیل شود. در نهایت با لایه‌های Dense دسته‌بندی انجام می‌شود. این ترکیب باعث می‌شود مدل هم روی **الگوهای واژگانی** و هم روی **ساختار ترتیبی جمله‌ها** حساس باشد.

فاز A: آموزش نگاشت E_2 = vec2word با شبکه عصبی ساده

یادگیری بردارهای واژه‌ها (embedding) با استفاده از شبکه عصبی.

استفاده از نگاشت E_1 (one-hot) به عنوان ورودی.

پیش‌بینی واژه‌های زمینه (context) به عنوان خروجی.

نگاشت $E_2(w_i)$ همان وزن‌های لایه مخفی شبکه خواهد بود.

بررسی ترکیب برداری مثل:

$$E2(queen)=E2(great)-E2(boring)+E2(recommend)$$

هدف این شبکه، یادگیری نگاشت عددی از واژه‌هاست به‌گونه‌ای که واژگان هم‌معنا یا هم‌جمله‌ای در فضای برداری نزدیک باشند. نگاشت E2 توسط لایه‌ی اول (Dense) یاد گرفته می‌شود. **واژه‌هایی که اغلب کنار هم ظاهر می‌شوند**، در فضای E2 نزدیک خواهند بود. درنهایت مدلی داریم که نگاشت $E1(w_i) \rightarrow E2(w_i)$ را با یک لایه مخفی آموزش بدهد و بتوانیم عملیات‌های معنایی روی آن انجام دهیم.

فاز B :استفاده از نگاشت یادگرفته‌شده برای دسته‌بندی جمله‌ها

هر جمله را به بردار میانگین نگاشت E_2 تبدیل.

آموزش دسته‌بندCNN ، RNNو ترکیبی

حال باید نگاشت E2 را که در پارت سوم توضیح داده شده بود را پیاده سازی کنیم:

```
#  در آموزش نگاشت RAM نسخه بهینه‌شده برای جلوگیری از کرش E2

# --- مرحله ۱: پیش‌پردازش ---
def preprocess(text):
    text = text.lower()
    text = re.sub(r"<.*?>", " ", text)
    text = re.sub(r"[^a-z\s]", " ", text)
    tokens = nltk.word_tokenize(text)
    return [w for w in tokens if w not in stop_words]

train_iter = list(IMDB(split='train'))
train_iter = random.sample(train_iter, len(train_iter) // 2)
train_tokens = [(label, preprocess(text)) for label, text in train_iter]

# حذف واژه‌های نادر
all_tokens = [token for _, tokens in train_tokens for token in tokens]
token_freq = Counter(all_tokens)
def remove_rare_words(tokens, min_freq=2):
    return [token for token in tokens if token_freq[token] >= min_freq]
train_tokens_filtered = [(label, remove_rare_words(tokens)) for label,
tokens in train_tokens]

# --- مرحله ۲: BOW و خوشه‌بندی ---
vocab = sorted(set(token for _, tokens in train_tokens_filtered for token
in tokens))
word2idx = {word: idx for idx, word in enumerate(vocab)}
vocab_size = len(vocab)
```

```
def E1(word):
    vec = np.zeros(vocab_size)
    vec[word2idx[word]] = 1
    return vec

def bow_vector(tokens):
    vec = np.zeros(vocab_size)
    for word in tokens:
        if word in word2idx:
            vec += E1(word)
    return vec

full_bow = [(label, bow_vector(tokens), tokens) for label, tokens in
train_tokens_filtered]
full_bow = full_bow[:2000]

labels, vectors, token_lists = zip(*full_bow)
X = np.array(vectors)

from sklearn.decomposition import TruncatedSVD
X_reduced = TruncatedSVD(n_components=100,
random_state=42).fit_transform(X)

from sklearn.cluster import KMeans
k = 2
clusters = KMeans(n_clusters=k, random_state=42).fit_predict(X_reduced)

# --- مرحله ۳: کاهش درصدی داده ها ---
from collections import defaultdict
clustered_sentences = defaultdict(list)
for i, vec in enumerate(X_reduced):
    clustered_sentences[clusters[i]].append((labels[i], vec,
token_lists[i]))

M_target = 300
cluster_sizes = {cid: len(clustered_sentences[cid]) for cid in
clustered_sentences}
total_size = sum(cluster_sizes.values())

mi_per_cluster = {
    cid: max(5, int(cluster_sizes[cid] * M_target / total_size))
    for cid in clustered_sentences
}
```

```
reduced_dataset = []
reduced_tokens_filtered = []
for cid, data in clustered_sentences.items():
    mi = mi_per_cluster[cid]
    selected = data if len(data) <= mi else random.sample(data, mi)
    for label, vec, tokens in selected:
        reduced_dataset.append((label, vec))
        reduced_tokens_filtered.append((label, tokens))

# --- آموزش نگاشت E2 با vocab محدود مرحله ٤: ---
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# محدودسازی واژگان با فراوانی بالا
token_freq = Counter([token for _, tokens in reduced_tokens_filtered for
token in tokens])
vocab = sorted(set(token for token in token_freq if token_freq[token] >=
5))
word2idx = {w: i for i, w in enumerate(vocab)}
idx2word = {i: w for w, i in word2idx.items()}
vocab_size = len(vocab)

X_train, y_train = [], []
window = 2
for _, tokens in reduced_tokens_filtered:
    for i, target in enumerate(tokens):
        if target not in word2idx:
            continue
        for j in range(max(0, i - window), min(len(tokens), i + window +
1)):
            if i != j and tokens[j] in word2idx:
                X_train.append(to_categorical(word2idx[target],
num_classes=vocab_size))
                y_train.append(to_categorical(word2idx[tokens[j]],
num_classes=vocab_size))

X_train = np.array(X_train)
y_train = np.array(y_train)

embedding_dim = 100
model = Sequential([
    Dense(embedding_dim, input_shape=(vocab_size,), activation='linear'),
    Dense(vocab_size, activation='softmax')
])
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy')
model.fit(X_train, y_train, epochs=30, batch_size=32, verbose=1)

embedding_matrix = model.layers[0].get_weights()[0]
def E2_custom(word):
    return embedding_matrix[word2idx[word]]

# --- مرحله ۵: عملیات برداری ترکیبی ---
def cosine_sim(vec1, vec2):
    return np.dot(vec1, vec2) / (np.linalg.norm(vec1) *
np.linalg.norm(vec2))

def most_similar_custom(vec):
    return sorted(
        [(w, cosine_sim(vec, E2_custom(w))) for w in vocab],
        key=lambda x: -x[1]
    )[:5]
```

ساخت نگاشت E2 یا همان vec2word که:

- به هر **واژه**، یک بردار **عددی** اختصاص که در واقع همان embedding آن واژه است.
- این بردارها به‌گونه‌ای آموزش که **شباهت معنایی واژگان** را در فضا حفظ کنند.
- در واقع، کاری شبیه به **Word2Vec (Skip-gram)** انجام ولی با شبکه عصبی ساده.
- داده ورودی: target word به صورت one-hot
- داده خروجی: context word در یک پنجره ±۲ کلمه، باز هم به صورت one-hot
- مدل :
  - لایه اول، بردار نگاشت E2 را می‌سازد.
  - لایه دوم، احتمال وقوع واژه‌های اطراف را پیش‌بینی می‌کند.
- مدل یاد می‌گیرد که:
  - اگر مثلاً کلمه‌ی "king" در کنار "man" و "royal" و "palace" ظاهر شده، نگاشت برداری‌اش باید به بردار آن‌ها نزدیک باشد.

