## 个人简介

面试官您好！我叫张哲槟，来自于四川绵阳。我是2012年考入浙江大学物理系，本科毕业后我获得保研资格，选择跨专业到计算机学院攻读博士学位，主攻神经网络NLP方向，期间从事Transformer神经网络模型的架构优化和在文本生成领域的应用。

毕业我加入了华为的2012实验室，两到三年时间里的工作内容主要可以分成两个部分，前半部分的工作是在智能问答场景的模型优化，期间我一方面是做前沿技术的研究，牵头完成一些论文发表和专利撰写工作，另一方面也结合实际业务需求，做模型的训练优化工作，在监督微调和强化学习训练方面都有实践经验。

后半部分的重点工作是智能体开发平台的算法研发，主要包括的提示词的自动生成自动优化，工作流的自动构建，最近也有在做一些多智能体相关的探索。

整体来说，我的科研和项目经历主要围绕着大模型技术和一些前沿落地场景，我的大体情况是这样。

## 离职原因

部门本来定位是偏向技术预研和探索，当时当前的业务比较重交付，而且把工程和算法的人混在一起用。

主要原因是感觉到在当前部门的发展比较受限。我目前在华为2012实验室下面的一个偏向预研性质的中台部门，一般来说是做一些前沿技术的研究和储备。这两年来部门迫于压力在做转型，我们开始承担一些向公司内部的其他产品业务部门的工程交付。这个转型过程其实对做算法的同学比较不友好，产品那边的策略比较保守，他们更愿意follow业界已有的产品特性去做开发，但是我们部门这边的成果呈现又是需要我们领先业界去做创新，这就导致我们设计研发的一些相关特性不能很快地在产品侧落地，所以我们很多的算法成果得不到应用。基于上面的原因，我们有很多算法的同学选择离职另求发展，部门的组织结构最近也在调整，我自己也在向外看一些机会，希望能够找到适合自己发展目标，并且业务目标明确且稳定的技术团队。

## 绩效情况

在华为期间属于中规中矩的正常绩效。A B/B+ C几个区间，我在b/b+之间。

## 自身优缺点

### 优点

我觉得我会有比较宽的责任意识边界。比如在任务交付上，除了做好我自身这个环节的工作以外，会比较多地考虑整个交付内容是否存在问题，会主动去和上下游对齐，保证不会出现问题

### 缺点

对于自己没有完全把握的事情，不会很快地去推进落实。比如接到一个任务，要做一个特性或者技术，我自己通过基本的调研和实施，搭建了一个demo的流程，但是没有完全验证它的有效性，这时候我会花更多时间去验证，但是可能就导致这个时间拖的比较久。根据我目前的工作经验来看，快速地把技术推出去，并且把问题抛出去，让整个团队一起来解决，其实是更有效率的工作方式。

## Agent平台

### Prompt自动生成优化

prompt自动生成和优化是我们最早在智能体开发平台上开发的特性之一。这个功能的目标是帮助用户快速地进行智能体开发，是根据用户的任务场景，自动生成有效的prompt，来指导大模型生成更准确、更相关的回复。他的核心功能点包含两个，一个是提示词的自动生成，还有一个是自动优化。

