# LMSYS - Chatbot Arena Human Preference Predictions

## Overview

这项比赛要求您在由大型语言模型 (LLM) 驱动的聊天机器人之间的正面对决中预测用户会喜欢哪些回答。您将获得来自 Chatbot Arena 的对话数据集，其中不同的 LLM 会生成对用户提示的回答。通过开发成功的机器学习模型，您将帮助改善聊天机器人与人类的互动方式，并确保它们更好地符合人类的偏好。

## Evaluation

提交的内容将根据预测概率和真实值之间的对数损失进行评估。

*Log Loss，也称为对数损失或交叉熵损失（Cross-Entropy Loss），是一种在机器学习中用于评估分类模型性能的损失函数，特别是在处理多类分类问题时。它衡量的是模型预测的概率分布与真实标签的概率分布之间的差异*。

对于一个给定的样本，假设其真实标签为 ( y )，模型预测的概率分布为 ( p )，Log Loss 可以通>过以下公式计算：

$$\text{Log Loss} = -\frac{1}{N} \sum\_{i=1}^{N} \sum\_{j=1}^{M} y\_{ij} >\log(p\_{ij}) \$$

参数解释:

* 是样本的总数。
* 是类别的总数。
* 是一个二进制指示器（0或1），如果样本 属于类别 ，则为1，否则为0。
* 是模型预测样本 属于类别 的概率。
* 通常是以自然对数为底的对数。

Log Loss 能够给予错误分类更高的惩罚，特别是当预测概率远离真实标签时。这促使模型生成更准确的预测概率，而不仅仅是预测正确的类别。Log Loss 常用于逻辑回归和神经网络等模型的优化中。

## Dataset Description

竞赛数据集由 ChatBot Arena 中的用户交互组成。在每次用户交互中，评委向两个不同的大型语言模型提供一个或多个提示，然后指出哪个模型给出了更令人满意的响应。竞赛的目标是预测评委的偏好，并确定给定的提示/响应对被选为获胜者的可能性。请注意，这是一场代码竞赛。当您的提交被评分时，此示例测试数据将替换为完整测试集。训练数据中有 55K 行，测试集中大约有 25,000 行。

### Files

**train.csv**

* id - A unique identifier for the row.
* model\_[a/b] - The identity of model\_[a/b]. Included in train.csv but not test.csv.
* prompt - The prompt that was given as an input (to both models).
* response\_[a/b] - The response from model\_[a/b] to the given prompt.
* winner\_model\_[a/b/tie] - Binary columns marking the judge's selection. The ground truth target column.

