این پروژه مربوط به **فاین‌تیونینگ مدل FLAN-T5** با استفاده از تکنیک **LoRA (Low-Rank Adaptation)** است. در اصل، مدل FLAN-T5 برای **ترجمه زبان (English to German)** استفاده می‌شود، اما در این پروژه برخی از لایه‌های آن با لایه‌های LoRA جایگزین شده‌اند تا فاین‌تیونینگ آن با پارامترهای کمتری انجام شود.

**خلاصه کارهای انجام‌شده در این پروژه:**

1. **بارگیری داده‌ها:**
   * از **مجموعه داده WMT16** برای ترجمه انگلیسی به آلمانی استفاده شده است.
   * داده‌ها پردازش و به فرمت مناسب برای ورودی مدل تبدیل شده‌اند.
2. **بارگیری و اصلاح مدل FLAN-T5:**
   * از مدل **FLAN-T5-small** که توسط گوگل توسعه یافته، استفاده شده است.
   * برخی از لایه‌های آن با **لایه‌های LoRA** جایگزین شده‌اند تا فاین‌تیونینگ با **پارامترهای کمتر و مصرف حافظه پایین‌تر** انجام شود.
3. **محاسبه تعداد پارامترهای قابل آموزش و غیرقابل آموزش:**
   * مشخص شده که تنها بخش‌هایی از مدل قابلیت یادگیری دارند، درحالی که سایر بخش‌ها ثابت نگه داشته شده‌اند.
4. **کامپایل و آموزش مدل:**
   * مدل با استفاده از **Adam Optimizer** و تابع **SparseCategoricalCrossentropy** آموزش داده شده است.
5. **ارزیابی مدل:**
   * مدل روی مجموعه تست ارزیابی شده و برای بررسی کیفیت ترجمه‌ها از **BLEU Score** استفاده شده است.

**هدف اصلی پروژه:**

✅ کاهش تعداد پارامترهای مورد نیاز برای فاین‌تیونینگ مدل **FLAN-T5** با استفاده از **LoRA**  
✅ بهینه‌سازی حافظه و منابع محاسباتی برای اجرای مدل‌های ترجمه زبان در مقیاس بزرگ

**📌 کاربرد:** این روش برای بهینه‌سازی مدل‌های NLP در **سخت‌افزارهای محدودتر (مثل موبایل، لپ‌تاپ، و سرورهای کوچک‌تر)** بسیار مفید است.

**TensorFlow Addons** به توسعه‌دهندگان اجازه می‌دهد تا از **قابلیت‌های پیشرفته و انعطاف‌پذیر در یادگیری عمیق** استفاده کنند که در نسخه استاندارد TensorFlow موجود نیستند. 🚀

!pip install transformers tensorflow datasets tensorflow\_addons

"""## Load and Preprocess the Dataset"""

from datasets import load\_dataset

# Load the WMT16 English-German dataset

dataset = load\_dataset('wmt16', 'de-en')

**examples یک دیکشنری است که شامل چندین نمونه از دیتاست است.** تابع preprocess\_data این داده‌ها را پردازش کرده و آنها را برای ورود به مدل آماده می‌کند.

# Display an example

print(dataset['train'][0])

import tensorflow as tf

from transformers import AutoTokenizer

# Load the tokenizer

tokenizer = AutoTokenizer.from\_pretrained('google/flan-t5-base')

# Preprocess the dataset for input into the model

def preprocess\_data(examples):

inputs = [f'Translate English to German: {example["en"]}' for example in examples['translation']]

targets = [example['de'] for example in examples['translation']]

model\_inputs = tokenizer(inputs, max\_length=128, truncation=True, padding='max\_length', return\_tensors='tf')

labels = tokenizer(targets, max\_length=128, truncation=True, padding='max\_length', return\_tensors='tf').input\_ids

model\_inputs['labels'] = labels

decoder\_inputs = tokenizer(targets, max\_length=128, truncation=True, padding="max\_length")

model\_inputs["decoder\_input\_ids"] = decoder\_inputs["input\_ids"]