* 首先是提示词的自动生成功能，它是根据用户提供的任务描述，直接生成一个完整的系统prompt。这个prompt自动生成算法的原理，其实也是利用大模型本身的内容理解和生成能力，让它基于一个提示词模板，来生成一个具体任务场景的prompt，我们把这个提示词模板叫做元模板。这个元模板的作用类似于一个大纲，引导大模型构建完整 prompt 结构并填充内容。我们起初是用一个统一的元模板来进行所有场景的prompt生成，生成prompt中会包含角色描述、技能描述、执行步骤、输出结构、约束条件等等这些元素。我们在算法效果优化过程中发现，使用这样一个单一的元模板，并不是适用于所有的任务和场景。比如当用户想创建一个角色扮演类型的智能体，那么相应的prompt中就不需要执行步骤或者输出结构这样的元素，反而需要增加语言风格、口吻语气这样的约束条件。因此我们对用户的任务场景做了细分，划分出了推理类、角色扮演类、文案撰写类、（计划制定类、工具调用类、信息分析类）这一系列的类别，每个类别分别配备了相应的元模板。我们的prompt生成算法在运行时，就会首先对用户的任务场景进行类别判定，然后路由到相应的元模板，进行prompt的生成。
  + 推理类：角色描述、任务描述、推理策略、执行步骤、输出结构
  + 角色扮演类：角色描述、风格语气、约束条件
  + 文案撰写类：角色描述、任务描述、输出结构、约束条件
  + 计划执行类：角色描述、任务描述、输出结构、约束条件
  + 工具调用类：角色描述、技能描述、约束条件
  + 信息分析类：角色描述、技能描述、执行步骤、输出结构
* 在有了提示词生成功能之后，我们还开发了一个提示词自动优化功能，它的目的是帮助开发者从繁琐的提示词优化工程中解脱出来，帮助他们的提示词快速地在测试集上达到目标准确率。Prompt自动优化算法其实在学术界有比较多的相关研究，包括一些文本梯度的方法和遗传变异的算法。我们基于这些算法，再结合我们常见的业务场景和目标，做出了一套自动优化的流程和工具。在用户使用自优化功能时，需要提供一个初始化的prompt，以及一个评测数据集，这个数据集中需要包含一些case，每个case包括用户的输入以及期望的输出，除此之外，用户还需要提供一个评分准则，我们依据这个准则，对大模型的实际输出和参考答案进行对比，从而得到一个打分，这个打分是提示词自动优化过程中的重要依据。我们提供的评分准则包括这么几种，首先是文本的完全匹配，它适用于分类或多项选择这种有确定标签答案的场景。然后就是json匹配它适用于需要大模型进行结构化输出的场景。除此之外，还有基于语意相似度的打分，也就是将大模型输出文本和参考答案文本分别通过一个embedding模型编码成向量，然后将两个向量的cosine相似度作为分数。上面说的三种打分策略，都是利用客观规则或者确定的程序进行打分，它们有各自的适用场景，如果在某些场景下，这些方法都不适用，比如在文案撰写场景，在评判大模型输出的时候，通常是基于一些主观的意见，例如文案的丰富度、结构是否清晰、是否有吸引力等等，针对这些主观评判的场景，我们还提供了一种通用的打分策略，就是利用大模型进行打分，用户可以以文本的形式输入一些评测的维度，例如丰富度、吸引力等等，那么大模型就会参考用户给出的维度，对输出进行打分。上面所说的初始prompt、评测数据集和评分准则都准备好之后，我们的自动优化算法就可以运行起来了。这个算法的运行是一个循环迭代的过程，在每一次迭代中，都会包含预测、打分、优化这样一系列步骤，首先预测过程，就是让目标大模型对评测集中的case进行推理，得到预测答案；然后是打分步骤，它会根据参考答案对预测答案进行打分，得到一个1-5分的分数，这个过程除了输出分数之外，还会包含一些额外信息，比如用户如果选择json匹配打分，那么假使大模型输出的json结构体不完整，导致解析报错，那么这些报错信息会被保存下来，它能够在后续的优化步骤中指导模型进行有针对性的提示词优化；在最后一个优化步骤中，就是基于前面的评估结果进行提示词的优化。这个步骤中，我们对常见的文本梯度和遗传变异两种算法进行了结合。首先，我们基于文本梯度的方法，根据反馈信息对prompt进行修正，并得到多个不同的修正版本，然后我们分别对这些版本的prompt以及他们的交叉融合的版本进行评估，选择效果较好的prompt进行下一轮迭代。经过一定的迭代次数，或者评测准确率达到目标区间之后，算法运行会终止，并将最终的prompt返回给用户。
* 上面讲的Prompt的自动生成和优化功能，其实在各个agent平台上都有提供，例如国外的langchain、claude或者国内的coze、dify。但我们的平台还是做出了一些差异化的特性，首先是在prompt生成这一块，像**coze上的自动生成功能，它针对任何场景生成的prompt都是比较同质化的**，都是只包含角色、技能和限制这三个元素，用户在实际使用这些prompt的过程中，就会发现可能效果达不到预期。**我们的prompt自动生成就会有很多的细分场景，基于这些精细化的模板去创建prompt**，在很多场景上验证，都会有一个明显的效果提升。当然也不能说我们生成的prompt就是完美的，任何的这种自动创建的prompt，它离实际投入生产使用，都需要一个后续的迭代调优过程。那只是说，我们这种生成算法，相较于业界其他产品，在效果上往前走了一步。除了自动生成之外，在自动优化这个功能点上，我们相较于友商产品，也是有一些功能上的增加。例如字节的火山引擎上就提供了这种类似的自动优化功能，我们看不到它背后的算法到底是什么，但是在用户体验上，**它的评分标准中只支持基于大模型的主观评测，而我们支持3种基于规则的和1种基于大模型的，一共四种评分标准**。我们开发多种平分标注，主要是考虑到优化效率的问题，有一些简单的评测标准，如果基于规则通常可以在几百毫秒以内得到结果，但是如果使用大模型推理的话，可能就需要几秒或者几十秒。这个评测的耗时是和评测集的数量成正比的，如果评测集非常大，那么使用大模型来推理评测结果的话，在评测的这个环节就会占用很多时间，而且会增加很多token消耗。还有一个特性是我们发现在**友商平台上所缺失的，那就是工具调用场景的提示词优化功能**。大模型的工具调用能力在智能体平台上是十分重要的，通过集成网页搜索、图片生成、信息查询等等这些工具，可以扩展智能体的能力边界。所以我们接触到的实际业务场景中，**优化大模型的工具调用效果，是一个非常普遍的诉求。**我们接收到来自业务方的反馈中，50%以上的效果优化需求都是在工具调用的这种场景。但是友商平台上，没太看到针对这种场景的prompt自动优化，它们提供的都是针对大模型文本回复的优化功能；我们平台上集成了这种工具调用的自动优化功能，在业务方那边也有比较多的使用。
* 在评测我们的算法效果的时候，因为我们这个功能是面向开发者的，不是直接面向终端用户的，所以很难有比较系统的效果反馈。所以我们是想办法在系统中加入一个打点信息，就是开发者从创建一个智能体、调用完自动生成算法开始，到他把提示词优化完毕、智能体完成上线的时间。从我们几个版本算法的不断优化来看，还是有一个明显的效率提升。当然也不排除开发者自身水平不断提高，加快了优化效率。但是从开发者给我们的点赞点踩数据来看，他们的满意度还是有一个不错的提升的。