**test.csv**

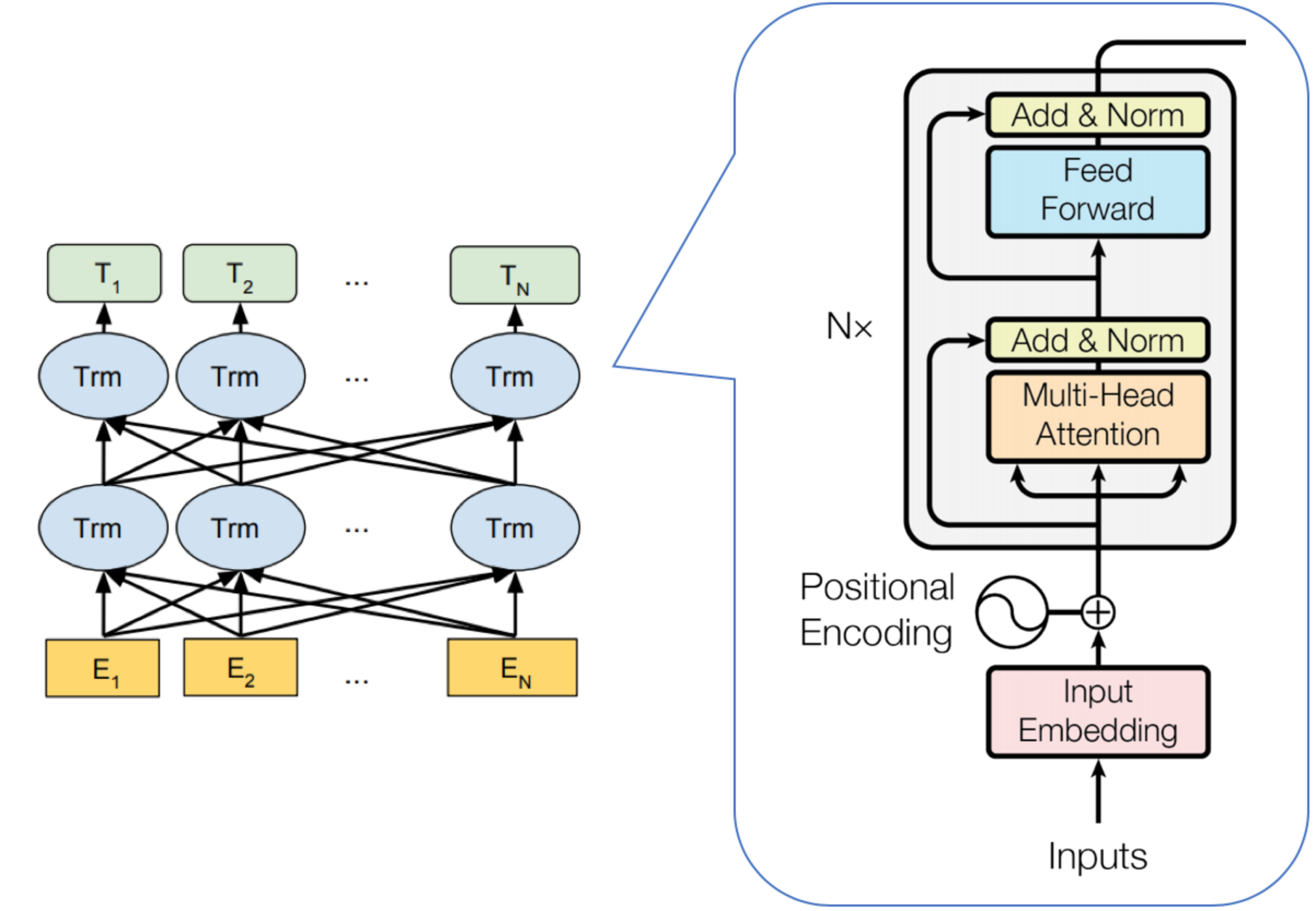
* id
* prompt
* response\_[a/b]