return model\_inputs

train\_dataset = dataset['train'].select(range(20000)).map(preprocess\_data, batched=True)

test\_dataset = dataset['test'].select(range(1000)).map(preprocess\_data, batched=True)

train\_dataset = train\_dataset.to\_tf\_dataset(

columns=['input\_ids', 'attention\_mask', 'decoder\_input\_ids'],

label\_cols=['labels'],

shuffle=True,

batch\_size=128,

collate\_fn=None

)

test\_dataset = test\_dataset.to\_tf\_dataset(

columns=['input\_ids', 'attention\_mask', 'decoder\_input\_ids'],

label\_cols=['labels'],

shuffle=False,

batch\_size=128,

collate\_fn=None

)

"""## Load the Pre-trained FLAN-T5 Model and Modify"""

from transformers import TFAutoModelForSeq2SeqLM, AutoTokenizer

import tensorflow\_addons as tfa

from tensorflow.keras.layers import Dense

# Load the model

model = TFAutoModelForSeq2SeqLM.from\_pretrained('google/flan-t5-small')

کلاس LoRALayer یک **لایه سفارشی** در TensorFlow است که یک **لایه LoRA (Low-Rank Adaptation)** را پیاده‌سازی می‌کند. این لایه جایگزین **لایه‌های چگال (Dense layers)** در یک مدل عصبی می‌شود و هدف آن **کاهش تعداد پارامترهای قابل آموزش در فرآیند Fine-Tuning** است.

🔹 **dense**: این پارامتر یک **لایه Dense موجود** را دریافت می‌کند که می‌خواهیم آن را به LoRA تبدیل کنیم.  
🔹 **rank**: مقدار rank تعداد ابعاد کم‌مرتبه‌ای (Low-rank) را مشخص می‌کند. مقدار کمتر = تعداد پارامتر کمتر = یادگیری کم‌هزینه‌تر.

# Replace the dense layers with LoRA layers

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

def \_\_init\_\_(self, dense, rank=4):

super().\_\_init\_\_()

self.dense = dense

self.rank = rank

**متد build: تعریف وزن‌های جدید برای LoRA**

🔹 w\_a: یک ماتریس وزن با ابعاد **(ورودی, rank)** که ورودی را به فضای کم‌مرتبه می‌برد.  
🔹 w\_b: یک ماتریس وزن با ابعاد **(rank, خروجی Dense)** که داده را دوباره به ابعاد اصلی بازمی‌گرداند.  
✅ در مجموع، این دو وزن، جایگزین **وزن‌های بزرگ Dense Layer اصلی** می‌شوند.

def build(self, input\_shape):

self.w\_a = self.add\_weight(shape=(input\_shape[-1], self.rank),

initializer='random\_normal',

trainable=True, name='w\_a')

self.w\_b = self.add\_weight(shape=(self.rank, self.dense.units),

initializer='random\_normal',

trainable=True, name='w\_b')

متد call: محاسبه خروجی لایه LoRA

🔹 ابتدا **خروجی لایه Dense اصلی** محاسبه می‌شود.  
🔹 سپس **خروجی LoRA** با استفاده از w\_a و w\_b محاسبه می‌شود.  
🔹 در نهایت، **Dense Layer ثابت نگه داشته می‌شود** و فقط وزن‌های جدید LoRA به‌روزرسانی می‌شوند.  
🔹 خروجی نهایی **ترکیبی از Dense اصلی + بهبود LoRA** است.