### NL2Workflow

在智能体平台上，workflow工作流是一个非常重要的功能，一个工作流中可以包含一系列的节点，通过对这些节点进行组合串联，可以自动化地完成非常复杂的任务。虽然说工作流的能力非常强大，但要搭建一个工作流却并不简单，这种困难主要来源于两个方面：首先，他需要业务人员十分熟悉平台上提供了哪些组件，以及每个组件有哪些功能和配置项，这样他才能根据业务场景，选择合适的组件进行工作流的搭建，这对于业务人员来说，有不少的学习成本；其次，在实际搭建过程中，每个组件都会有很多配置项，在组件较多，而且数据字段较多的情况下，这个配置过程其实十分繁琐。于是，我们想说，利用大模型的能力，辅助开发人员快速地完成工作流的搭建，nl2workflow，就是自然语言转workflow，让开发人员通过向大模型发送文本指令，就能自动创建一个完整的工作流。虽然说这个特性听起来非常强大，但是相关算法的实现和优化却是一个庞大和复杂的流程，所以我们是分阶段来逐步完成这个目标的。

* 首先第一个阶段，是完成组件粒度的自动配置。用户首先需要能够自己通过拖拉拽组件，能穿起一个大致的工作流，我们首先提供了一个功能，就是让用户在配置单个组件的时候，能够通过与大模型对话，一键式完成这个组件的输入输出项，以及配置项的填写。例如，当用户配置一个大模型组件的时候，他可以直接输入命令：从前面的组件中获取收件人名称、邮箱地址和会议时间，使用这个大模型组件生成一封邮件，包含主题、正文。那么大模型在收到这条命令后，会读入当前整个workflow的信息，输出一个当前组件的配置文件，例如这个组件的输入项，就会包含姓名、邮箱地址和开会时间三个字段，并且他会自动找到前面是哪个组件会输出这些信息，并生成相应字段的数据引用。然后在输出项中，他会生成邮件标题和邮件正文这两个字段，他还会为这个大模型组件配置一个prompt，让他根据输入的信息，来生成一个会议邀请的邮件内容。开发者还可以通过对话，来更新组件的配置，例如，用户可以说，邮件内容中再增加一项会议地点，然后就会自动增加一个输入项，prompt也会自动进行相应的修改。
* 刚才说的是我们在第一个阶段实现的组件粒度的自动生成特性，我们第二个阶段做的，就是工作流粒度的自动创建，也就是给一个工作流描述，大模型能够直接创建出整个工作流。但是我们做一些试验，如果要让大模型自主完成工作流创建，需要对工作流中用到的组件提供比较细致的描述，比如，要创建一个文档问答助手的工作流时，需要描述说，“先使用一个大模型组件用于意图判断和query改写，然后使用知识库组件进行检索，最后使用大模型组件基于检索结果生成回复”。但是开发者写的描述一般很简单，可能就是说比如“根据用户的问题，从知识库中检索文档，并生成回复”，我们很难期望开发者用很详尽的语言去描述他的工作流。为了让大模型能够理解开发者的诉求，并且能够生成一个完整的工作流，仅凭大模型本身的知识可能是不够的，我们需要给大模型输入一些经验。因此我们首先做的一件事，就是创建一个的经验库，我们在里面预置了很多的工作流示例。每个工作流示例包含三个部分，一个部分是工作流描述，还有一部分就是创建好的工作流配置文件，还有一个部分是CoT思考过程，它是一段文字描述，去描述怎么根据工作流描述来设计对应的工作流。那么用了这个经验库之后，当开发者想要创建一个新的工作流时，我们会先在经验库中检索，根据他提供的工作流描述找到一个类似的工作流，然后把对应工作流的配置文件信息和思考过程都输入给大模型。这样，大模型通过参考一个类似工作流构建经验，就能更准确地完成新的工作流构建。在验证我们的算法成功率的时候，我们直接将模型生成的工作流放到系统上去运行，能顺利执行，就表示它是ok的。这个算法的成功率受到几个因素的影响，一个是经验库，如果我们的经验库足够丰富，并且里面的工作流质量足够高，那么就能给大模型提供一个更好的参考，因此我们在智能体平台上是可以不断动态更新这个经验库的，当开发者在平台上创建了一个新的工作流，并且通过调试运行成功之后，我们就会把工作流保存下来，存到经验库中，这样随着开发者的不断使用，经验库不断更新，就会变得更好用。还有个因素是模型本身，我们之前使用的是公司内部的一个大模型，他的能力并没有那么强，我们评测下来的成功率也就在50%上下，最近我们也开始切换deepseek的r1模型，有一个非常可观的成功率提升，提升了将近20%。目前NL2Workflow类似的算法，我们在一些论文中看到一些相关的研究，但是在友商平台上我们还没看到类似的功能。所以我们的研究还是比较前沿的，目前我们实现的效果，它能够一定程度上提升开发者的效率，但是要做到完全的一键式成功创建，目前还有比较大的优化空间。