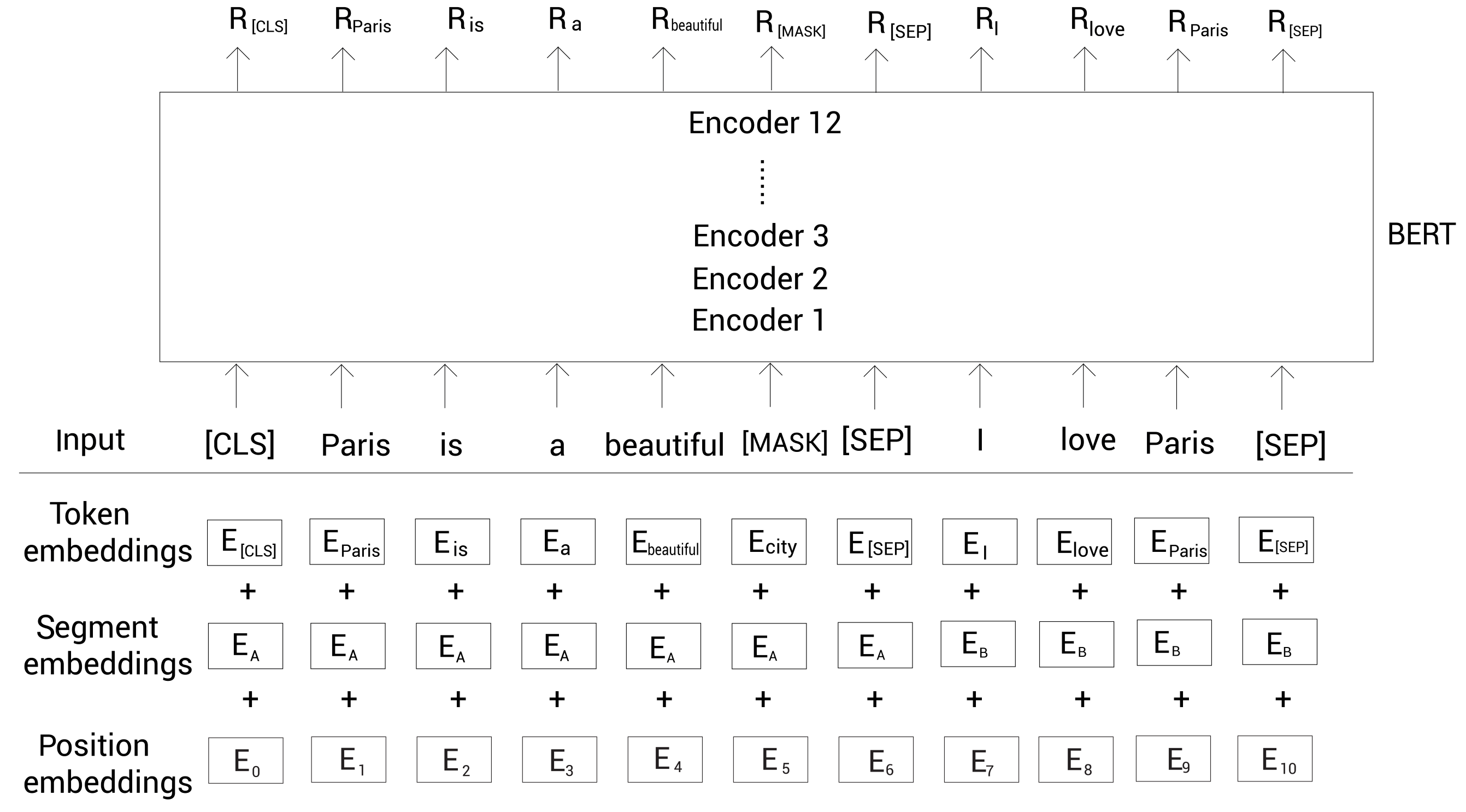
**sample\_submission.csv** A submission file in the correct format.

* id
* winner\_model\_[a/b/tie] - This is what is predicted from the test set.

## Model

### BERT





### DeBERTa

**DeBERTa 模型使用了两种新技术改进了 BERT 和 RoBERTa 模型，同时还引入了一种新的微调方法以提高模型的泛化能力。**

RoBERTa模型（Robustly Optimized BERT Approach）  
概述：  
RoBERTa（Robustly Optimized BERT Approach）是由Facebook AI于2019年提出的一种BERT模型的改进版本。RoBERTa通过对BERT的训练过程进行优化，进一步提升了模型的性能。

工作原理：

* 更大的数据集和更长的训练时间：RoBERTa在更大的数据集上进行了更长时间的预训练，确保模型能够学习到更多的语言知识。
* 去掉Next Sentence Prediction任务：BERT在预训练时包括两个任务：掩码语言模型和下一句预测（Next Sentence Prediction, NSP）。RoBERTa发现NSP任务对模型性能提升有限，因此在预训练中去掉了NSP任务，只保留MLM任务。
* 动态掩码：RoBERTa在每个训练周期动态生成掩码，而不是像BERT那样在训练开始前就固定掩码位置。这种方法增加了训练数据的多样性，提高了模型的泛化能力。

两种新技术的改进：

* **注意力解耦机制**：DeBERTa采用拆分注意力机制，每个词由两个向量表示其内容和位置信息。词对之间的注意力权重通过分离矩阵计算内容和相对位置信息。
* **增强的掩码解码器** ：DeBERTa在解码层中引入词的绝对位置嵌入，用于MLM预训练任务中的掩盖词预测。

