



دانشگاه تهران

دانشکده علوم و فنون نوین

تمرین شماره یک درس پردازش داده‌گان انبوه

نام و نام خانوادگی	فاطمه چیت ساز
شماره دانشجویی	830402092
تاریخ ارسال گزارش	17 خرداد 1403

سوال یک

قسمت اول:

فاصله بین دو توزیع (مانند توزیع گاوسی) در یادگیری ماشین بسیار مهم است زیرا این فاصله اطلاعات مهمی را درباره شباهت یا تفاوت بین دو مجموعه داده فراهم می‌کند. این اطلاعات می‌تواند در تصمیم‌گیری‌ها، بهینه‌سازی مدل‌ها و ارزیابی عملکرد آن‌ها نقش کلیدی داشته باشد. به عبارت دیگر، اندازه‌گیری فاصله بین دو توزیع می‌تواند به ما کمک کند تا بفهمیم که دو مجموعه داده چقدر به هم نزدیک هستند و چقدر می‌توانند به طور مشابه پردازش شوند.

کاربردها در روش‌های یادگیری ماشین

1. Transfer Learning

در انتقال یادگیری، مدل از دانشی که از یک مجموعه داده آموخته شده است برای بهبود عملکرد در یک مجموعه داده جدید استفاده می‌کند. اگر فاصله بین توزیع داده‌های منبع و مقصد کم باشد، احتمال موفقیت انتقال یادگیری بیشتر است. به عبارت دیگر، نزدیکی دو توزیع می‌تواند نشان‌دهنده شباهت بین دو وظیفه باشد، که این شباهت امکان انتقال موفقیت‌آمیز دانش را افزایش می‌دهد.

2. Anomaly Detection

در تشخیص ناهنجاری، مدل تلاش می‌کند نمونه‌هایی را که با توزیع داده‌های نرمال تفاوت دارند شناسایی کند. محاسبه فاصله بین توزیع داده‌های نرمال و توزیع داده‌های جدید می‌تواند به شناسایی ناهنجاری‌ها کمک کند. اگر فاصله بین این دو توزیع زیاد باشد، به احتمال زیاد داده‌های جدید شامل ناهنجاری‌هایی هستند که نیاز به بررسی دارند.

روش برای بدست آوردن فاصله میان دو توزیع:

یکی از روش‌های مشهور برای اندازه‌گیری فاصله بین دو توزیع، فاصله کولبک‌لایبلر است. فاصله KL، میزان اطلاعاتی را که با جایگزینی یک توزیع با توزیع دیگر از دست می‌رود، اندازه‌گیری می‌کند. به عبارت دیگر، این فاصله تفاوت بین دو توزیع احتمال را به صورت نامتقارن نشان می‌دهد.

فرمول فاصله کولبک‌لایبلر به صورت زیر است:

$$D_{KL}(p||q) = \sum_{i=1}^N p(x_i) \cdot \log\left(\frac{p(x_i)}{q(x_i)}\right)$$

در این فرمول، P و Q دو توزیع احتمال هستند که فاصله بین آن‌ها اندازه‌گیری می‌شود.

قسمت دوم :

یکی از روش‌های موثر برای نزدیک کردن دو توزیع به یکدیگر، استفاده از Regularization است. به طور خاص، تکنیک Distribution Matching می‌تواند به این منظور به کار گرفته شود.

شرح روش Distribution Matching

هدف از تطبیق توزیع، کاهش فاصله بین دو توزیع مختلف است. یکی از راه‌های انجام این کار، مینیمم‌سازی واگرایی کولبک‌لایبلر بین دو توزیع است. در این روش، مدل به گونه‌ای آموزش داده می‌شود که توزیع خروجی‌های آن (یا توزیع ویژگی‌های میانی) به توزیع هدف نزدیک‌تر شود.

مراحل کار:

1. تعریف توزیع‌های هدف و مدل:

توزیع هدف P توزیعی است که می‌خواهیم به آن نزدیک شویم.
توزیع مدل Q توزیعی است که مدل ما تولید می‌کند و نیاز به بهینه‌سازی دارد.

2. محاسبه واگرایی کولبکلایبیلر:

واگرایی کولبکلایبیلر بین دو توزیع P و Q محاسبه می‌شود.

فرمول آن به صورت زیر است:

$$D_{KL}(p||q) = \sum_{i=1}^N p(x_i) \cdot \log\left(\frac{p(x_i)}{q(x_i)}\right)$$

3. بهینه‌سازی مدل:

هدف از بهینه‌سازی، کاهش واگرایی کولبکلایبیلر است. این کار با استفاده از الگوریتم‌های بهینه‌سازی مانند گرادیان دیسنت انجام می‌شود.

پارامترهای مدل به گونه‌ای تنظیم می‌شوند که واگرایی کولبکلایبیلر کمینه شود.

4. بروز رسانی مدل:

در هر مرحله از بهینه‌سازی، پارامترهای مدل بروز رسانی می‌شوند تا مدل بتواند توزیع خود را به توزیع هدف نزدیک‌تر کند.

مثال :

یکی از مثال‌های معروف این روش، استفاده از GANs است. در GANها، یک شبکه Generator تلاش می‌کند داده‌هایی تولید کند که شبیه به داده‌های واقعی باشند، و یک شبکه Discriminator تلاش می‌کند تفاوت بین داده‌های واقعی و داده‌های تولید شده را تشخیص دهد. شبکه Generator با هدف کمینه سازی واگرایی کولبکلايبلر یا واگرایی‌های مشابه آموزش می‌بیند

قسمت سوم:

AutoML به فرآیند خودکارسازی تمامی یا اکثر مراحل مختلف فرآیند ساخت و پیاده‌سازی مدل‌های یادگیری ماشین گفته می‌شود. هدف از AutoML کاهش نیاز به تخصص عمیق در یادگیری ماشین و بهینه‌سازی زمان و منابع مورد نیاز برای ایجاد مدل‌های دقیق و کارآمد است. AutoML شامل مراحل اولیه مانند پیش‌پردازش داده‌ها، انتخاب ویژگی‌ها، انتخاب مدل، تنظیم هایپرپارامترها و ارزیابی مدل می‌باشد.

یک روش AutoML در حوزه یادگیری عمیق:

یکی از روش‌های برجسته AutoML در حوزه یادگیری عمیق، Neural Architecture Search است.

NAS به معنای جستجوی خودکار برای یافتن معماری بهینه شبکه‌های عصبی است. این روش به جای طراحی دستی معماری شبکه‌های عصبی توسط متخصصان، از الگوریتم‌های خودکار برای کشف بهترین معماری‌ها استفاده می‌کند. NAS شامل سه بخش اصلی است:

1. Search Space

مجموعه‌ای از تمامی معماری‌های ممکن که NAS می‌تواند از بین آن‌ها انتخاب کند. این فضا شامل اجزای مختلف شبکه عصبی مانند لایه‌ها، نوع اتصال‌ها، و پارامترهای دیگر است.

2. Search Strategy

روش یا الگوریتمی که برای جستجو در فضای جستجو استفاده می‌شود. استراتژی‌های مختلفی مانند جستجوی تصادفی، الگوریتم‌های تکاملی، و یادگیری تقویتی می‌توانند به کار روند.