### Multi-Agent

Multi-agent也是当前的智能体系统中比较受关注的一个领域，它的主要目的是，使用多个基于大模型的智能体，通过他们的交互协作，共同完成一个任务。它和workflow的一个重要区别是，它不是一个固定的工作流程，而是能够根据任务的复杂度，自适应地调整多个智能体的交互流程。我们这边也在布局multiagent相关的能力，目前我们主要做的有两个方面的算法，一个是agent的自动创建，一个是任务流程的规划。

* 首先是agent的自动创建，针对一个用户给定的任务，我们需要自动创建出能够共同完成这个任务的各个智能体。我们首先通过大模型对用户的任务进行分析，然后生成一个智能体列表，分别描述了每个智能体的角色，和他需要履行的职责。比如用户想做一个软件开发任务，那么相应的智能体角色可能就包括产品经理、需求分析师、架构设计师、代码开发工程师、测试工程师。第二步，我们就需要根据上面的角色信息来完成相应的智能体配置。每个智能体的配置信息，需要包含两个部分，一个是他的系统提示词，这里我们复用简历上前面提到的prompt自动生成算法，来生成相应的提示词；同时，我们还需要为每个agent配备一些外部工具，比如产品经理这个角色，可能需要做一些市场调研，那我们应该给他配备一个网页搜索工具。这个工具配置过程，也是通过大模型来实现的，我们把候选工具喂给大模型，让模型对智能体角色的职责进行理解，然后为他选择合适的工具。
* 除此之外，我们还做了一个流程规划的算法。这个算法有两个目的，第一个目的，一个是对将任务拆解成子任务，并监控和更新任务的执行进度。还有就是在执行每个子任务时，可以控制哪些agent需要参与当前子任务的执行，因为用户在和多agent系统进行交互的时候，用户提出的query，并不一定需要所有的agent参与来解决。还是以软件开发这个场景为例，比如当用户提出想要分析一下市面上的开源agent框架，那么这时候只需要产品经理和需求分析师参与讨论就可以了，因为用户的query还没涉及到代码的开发，不需要其他角色参与。

我们目前把多agent这个特性做成一个聊天群的形式，我们把用户和多agent之间的交互做成一种类似群聊的体验，用户发出一个请求之后，可以清楚地看到各个智能体的发言过程，他们如何进行协作来完成这个任务，并且用户可以在多agent的协作过程中进行打断，并且向群聊中发送反馈信息，我们的流程规划算法会纳入这些反馈意见，对当前的执行规划进行更新，从而调整后序的执行过程。

## IAG

背景项目：在常规的问答任务中，我们通常借助检索模块来补充与问题相关的知识，以此辅助生成模型给出更准确的回答。然而，知识库通常会存在噪声或者覆盖率低的问题，导致对于一些特定的问答任务，无法给予有效的知识补充。例如，在推理任务中，像 commonsense QA 和 strategy QA 这些标准评测集里，就包含很多常识性推理问题，像 “果汁是否能把人淹死”“苏格拉底是否使用过笔记本电脑”，通常的知识库中很难为这类问题提供相应的线索。 与此同时，语言模型作为另一种形式的知识载体，能够以自由文本的形式产出知识内容，因此，我们可以利用语言模型作为知识库的另外一种补充知识来源，辅助进行答案推理。