**جمع‌بندی:**

کلاس LoRALayer به **جای تغییر کل لایه Dense**، یک ماژول **کم‌مرتبه (Low-Rank Adaptation)** روی آن اعمال می‌کند. این روش باعث **افزایش بهره‌وری، کاهش نیاز به حافظه و سرعت بیشتر در Fine-Tuning مدل‌های بزرگ** می‌شود.

def call(self, inputs):

original\_output = self.dense(inputs)

lora\_output = tf.matmul(tf.matmul(inputs, self.w\_a), self.w\_b)

self.dense.trainable = False

return original\_output + lora\_output

🔹 **ستون اول (Layer type)** → نام و نوع هر لایه (مثلاً Dense، LSTM، Conv2D و غیره).  
🔹 **ستون دوم (Output Shape)** → شکل خروجی هر لایه، که نشان می‌دهد داده‌ها در آن لایه چگونه پردازش می‌شوند.  
🔹 **ستون سوم (Param #)** → تعداد پارامترهای قابل یادگیری برای هر لایه.

model.summary()

import tf\_keras

model.get\_layer('encoder').trainable = False

model.get\_layer('shared').trainable = False

model.get\_layer('decoder').trainable = False

model.layers[3] = LoRALayer(model.get\_layer('lm\_head'))

✅ در **Fine-Tuning** مشخص می‌کند که **چند پارامتر مدل در حال یادگیری است**.  
✅ در **LoRA** بررسی می‌کند که **فقط لایه‌های LoRA فعال شده باشند**.  
✅ **بهینه‌سازی حافظه**: می‌توان دید که چقدر حافظه برای یادگیری مدل استفاده می‌شود.

🚀 **جمع‌بندی:** این کد **تعداد کل پارامترهای قابل یادگیری و غیرقابل یادگیری مدل را محاسبه و نمایش می‌دهد** که در تنظیم مدل و بررسی عملکرد آن بسیار مفید است.

"""## Count Trainable and Non-Trainable Parameters"""

trainable\_params = tf.reduce\_sum([tf.reduce\_prod(v.shape) for v in model.trainable\_variables])

non\_trainable\_params = tf.reduce\_sum([tf.reduce\_prod(v.shape) for v in model.non\_trainable\_variables])

print(f'Trainable parameters: {trainable\_params.numpy()}')

print(f'Non-trainable parameters: {non\_trainable\_params.numpy()}')

"""## Compile and Train the Model"""

model.summary()

"""## Train the Model"""

این قطعه کد **مدل را کامپایل می‌کند و سپس آن را روی داده‌های آموزشی آموزش می‌دهد**. بیایید خط به خط بررسی کنیم:

این خط مدل را برای **آموزش آماده می‌کند** و شامل تنظیمات زیر است:

✅ **optimizer=tf\_keras.optimizers.Adam(learning\_rate=1e-3)**  
🔹 از بهینه‌ساز **Adam** برای به‌روزرسانی وزن‌های مدل استفاده می‌شود.  
🔹 **learning\_rate=1e-3** یعنی نرخ یادگیری **0.001** تنظیم شده است.

✅ **loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True)**  
🔹 تابع **خطای Sparse Categorical Crossentropy** برای **مسائل دسته‌بندی (Classification)** استفاده می‌شود.  
🔹 **from\_logits=True** یعنی مدل خروجی **احتمالات Softmax ندارد** و هنوز به softmax تبدیل نشده است.  
(اگر مدل خروجی **احتمالات Softmax** می‌داد، باید from\_logits=False تنظیم می‌شد.)

# Compile the model

model.compile(optimizer=tf\_keras.optimizers.Adam(learning\_rate=1e-3),

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True))

۲. آموزش مدل (model.fit(...))

✅ **train\_dataset** → داده‌های آموزشی را به مدل می‌دهد.  
✅ **validation\_data=test\_dataset** → مدل را روی داده‌های **تست** اعتبارسنجی می‌کند.  
✅ **epochs=3** → آموزش مدل را برای **۳ دور (Epoch)** تکرار می‌کند.

🔹 در هر **epoch**، مدل **همه داده‌های آموزشی** را یک بار پردازش می‌کند و وزن‌هایش را آپدیت می‌کند.  
🔹 پس از هر epoch، **میزان خطا و دقت مدل روی داده‌های اعتبارسنجی محاسبه می‌شود**.