新的微调方法：**虚拟对抗训练方法**。

* 虚拟对抗训练通过在输入数据上添加小扰动，使模型在面对稍微变化的输入时仍能保持稳定的输出，提高模型的鲁棒性。

我们使用了Keras框架的Bert模型的变体deberta\_v3\_extra\_small\_en，参数为70.86M，12 层 DeBERTaV3 模型，其中外壳保持不变。接受过英语维基百科、BookCorpus 和 OpenWebText 的训练。

## Code

环境：kaggle notebook

*使用 KerasNLP 的共享权重策略微调本次比赛的 DebertaV3 模型*。

**共享权重策略**：在不同的任务或模型部分之间共享权重，以利用不同任务之间的相关性，提高模型的效率和泛化能力。

### Import Libraries

os.environ["KERAS\_BACKEND"]

* Python用于设置环境变量的一种方式。在这种情况下，它被用来指定Keras后端，即Keras在执行其操作时所依赖的底层计算库。

Keras是一个高级神经网络API，它允许用户快速构建和训练深度学习模型。Keras本身不执行计算，而是依赖于后端库来执行这些操作。Keras支持多种后端，包括TensorFlow、Theano、CNTK等。这些后端提供了底层的计算能力，使得Keras能够执行复杂的数学运算Import。

### Prompt and Responses

有 14 行重复，形成 7 组，我们只保留每组一行。

df = df.drop\_duplicates(keep="first", ignore\_index=True)

我们检查这三列中是否有任何缺失值，由于每个条目都以字符串 类型 表示，因此我们需要事先调用 eval()。可以看出，eval() 对两个响应列都失败，因为无法处理 null。

* eval(x) 是 Python 中的一个内置函数，它用于动态执行字符串形式的表达式，并返回表达式的结果。这个函数会将传入的字符串 x 作为一个有效的 Python 表达式进行求值，并返回其计算结果。

for col in ["prompt", "response\_a", "response\_b"]:  
try:  
 train[col] = train[col].apply(lambda x: eval(x))  
except Exception as e:  
 print(f"eval() fails for column {col}...")  
 print(f"Error: {e}")

eval() fails for column response\_a...  
Error: name 'null' is not defined  
eval() fails for column response\_b...  
Error: name 'null' is not defined

如果至少有一个响应为空，而不是“null”，则 eval() 会失败。为了解决这个问题，我们只需将 null 替换为 None，一切就都正常了！

for col in ["prompt"]:  
 df[col] = df[col].apply(lambda x: eval(x))  
 test\_df[col] = test\_df[col].apply(lambda x: eval(x))  
for col in ["response\_a", "response\_b"]:  
 df[col] = df[col].apply(lambda x: eval(x.replace("null", "None")))  
 test\_df[col] = test\_df[col].apply(lambda x: eval(x.replace("null", "None")))

### Contextualize Response with Prompt

我们将使用提示语境化每个响应，而不是对所有响应使用单个提示。这意味着对于每个响应，我们将为模型提供同一组提示及其各自的响应（例如，（P + R\_A），（P + R\_B）等）。

* 某些提示和响应可能包含无法识别的Unicode字符，从而导致创建数据加载器时出错。在这种情况下，我们将用空字符串替换它们。

# 文本清洗，例如去除无法识别的Unicode字符或替换它们  
clean\_response\_a\_str = "".join(filter(lambda x: ord(x) < 128, response\_a\_str))  
clean\_response\_a\_str = "".join(filter(lambda x: ord(x) < 128, response\_b\_str))

### EDA

#### Number of Turns

* 86.88% 的对话是单轮对话。
* 超过 99.19% 的对话少于 6 轮。
* 最大轮次数为 36。

#### Response Length

LLM 和人类的冗长偏见（即 LLM 和人类有时喜欢较长的答案，即使质量没有提高）

* 回应长度的分布同样呈现右偏，意味着较长回应的样本较少。
* 两个模型的回应长度分布存在差异，尤其是在平均长度和最大长度上。

#### Pair Length Relationship

* 分析了模型A和B的回应长度之间的关系，以及提示长度与回应长度之间的关系。
* 发现两个模型的回应长度之间存在强正相关，可能是因为两个模型都接收到相同的提示。
* 提示长度与回应长度之间的相关性则显得较弱。