- به این صورت می‌توانیم مشابهت‌ها رو بعداً با cosine_similarity بسنجیم یا عملیات برداری مثل:

E2("king") - E2("man") + E2("woman") ≈ E2("queen")

- با کاهش هوشمندانه داده‌ها، RAM دیگر کرش نکرده زیرا در کولب با محدودیت سخت افزاری روبرو هستیم و باید هم دیتا را باز هم کاهش داد و هم سخت گیری بیشتری روی حذف نادر ها داشته باشیم و درواقع در دو حالت خوشه بندی ۷ تایی و ۲ تایی تست کرده که البته با تعداد خوشه ۲ تایی مدل خیلی دقیق اورفیت می‌شود و روی دیتای کم به شدت قوی عمل می‌کند که در دنیای واقعی اصلا واقع بینانه نیست.

مدل	لایه‌ها	دلیل استفاده
RNN	LSTM + Dropout + Dense	مناسب برای توالی‌های معنایی
CNN	Conv1D + MaxPool + Dense	مناسب برای تشخیص الگوهای موضعی (n-gram)
ترکیبی	Conv1D → LSTM	اول استخراج ویژگی محلی، سپس وابستگی زمانی

چرا تا اینجا دقت ما اغلب ۱.۰ می‌شود؟

چون مدل روی داده‌ی بسیار کم و بسیار ساده‌ای آموزش می‌بیند، و اکثر موارد نمونه‌ها خیلی تمیز برچسب‌گذاری شدند و این یعنی مدل خیلی راحت می‌تواند همه چیز را حفظ کند، نه اینکه واقعاً یاد بگیرد.

راه‌حل برای واقعی‌تر شدن دقت:

ما دو استراتژی به‌صورت ترکیبی استفاده می‌کنیم:

روش اول: استفاده از بردار میانگین جمله به‌جای توالی برای MLP

```
def sentence_vector_average(tokens):
    vecs = [E2_custom(w) for w in tokens if w in word2idx]
```

```
    if vecs:
        return np.mean(vecs, axis=0)
    else:
        return np.zeros(100)
```

بردار میانگین اطلاعات ترتیب واژه‌ها رو حذف می‌کند و نمای خلاصه‌تری از جمله می‌دهد، که باعث کاهش قدرت مدل و جلوگیری از Overfitting می‌شود.

روش دوم: اعمال Dropout شدیدتر و کاهش تعداد نرون‌ها در شبکه

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

model_mlp = Sequential([
    Dense(64, activation='relu', input_shape=(100,)),
    Dropout(0.5),  # شدیدتر از قبل
    Dense(32, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

چرا کاهش دقت اتفاق افتاد وقتی از میانگین بردار جمله (MLP) استفاده کردیم؟

وقتی توالی بردارها را برای RNN یا CNN به مدل می‌دهیم:

- مدل می‌تواند ساختار زمانی و ترتیبی جمله را یاد بگیرد، اما با داده‌ی کم، مدل خیلی راحت overfit می‌شود.
- مخصوصاً چون توکن‌ها به صورت embedding داده شدند و مدل می‌تواند بر اساس ترتیب آن‌ها مثال‌ها را حفظ کند.

اما در روش **میانگین‌گیری:(Average Embedding)**

- اطلاعات ترتیبی جمله حذف می‌شود.
- فقط معنای کلی جمله (میانگین معنای کلمات) باقی می‌ماند.
- بنابراین مدل باید واقعا یاد بگیرد که «چه برداری» نشان‌دهنده جمله مثبت یا منفی هست و نه اینکه فقط ترتیب یا ساختار حفظ کند.

و چون جمله‌ها **تکراری نیستند**، مدل **overfitنمی‌کند** و واقع‌بینانه یاد می‌گیرد.

آیا می‌توان بهبود بخشید؟

بله! ولی با **این حجم داده و سخت‌افزار محدود**، فقط راه‌های زیر واقع‌بینانه هستند:

: Dropout

برای جلوگیری از یادگیری بیش از حد.

: Data Augmentation

مثل حذف یا جایگزینی تصادفی برخی کلمات غیر مهم (stopwords) برای افزایش تنوع داده.

استفاده ترکیبی از بردار میانگین + بردارmax pooling :

می‌تونه خلاصه بهتری از جمله بسازد.

استفاده از pre-trained embedding مثلWord2Vec :

اگر نگاشت خودمان یعنیE2 دقیق نباشد، Word2Vec می‌تواند مدل را تقویت کند.

مشکلات فعلی

۱. **برچسب‌ها (Labels) نامتوازن یا تکراری:** کلاس‌ها نسبتاً متوازن نیست و یا تنوع در جملات کم است. در نتیجه، مدل‌ها به طور پیش‌فرض کلاس غالب را حدس می‌زنند (مثلاً همیشه کلاس ۱).

۲. **تعداد نمونه‌های آموزش خیلی کم است (کمتر از ۳۰۰ داده):** برای مدل‌های عمیق مثل LSTM و CNN، ۲۰۰–۳۰۰ نمونه شدیداً ناکافی است. این باعث overfitting روی یک مقدار ثابت می‌شود یا مدل هیچ الگوی خاصی را یاد نمی‌گیرد. که نمی‌توان این مقدار را به علت کرش کردن اجرا، انجام داد!

۳. **استفاده از میانگین بردارها یا نگاشت‌های خیلی ساده:** میانگین‌گیری بردار کلمات باعث از بین رفتن ترتیب کلمات می‌شود. این برای RNN و CNN مفید نیست چون آن‌ها برای داده‌های ترتیبی طراحی شده‌اند.

٤. **فشرده‌سازی شدید خوشه‌بندی:** در مرحله‌ی کاهش داده‌ها با خوشه‌بندی، عملاً بیش‌ازحد اطلاعات حذف شده است.

مدل	دقت نهایی روی تست(val_accuracy)
MLP	**0.77** بالاترین دقت
ترکیبی(CNN + LSTM)	0.67
RNN	0.65
CNN	0.60

MLPدقت بالاتری دارد؟ چرا؟

- زیرا از **میانگین بردار کلمات (sentence-level embedding)** برای MLP استفاده کردیم.
- در داده‌های کوچک و بدون ترتیب زمانی پیچیده، این بردار میانگین می‌تواند **خیلی خوب الگوها را خلاصه کند.**
- MLPبا dropout مناسب و class_weight توانسته به‌خوبی تفکیک کند.
- بنابراین دقت بالای MLP در اینجا **قابل قبول** و **واقع‌بینانه** است.