**چرا این کد مهم است؟**

✅ **مدل را برای یادگیری آماده می‌کند** (با تنظیم optimizer و loss function).  
✅ **مدل را روی داده‌های آموزشی تمرین می‌دهد** و وزن‌هایش را به‌روزرسانی می‌کند.  
✅ **عملکرد مدل را روی داده‌های تست بررسی می‌کند** تا ببینیم آیا بیش‌برازش (overfitting) رخ داده یا نه.

🚀 **جمع‌بندی:** این کد **مدل را کامپایل می‌کند و سپس آن را برای ۳ دور روی داده‌های آموزشی تمرین می‌دهد** تا مدل بتواند یاد بگیرد

# Train the model

model.fit(train\_dataset, validation\_data=test\_dataset, epochs=3)

"""## Evaluate the Model"""

# Evaluate the model

model.evaluate(test\_dataset)

"""## Verify Translations with BLEU"""

from nltk.translate.bleu\_score import sentence\_bleu, SmoothingFunction

import nltk

nltk.download('punkt')

# Function to calculate BLEU score

def calculate\_bleu(reference, hypothesis):

reference\_tokens = [nltk.word\_tokenize(reference)]

hypothesis\_tokens = nltk.word\_tokenize(hypothesis)

bleu\_score = sentence\_bleu(reference\_tokens, hypothesis\_tokens, smoothing\_function=SmoothingFunction().method4)

return bleu\_score

# Function to translate and evaluate

def translate\_and\_evaluate(dataset):

bleu\_scores = []

batch = next(iter(dataset))

inputs = batch[0]['input\_ids']

references = batch[1]

outputs = model.generate(inputs, max\_length=128, num\_beams=4, early\_stopping=True)

for i in range(len(inputs)):

reference = tokenizer.decode(references[i], skip\_special\_tokens=True)

hypothesis = tokenizer.decode(outputs[i], skip\_special\_tokens=True)

bleu\_score = calculate\_bleu(reference, hypothesis)

bleu\_scores.append(bleu\_score)

return sum(bleu\_scores) / len(bleu\_scores)

# Evaluate on the validation set

average\_bleu\_score = translate\_and\_evaluate(test\_dataset)

print(f'Average BLEU score on validation set: {average\_bleu\_score}')

A **Dense layer** (Fully Connected Layer) in neural networks connects **every neuron** from the previous layer to **every neuron** in the current layer. It applies a **weight matrix (W)** and **bias (b)** to the input, followed by an **activation function** like ReLU or Softmax.

لایه **Dense** یک **لایه کاملاً متصل (Fully Connected Layer)** است که وزن‌ها و بایاس را روی ورودی اعمال می‌کند و سپس یک تابع فعال‌سازی روی آن اعمال می‌شود.

**۱. بازه‌ی BLEU Score**

* **۰ → ترجمه کاملاً نادرست**
* **۱۰۰ → ترجمه کاملاً بی‌نقص (تقریباً غیرممکن)**
* **معمولاً:**
  + **BLEU < 10** → خیلی ضعیف
  + **10 ≤ BLEU < 30** → قابل قبول، اما بهبود لازم است
  + **30 ≤ BLEU < 50** → ترجمه خوب
  + **BLEU > 50** → ترجمه بسیار خوب

**۲. مقایسه با ترجمه‌های انسانی**

* **ترجمه‌های انسانی معمولاً BLEU بین ۳۰ تا ۷۰ دارند.**
* **مدل‌های قوی (مانند Google Translate) BLEU حدود ۴۰-۶۰ دارند.**
* **BLEU بالای ۷۰ معمولاً به دلیل یادگیری نادرست یا تکرار زیاد کلمات است.**

**۳. بهتر کردن BLEU Score**

✅ بهبود کیفیت داده‌های آموزشی  
✅ استفاده از مدل‌های قوی‌تر (مثل FLAN-T5، mT5)  
✅ استفاده از تکنیک‌هایی مثل **Beam Search** یا **LoRA Fine-Tuning**

🚀 **نتیجه:** BLEU معمولاً **بین ۳۰ تا ۵۰ یک امتیاز خوب** محسوب می‌شود، اما باید با ترجمه‌های انسانی مقایسه شود.

4o