我们提出的IAG架构，就是在传统RAG架构的基础上，增加一个用于知识推理的语言模型。这个架构的核心在于让语言模型生成合理的推理知识，能够对答案推理产生正向的帮助，我们通过分析人类的归纳推理思维过程，总结出一种归纳推理模式，就是先列出基本知识，再结合事实进行归纳总结。比如 “果汁是否能把人淹死” 这个问题，人类的思考过程可能是：果汁和水类似，都属于液体，人在液体中可能溺死，所以推断果汁也能把人淹死。我们让语言模型去学习这样的归纳推理模式，因此我们把这个模块也叫做归纳器模型。

我们使用了T5模型作为让归纳器模型，为了让它能够学习归纳推理过程，我们首先需要用一批归纳推理的数据对模型进行微调。因此我们利用了GPT-3模型来生成了一批推理数据，构造过程就是使用一个few-shot的prompt模板，然后再把当前query拼接到prompt，让模型生成多条的归纳推理知识，最后对它们进行了过滤。我们分别把归纳推理知识扔个答案生成模型，我们选择获得正确答案概率相较baseline更高的归纳推理知识，将它们保留作为微调数据。

微调训练后的归纳器能模仿 GPT 生成归纳知识，但还缺少给予端到端损失信号的训练优化。由于框架中有两个生成器模型，第一个生成器生成的归纳推理知识以离散 token 形式输入给第二个生成模型，导致端到端的损失函数无法通过梯度回传更新第一个生成模型的参数。因此，我们采用了一种类似强化学习的方式：让归纳器针对同一个问题生成多条不同的归纳推理过程，分别输入答案生成器模型，计算得到正确答案的概率，这个概率就相当于奖励（reward）。概率越高，说明对应的归纳推理知识越好。对于归纳器模型来说，我们可以使用每条归纳知识作为目标，分别得到一个loss，我们再基于每条归纳知识的reward 值，对相应的loss进行加权求和，作为总的loss做梯度回传更新归纳器参数，从而鼓励他去生成更好的那条归纳知识。这就是我们对归纳器模型的主要训练过程。

在这个项目中，我主要负责框架设计、归纳器的训练工作，以及论文的撰写与投稿。我们遇到的主要困难，就是在这样一个包含两个生成器模型的框架里，做端到端的优化训练。刚才也讲到，第一个生成器模型的输出是以离散token的形式作为第二个生成器模型的输入的，所以无法通过神经网络的反向梯度传播的形式进行优化。我们当时对学术界中相关的训练算法进行了调研和分析，最终我们决定尝试类似于强化学习的算法。我们设计的算法借鉴了强化学习的思想，但是实现上完全不同，我们把归纳器模型当作策略模型，端到端的准确率等效于 reward 模型，它能帮助我们判断归纳器生成的不同推理知识的优劣，进而基于这个权重信息构造 loss 函数，更新归纳器参数。训练难度和成本比强化学习低很多，这个训练方法也成为了我们论文的核心创新点之一。

当然，从现在的视角去看，这个算法也存在一些不足。在 loss 设计方面，我们当时没有考虑模型过度训练的问题，如果参考现在主流的 DPO 算法，可以考虑将初始模型作为 ref 模型，在 loss 中加入惩罚项，以防止模型过度训练导致性能劣化。

## RLHF

**项目背景**

我们部门之前从事搜索引擎业务，实验室有个 RAG 项目，它能根据用户的 query 进行网页搜索，然后基于检索内容生成问答，并在搜索引擎上以卡片形式展示答案。

**当前方案的问题**

我们当时使用的生成模型是 t5 - 3b，经过 sft 优化后，能对网页检索内容做简单摘要并回答用户问题。但在准确性、全面性和一致性这些指标上，还有提升空间。InstructGPT 论文发表后，很多工作开始用 RLHF 这种强化学习算法来优化生成模型。鉴于我们的 sft 优化遇到瓶颈，所以尝试用这种新算法提升模型效果。

**Reward 模型训练**

首先要训练一个 reward 模型，这就需要大量数据标注。基于当前模型的缺陷，我们从有用性、全面性、一致性这三个维度打分，分数范围是 1 - 5 分，再根据维度重要性加权得到综合分数，以此构建正负例。

为了降低数据标注的成本，我们借助 GPT 进行数据标注。但直接用 GPT 打分不太靠谱，所以采用了双重打分策略。第一种策略是把三个维度分开，用三个独立 Prompt，让 GPT 分别从不同维度给模型回复打分。构造 Prompt 时，把模型多条回复一起给 GPT，加入 CoT 指令，让它给出思考过程和打分依据，最后计算综合分数，对回复进行优劣排序。第二种策略是抛开具体的打分维度，让 GPT 从综合角度直接对模型多条回复进行优劣排序，同样加入 CoT 指令提升准确性。通过这两种策略得到两个不同的排序结果，为了得到更可靠的标注样本，我们将两个排序结果进行比对，假使第一种排序结果是回复a比回复b好，但是第二种排序结果是回复b比回复a好，那么我们就会排除掉a和b这一对数据，只保留排序结果一致的数据，从而得到更高质量的标注。最终用 GPT 构造了近 1 万条数据，经人工评估，GPT 打分准确率接近 90%。用这批数据微调 reward 模型，模型准确率达到 70% 多，不到 80%。