آیا مدل ترکیبی (CNN + LSTM) بهتر از RNN و CNN تنها شده؟

بله، ولی فقط کمی بهتر شده است (۰.۶۷ در برابر ۰.۶۵ و ۰.۶۰)

دلیل:

- **CNN** ویژگی‌های محلی وn-gram ها رو می‌گیره.
- **LSTM** وابستگی ترتیبی رو یاد می‌گیره.
- ترکیب این دو باعث می‌شود مدل **ویژگی‌های محلی و طولانی‌تر** رو همزمان بفهمد.

اما چرا خیلی بهتر نشده؟

- چون داده‌ها کم بوده (در مرحله‌ی خوشه‌بندی فقط ۳۰۰ جمله نگه داشتیم).
- در داده‌های کم، مدل پیچیده مثل Hybrid ممکن است **overfitکند یا به پتانسیل کامل نرسد.**

در همه مدل‌ها به‌خوبی loss کم شده، ولی val_loss در مدل‌های پیچیده مثل hybrid و RNN از یک نقطه شروع به افزایش می‌کند. این یعنی **overfitting** شروع و باید:

- یا EarlyStopping اضافه
- یا Dropout بیشتر
- یا داده بیشتر وارد

نکات مهم در آموزش:

۱- لیبل‌ها باید به درستی از ۲/۱ به ۱/۰ تبدیل شوند :

- در بخش نگاشت E2 یعنی یادگیری بردارهای کلمات با شبکه عصبی: ما **اصلاً از لیبل‌ها برای آموزش استفاده نمی‌کنیم**. زیرا:

```
reduced_tokens_filtered = [(label, tokens)]
```

فقط از قسمت tokens استفاده می‌شوند تا مدل پیش‌بینی کلمه مجاور رو یاد بگیرد.

- استفاده از sparse_categorical_crossentropy به‌جای binary_crossentropy
  - چون مدل خروجی Dense(1, activation='sigmoid') فقط برای برچسب‌های ۰ و ۱ طراحی شده. اما چون داده‌های ما برچسب ۱ و ۲ دارند، باید:

```
y = np.array([1 if label == 2 else 0 for label, _ in reduced_tokens_filtered])
```

بهینه‌ترین کار **تبدیلlabel های ۱ و ۲ به ۰ و ۱** هست. چون هم:

- با معماری فعلی مدل‌ها سازگاراست،
- هم نیاز به تغییر خروجی مدل یا loss function نداریم.

۲- داده‌ها با pad_sequences برای مدل‌های ترتیبی آماده شوند.

۳- از class_weight برای بالانس کردن کلاس‌ها استفاده کرد.

1 . نگاشت Word2Vec آماده نسبت به نگاشت E2

- مدل Google Word2Vec روی حجم عظیمی از داده (Google News) آموزش دیده، بنابراین شباهت‌های زبانی را بسیار دقیق‌تر از نگاشت سفارشی E2 که روی حدود ۳۰۰ نمونه آموزش داده‌ایم، درک می‌کند.
- **واژگان رایج‌تر و عام‌تر در Word2Vec به‌خوبی بازنمایی می‌شوند** ولی ممکن است نسبت به داده‌ی خاص فیلم‌ها، سفارشی‌سازی نشده باشند.

2 . چرا CNN با Word2Vec آماده بهتر شد؟

- چون CNN به‌جای درک ترتیب زمانی، تمرکز روی الگوهای محلی از بردارها دارد و مانند n-gram embeddingهاست.
- Word2Vec با بردارهای معنایی دقیق‌تر، به CNN کمک کرده تا الگوهای قوی‌تری از جملات را بفهمد.
- به همین دلیل، **CNNبا Word2Vec بهترین عملکرد را بین همه مدل‌ها** داشته است.

3 . چرا مدل ترکیبی بهتر از RNN ولی بدتر از CNN است؟

- در داده‌های کوچک (مثل دیتای ۳۰۰ جمله‌ای شما)، شبکه ترکیبی پیچیده‌تر می‌شود و **overfitting** به‌راحتی رخ می‌دهد.
- CNNساده و سریع‌تر است و وقتی کیفیت بردارها بالا باشد مثلWord2Vec ، بهتر از RNN عمل می‌کند.
- در مدل ترکیبی، LSTM ممکن است ویژگی‌هایی را که CNN خوب یاد گرفته، دوباره "از بین ببرد" یا وزن بدهد.

مدل	در چه حالتی بهتر است؟
Word2Vec + CNN	بهترین گزینه برای داده کم با embedding آماده
E2 + MLP	اگر داده بیشتر شود و embedding سفارشی دقیق‌تر شود، عملکرد خوبی دارد
ترکیبی(Hybrid)	فقط در داده‌های بزرگ بهینه است
Word2Vec + RNN	نسبتاً خوب، ولی حساس به ترتیب جملات وnoise

مدل	نگاشت E2 (سفارشی)	Word2Vec آماده
MLP (میانگین)	~0.76	**0.73**
RNN	~0.65	~0.65
CNN	~0.60	**0.75**
Hybrid	~0.67	~0.66

- مدل MLP به دلیل استفاده از **میانگین بردار کلمات**، به ترتیب واژگان حساس نیست. بنابراین برای داده‌های کوچک یا نویزی، عملکرد پایدارتری دارد.
- بردارهای Word2Vec چون روی داده بزرگی آموزش دیده‌اند(Google News) ، **در بازنمایی معنای کلمات دقیق‌تر** هستند و کمک می‌کنند MLP ویژگی‌های مهم‌تری از جمله استخراج کند.

```
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Conv1D, GlobalMaxPooling1D,
Dense, Dropout, MaxPooling1D
from tensorflow.keras.preprocessing.sequence import pad_sequences
# import gensim.downloader as api

# آماده گوگل بارگذاری word2vec
# wv = api.load("word2vec-google-news-300")
EMBED_DIM = 300
MAX_LEN = 50

# ساخت توالی برداری
def sentence_vector_sequence(tokens):
    return [wv[word] for word in tokens if word in wv]

# فرض: reduced_tokens_filtered = [(label, tokens)]
X_seq = [sentence_vector_sequence(tokens) for _, tokens in
reduced_tokens_filtered]
y = np.array([0 if label == 1 else 1 for label, _ in
reduced_tokens_filtered])

# پدینگ توالی‌ها
X_seq_padded = pad_sequences(X_seq, maxlen=MAX_LEN, dtype='float32',
padding='post', truncating='post')

# تقسیم داده ها
X_train, X_test, y_train, y_test = train_test_split(X_seq_padded, y,
test_size=0.2, random_state=42)
```

```
# ------------------------------
# مدل RNN
# ------------------------------
model_rnn = Sequential([
    LSTM(64, input_shape=(MAX_LEN, EMBED_DIM), dropout=0.2,
recurrent_dropout=0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
model_rnn.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_rnn.fit(X_train, y_train, epochs=20, batch_size=32,
validation_data=(X_test, y_test))
print(" دقت RNN:", model_rnn.evaluate(X_test, y_test, verbose=0)[1])

# ------------------------------
# مدل CNN
# ------------------------------
model_cnn = Sequential([
    Conv1D(128, 5, activation='relu', input_shape=(MAX_LEN, EMBED_DIM)),
    GlobalMaxPooling1D(),
    Dropout(0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
model_cnn.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_cnn.fit(X_train, y_train, epochs=20, batch_size=32,
validation_data=(X_test, y_test))
print(" دقت CNN:", model_cnn.evaluate(X_test, y_test, verbose=0)[1])

# ------------------------------
# مدل ترکیبی CNN + LSTM
# ------------------------------
model_hybrid = Sequential([
    Conv1D(64, 5, activation='relu', input_shape=(MAX_LEN, EMBED_DIM)),
    MaxPooling1D(pool_size=2),
    Dropout(0.2),
    LSTM(64, dropout=0.2, recurrent_dropout=0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
```