3. Performance Estimation Strategy

متدی برای ارزیابی عملکرد معماری‌های مختلف در فضای جستجو. این مرحله شامل آموزش و ارزیابی شبکه‌های عصبی کاندید است. برای کاهش هزینه‌های محاسباتی، ممکن است از تکنیک‌هایی مانند کاهش رزولوشن، آموزش با تعداد کمتری از داده‌ها، یا شبیه‌سازی استفاده شود.

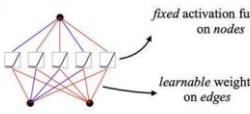
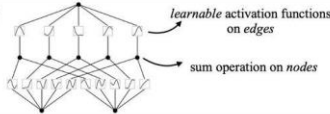
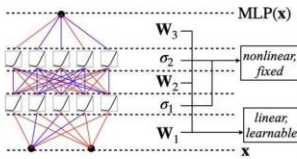
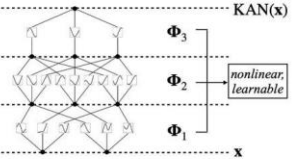
قسمت چهارم:

KAN بر پایه‌ی یک قضیه ریاضی استوار است که توسط دو ریاضی‌دان معروف، کولموگروف و آرنولد، پیشنهاد شده. این قضیه بیان می‌کند که هر تابع پیچیده‌ای را می‌توان به صورت ترکیبی از توابع ساده‌تر تک‌متغیره نمایش داد. حالا MIT آمده و از این قضیه برای ساخت یک شبکه عصبی جدید استفاده کرده که به جای استفاده از توابع ثابت و غیرقابل یادگیری (مثل ReLU یا سیگموئید)، از توابع تک‌متغیره‌ای استفاده می‌کند که در طول فرایند یادگیری بهینه می‌شوند.

چگونه کار می کند؟

برای درک بهتر، بگذارید کمی به ساختار شبکه های عصبی سنتی نگاهی بیندازیم. در شبکه های چند لایه (MLP)، ما ورودی ها را می گیریم، آنها را با وزن ها ضرب می کنیم، یک بایاس اضافه می کنیم و سپس یک تابع فعال سازی اعمال می کنیم. این روش خیلی خوب جواب می دهد، ولی گاهی اوقات نیاز به چیزی پیچیده تر و کارآمدتر داریم.

KAN این روش را تغییر می دهد. در KAN، توابع تک متغیره ای داریم که به عنوان وزن و تابع فعال سازی عمل می کنند. یعنی به جای اینکه فقط وزن ها را یاد بگیریم، این توابع را هم یاد می گیریم. نتیجه؟ شبکه ای که می تواند بهتر و دقیق تر توابع پیچیده را تقریب بزند.

Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(e)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^n \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a) 	(b) 
Formula (Deep)	$\text{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\text{KAN}(\mathbf{x}) = (\Phi_3 \circ \Phi_2 \circ \Phi_1)(\mathbf{x})$
Model (Deep)	(c) 	(d) 

چرا این مهم است؟

کاهش پیچیدگی محاسباتی: به دلیل استفاده از توابع تک متغیره، محاسبات ساده تر و سریع تر انجام می شوند.

سرعت آموزش بالا: بهینه سازی توابع تک متغیره باعث می شود شبکه با سرعت بیشتری آموزش ببیند.

دقت بیشتر: شبکه KAN می تواند با دقت بیشتری مسائل پیچیده را حل کند، چون توابع تک متغیره ای که یاد می گیرد، خیلی خوب با داده ها سازگار می شوند.

سوال دو

$$J = - \sum y \ln(\hat{y}) \quad \boxed{\frac{\partial J}{\partial \hat{y}} = - \sum y \times \frac{1}{\hat{y}}}$$

$$\frac{\partial \hat{y}}{\partial \theta} = \begin{cases} \hat{y}_i (1 - \hat{y}_i) & i = j \\ -\hat{y}_i \hat{y}_j & i \neq j \end{cases}$$

$$\frac{\partial J}{\partial \theta} = - \frac{y_i}{\hat{y}_i} \times \hat{y}_i (1 - \hat{y}_i) + \sum_{i \neq j} - \frac{y_j}{\hat{y}_j} (-\hat{y}_i \hat{y}_j)$$

$$\rightarrow \hat{y}_i - y_i \rightarrow \boxed{\hat{y} - y}$$

مسئله

موضوع

دانش

تاریخ

$$\frac{\partial J}{\partial z_r} = \frac{\partial J}{\partial \theta} \times \frac{\partial \theta}{\partial h_r} \times \frac{\partial h_r}{\partial z_r}$$

$$\Rightarrow (\hat{y} - y) \times 1 \times \text{Relu}'(z_r)$$

$$\begin{cases} z_r > 0 \rightarrow 1 \\ \text{other} \rightarrow 0 \end{cases}$$

مسئله

$$\frac{\partial J}{\partial d} = \frac{\partial J}{\partial \theta} \times \frac{\partial \theta}{\partial d} + \frac{\partial J}{\partial z_r} \times \frac{\partial z_r}{\partial d}$$

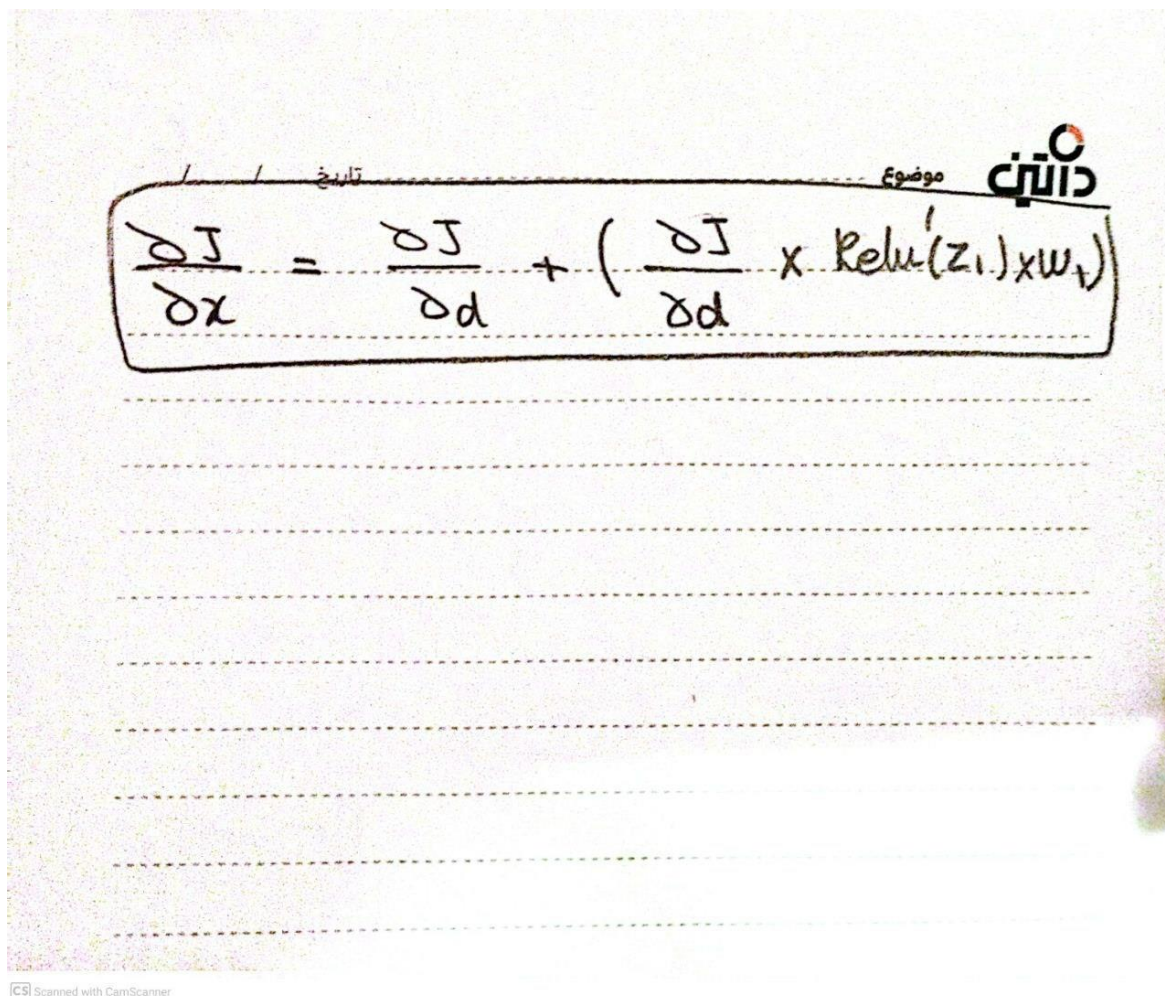
$$\Rightarrow (\hat{y} - y) \times 1 + \frac{\partial J}{\partial z_r} \times W_r$$