**PPO 训练**

我们采用了 GitHub 上的一个开源强化学习框架 trlx 来进行 PPO 训练，其训练过程和主流的基于 GAE（也就是广义优势估计方法）的训练方式大致相同。

**涉及的模型**

在训练中主要涉及四个模型，分别是 reward 模型、policy 模型、reference 模型和 critic 模型。其中，policy 模型就是我们要重点训练的 T5 生成模型。reference 模型与 policy 模型使用相同的架构，但它不参与训练过程，主要起到一个参考的作用。reward 模型在整个训练过程中，其参数始终处于冻结状态，不会进行更新。而 critic 模型则是参与训练的。

**训练过程**

GAE 方法的核心思想是，在计算每个 token 的收益时，并非直接计算其总体的 reward，而是计算生成这个 token 的动作所产生的收益相较于平均收益多出的那部分。这里的平均收益，是通过 critic 模型计算得到的 value 值。

具体来说，对于目标序列中任意一个 token 位置，它的优势函数值由以下几个部分组成：首先是 reward 模型计算出来的 reward 值，然后减去当前位置上的平均价值收益（也就是 critic 模型输出的 value 值），接着加上生成该 token 之后未来的平均价值收益（即 critic 模型在下一个 token 位置输出的值），最后还要减去一个 policy 模型和 reference 模型输出之间的 KL 散度。引入 KL 散度的目的是约束 policy 模型在更新之后，其参数不要距离原本模型的参数太远。

在构造损失函数时，policy 模型的损失是优势函数的值乘以一个重要性采样项。这个重要性采样项是当前 policy 模型和老的 policy 模型生成 token 的概率比值，并且会对这个比值进行裁剪，确保它处于一个合理的范围之内，比如 0.8 到 1.2 之间。而 critic 模型的损失，是由优势函数值加上老的 critic 模型的 value 值与当前 critic 模型计算出来的 value 值之间的均方误差损失（MSE loss）构成。

**训练中出现的问题及解决措施**

在实际训练过程中，我们遇到了一些问题。刚开始训练时，可以看到 reward 随着训练的进行逐渐增长，然而训练到后期，模型生成的内容质量明显下降，会生成一些不合理的回复，例如回复中出现大量重复的文字。

经过分析，我们发现生成重复文字问题的一部分原因是 reward 模型打分错误。它会给包含大量重复文字的回复很高的分数，这可能是 reward 训练过程中常常出现的 reward hacking 问题。也就是说，如果我们的训练数据存在偏差或者分布不均匀，会导致 reward 模型基于一些特殊的模式去打分。此外，我们还发现存在长度 hacking 的问题，reward 模型更倾向于给较长的回复打高分。

针对这些问题，我们采取了以下解决措施：

* **针对 reward 模型**：我们对训练语料的分布进行了处理，使得正负例的相对长度分布均匀，以此来解决长度 hacking 的问题。同时，根据实际发现的一些打分错误的情况，比如给包含重复文字的回复打高分，我们人工构造了一些包含重复文字的回复作为负例，然后使用优化后的训练集重新训练了 reward 模型。另外，我们还对 reward 值增加了截断处理，并且根据实验结果将截断值设置在 -10 到 10 之间。
* **提升训练稳定性**：除了 reward 打分问题之外，我们还发现模型在训练过程中会发生参数崩溃的情况，这主要是由于训练的稳定性不佳导致的。为了增强训练的稳定性，一方面我们进行了常规的超参数调优，例如调整 KL 散度的系数和学习率调度（lr schedule）；另一方面，我们将训练过程划分为两个阶段。第一阶段，我们将 policy 模型冻结，只训练 critic 模型；第二阶段，再同时训练 policy 模型和 critic 模型。这样做的原因是，我们的 critic 模型是用 reward 模型进行初始化的，在训练初期它对于平均价值的估计会存在较大的偏差。如果在这个时候就让 policy 模型跟着一起训练，很可能会导致训练不稳定。所以，我们在训练初期专门使用一些步骤来训练 critic 模型，这样在真正更新 policy 模型之前，就能够有一个可以进行合理价值估计的 critic 模型，从而帮助整个模型更稳定地进行迭代。
* **解决显存问题**：由于同时要放置四个模型进行训练，显存压力较大。我们采取了一系列措施来解决这个问题，包括使用 gradient checkpointing 技术，让 critic 模型和 reference 模型共享一些层的参数，并且让 critic 模型只对上面几层进行更新，这样，显存中的保存的部分参数可以复用在两个不同的模型中，减少显存消耗。