#### Verbosity Bias

* 探讨了LLMs和人类评判者是否存在偏好更长回答的倾向，即使回答的质量并未提高。
* 通过分析回应长度差异与评判者选择的关系，发现当模型B的回应比模型A长时，评判者倾向于选择模型B。
* 通过创建更细粒度的区间，发现在(-1, 0]区间内，平局的概率超过0.7，这表明数据集中存在冗余性偏好。
* **创建长度差异区间（Buckets）**将回应长度差异分为不同的区间，以更细致地观察不同区间内评判者的偏好。当创建更细粒度的区间时，注意到在区间(-1, 0]内，平局的概率超过了0.7。这表明在这个区间内，评判者很难区分哪个模型的回答更好，即两个模型的回答长度差异不大时，评判者倾向于认为是平局。

#### Feature

最终我们提取了以下的特征，先做归一化处理最后作为数值特征嵌入文本向量中

numerical\_feature\_columns = ["res\_a\_len\_sum","res\_a\_len\_mean","res\_a\_len\_max","res\_a\_len\_sum\_log","res\_a\_len\_mean\_log","res\_a\_len\_max\_log","res\_a\_len\_med","res\_a\_len\_std","res\_a\_len\_eff\_mean","p\_a\_sum\_diff","p\_a\_mean\_diff","p\_a\_max\_diff","p\_a\_med\_diff","p\_a\_eff\_mean\_diff","res\_b\_len\_sum","res\_b\_len\_mean","res\_b\_len\_max","res\_b\_len\_sum\_log","res\_b\_len\_mean\_log","res\_b\_len\_max\_log","res\_b\_len\_med","res\_b\_len\_std","res\_b\_len\_eff\_mean","p\_b\_sum\_diff","p\_b\_mean\_diff","p\_b\_max\_diff","p\_b\_med\_diff","p\_b\_eff\_mean\_diff"]

### Preprocessing

预处理器获取输入字符串并将其转换为包含预处理张量的字典 (token\_ids、padding\_mask)。

* 这个过程从分词开始，将输入字符串转换为标记 ID 序列。

preprocessor = keras\_nlp.models.DebertaV3Preprocessor.from\_preset(  
 preset=CFG.preset,   
 sequence\_length=CFG.sequence\_length,   
)

### AWP

AWP（Adversarial Weight Perturbation）是一种提高模型鲁棒性和泛化能力的技术，通过在训练过程中对模型权重添加扰动，使模型在面对对抗性输入时表现更好。下面是代码中 AWP 的具体实现和作用解析：

# 定义 AWP 扰动函数  
def awp\_perturb(model, epsilon=1e-4):  
 for layer in model.layers:  
 if hasattr(layer, 'kernel'):  
 # 获取权重  
 weights = layer.kernel  
 # 计算扰动  
 perturbation = tf.random.normal(weights.shape, stddev=epsilon)  
 # 应用扰动  
 layer.kernel.assign\_add(perturbation)

这个函数的作用是对模型的每一层添加小的随机扰动：

1. **遍历模型的每一层**：通过 for layer in model.layers 遍历模型的所有层。
2. **检查是否具有 kernel 属性**：通过 hasattr(layer, 'kernel') 检查该层是否有权重（kernel），即只有对包含权重的层添加扰动。
3. **获取权重**：将层的权重赋值给 weights。
4. **计算扰动**：使用 tf.random.normal(weights.shape, stddev=epsilon) 生成与权重形状相同的随机扰动，标准差为 epsilon。
5. **应用扰动**：通过 layer.kernel.assign\_add(perturbation) 将扰动加到权重上。