مسئله

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial d} \times \frac{\partial d}{\partial x} + \frac{\partial J}{\partial z_1} \times \frac{\partial z_1}{\partial x}$$

$$\delta_x \quad \delta_r \times 1 \quad + \quad \frac{\partial J}{\partial h_1} \times \frac{\partial h_r}{\partial z_1} \times W_1$$

$$\frac{\partial J}{\partial h_1} = \frac{\partial J}{\partial d} \times \frac{\partial d}{\partial h_1} \quad \begin{cases} z_1 > 0 \rightarrow 1 \\ z_1 < 0 \rightarrow 0 \end{cases} \quad \text{Relu}'(z_1)$$



سوال سه

قسمت اول :

می‌خواهیم با مدل Whisper از OpenAI کار کنیم تا به جمله صوتی رو به متن تبدیل کنیم و زبانشو تشخیص بدیم. برای شروع، اول باید این کتابخونه رو نصب کنیم:

```
pip install -U openai-whisper
```

Collecting openai-whisper

Downloading openai-whisper-20231117.tar.gz (798 kB)

0.0/798.6 kB ? eta -:--:--

256.0/798.6 kB 7.5 MB/s eta 0:00:01

798.6/798.6 kB 13.5 MB/s eta 0:00:00

```
Installing build dependencies ... done
```

```
Getting requirements to build wheel ... done
```

```
Preparing metadata (pyproject.toml) ... done
```

```
Requirement already satisfied: triton<3,>=2.0.0 in /usr/local/lib/python3.10/dist-packages
```

```
Requirement already satisfied: numba in /usr/local/lib/python3.10/dist-packages (from d
```

Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from d

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from d
```

```
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from o
```

Requirement already satisfied: more-itertools in /usr/local/lib/python3.10/dist-packages

Collecting tiktoken (from openai-whisper)

```
Downloading tiktoken-0.7.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
```

```
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt)
```

```
Requirement already satisfied: llvmlite<0.42,>=0.41.0dev0 in /usr/local/lib/python3.10
```

```
Requirement already satisfied: regex>=2022.1.18 in /usr/local/lib/python3.10/dist-pack
```

```
Requirement already satisfied: requests>=2.26.0 in /usr/local/lib/python3.10/dist-pack
```

```
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (4.9.0)
```

```
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from ...)
```

حالا که نصب شد، باید بیایم و ازش استفاده کنیم. اول باید این کتابخونه رو وارد کد کنیم:

بعدش باید مدل رو بارگذاری کنیم. من اینجا از مدل پایه استفاده کردم که سبک و

سریع تره

حالا نوبت به فایل صوتی می‌رسه. فرض کنیم یه فایل صوتی داریم به اسم

Recording.wav که توی مسیر /content/ قرار داره. باید آدرس این فایل رو به کد

بدیم:

```
import whisper
```

```
model = whisper.load_model("base")
```

```
audio file = "/content/Recording.wav"
```

[illegible]

خب، حالا این مدل فایل صوتی رو پردازش می‌کنه و به ما می‌گه که زبانش چیه و متنش چی می‌گه:

```
result = model.transcribe(audio_file)

detected_language = result["language"]
transcribed_text = result["text"]

print(f"Detected Language: {detected_language}")
print(f"Transcribed Text: {transcribed_text}")
```

بعد از این که مدل کارش رو انجام داد، ما نتایج رو استخراج می‌کنیم. یعنی زبانی که تشخیص داده رو می‌گیریم و متنی که استخراج کرده رو هم می‌گیریم و چاپ می‌کنیم:

```
Detected Language: en
Transcribed Text: Hello everyone.
```

قسمت دوم :

می‌خواهیم یه فایل صوتی چینی رو بگیریم و متنش رو به انگلیسی ترجمه کنیم. برای این کار از کتابخانه‌های Transformers و Librosa استفاده می‌کنیم.

نصب کتابخانه‌ها

اول باید کتابخانه‌های لازم رو نصب کنیم. دو تا کتابخانه نیاز داریم: یکی برای پردازش صوت (Librosa) و یکی برای مدل‌های زبانی (Transformers).

```
!pip install transformers
!pip install librosa
```

بارگذاری مدل و پردازشگر

ما از مدل Whisper برای ترجمه استفاده می‌کنیم. مدل و پردازشگر رو از پیش‌ساخته‌های whispermedium از OpenAI بارگذاری می‌کنیم:

```
model = WhisperForConditionalGeneration.from_pretrained("openai/whisper-medium")
```

تنظیمات ترجمه به انگلیسی:

باید به مدل بگوییم که می‌خواهیم فایل صوتی چینی رو به انگلیسی ترجمه کنه. این کار رو با forced_decoder_ids انجام می‌دیم:

```
forced_decoder_ids = processor.get_decoder_prompt_ids(language="zh", task="translate")
```

بارگذاری و پردازش فایل صوتی

حالا فایل صوتی چینی مون رو بارگذاری می‌کنیم. فرض کنیم فایل صوتی مون به اسم chinese_hello.mp3 توی مسیر /content/ قرار داره:

```
audio_file = "/content/chinese_hello.mp3"
input_speech, original_sampling_rate = librosa.load(audio_file, sr=None)
```

برای اینکه مدل بهتر کار کنه، باید نمونه‌برداری صوتی رو به 16000 هرتز تغییر بدیم:

```
target_sampling_rate = 16000
input_speech_16k = librosa.resample(input_speech, orig_sr=original_sampling_rate, target_sr=target_sampling_rate)
```

آماده‌سازی ورودی برای مدل

باید ویژگی‌های صوتی رو برای مدل آماده کنیم:


```
input_features = processor(input_speech_16k, sampling_rate=target_sampling_rate, return_tensors="pt").input_features

predicted_ids = model.generate(input_features, forced_decoder_ids=forced_decoder_ids)
```

تولید ترجمه

حالا وقتشه که مدل رو اجرا کنیم تا ترجمه رو تولید کنه:

```
transcription = processor.batch_decode(predicted_ids, skip_special_tokens=True)

print("Translated Text:", transcription[0])
```

خروجی:

Translated Text: Hello.

قسمت سوم :

مزیت‌های یادگیری انتقالی

1. صرفه‌جویی در زمان و منابع:

با استفاده از مدل‌های پیش‌آموزش داده‌شده، نیاز به آموزش مدل‌ها از ابتدا کاهش می‌یابد، که این امر می‌تواند به صرفه‌جویی در زمان و منابع محاسباتی منجر شود.

2. کارایی بهتر با داده‌های کم:

یادگیری انتقالی می‌تواند در شرایطی که داده‌های آموزش جدید کم هستند، نتایج بهتری ارائه دهد. مدل‌هایی که پیش از این روی مجموعه داده‌های بزرگ آموزش دیده‌اند، می‌توانند دانش خود را به مجموعه داده‌های کوچک‌تر انتقال دهند و عملکرد خوبی داشته باشند.

3. بهبود دقت و کارایی:

مدل‌هایی که از یادگیری انتقالی استفاده می‌کنند، معمولاً دقت و کارایی بهتری نسبت به مدل‌هایی که از ابتدا آموزش داده شده‌اند، دارند. این به دلیل این است که مدل از دانش کسب‌شده در مراحل پیشین استفاده می‌کند و بهتر می‌تواند الگوهای پیچیده را شناسایی کند.

دو روش دیگر از یادگیری انتقالی:

1. Feature Extraction

در این روش، از لایه‌های ابتدایی یک مدل پیش‌آموزش داده‌شده به عنوان استخراج‌کننده ویژگی استفاده می‌شود. به عبارت دیگر، از لایه‌های اولیه مدل که به خوبی ویژگی‌های کلی داده‌ها را استخراج می‌کنند، بدون تغییر استفاده می‌شود و تنها لایه‌های انتهایی مدل که مختص به وظیفه جدید هستند، دوباره آموزش داده می‌شوند. مزیت: این روش ساده و سریع است و نیاز به تغییرات کم در مدل پیش‌آموزش داده‌شده دارد.

2. Finetuning

در این روش، کل مدل پیش‌آموزش داده‌شده دوباره آموزش داده می‌شود، اما با نرخ یادگیری کمتر. این کار باعث می‌شود تا وزن‌های مدل با مجموعه داده جدید وفق پیدا کنند و دقت و کارایی بهتری ارائه دهند.

مزیت: این روش انعطاف‌پذیرتر است و می‌تواند عملکرد بهتری در تطبیق مدل با وظیفه جدید داشته باشد.

قسمت چهارم:

میخوایم finetuning کنیم

نصب کتابخانه‌ها:

ترنسفرمر و دیتاست را برای فرایند فاین تون نصب میکنیم

Tqdm هم برای نمایش progress مینصبیم

```
!pip install tqdm
!pip install --upgrade --quiet datasets
!pip install --upgrade --quiet pip
!pip install --upgrade --quiet transformers accelerate evaluate jiwer tensorboard gradio
```

2. بارگذاری داده‌ها:

داده‌ها رو از فایل‌های صوتی و اکسل رو از گوگل درایو دان میکنیم و فایل زیپ رو باز میکنیم . librosa برای خواندن فایل‌های صوتی و pandas برای خواندن

```
!gdown 1cCwH_eoa4Nq17XDHn6e1WfHmdGWPKO

Downloading...
From (original): https://drive.google.com/uc?id=1cCwH\_eoa4Nq17XDHn6e1WfHmdGWPKO
From (redirected): https://drive.google.com/uc?id=1cCwH\_eoa4Nq17XDHn6e1WfHmdGWPKO&confirm=t&uuid=7c8db867-f7cd-4f63-b140-6f1022def4af
to: /content/myaudio_tiny.tar.gz
100% 248M/248M [00:07<00:00, 32.4MB/s]

!tar -xzf "/content/myaudio_tiny.tar.gz" -C "/content/"

myaudio_tiny/
myaudio_tiny/myaudio_tiny.xlsx
myaudio_tiny/myaudio/
myaudio_tiny/myaudio/12440358.wav
myaudio_tiny/myaudio/12560343.wav
myaudio_tiny/myaudio/12560173.wav
myaudio_tiny/myaudio/450625.wav
myaudio_tiny/myaudio/1241467.wav
myaudio_tiny/myaudio/12220095.wav
myaudio_tiny/myaudio/12440241.wav
myaudio_tiny/myaudio/12440435.wav
myaudio_tiny/myaudio/12440093.wav
myaudio_tiny/myaudio/450026.wav
```

3. تقسیم داده‌ها:

داده‌ها را به مجموعه‌های آموزش، ارزیابی و تست تقسیم می‌کنیم. ابتدا 80 درصد برای آموزش و ارزیابی و 20 درصد برای تست. سپس از 80 درصد، 20 درصد برای ارزیابی جدا می‌کنیم:

```
data_path = '/content/myaudio_tiny/'
transcriptions = pd.read_excel('/content/myaudio_tiny/myaudio_tiny.xlsx')

audio_files = []
texts = []
srs = []
for idx, row in transcriptions.iterrows():
    file_path = os.path.join(data_path, row['audio'])
    y, sr = librosa.load(file_path, sr=None)
    srs.append(sr)
    audio_files.append(y)
    texts.append(row['text'])

train_audio, test_audio, train_texts, test_texts, train_srs, test_srs = train_test_split(audio_files, texts, srs, test_size=0.2, random_state=42)
train_audio, val_audio, train_texts, val_texts, train_srs, val_srs = train_test_split(train_audio, train_texts, train_srs, test_size=0.2, random_state=42)
```

نمونه داده :


```
train_audio[0],train_texts[0],train_srs[0]

(array([0.01031494, 0.01544189, 0.01211548, ..., 0.00222778, 0.00375366,
        0.00357056], dtype=float32),
'که بچه ها در آن جا به پشت سر می خوردند و در گوشه کناری آدم را غافلگیر می کردند سرانجام'
16000)
```

ارزیابی اولیه با مدل Whisper:

1. بارگذاری مدل Whisper:

```
model = whisper.load_model("small")
```

از مدل small از Whisper استفاده می کنیم. این مدل از پیش آموزش دیده و برای تبدیل گفتار به متن استفاده می شود:

2. پردازش فایل های صوتی:

```
def process_audio(audio):
    result = model.transcribe(audio)
    return result["text"]

if __name__ == "__main__":
    with mp.Pool(processes=mp.cpu_count()) as pool:
        predictions = list(tqdm(pool.imap(process_audio, test_audio), total=len(test_audio), desc="Processing audio files"))
```

فایل های صوتی تست را با مدل Whisper پردازش می کنیم و متن های پیش بینی رو بدست میاریم بعدش wer را حساب میکنیم

```
error = wer(test_texts, predictions)
print(f"Initial WER: {error * 100:.2f}%")
```

خروجی:

```
Processing audio files: 100%|██████████| 217/217 [1:15:39<00:00, 20.92s/it]
Initial WER: 59.07%
```

مقدار wer خیلی بالاس یعنی مدل زیاد کارش درست نبوده

آموزش مدل Whisper با داده‌های فارسی:

1. آماده‌سازی داده‌ها برای آموزش:

ابتدا داده‌ها را برای آموزش آماده می‌کنیم. از کتابخانه transformers و توکنایزر Whisper استفاده می‌کنیم تا داده‌های صوتی رو بدیم به توکنایزر و فیچر ها رو در بیاریم و لیبل ها که همون متن ما هستند هم با توکنایزر به فرمتی که مدل میفهمه برسونیم و برای این کار اول داده ها رو به dataset تبدیل کرده بعد از map استفاده میکنیم که سریع تر برامون این کارو انجام بده

```
def prepare_dataset(batch):  
  
    # compute log-Mel input features from input audio array  
    batch["input_features"] = feature_extractor(batch["audio"], sampling_rate=batch["sr"]).input_features[0]  
  
    # encode target text to label ids  
    batch["labels"] = tokenizer(batch["text"]).input_ids  
    return batch
```

مپ کردن :

```
processed_train_data = train_dataset.map(prepare_dataset, remove_columns=train_dataset.column_names)  
Map: 100% ██████████ 692/692 [01:04<00:00, 13.88 examples/s]  
  
processed_val_data = val_dataset.map(prepare_dataset, remove_columns=val_dataset.column_names)  
Map: 100% ██████████ 173/173 [00:14<00:00, 11.31 examples/s]
```

2. آموزش مدل:

مدل را با داده‌های فارسی fine-tune می‌کنیم. از Seq2SeqTrainer از کتابخانه transformers استفاده می‌کنیم تا مدل را آموزش بدیم و می‌گیم که پنج تا epoch آموزش بده (بیشتر می‌داشتیم تا صب طول میکشید)

```

training_args = Seq2SeqTrainingArguments(
    output_dir="./results",
    per_device_train_batch_size=16,
    evaluation_strategy="epoch",
    num_train_epochs=5,
    save_strategy="epoch",
    logging_dir="./logs",
    gradient_accumulation_steps=1, # increase by 2x for every 2x decrease in batch size
    learning_rate=1e-5,
    gradient_checkpointing=True,
    fp16=True,
    predict_with_generate=True,
    logging_steps=25,
    report_to=["tensorboard"],
    load_best_model_at_end=True,
    metric_for_best_model="wer",
    greater_is_better=False,
)

```

/usr/local/lib/python3.10/dist-packages/transformers/training_args.py:1474: FutureWarning: `evaluation_strat

ترین مدل :

```

from transformers import Seq2SeqTrainer

trainer = Seq2SeqTrainer(
    args=training_args,
    model=model,
    train_dataset=processed_train_data,
    eval_dataset=processed_val_data,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
    tokenizer=processor.feature_extractor,
)
trainer.train()

```

/usr/local/lib/python3.10/dist-packages/torch/utils/checkpoint.py:464: UserWarning: torch.utils.checkpoint: the use_reentrant parameter should be passed explicitly
warnings.warn(
`use_cache = True` is incompatible with gradient checkpointing. Setting `use_cache = False`...
[220/220 27:13, Epoch 5/5]

Epoch	Training Loss	Validation Loss	Wer
1	0.626700	0.387128	34.218100
2	0.218200	0.329496	29.804925
3	0.106000	0.326968	29.677007
4	0.062200	0.326269	28.845539
5	0.044600	0.330407	29.165334

Some non-default generation parameters are set in the model config. These should go into a GenerationConfig file (https://huggingface.co/docs/transformers/main_classes/generation#transformers.GenerationConfig).

ارزیابی مدل

1. بارگذاری مدل آموزش دیده:

مدل آموزش دیده را بارگذاری می کنیم و آماده ارزیابی می شویم

```

model_path = "/content/results/checkpoint-220"
fine_tuned_model = WhisperForConditionalGeneration.from_pretrained(model_path)

```

3. آماده سازی داده های تست:

داده‌های تست را پردازش می‌کنیم عین داده‌های ترین و val که تبدیلشون کردیم به فیچرهای صدا و لیبل‌ها

```
test_data = [{"audio": audio, "text": text, "sr": sr} for audio, text, sr in zip(test_audio, test_texts, test_srs)]
test_dataset = Dataset.from_list(test_data)
processed_test_data = test_dataset.map(prepare_dataset, remove_columns=test_dataset.column_names)

Map: 100% 217/217 [00:19<00:00, 13.02 examples/s]

test_dataloader = torch.utils.data.DataLoader(
    processed_test_data,
    batch_size=training_args.per_device_train_batch_size,
    collate_fn=data_collator,
    num_workers=4,
)
```

داده‌های تست :

```
processed_test_data

Dataset({
  features: ['input_features', 'labels'],
  num_rows: 217
})

from tadm import tadm
```

حالا این داده‌های آماده شده رو باید بدیم به مدلی که ساختیم و بعد بارگذاری کردیم و بگیم generate کنه بعد خروجی رو decode کنیم و با label های اصلی مقایسه کنیم و wer را اینبار بعد finetune روی داده‌های تست محاسبه کنیم

```

from tqdm import tqdm
def evaluate_model(model, dataloader, processor):
    model.eval()
    predictions = []
    references = []

    for batch in tqdm(dataloader, desc="Evaluating"):
        with torch.no_grad():
            outputs = model.generate(batch["input_features"].to(model.device))
            pred_texts = processor.batch_decode(outputs, skip_special_tokens=True)
            ref_texts = processor.batch_decode(batch["labels"], skip_special_tokens=True)

            predictions.extend(pred_texts)
            references.extend(ref_texts)

    wer = metric.compute(predictions=predictions, references=references)
    return wer

wer_score = evaluate_model(fine_tuned_model, test_dataloader, processor)
print(f"Test WER: {wer_score * 100:.2f}%")

```

Wer روی داده های تست بعد fine tuning

```

Evaluating: 0%|          | 0/14 [00:00<?, ?it/s]/usr/lib/python3.10/multiprocessing/popen_fork.py:66: Runtime
self.pid = os.fork()
Evaluating: 100%|██████████| 14/14 [26:44<00:00, 114.63s/it]Test WER: 26.73%

```

همونطور که میبینیم wer خیلی کمتر شده و خب یعنی مدل اوضاعش بهتر شده

سوال چهار

قسمت اول:

میخوایم ی کد برای تشخیص ژانرهای موسیقی با استفاده از شبکه های LSTM طراحی کنیم

نصب و راه اندازی کتابخانه ها و دانلود داده ها:

```
!pip install -q kaggle

!mkdir -p ~/.kaggle
!cp /content/kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

cp: cannot stat '/content/kaggle.json': No such file or directory
chmod: cannot access '/root/.kaggle/kaggle.json': No such file or directory

! kaggle datasets download andradaolteanu/gtzan-dataset-music-genre-classification

Dataset URL: https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification
License(s): other
Downloading gtzan-dataset-music-genre-classification.zip to /content
 99% 1.20G/1.21G [00:17<00:00, 103MB/s]
100% 1.21G/1.21G [00:17<00:00, 76.3MB/s]

! unzip /content/gtzan-dataset-music-genre-classification.zip
```

Kaggle: نصب کتابخانه Kaggle برای دسترسی به داده‌ها.

کپی فایل JSON: فایل kaggle.json حاوی API Key شماست که برای دانلود داده‌ها از Kaggle استفاده می‌شود.

دانلود و استخراج: داده‌های ژانر موسیقی را از Kaggle دانلود و از حالت فشرده خارج می‌کنیم.

وارد کردن کتابخانه‌های مورد نیاز:

```
import os
import numpy as np
import librosa
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional
```

وارد کردن کتابخانه‌های مورد نیاز شامل numpy برای محاسبات عددی، librosa برای پردازش صوتی، و tensorflow برای ساخت مدل‌های یادگیری عمیق. تنظیمات اولیه:

```
dataset_path = '/content/Data/genres_original'
```

```
genres = ['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz', 'metal', 'pop', 'reggae', 'rock']
```

[+ Code](#)[+ Markdown](#)

```
n_mfcc = 13  
frame_length = 25 # in milliseconds
```

مسیر داده‌ها: مسیر داده‌های ژانرا

ژانرها: لیست ژانرا

تعداد ویژگی‌های MFCC و طول فریم: تنظیمات مربوط به استخراج ویژگی‌ها.

استخراج ویژگی‌های MFCC:

```
def extract_features(file_path):  
    y, sr = librosa.load(file_path)  
    frame_length_samples = int(sr * frame_length / 1000)  
    mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n_mfcc, n_fft=frame_length_samples)  
    return np.mean(mfcc, axis=1)
```

استخراج ویژگی‌ها: تابعی برای استخراج ویژگی‌های MFCC از فایل صوتی. این ویژگی‌ها به ما کمک می‌کنند تا مشخصات صوتی هر ژانر را بدست بیاوریم

خواندن داده‌ها و استخراج ویژگی‌ها:

```
data = []  
labels = []  
  
for genre in genres:  
    genre_path = os.path.join(dataset_path, genre)  
    for file_name in os.listdir(genre_path):  
        try:  
            file_path = os.path.join(genre_path, file_name)  
            features = extract_features(file_path)  
            data.append(features)  
            labels.append(genres.index(genre))  
        except Exception as e:  
            print(f"Error loading {file_path}: {str(e)}")
```

خواندن فایل‌ها: در این قسمت، برای هر ژانر، فایل‌های صوتی مربوطه را می‌خوانیم و ویژگی‌های MFCC آن‌ها را استخراج می‌کنیم.

برچسب‌گذاری: به هر فایل صوتی یک برچسب (label) اختصاص می‌دهیم که نشان‌دهنده ژانر اونه

آماده‌سازی داده‌ها برای آموزش:

آرایه‌های numpy: تبدیل لیست‌ها به آرایه‌های numpy.

```
data = np.array(data)
labels = np.array(labels)
```

```
data.shape
```

```
(999, 13)
```

```
labels.shape
```

```
(999,)
```

کدگذاری برچسب‌ها: برچسب‌های متنی را به اعداد تبدیل می‌کنیم.

```
# Encode labels
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels)
```

```
X_train, X_val, y_train, y_val = train_test_split(data, labels_encoded, test_size=0.2, random_state=42)
```

```
X_train.shape
```

```
(799, 13)
```

تقسیم داده‌ها: داده‌ها را به دو بخش آموزشی (80٪) و ارزیابی (20٪) تقسیم می‌کنیم.


```
# Encode labels
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels)

X_train, X_val, y_train, y_val = train_test_split(data, labels_encoded, test_size=0.2, random_state=42)

X_train.shape

(799, 13)
```

تغییر شکل داده‌ها:

```
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_val = X_val.reshape(X_val.shape[0], X_val.shape[1], 1)

X_train.shape

(799, 13, 1)
```

تغییر شکل داده‌ها: برای استفاده در مدل LSTM، شکل داده‌ها را تغییر می‌دهیم تا هر نمونه دارای ابعاد (تعداد ویژگی‌ها، 1) باشد.

ساخت مدل LSTM

```
model = Sequential([
    LSTM(64, return_sequences=True, input_shape=(n_mfcc, 1)),
    LSTM(64),
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])
```

+ Code

+ Markdown

```
model.summary()
```

ساخت مدل: مدل شامل دو لایه LSTM با 64 نورون، یک لایه Dense با 64 نورون و تابع فعال‌سازی ReLU، یک لایه Dropout برای جلوگیری از overfitting و در نهایت یک لایه Dense با 10 نورون (برای 10 ژانر) و تابع فعال‌سازی Softmax.

ساختار مدل:

Model: "sequential_13"

Layer (type)	Output Shape	Param #
lstm_21 (LSTM)	(None, 13, 64)	16896
lstm_22 (LSTM)	(None, 64)	33024
dense_22 (Dense)	(None, 64)	4160
dropout_11 (Dropout)	(None, 64)	0
dense_23 (Dense)	(None, 10)	650

=====
Total params: 54730 (213.79 KB)
Trainable params: 54730 (213.79 KB)
Non-trainable params: 0 (0.00 Byte)

کامپایل و آموزش مدل:

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),  
              loss='sparse_categorical_crossentropy',  
              metrics=['accuracy'])  
  
history = model.fit(X_train, y_train, epochs=30, batch_size=32, validation_data=(X_val, y_val))  
  
train_accuracy = history.history['accuracy']  
val_accuracy = history.history['val_accuracy']
```

کامپایل مدل: استفاده از بهینه‌ساز Adam و تابع هزینه sparse_categorical_crossentropy

آموزش مدل: مدل را به مدت 30 دوره با batch size برابر 32 آموزش می‌دهیم و در هر دوره دقت آموزش و ارزیابی را گزارش می‌دهیم.