**训练超参数设置**

我们的训练超参数设置如下：训练步数将近 1000 步，梯度累积步数设置为 16，batch 大小为 2，每 100 个 step 为一个 epoch，总共使用了将近一万条样本。KL 散度系数设置在 0.1 - 0.2 之间，学习率（lr）为 10^ - 6，重要性采样阶段的值为 0.2，gamma 和 lambda 的值都设置在 1 左右。

**最终效果**

经过强化学习训练后的 T5 生成模型，在 50 个样例上测试 GB 比是 15：6，并且成功上线到花瓣搜索。

**我负责的工作**

我带领两个同事一起完成这个项目。主要负责总体方案设计，包括数据标注方案、reward 模型选型和构造；在 PPO 训练过程中，遇到问题就寻找解决方案，比如使用开源 trlx 框架搭建网络和构建训练流程代码时，一起 debug，同事主要负责训练任务管理，以及整理实验数据；训练中遇到训练不稳定、reward hacking 等问题，也负责解决。

**做得好的地方**

我们小组之前没有强化学习训练经验，但能快速跟进和复现业界最新技术。尽管遇到诸多问题，还是取得了不错的提升效果，积累了宝贵经验。

**不好的地方**

虽然在 webbrain 生成模型场景成功应用了 PPO 算法，但感觉没有完全掌握它。实际训练中问题频发，有些问题（如模型崩溃问题，通过控制学习率、训练轮数、KL 散度比重等方式）也没有彻底解决。

**优势函数的构造**

优势函数构造方面，reward 只有在最后一个 token 有值，由 reward 模型输出得到，其余位置为 0。kl 散度用于约束 policy 模型与 ref 模型之间的分布差距，通过每个位置上 token 的概率分布计算，所以 reward 要减去 kl 散度。再考虑 critic 模型预测的价值，这是对当前状态平均收益的预估，每个位置上优势函数还需加上采取下个动作的预期收益与当前状态平均收益的差值，这个差值体现了当前策略收益相较于平均收益的增量，加上这个值就是优势函数。

**DPO vs PPO**

PPO 依赖 reward 模型学习人类对大模型生成回复的偏好，训练时 reward 给策略模型生成的回复打分，结合 critic 模型计算损失函数，更新策略模型参数。DPO 则证明了 reward 函数与优劣策略的映射关系，将 reward 最大化问题转化为优劣策略分类问题，最大区别是不依赖 reward 函数，直接用人工标注的偏好数据集训练，本质是微调训练，其 loss 推导是对 PPO 的 reward 模型 loss 的等效简化。

不少论文探讨过两种算法优劣。理论上，PPO 适用于各种复杂强化学习场景，只要能构造合理的 reward 模型，就能提升策略模型效果。但实际训练中，PPO 受超参设置影响大，训练不稳定，训练开销也大，训练中要迭代训练 policy 和 critic 两个模型，多步骤训练时还需重新采样标注和训练 reward 模型。DPO 训练过程简单，对计算资源消耗少，因为本质是微调。但它依赖事先标注的数据集，适用场景不如 PPO 多，泛化能力也不如 PPO，即下限比 PPO 高，但上限不如 PPO。

## 鸿蒙问答助手

**项目背景**

华为鸿蒙系统开发的时候，推出前端开发语言 ArkTs，就有很多新手开发者需要学习和掌握这门全新的语言。为了给这些开发者提供便捷的帮助，我们需要开发一个智能问答助手，将其直接部署在开发所用的 IDE 界面中，这样开发者在编码过程中遇到问题，就能随时随地快速获取解答。

**项目挑战**

在项目推进过程中，我们主要遭遇的挑战，来源于模型指令遵循能力差和幻觉问题。当时我们使用的是公司内部自主训练的一个总参数达到 7B 的混合专家 moe 模型。虽然这个模型在通用领域有一定的表现，但其在指令遵从等方面的能力明显不足，尤其是当模型的输入中包含一些代码块，并且用户的问题与这些代码相关时，模型也自主生成一些代码，这些代码内容有时候并不是参数输入的文档，而是它自己编造出来的，这严重影响了答案的准确性和可靠性。我们起初尝试了很多prompt工程的方法，通过优化prompt，试图解决这个问题，但是最终收效甚微。