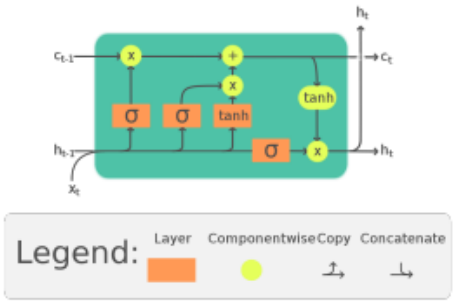
```
])
model_hybrid.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_hybrid.fit(X_train, y_train, epochs=30, batch_size=32,
validation_data=(X_test, y_test))
print(" دقت ترکیبی:", model_hybrid.evaluate(X_test, y_test, verbose=0)[1])

print("************************************************************")
# ارزیابی نهایی
print("\n دقت‌ها:")
print(" دقت RNN:", model_rnn.evaluate(X_test, y_test, verbose=0)[1])
print(" دقت CNN:", model_cnn.evaluate(X_test, y_test, verbose=0)[1])
print(" دقت ترکیبی:", model_hybrid.evaluate(X_test, y_test, verbose=0)[1])
```

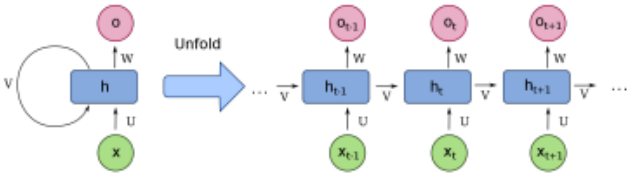
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils import class_weight
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM, Conv1D,
GlobalMaxPooling1D
from tensorflow.keras.preprocessing.sequence import pad_sequences

# فرض: reduced_tokens_filtered = [(label, [tokens])]
# فرض: E2_custom(word) و word2idx از قبل تعریف شده‌اند

MAX_LEN = 50
EMBED_DIM = 100

# --- ساخت توالی و میانگین بردار ---
def sentence_vector_sequence(tokens):
    return [E2_custom(w) for w in tokens if w in word2idx]

def sentence_vector_average(tokens):
    vecs = [E2_custom(w) for w in tokens if w in word2idx]
    return np.mean(vecs, axis=0) if vecs else np.zeros(EMBED_DIM)

# --- آماده‌سازی داده‌ها ---
X_seq = [sentence_vector_sequence(tokens) for _, tokens in
reduced_tokens_filtered]
X_seq_padded = pad_sequences(X_seq, maxlen=MAX_LEN, dtype='float32',
padding='post', truncating='post')
X_avg = np.array([sentence_vector_average(tokens) for _, tokens in
reduced_tokens_filtered])
```

```
# تبدیل label۱→۲ و ۰→۱ های
y = np.array([0 if label == 1 else 1 for label, _ in
reduced_tokens_filtered])

# --- تقسیم داده ---
X_train_seq, X_test_seq, y_train_seq, y_test_seq =
train_test_split(X_seq_padded, y, test_size=0.2, random_state=42)
X_train_avg, X_test_avg, y_train_avg, y_test_avg = train_test_split(X_avg,
y, test_size=0.2, random_state=42)

# --- محاسبه وزن کلاس‌ها برای مقابله با عدم‌تعادل ---
cw_array = class_weight.compute_class_weight(class_weight='balanced',
classes=np.unique(y), y=y)
cw = {0: cw_array[0], 1: cw_array[1]}

# -------------------------
#  مدل MLP
# -------------------------
model_mlp = Sequential([
    Dense(64, activation='relu', input_shape=(EMBED_DIM,)),
    Dropout(0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
model_mlp.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_mlp.fit(X_train_avg, y_train_avg, epochs=20, batch_size=16,
validation_data=(X_test_avg, y_test_avg), class_weight=cw)
print("==========================================")
# -------------------------
#  مدل RNN
# -------------------------
model_rnn = Sequential([
    LSTM(64, input_shape=(MAX_LEN, EMBED_DIM), dropout=0.2,
recurrent_dropout=0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
model_rnn.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_rnn.fit(X_train_seq, y_train_seq, epochs=20, batch_size=32,
validation_data=(X_test_seq, y_test_seq), class_weight=cw)
```

```
print("============================================")
# -------------------------
#  مدل CNN
# -------------------------
model_cnn = Sequential([
    Conv1D(128, 5, activation='relu', input_shape=(MAX_LEN, EMBED_DIM)),
    GlobalMaxPooling1D(),
    Dropout(0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
model_cnn.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_cnn.fit(X_train_seq, y_train_seq, epochs=20, batch_size=32,
validation_data=(X_test_seq, y_test_seq), class_weight=cw)
print("============================================")
# -------------------------
#  مدل ترکیبی Hybrid (CNN + LSTM)
# -------------------------
from tensorflow.keras.layers import MaxPooling1D

model_hybrid = Sequential([
    Conv1D(64, 5, activation='relu', input_shape=(MAX_LEN, EMBED_DIM)),
    MaxPooling1D(pool_size=2),
    Dropout(0.2),
    LSTM(64, dropout=0.2, recurrent_dropout=0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
model_hybrid.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_hybrid.fit(X_train_seq, y_train_seq, epochs=30, batch_size=32,
validation_data=(X_test_seq, y_test_seq), class_weight=cw)
print("============================================")
# -------------------------
#  ارزیابی نهایی
# -------------------------
print("\n دقت‌ها:")
print(" دقت MLP:", model_mlp.evaluate(X_test_avg, y_test_avg,
verbose=0)[1])
print(" دقت RNN:", model_rnn.evaluate(X_test_seq, y_test_seq,
verbose=0)[1])
```