نتایج:

```
Epoch 1: Train Accuracy = 0.39299124479293823, Validation Accuracy = 0.3449999988079071
Epoch 2: Train Accuracy = 0.3767209053039551, Validation Accuracy = 0.3199999928474426
Epoch 3: Train Accuracy = 0.40175220370292664, Validation Accuracy = 0.33000001311302185
Epoch 4: Train Accuracy = 0.400500625371933, Validation Accuracy = 0.3400000035762787
Epoch 5: Train Accuracy = 0.3917396664619446, Validation Accuracy = 0.33000001311302185
Epoch 6: Train Accuracy = 0.4055068790912628, Validation Accuracy = 0.3400000035762787
Epoch 7: Train Accuracy = 0.4030037522315979, Validation Accuracy = 0.35499998927116394
Epoch 8: Train Accuracy = 0.40675845742225647, Validation Accuracy = 0.3499999940395355
Epoch 9: Train Accuracy = 0.41051313281059265, Validation Accuracy = 0.3799999952316284
Epoch 10: Train Accuracy = 0.42803505063056946, Validation Accuracy = 0.35499998927116394
Epoch 11: Train Accuracy = 0.41051313281059265, Validation Accuracy = 0.35499998927116394
Epoch 12: Train Accuracy = 0.4230287969112396, Validation Accuracy = 0.375
Epoch 13: Train Accuracy = 0.42553192377090454, Validation Accuracy = 0.3799999952316284
Epoch 14: Train Accuracy = 0.41051313281059265, Validation Accuracy = 0.38499999046325684
Epoch 15: Train Accuracy = 0.39799749851226807, Validation Accuracy = 0.3799999952316284
Epoch 16: Train Accuracy = 0.43178972601890564, Validation Accuracy = 0.375
Epoch 17: Train Accuracy = 0.4355444312095642, Validation Accuracy = 0.38999998569488525
Epoch 18: Train Accuracy = 0.4392991364002228, Validation Accuracy = 0.3799999952316284
Epoch 19: Train Accuracy = 0.44305381178855896, Validation Accuracy = 0.41499999165534973
Epoch 20: Train Accuracy = 0.45306631922721863, Validation Accuracy = 0.38999998569488525
Epoch 21: Train Accuracy = 0.4493116438388245, Validation Accuracy = 0.38499999046325684
Epoch 22: Train Accuracy = 0.46057572960853577, Validation Accuracy = 0.38999998569488525
Epoch 23: Train Accuracy = 0.4593241512775421, Validation Accuracy = 0.375
Epoch 24: Train Accuracy = 0.4355444312095642, Validation Accuracy = 0.3799999952316284
Epoch 25: Train Accuracy = 0.47309136390686035, Validation Accuracy = 0.4000000059604645
...
Epoch 27: Train Accuracy = 0.4693366587162018, Validation Accuracy = 0.4099999964237213
Epoch 28: Train Accuracy = 0.4655819833278656, Validation Accuracy = 0.4099999964237213
Epoch 29: Train Accuracy = 0.4630788564682007, Validation Accuracy = 0.4050000011920929
Epoch 30: Train Accuracy = 0.4380475580692291, Validation Accuracy = 0.41999998688697815
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings...](#)

عواملی که تاثیر میذاره روی دقت (که الان کمتر از cnn شد :/)

1. کیفیت داده‌ها:

داده‌های نویزی: اگر فایل‌های صوتی دارای نویز باشند، ویژگی‌های MFCC ممکن است دقیق نباشند و مدل نتواند به درستی یاد بگیرد.

2. تنظیمات مدل:

تعداد نوروں‌ها و لایه‌ها: ممکن است تعداد نوروں‌ها و لایه‌های LSTM کم یا زیاد باشد. مثلاً اگر نوروں‌ها کم باشند، مدل قدرت یادگیری کافی ندارد و اگر زیاد باشند، مدل ممکن است دچار overfitting شود.

نرخ یادگیری: نرخ یادگیری (learning rate) ممکن است مناسب نباشد. اگر نرخ یادگیری خیلی کم باشد، مدل به کندی یاد می‌گیرد و اگر زیاد باشد، مدل ممکن است نتواند به درستی بهینه شود.

3. پیش‌پردازش داده‌ها:

ویژگی‌های MFCC: تعداد و تنظیمات ویژگی‌های MFCC ممکن است مناسب نباشد. مثلاً تعداد n_mfcc یا طول فریم ممکن است بهینه نباشد.

4. تنظیمات آموزش:

تعداد epochها: تعداد دوره‌های آموزش (epochs) ممکن است کم یا زیاد باشد. اگر تعداد epochs کم باشد، مدل به درستی یاد نمی‌گیرد و اگر زیاد باشد، مدل overfit می‌شود.

batch size: اندازه batch نیز مهم است. اگر خیلی کوچک باشد، نویز زیادی در گرادینت‌ها داریم و اگر خیلی بزرگ باشد، مدل ممکن است بهینه نشود.

مقایسات:

CNN برای داده‌های صوتی:

CNN بیشتر برای داده‌های تصویری و دو بعدی مناسب است. با این حال، می‌توان از CNN برای داده‌های صوتی نیز استفاده کرد. در این صورت، داده‌های صوتی به صورت تصاویر (spectrogram) تبدیل می‌شوند.

Spectrograms: با تبدیل داده‌های صوتی به تصاویر spectrogram، می‌توان از CNN استفاده کرد که برای شناسایی الگوهای مکانی در تصاویر بسیار مناسب است.

پیش‌پردازش: نیاز به تبدیل داده‌های صوتی به تصاویر spectrogram داریم که ممکن است زمان‌بر باشد ولی نتایج بهتری نسبت به استفاده مستقیم از ویژگی‌های MFCC بدهد.

LSTM برای داده‌های صوتی:

LSTM به دلیل حافظه طولانی‌مدت و کوتاه‌مدت، برای داده‌های ترتیبی مثل صدا و سری‌های زمانی مناسب است. LSTM می‌تواند توالی زمانی را بهتر مدل کند.

توالی زمانی: داده‌های صوتی به صورت توالی زمانی هستند و LSTM برای مدل‌سازی این توالی‌ها بسیار مناسب است.

کارایی :

دقت: بسته به داده‌ها و تنظیمات، یکی از مدل‌ها ممکن است دقت بالاتری داشته باشد. اگر داده‌های صوتی به خوبی پیش‌پردازش شده باشند و به تصاویر spectrogram تبدیل شوند، CNN می‌تواند دقت بالاتری داشته باشد. اما اگر از ویژگی‌های MFCC استفاده کنیم، LSTM معمولاً بهتر عمل می‌کند.

زمان آموزش: CNN ممکن است زمان آموزش بیشتری نیاز داشته باشد به دلیل تعداد پارامترهای بیشتر.

پیچیدگی مدل: LSTM ممکن است مدل پیچیده‌تری باشد به دلیل نیاز به مدل‌سازی توالی‌های زمانی.

قسمت دوم :

سوال: چرا به جای دو لایه LSTM با hidden state 64، از یک لایه LSTM با 128 hidden state استفاده نشده؟

استفاده از دو لایه LSTM با 64 نورون می‌تواند باعث شود مدل ویژگی‌های پیچیده‌تری را یاد بگیرد، زیرا هر لایه می‌تواند الگوهای متفاوتی را استخراج کند. لایه‌های بیشتر می‌توانند توانایی مدل را برای یادگیری روابط طولانی‌مدت افزایش دهند.

مزایا و معایب:

دو لایه LSTM با 64 نورون: این مدل می‌تواند الگوهای پیچیده‌تری را یاد بگیرد ولی احتمالاً زمان بیشتری برای آموزش نیاز دارد.

یک لایه LSTM با 128 نورون: این مدل ممکن است سریع‌تر آموزش ببیند، ولی شاید نتواند به همان دقتی برسد که دو لایه می‌تواند

کد :

همون کد قدیمیم ولی فقط لایه اول را عوض کردیم :

```
model = Sequential()
model.add(LSTM(128, input_shape=(n_mfcc, 1)))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(len(genres), activation='softmax'))
```

ساختار:

```
model.summary()

Model: "sequential_16"

```

Layer (type)	Output Shape	Param #
lstm_26 (LSTM)	(None, 128)	66560
dense_28 (Dense)	(None, 64)	8256
dropout_14 (Dropout)	(None, 64)	0
dense_29 (Dense)	(None, 10)	650

```

Total params: 75466 (294.79 KB)
Trainable params: 75466 (294.79 KB)
Non-trainable params: 0 (0.00 Byte)
```

دقت :