# 创建 AWP 回调函数  
class AWPCallback(keras.callbacks.Callback):  
 def \_\_init\_\_(self, epsilon):  
 super(AWPCallback, self).\_\_init\_\_()  
 self.epsilon = epsilon  
  
 def on\_batch\_begin(self, batch, logs=None):  
 # 在每个批次开始时应用 AWP 扰动  
 awp\_perturb(self.model, self.epsilon)

这个回调函数的作用是在每个训练批次开始时应用 AWP 扰动：

1. **初始化 epsilon 参数**：通过 \_\_init\_\_ 方法设置扰动的标准差 epsilon。
2. **在批次开始时应用扰动**：通过重写 on\_batch\_begin 方法，在每个训练批次开始时调用 awp\_perturb 函数对模型添加扰动。

通过这种方式，AWP 在训练过程中不断对模型权重添加小的随机扰动，迫使模型在权重空间中寻找更加鲁棒的解，从而提高模型对对抗性输入的抵抗能力和泛化性能。

### DataLoader

使用 tf.data.Dataset 为数据处理建立了一个强大的数据流管道。

整体流程

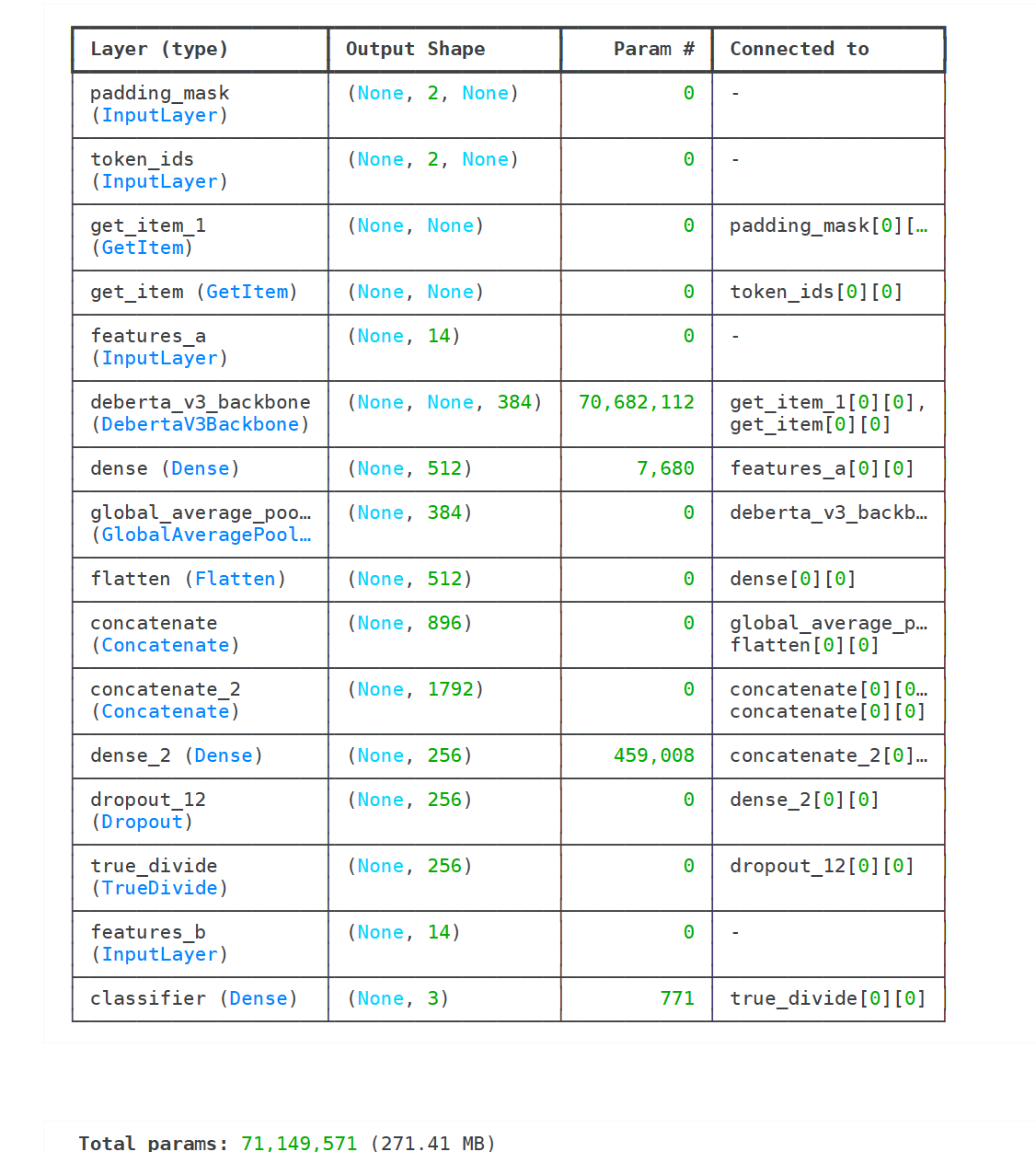
1. **创建切片**：根据输入文本、标签和特征创建数据切片。
2. **创建数据集**：从切片中创建 TensorFlow 数据集。
3. **缓存数据集**：根据需要缓存数据集。
4. **预处理数据**：使用预处理函数对数据进行处理。
5. **打乱数据集**：根据需要打乱数据集。
6. **批处理和预取**：按批次处理数据并预取数据。
7. **返回数据集**：返回处理好的数据集。