```
print(" دقت CNN:", model_cnn.evaluate(X_test_seq, y_test_seq,
verbose=0)[1])
print(" دقت ترکیبی:", model_hybrid.evaluate(X_test_seq, y_test_seq,
verbose=0)[1])
```





ترکیب شبکه کانولوشن یک‌بعدی و شبکه بازگشتی(CNN + RNN)

چرا ترکیب ؟

در مسائل پردازش زبان طبیعی (NLP) ما نیاز داریم:

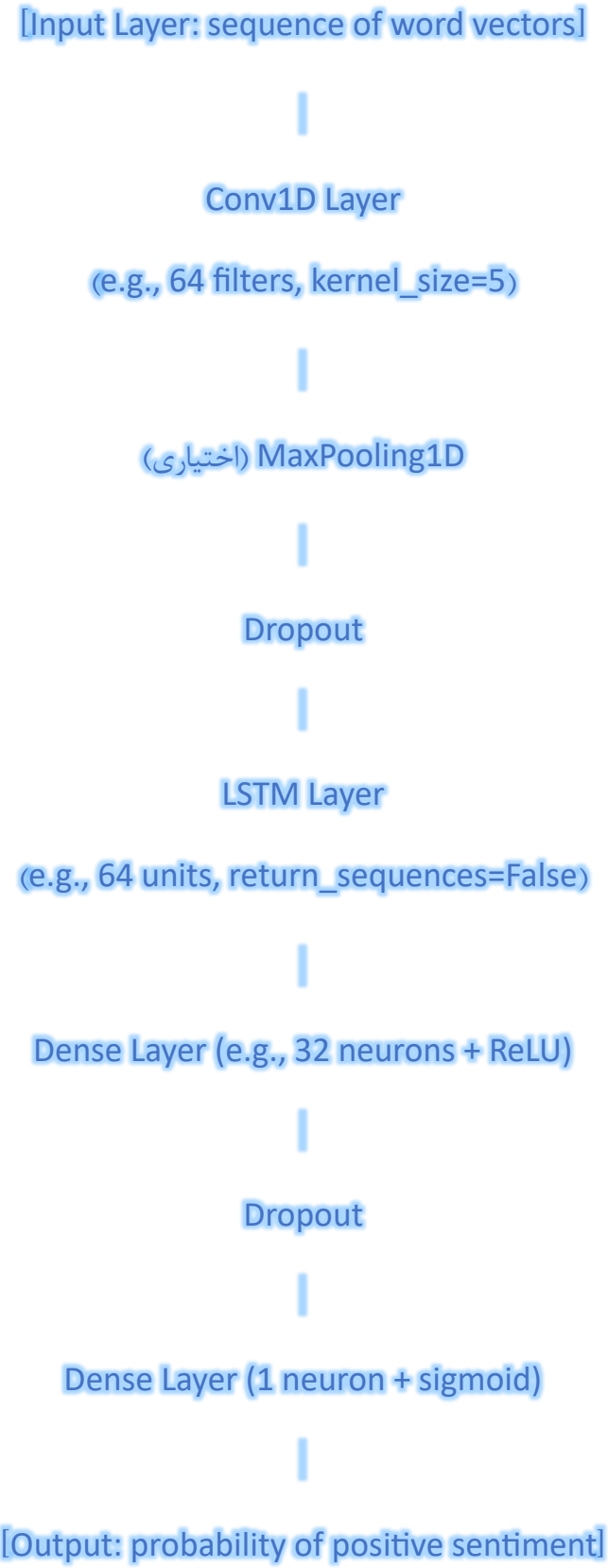
۱. **الگوهای محلی و معنایی بین کلمات** را تشخیص دهیم مثل عبارت "not bad" که در مجموع مثبت است .

۲. **وابستگی‌های زمانی بلندمدت** بین بخش‌های مختلف جمله را در نظر بگیریم (مثل اینکه جمله در ابتدا منفی است اما در انتها مثبت می‌شود).

شبکه‌های **کانولوشن یک‌بعدی (CNN)** برای کشف ویژگی‌های محلی (Local Patterns) مانندN-gram ها بسیار مؤثرند، زیرا با فیلترهای خود، به صورت اسلایدی روی توالی حرکت می‌کنند و ویژگی‌های مهم را استخراج می‌کنند.

شبکه‌های **بازگشتی (RNN)** و به‌ویژه نوعLSTM ، برای درک ساختار ترتیبی داده‌ها (مانند ترتیب کلمات در جمله) و حفظ حافظه بلندمدت مناسب هستند. بنابراین با ترکیب این دو:

- **CNN** به‌عنوان استخراج‌کننده ویژگی عمل می‌کند(feature extractor)
- **RNN** به‌عنوان مدل ترتیبی، وابستگی زمانی را در ویژگی‌های استخراج‌شده یاد می‌گیرد.


[Input Layer: sequence of word vectors]
Conv1D Layer
(e.g., 64 filters, kernel_size=5)
(اختیاری) MaxPooling1D
Dropout
LSTM Layer
(e.g., 64 units, return_sequences=False)
Dense Layer (e.g., 32 neurons + ReLU)
Dropout
Dense Layer (1 neuron + sigmoid)
[Output: probability of positive sentiment]

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, LSTM, Dense, Dropout

model_hybrid = Sequential([
    Conv1D(filters=64, kernel_size=5, activation='relu', input_shape=(MAX_LEN, EMBED_DIM)),
    MaxPooling1D(pool_size=2),
    Dropout(0.2),
    LSTM(64, dropout=0.2, recurrent_dropout=0.2),
    Dense(32, activation='relu'),
    Dropout(0.2),
    Dense(1, activation='sigmoid')
])
```

لایه‌ها:

- Conv1D : نواحی معنایی محلی جمله را شناسایی می‌کند (مانند ترکیب‌های مثبت یا منفی).
- MaxPooling1D : اندازه‌ی خروجی کانولوشن را کاهش داده و ویژگی‌های غالب را انتخاب می‌کند.
- LSTM : توالی ویژگی‌های استخراج‌شده را تحلیل کرده و وابستگی‌های زمانی را یاد می‌گیرد.
- Dense + sigmoid : تصمیم نهایی را در قالب احتمال دوکلاسه (مثبت / منفی) ارائه می‌دهد.

بخش سوم: خوانش مقاله

۱- Attention is All You Need :

هدف این مقاله معرفی یک معماری جدید برای مدل‌های ترجمه ماشینی و مدل‌سازی دنباله‌ها به نام Transformer که فقط بر پایه توجه (Attention) ساخته شده و از RNN و CNN استفاده نمی‌کند.