```
Epoch 1: Train Accuracy = 0.1401752233505249, Validation Accuracy = 0.17499999701976776
Epoch 2: Train Accuracy = 0.20150187611579895, Validation Accuracy = 0.19499999284744263
Epoch 3: Train Accuracy = 0.21652065217494965, Validation Accuracy = 0.2800000011920929
Epoch 4: Train Accuracy = 0.2778473198413849, Validation Accuracy = 0.24500000476837158
Epoch 5: Train Accuracy = 0.2803504467010498, Validation Accuracy = 0.26499998569488525
Epoch 6: Train Accuracy = 0.27909886837005615, Validation Accuracy = 0.2750000059604645
Epoch 7: Train Accuracy = 0.28535670042037964, Validation Accuracy = 0.2849999964237213
Epoch 8: Train Accuracy = 0.3078848421573639, Validation Accuracy = 0.2750000059604645
Epoch 9: Train Accuracy = 0.3128911256790161, Validation Accuracy = 0.2949999868697815
Epoch 10: Train Accuracy = 0.2978723347187042, Validation Accuracy = 0.2849999964237213
Epoch 11: Train Accuracy = 0.32665830850601196, Validation Accuracy = 0.28999999165534973
Epoch 12: Train Accuracy = 0.32040050625801086, Validation Accuracy = 0.28999999165534973
Epoch 13: Train Accuracy = 0.3341677188873291, Validation Accuracy = 0.2949999868697815
Epoch 14: Train Accuracy = 0.336670845746994, Validation Accuracy = 0.3050000071525574
Epoch 15: Train Accuracy = 0.32040050625801086, Validation Accuracy = 0.30000001192092896
Epoch 16: Train Accuracy = 0.3404255211353302, Validation Accuracy = 0.3100000023841858
Epoch 17: Train Accuracy = 0.34167709946632385, Validation Accuracy = 0.2849999964237213
Epoch 18: Train Accuracy = 0.34418022632598877, Validation Accuracy = 0.2949999868697815
Epoch 19: Train Accuracy = 0.3466833531856537, Validation Accuracy = 0.3050000071525574
Epoch 20: Train Accuracy = 0.32415518164634705, Validation Accuracy = 0.28999999165534973
Epoch 21: Train Accuracy = 0.3341677188873291, Validation Accuracy = 0.2949999868697815
Epoch 22: Train Accuracy = 0.3729662001132965, Validation Accuracy = 0.2849999964237213
Epoch 23: Train Accuracy = 0.3642052710056305, Validation Accuracy = 0.3050000071525574
Epoch 24: Train Accuracy = 0.3379223942756653, Validation Accuracy = 0.30000001192092896
Epoch 25: Train Accuracy = 0.35419273376464844, Validation Accuracy = 0.3050000071525574
...
Epoch 27: Train Accuracy = 0.37797245383262634, Validation Accuracy = 0.28999999165534973
Epoch 28: Train Accuracy = 0.38047558069229126, Validation Accuracy = 0.3050000071525574
Epoch 29: Train Accuracy = 0.37797245383262634, Validation Accuracy = 0.3100000023841858
Epoch 30: Train Accuracy = 0.3729662001132965, Validation Accuracy = 0.3050000071525574
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

همون طور که حدس میزدیم دقت کم شده

قسمت سوم :

شبکه یک طرفه (Single Directional LSTM): این شبکه تنها از توالی‌های زمانی در

یک جهت (معمولاً از گذشته به آینده) یاد می‌گیرد

شبکه دو طرفه (Bidirectional LSTM): این شبکه از هر دو جهت یاد می‌گیرد،

بنابراین می‌تواند ویژگی‌های بیشتری را از توالی‌های زمانی استخراج کند پس‌س دقت میره

بالا

کدش:

لایه ها رو Bidirectional میکنیم

Model: "sequential_17"

Layer (type)	Output Shape	Param #
bidirectional_4 (Bidirectional)	(None, 13, 128)	33792
bidirectional_5 (Bidirectional)	(None, 128)	98816
dense_30 (Dense)	(None, 64)	8256
dropout_15 (Dropout)	(None, 64)	0
dense_31 (Dense)	(None, 10)	650

=====
Total params: 141514 (552.79 KB)
Trainable params: 141514 (552.79 KB)
Non-trainable params: 0 (0.00 Byte)

نتایج اینگونه میشه :

```
Epoch 1: Train Accuracy = 0.36295369267463684, Validation Accuracy = 0.3050000071525574
Epoch 2: Train Accuracy = 0.36545681953430176, Validation Accuracy = 0.33500000834465027
Epoch 3: Train Accuracy = 0.38297873735427856, Validation Accuracy = 0.3449999988079071
Epoch 4: Train Accuracy = 0.35919898748397827, Validation Accuracy = 0.3100000023841858
Epoch 5: Train Accuracy = 0.37797245383262634, Validation Accuracy = 0.33500000834465027
Epoch 6: Train Accuracy = 0.3767209053039551, Validation Accuracy = 0.3149999976158142
Epoch 7: Train Accuracy = 0.3904881179332733, Validation Accuracy = 0.3449999988079071
Epoch 8: Train Accuracy = 0.3679599463939667, Validation Accuracy = 0.3400000035762787
Epoch 9: Train Accuracy = 0.3917396664619446, Validation Accuracy = 0.33500000834465027
Epoch 10: Train Accuracy = 0.39299124479293823, Validation Accuracy = 0.324999998807907104
Epoch 11: Train Accuracy = 0.38923653960227966, Validation Accuracy = 0.3449999988079071
Epoch 12: Train Accuracy = 0.38047558069229126, Validation Accuracy = 0.324999998807907104
Epoch 13: Train Accuracy = 0.38297873735427856, Validation Accuracy = 0.3449999988079071
Epoch 14: Train Accuracy = 0.4155193865299225, Validation Accuracy = 0.3400000035762787
Epoch 15: Train Accuracy = 0.4155193865299225, Validation Accuracy = 0.324999998807907104
Epoch 16: Train Accuracy = 0.40675845742225647, Validation Accuracy = 0.3449999988079071
Epoch 17: Train Accuracy = 0.40675845742225647, Validation Accuracy = 0.3400000035762787
Epoch 18: Train Accuracy = 0.4205256700515747, Validation Accuracy = 0.3499999940395355
Epoch 19: Train Accuracy = 0.4205256700515747, Validation Accuracy = 0.3400000035762787
Epoch 20: Train Accuracy = 0.42177721858024597, Validation Accuracy = 0.3100000023841858
Epoch 21: Train Accuracy = 0.4155193865299225, Validation Accuracy = 0.3499999940395355
Epoch 22: Train Accuracy = 0.4392991364002228, Validation Accuracy = 0.36000001430511475
Epoch 23: Train Accuracy = 0.41927409172058105, Validation Accuracy = 0.3400000035762787
Epoch 24: Train Accuracy = 0.43178972601890564, Validation Accuracy = 0.36500000953674316
Epoch 25: Train Accuracy = 0.44680851697921753, Validation Accuracy = 0.3799999952316284
...
Epoch 27: Train Accuracy = 0.4718397855758667, Validation Accuracy = 0.375
Epoch 28: Train Accuracy = 0.4543178975582123, Validation Accuracy = 0.3799999952316284
Epoch 29: Train Accuracy = 0.4418022632598877, Validation Accuracy = 0.375
Epoch 30: Train Accuracy = 0.45181477069854736, Validation Accuracy = 0.39500001072883606
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

که اتفاق جالبی افتاده روی ترین بهتر شده اما روی تست بدتر شده یعنی این دو طرفه بودن ماجرا یکم باعث overfit شده

مدل Bidirectional LSTM پیچیده‌تر است و اگر داده‌های آموزشی کافی نباشند یا مدل به درستی regularize نشده باشد، ممکن است دچار overfitting شود، یعنی در داده‌های آموزشی عملکرد خوبی داشته باشد ولی در داده‌های ارزیابی نه.