def build\_dataset\_with\_features(texts, labels=None, features\_a=None, features\_b=None, batch\_size=32, cache=True, shuffle=1024):  
 AUTO = tf.data.AUTOTUNE  
 if (features\_a is not None) and (features\_b is not None):  
 slices = (texts, None, features\_a, features\_b) if labels is None else (texts, keras.utils.to\_categorical(labels, num\_classes=3), features\_a, features\_b) # Create slices  
 else:  
 slices = (texts,) if labels is None else (texts, keras.utils.to\_categorical(labels, num\_classes=3)) # Create slices  
 ds = tf.data.Dataset.from\_tensor\_slices(slices)  
 ds = ds.cache() if cache else ds  
 ds = ds.map(preprocess\_fn, num\_parallel\_calls=AUTO)  
 opt = tf.data.Options()  
 if shuffle:  
 ds = ds.shuffle(shuffle, seed=CFG.seed)  
 opt.experimental\_deterministic = False  
 ds = ds.with\_options(opt)  
 ds = ds.batch(batch\_size, drop\_remainder=False)  
 # drop\_remainder=False 表示在进行批处理时不会丢弃不完整的最后一个批次。  
 ds = ds.prefetch(AUTO)  
   
 return ds

### Model

with strategy.scope():  
  
 # 将所有输入层整合到一个字典中  
 inputs = {  
 "token\_ids": keras.layers.Input(shape=(2, None), dtype=tf.int32, name="token\_ids"),  
 "padding\_mask": keras.layers.Input(shape=(2, None), dtype=tf.int32, name="padding\_mask"),  
 "features\_a": keras.layers.Input(shape=(14,), name="features\_a", dtype=tf.float32),  
 "features\_b": keras.layers.Input(shape=(14,), name="features\_b", dtype=tf.float32),  
 }  
   
 # Create a DebertaV3Classifier backbone  
 backbone = keras\_nlp.models.DebertaV3Backbone.from\_preset(  
 CFG.preset,  
 )  
  
 # 修改 response\_a 和 response\_b 的创建方式，包含 padding\_mask  
 response\_a = {  
 "token\_ids": inputs["token\_ids"][:, 0, :],  
 "padding\_mask": inputs["padding\_mask"][:, 0, :]  
 }  
 embed\_a = backbone(response\_a)  
  
 response\_b = {  
 "token\_ids": inputs["token\_ids"][:, 1, :],  
 "padding\_mask": inputs["padding\_mask"][:, 1, :]  
 }  
 embed\_b = backbone(response\_b)  
   