در مدل‌های رایج sequence-to-sequence مثل RNN, GRU وLSTM ، مسئله اصلی **ماهیت ترتیبی و غیرقابل موازی‌سازی بودن** است که باعث می‌شود آموزش در توالی‌های بلند، پرهزینه و کند باشد. استفاده از ساختاری کاملاً مبتنی بر **self-attention** که اجازه می‌دهد تمام وابستگی‌ها بین کلمات بدون توجه به فاصله‌شان مدل‌سازی شود و **کاملاً قابل موازی‌سازی باشد.**

Transformer از دو بخش تشکیل شده:

- **Encoder Stack** : شامل ۶ لایه تکرارشونده با Self-Attention و Fully Connected
- **Decoder Stack** : ۶ لایه تکرارشونده مشابهEncoder با یک Attention به خروجی‌های انکدر

هر لایه شامل:

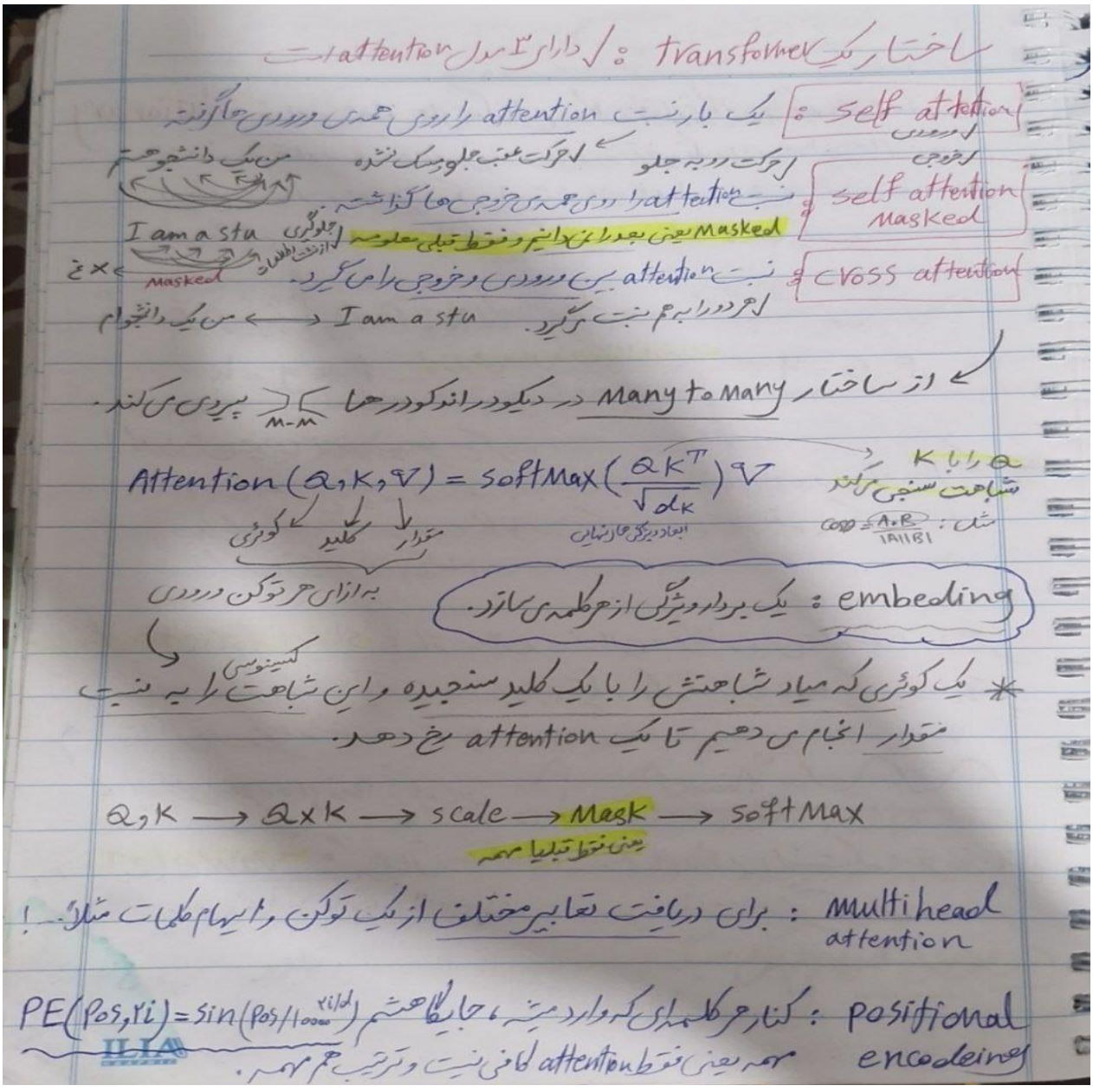
۱. Multi-Head Self-Attention

۲. Feed Forward Layer (FFN)

۳. Residual Connection + Layer Normalization

معادله اصلی:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



: Multi-Head Attention

استفاده از چندین attention head با subspace های متفاوت، که کمک می‌کند مدل اطلاعات را از زوایای مختلف ببیند.

: Position-wise Feed Forward Network

دو لایه Dense که به ازای هر مکان در توالی به‌صورت جداگانه اعمال می‌شود.

: Positional Encoding

چون مدل Recurrent نیست، ترتیب کلمات را با جمع کردن embedding ها با sin/cos encoding به مدل می‌دهد.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

- **Self-Attention** :پیچیدگی مطابق جدول ولی با **موازی‌سازی کامل**
- **RNN** :پیچیدگی مطابق جدول ولی **ترتیبی**
- **CNN** :پیچیدگی وابسته به kernel size ولی طول مسیر بین نقاط دور زیاد است.

آموزش مدل:

- استفاده از **Adam optimizer**
- **Label smoothing**
- **Dropout = 0.1**
- **Warmup steps = 4000**

Transformer از همه مدل‌های قبلی (مثل GNMT, ConvS2S, ByteNet) بهتر عمل کرد.

۲- On Layer Normalization in the Transformer Architecture :

هدف مقاله در این است که چرا در ترنسفورمر استفاده از **warm-up برای نرخ یادگیری** ضروری است و چه زمانی می‌توان آن را حذف کرد. پس تمرکز روی **محل قرارگیری Layer Normalization** یعنی Pre-LN یا Post-LN است و اینکه این محل چه تأثیری روی **پایداری گرادیان‌ها** و سرعت آموزش دارد.

ترنسفورمر شامل لایه‌هایی است که در هر لایه دو زیرساخت دارد:

۱. چندسر Self-Attention

۲. شبکه Fully Connected (FFN)

در هر دو زیرلایه، از اتصال باقی‌مانده (residual) و نرمال‌سازی لایه (LayerNorm) استفاده می‌شود.