最终为了解决这个问题，我从三个方面入手，采取了一系列针对性的措施：

1. **RAG 流程改进**：在对文档进行切分处理的时候，我们利用代码抽取技术，将文档中的代码部分抽取出来，然后将其替换成一个特定的代码片段占位符。当模型生成答案时，如果涉及到相关代码的生成，就让模型生成相应的代码占位符，而不是直接生成代码。最后，在向用户呈现答案的时候，再将这些代码占位符替换回实际的代码块。通过这种方式，我们能够确保答案中包含的代码块一定是来源于原始文档，有效避免了模型编造代码的问题。
2. **示例动态填充**：我们构建了一个示例动态召回填充模块。首先，我们精心搭建了一个包含几千个问答对的示例底库，这些问答对涵盖了各种常见的开发问题和场景。然后，我们训练了一个 Bert 模型，让它能够根据用户输入的 query，从示例底库中找到一些类似的问答对示例。这些示例会被填充到模型的输入 prompt 中，通过提供更多的上下文和示例信息，引导模型遵循特定的格式进行回复，从而有效提升了模型回复的规范性和准确性。
3. **MoE 模型训练**：由于我们所使用的基础模型是面向通用问答场景的，并不专门针对代码相关的问答任务进行优化。因此，我们对模型进行了有针对性的微调训练。在数据构造方面，我们借助 GPT 模型的强大能力，针对代码知识问答这个场景，分别构造了几种不同类型的 query，包括知识点类 query、总结类 query、对比类 query、开放类 query、拒答类 query 以及多轮问答的数据。除了针对我们目标场景的数据构造之外，为了让模型能够保留在通用领域的指令遵从能力，我们还特意混入了一些通用的指令数据，占比约为 30%。通过精心构造的数据，我们总共生成了近 1w 条用于模型微调的数据。在微调 moe 模型的时候，我们遇到了 moe 模型常见的负载均衡问题，即每个专家接收到的 token 数量不一致，这会导致有的专家训练不充分，影响模型的整体性能。为了解决这个问题，我们主要对训练阶段的网络结构进行了调整。一方面，在 router 网络中增加噪声和 dropout，这样可以增加模型训练的随机性，避免模型陷入局部最优解。另一方面，在专家选择的时候，通过概率采样选择 top-2，这样能够确保每个专家都有更多的机会被选中参与训练，有效避免了有的专家一直不被选中而得不到充分训练的问题。同时，经过综合考虑，我们并没有在 loss 中加入负载均衡的辅助损失。主要基于两方面原因：一是模型本身已经经过了充分的预训练，我们只是进行小规模的 sft（Supervised Fine - Tuning），因此专家的负载不均问题在这个阶段并没有那么严重；二是辅助损失本身主要是为了平衡训练阶段各个专家的利用率，但是对于端到端的推理效果来说，并不能提供实质性的帮助。

**项目成果**

经过我们一系列的优化和改进，在验证集（大约包含 300 条数据）上，答案的满足度达到了 85%（事实性、帮助性、一致性）。这一成果不仅证明了我们所采取的优化措施的有效性，也为开发者提供了一个具有较高实用价值的智能问答助手。

**个人职责**

我们组承接这个项目，任务是将整个流程串通，并且方案能够达到基本的准确率目标，然后向产品方交付一个demo，让他们做产品落地。这个demo的构建和效果优化过程是我一个人完成的，但是也有借助原有的一些技术积累，例如做文档优选召回的时候，使用的模型是我们组之前就有现成模型，然后moe基础模型也是其他部门提供的，我这边主要做一些微调和优化。除了这两个部分，其他部分，都是我独立完成的demo搭建。

**反思总结**

之前提到有个生成代码的幻觉问题，我们采用的代码抽取和回填的方式在我看来并不是十分完美，由于最终呈现的答案中有一部分是模型生成，有一部分是抽取回填的，就会导致答案有时候看上去不太连贯。当时是为了快速解决问题交付，采取了这样一个工程化的方案。其实产生这个现象的本质原因，是ArkTs语言是一个垂域的知识，它不在模型训练语料中，所以模型生成这些代码时会有严重的幻觉问题。如果要较好地解决这个问题，最好使用相关语料，对基础模型进行一个继续预训练，我预计能够极大地改善问题。

## 提问

* 部门的主要业务是什么，研究的技术应用在什么产品上？
* 我面试的这个岗位，既有业务目标，也有科研，科研（论文、专利）这些的硬性指标
* 在个人的发展晋升一般是什么样的路径，职级结构是什么样的。
* 你们应该是属于中国移动的子公司，员工的福利待遇上和总公司有什么区别吗？
* 薪资的构成是什么样的，base和年终奖
* 这边的工作和加班节奏大概是什么样的

Attention优化工作

T5和Bart区别

* 工作内容：交付、探索研究
* 考核要求

豆包：检索链路，各种源的检索

模型总结：意图判断、总结、图文

## 淘天

1. 模型训练对齐目标

2. 有相关竞品或者对标产品吗

3. 怎么对于算法效果的评价体系

3. 考核指标是什么

4.