 # 将数值特征嵌入  
 len\_features\_a\_embedding = keras.layers.Dense(512, activation='relu')(inputs["features\_a"])  
 len\_features\_b\_embedding = keras.layers.Dense(512, activation='relu')(inputs["features\_b"])  
   
 # 使用 Flatten 层将数值特征嵌入展平为二维张量  
 flattened\_len\_features\_a = keras.layers.Flatten()(len\_features\_a\_embedding)  
 flattened\_len\_features\_b = keras.layers.Flatten()(len\_features\_b\_embedding)  
   
 embed\_a = keras.layers.GlobalAveragePooling1D()(embed\_a)  
 embed\_b = keras.layers.GlobalAveragePooling1D()(embed\_b)  
 embeds\_text\_features\_a = keras.layers.Concatenate(axis=-1)([embed\_a, flattened\_len\_features\_a])  
 embeds\_text\_features\_b = keras.layers.Concatenate(axis=-1)([embed\_b, flattened\_len\_features\_b])  
   
# # 合并文本嵌入和数值特征嵌入  
 combined\_embeds = keras.layers.Concatenate(axis=-1)([embeds\_text\_features\_a, embeds\_text\_features\_a])  
   
 # 定义 temperature\_scale 函数  
 def temperature\_scale(logits, T=1.0):  
 return logits / T  
   
 # 定义温度参数 T  
 T = 0.85 # 这个值可以根据需要调整  
 # 应用温度缩放  
 scaled\_logits = temperature\_scale(combined\_embeds, T)  
  
 outputs = keras.layers.Dense(3, activation="softmax", name="classifier")(scaled\_logits)  
   
 model = keras.Model(inputs, outputs)  
   
 # Compile the model with optimizer, loss, and metrics  
 model.compile(  
 optimizer=keras.optimizers.Adam(learning\_rate=1e-6, clipnorm=1.0),  
 loss=keras.losses.CategoricalCrossentropy(label\_smoothing=0.1, from\_logits=False),  
 metrics=[  
 log\_loss,  
 keras.metrics.CategoricalAccuracy(name="accuracy"),  
 ],  
 )

模型的层级视图



**输入层**：

* token\_ids 和 padding\_mask 是模型的文本输入，形状为 (2, None)，没有参数（0）。
* features\_a 和 features\_b 是数值特征的输入，形状为 (14,)，也没有参数（0）。

**DebertaV3 主干网络**：

* deberta\_v3\_backbone 是从 token\_ids 和 padding\_mask 输入生成的文本嵌入部分，输出形状为 (None, None, 384)，有约 70,682,112 个参数。

**数值特征嵌入和展平**：

* dense 层将 features\_a 嵌入到 (None, 512) 的形状，有 7,680 个参数。
* flatten 层将其展平为 (None, 512)，没有额外参数。

**文本嵌入池化和特征合并**：

* global\_average\_pooling1d 对文本嵌入进行池化操作，输出形状为 (None, 384)，没有参数。
* concatenate 将文本嵌入与展平的数值特征合并，形状为 (None, 896)，没有额外参数。

**特征合并和正则化**：

* concatenate\_2 将前面合并的特征再次合并，形状为 (None, 1792)，没有额外参数。
* dense\_2 添加了 459,008 个参数的 Dense 层，形状为 (None, 256)。
* dropout\_12 是一个丢弃率为 0.5 的 Dropout 层，没有额外参数。

**温度缩放和输出层**：

* true\_divide 对 dense\_2 的输出进行温度缩放，形状保持 (None, 256)，没有额外参数。
* classifier 是最终的输出层，输出形状为 (None, 3)，有 771 个参数。

### Training

try:
  
 history = model.fit(
  
 train\_ds,
  
 epochs=CFG.epochs,
  
 validation\_data=valid\_ds,
  
 callbacks=[lr\_cb, ckpt\_cb, awp\_cb] # 将 AWP 回调添加到训练回调列表中
  
 )
  
except tf.errors.InvalidArgumentError as e:
  
 print(f"出现无效参数错误：{e}")