Post-LN :

- نرمال‌سازی لایه **بعد از** اتصال باقی‌مانده قرار می‌گیرد.
- در Post-LN ، در ابتدای آموزش، گرادیان لایه‌های انتهایی **خیلی بزرگ** می‌شوند. به همین دلیل نیاز به **warm-up** وجود دارد تا آموزش پایدار شود.
- ساختار کلی:

x → SubLayer → Add(x) → LayerNorm

Pre-LN :

- نرمال‌سازی لایه **قبل از** زیرلایه یعنی قبل از attention یا FFN قرار می‌گیرد.
- در Pre-LN ، گرادیان‌ها پایدار و «well-behaved» هستند. بنابراین می‌توان warm-up را حذف کرد و سرعت آموزش بالا می‌رود.
- ساختار کلی:

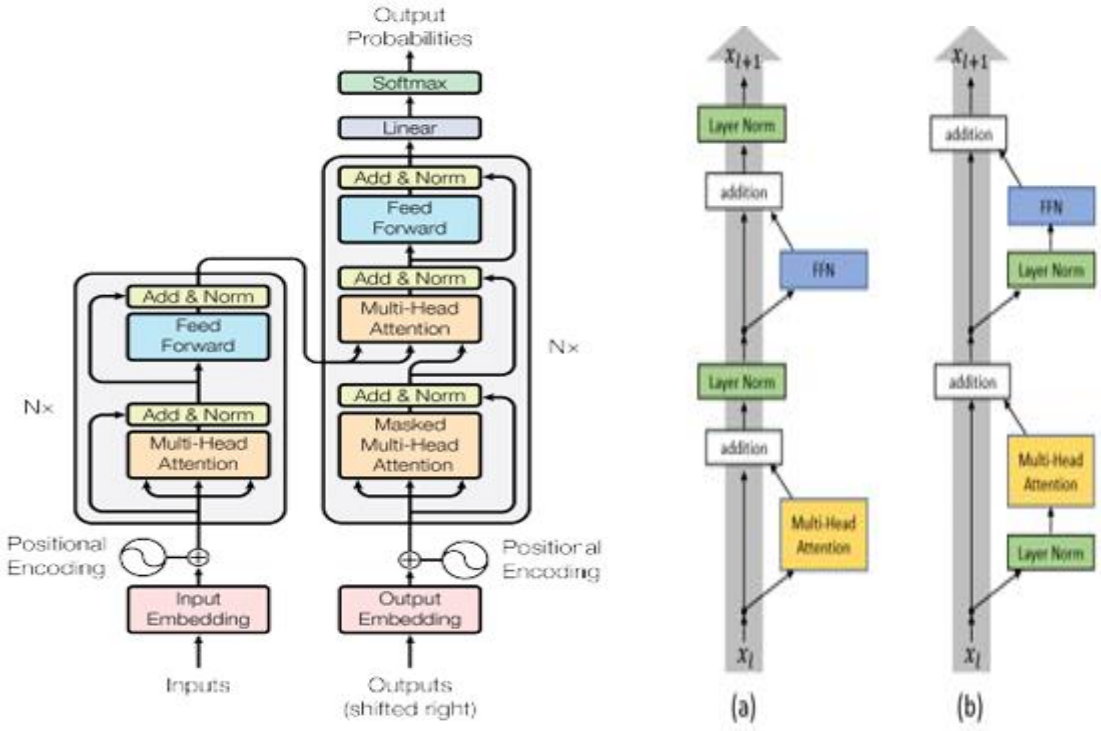
x → LayerNorm → SubLayer → Add(x)

در Pre-LN ، یک LayerNorm نهایی هم در پایان افزوده شده است.(Final LayerNorm)

در چندین وظیفه از جمله:

- ترجمه ماشین(IWSLT14, WMT14)
- پیش‌تمرین BERT
- تسک‌های پایین‌دستی مثل MRPC ، RTE

نتایج نشان دادند که Pre-LN بدون warm-up به همان دقت نهایی می‌رسد ولی سرعت همگرایی بسیار بیشتر است و تعداد پارامترهای تنظیمی کاهش می‌یابد.



در مقاله‌ی اول(NIPS 2017)، **Post-LN** استفاده شده است:

- Self-Attention → Add → LayerNorm
- FFN → Add → LayerNorm

اما مقاله‌ی دوم پیشنهاد می‌کند **LayerNorm را قبل از هر زیرلایه قرار دهیم** (Pre-LN) تا:

- از **نوسانات گرادیان جلوگیری شود و** مدل راحت‌تر و سریع‌تر **همگرا** شود.
- نیازی به **warm-up طولانی** نباشد.

ویژگی‌ها	Post-LN	Pre-LN
جایگاه LayerNorm	بعد از residual	قبل از attention/FFN
نیاز به warm-up	دارد	ندارد
پایداری گرادیان	ناپایدار	پایدار
سرعت همگرایی	کندتر	سریع‌تر
دقت نهایی	مشابه	مشابه یا بهتر

تصویر ترنسفورمر در مقاله Attention is all you need :

شکل ۱ در تمرین، ساختار کلی Transformer را از مقاله اول نشان می‌دهد :

- لایه Attention و FFN به همراه Residual Connection و **سپس** LayerNorm در هر مرحله.
- این معماری از نوع **Post-LN** است.

اما طبق مقاله دوم، در Post-LN چون LayerNorm بعد از residual قرار دارد، مقدار خروجی قبل از Softmax می‌تواند دارای گرادیان‌های بزرگ شود، و این باعث می‌شود در آموزش اولیه به‌خصوص با یادگیری زیاد، مدل واگرا شود. برای جلوگیری از این مشکل، از مرحله Warm-up استفاده می‌شود که نرخ یادگیری به آرامی زیاد می‌شود. حال در Pre-LN، با اینکه وزن‌های اولیه مشابه‌اند، چون LayerNorm قبل از هر زیرلایه قرار دارد، باعث می‌شود مقدار گرادیان به‌صورت کنترل‌شده در هر مرحله حرکت کند. پس هم بدون warm-up قابل آموزش است و هم سریع‌تر به دقت بالا می‌رسد.

تفاوت اصلی این دو ساختار در **محل به‌کارگیری LayerNorm** است که تأثیر قابل‌توجهی در پایداری گرادیان‌ها و کیفیت نهایی مدل دارد.

Encoder Stack سمت چپ:

هر encoder block شامل **۶ لایه تکراری** با ساختار زیر است:

۱. **Input Embedding**

- تبدیل توکن‌ها (کلمات) به بردارهای عددی با اندازه ثابت (مثلاً ۵۱۲ بعدی).
- این embedding برای هر کلمه از جمله استفاده می‌شود.

۲. **Positional Encoding**

- چون attention ترتیب توکن‌ها را در نظر نمی‌گیرد، اطلاعات موقعیت به embedding افزوده می‌شود.
- این مقدار به صورت جمع (element-wise) به Input Embedding اضافه می‌شود. (مطابق عکس از جزوه‌ی نوشته شده)

۳. **N لایه تکراری(Nx)**

هر لایه شامل دو بخش است:

- **Multi-Head Self-Attention**

- هر توکن به تمام توکن‌های دیگر در جمله توجه می‌کند.
- چند "سر" مختلف توجه دارند که اطلاعات متنوعی استخراج شود.(پوشش ایهام و ...)

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

- **Feed Forward**

- دو لایه Dense (یک لایه ReLU بینشان)
- این بخش مستقل از توالی است و به‌صورت position-wise انجام می‌شود.
- **Residual Connection + Layer Normalization (Add & Norm)**
- خروجی هر بلوک با ورودی جمع می‌شود.(residual)
- سپس عملیات normalization انجام می‌شود.

Decoder Stack سمت راست:

ساختاری مشابه Encoder با **سه بخش در هر لایه**:

۱. **Output Embedding + Positional Encoding**

- همانندencoder ، توکن‌های خروجی با موقعیت ترکیب می‌شوند.
- ورودی‌ها "shifted right" هستند تا مدل فقط به توکن‌های قبلی دسترسی داشته باشد که اتفاق مهم برای auto-regression و جلوگیری از نشت اطلاعات است .

۲. **N لایه تکراری(Nx) که** هر لایه شامل سه بلوک است:

- **Masked Multi-Head Self-Attention.**
  - مشابه attention قبلی، اما با *masking* برای جلوگیری از دیدن آینده.
- **Encoder-Decoder Attention.**
  - هر توکن خروجی به تمام توکن‌های encoder توجه می‌کند.
  - اطلاعات متنی را با اطلاعات ورودی ترکیب می‌کند.
- **Feed Forward + Add & Norm.**
  - مانند encoder

Final Linear + Softmax:

خروجی decoder از آخرین لایه به یک لایه Dense داده می‌شود و سپس softmax برای پیش‌بینی احتمالات کلمات بعدی بکار می‌رود.

تفاوت عملکرد:

Post-LN مقاله اول ترنسفورمر

- لایه attention یا feed-forward اجرا می‌شود
- سپس خروجی با ورودی جمع می‌شود
- و سپس روی جمع نهایی LayerNorm انجام می‌شود

مزایا:

- منطبق با معماری اولیه، نتایج تجربی قوی در آن زمان
- برای مدل‌های نسبتاً کوچک پایدار است

معایب:

- در مدل‌های عمیق یا بسیار بزرگ، گرادیان‌ها در backpropagation دچار مشکل می‌شوند
- نیاز شدید به warm-up و تنظیم دقیقlearning rate

Warm-up یعنی:

در ابتدای آموزش، نرخ یادگیری (learning rate) را به‌صورت تدریجی بالا ببریم.

- چون اگر از همان اول نرخ یادگیری بزرگ داشته باشیم، ممکن است مدل به‌دلیل وزن‌های تصادفی اولیه ناپایدار شود.
- مخصوصاً در **Post-LN** که گرادیان‌ها ناپایدارترند، نیاز به warm-up بیشتر حس می‌شود.

برای همین در مقاله (Attention is All You Need) نرخ یادگیری را در چند هزار batch ابتدایی به‌آرامی بالا می‌برد، سپس کاهش می‌دهد.

Pre-LN مقاله دوم On Layer Normalization in the Transformer :

- ابتدا ورودی را نرمال‌سازی می‌کنیم
- سپس زیرلایه attention یا FFN را روی آن اجرا می‌کنیم
- و در آخر residual addition را انجام می‌دهیم.

اتصال باقی‌مانده (Residual Connection) :

اگر یک تابع f(x) باشد، در ترنسفورمر به‌جای فقط اعمال f(x)، خروجی را به فرم زیر می‌نویسند:

$$\mathbf{output} = \mathbf{x} + \mathbf{f(x)}$$

یعنی **ورودی اصلی را حفظ** می‌کنیم و فقط خروجی تابع را به آن اضافه می‌کنیم. تا:

- از **ناپدید شدن گرادیان** در شبکه‌های عمیق جلوگیری می‌شود.
- مسیر یادگیری ساده‌تر و پایدارتر می‌شود.

در ترنسفورمر، این ساختار بارها در attention و feed-forward استفاده می‌شود.

لایه Feed Forward در ترنسفورمر :

این لایه در واقع یک **شبکه عصبی ساده دو لایه** است که روی هر موقعیت (token) به‌طور **مجزا** اعمال می‌شود.

$$\mathbf{output} = \mathbf{max}(۰, \mathbf{xW۱} + \mathbf{b۱})\,\mathbf{W۲} + \mathbf{b۲}$$

$$\mathbf{output} = \mathbf{W۲} \circledast \mathbf{ReLU}(\mathbf{W۱} \circledast \mathbf{x} + \mathbf{b۱}) + \mathbf{b۲}$$

یعنی:

۱. یک لایه خطی (Dense) اعمال می‌شود.

۲. تابع ReLU روی آن زده می‌شود.

۳. دوباره یک لایه Dense دیگر.

این ساختار روی **هر توکن** به‌صورت مستقل (position-wise) اعمال می‌شود و اطلاعات را تقویت یا پالایش می‌کند . پس اگر attention مثل یک سیستم اشتراک‌گذاری اطلاعات بین کلمات باشد، **Feed Forward مثل سیستم "درون‌پردازش" هر کلمه است.**

یعنی:

۱. بردار x توسط W۱ تغییر شکل پیدا می‌کند و از d_model به d_ff می‌رود.

۲. یک **تابع غیرخطی ReLU** اعمال می‌شود و باعث شده تا مدل بتواند ویژگی‌های پیچیده یاد بگیرد

۳. با W۲ به اندازه اولیه (d_model) برمی‌گردد.

٤. نتیجه به لایه بعدی می‌رود.

این مراحل **برای هر توکن جداگانه** انجام شده و هیچ تعامل بین کلمات داخل این لایه نیست! پس اطلاعاتی که attention جمع کرده‌است، توسط FFN **به شکل نهایی تبدیل** می‌شوند.

= **Shift Right : یک گام به جلو پیش‌بینی کردن.**

هنگامی که Decoder درحال پیش بینی **کلمه‌ی بعدی است،** نباید به خودش نگاه کند و تقلب کند. پس در زمان آموزش، ورودی Decoder را یک گام به راست شیفت می‌دهیم.

مزایا:

- آموزش بسیار پایدارتر
- نیاز کمتر به warm-up
- پخش مناسب گرادیان در شبکه‌های عمیق

معایب:

- گاهی نرمال‌سازی اولیه باعث کاهش ظرفیت غیرخطی می‌شود.

در مدل‌های عصبی، **ظرفیت غیرخطی** (Nonlinear Capacity) به توانایی مدل برای یادگیری روابط پیچیده در داده‌ها اشاره دارد. در ساختار **Pre-LN**، لایه LayerNorm **قبل** از اعمال توابع غیرفقط‌خطی مثل attentionیا feed-forward قرار می‌گیرد. این یعنی ورودی‌هایی که می‌رسند، نرمال شده‌اند و نرمال‌سازی ممکن است محدوده‌ی مقدارها را محدود کند. وقتی لایه attention یا feed-forward ورودی محدودی می‌بیند، ممکن است در یادگیری رفتارهای بسیار پیچیده دچار محدودیت شود. به‌همین خاطر Pre-LN ممکن است «ظرفیت غیرخطی» را کاهش دهد که **هنوز قطعی نیست** و بستگی به نوع داده و معماری دارد.