稀疏是通往AGI的必由之路

实现通用人工智能（AGI）道路之我见 II

前言 1

稀疏概况 1
- 什么是稠密 1
- 什么是稀疏 1
  - CNN 2
  - 软件1.0 vs 2.0 2
- 稀疏的种类 3
- 稀疏发展的挑战 4
- 稀疏发展的趋势 5
  - 类比Dennard scaling 5
- 稀疏对AGI的重要性 5

稀疏超级大模型 6
- 数据复用 7
  - 现今稠密模型的数据复用 8
  - 稀疏模型的数据复用 9
- 异步计算 10
  - 编程模型 10
  - 软件影响 10
  - 架构影响 10
  - 异步电路 11
  - 异步计算的思考 11
- 算法、软件、硬件协同设计 12

Sparsity on memory 13

Sparsity on composition 13

Few shot learning 13

Sparsity on evolution 14

Sparsity on federated learning 15

总结 15

References 16

前言

我对实现通用人工智能（AGI）有着非常浓厚的兴趣，平时经常思考一些实现AGI的最短路径，工作中也尽量选择一些能为AGI长期研究有所帮助的方向。但我深知自己能力有限，很多方面缺乏系统的理论指导。这次我把我的思考写下来，希望能够和大家一起讨论这个议题，得到各位专家的指正。

这一系列分为三篇文章：“自上而下思考实现AGI技术难点及可能方法”，“稀疏是通往AGI的必由之路”，和“AGI，从我做起”。

这篇文章源自这些年我对神经网络的思考。有些问题是要先解决的，有些问题是要等先解决的问题研究到一定程度后再开始的。在“自上而下思考实现AGI技术难点及可能方法”这篇文章讨论实现AGI所要攻克的大的技术难点后，这篇文章是我自下而上思考要攻克的第一个难点。主要是由稀疏的基础性和重要性，以及稀疏在现阶段的研究成果两方面决定的。

稀疏概况

这一节主要介绍广义的稀疏是什么以及为什么重要，后面几节更具体的介绍在各方面的影响。

神经网络由权重和激励相互作用得到结果。其中最主要的运算是乘加运算，并辅以一些其他的算子，如ReLu，Pooling，BN等。因为乘加运算在神经网络计算中占了主导地位，我们讨论主要集中在乘加运算上。

什么是稠密

在介绍稀疏之前，我们先定义一下什么是稠密。

> 稠密是每一个权重和每一个激励相互作用，形成全联接（Fully connected）。

MLP是一个典型的稠密网络，其他计算都归一到MLP来判断是否稠密。

由此可见，在稠密计算中，每一个权重和每一个激励的地位是相同的，都消耗相同的计算资源。但是，不同权重和激励对模型结果的贡献是不同的。这种投入相同但产出不同，造成了稠密计算效率不高的更本原因。

这种效率低下在中小型模型中不容易显示出来，尤其是硬件对稠密计算的支持非常高效的情况下。但这种劣势在超大规模模型上就会表现得非常明显。

什么是稀疏

类似于稠密的定义，我们也给稀疏做一个定义：

> 稀疏是一种选择性计算相关权重和激励的方式。
> A mechanism to compute on the **relevant** features and weights in a model.

这里我们并没有提到零（0），因为零是相关性在权重稀疏上的表现形式。

稠密其实也是稀疏的一种极端情况：所有的权重和激励都是相关的。

既然稠密对所有权重和激励可以无差别对待，直接导致了我们处理稠密计算时可以相邻元素一个一个顺序处理（sequential access）。然而，由于稀疏的定义是只处理相关元素，导致了稀疏对权重和稀疏时需要随机处理（random access）。这是稠密和稀疏从定义上导致的核心区别。

虽然，由于顺序处理更加高效，可以从稀疏的存储方式、算法改进等方法来增加稀疏的顺序处理能力，但这种从定义导出的特性从本质上是不可改变的。

CNN

在了解的稠密和稀疏的定义之后，我问一个小问题吧：CNN是稀疏吗？

答案是：Yes。如果我们把CNN转换为MLP来考虑，我们会发现有两个特性：

1.权重和很多激励是没有关系的。即，我们不需要为之做计算。这就是稀疏的定义。但是，这种不需要做的计算不是随机的，而是符合一种简单的规律，可以通过im2col的方式转换成稠密计算。im2col本身对激励做了规则的随机读取。为了使这种方式做的更加高效，很多DNN加速器有专门硬件来产生隐式iim2col，即，不会在机器中显式产生转换完的矩阵，而是在计算过程中动态产生。

2. 如果我们将CNN转换成MLP的计算的话，可以发现除了上面一点之外，有关系的权重和激励的计算是correlated，即，参与不同的输出计算的权重是相同的。这一点并不能从稀疏中得到，说明稀疏并不能解决所有问题，而需要辅以其他方式来完善。

其实上面两个特性可以用之前说的factory的方法来解决。CNN就是一个fixed projection函数而已。

CNN是一个特例，在1958年发明MLP[1]之后，人们用自己的聪明才智在80年代发明了CNN[2]，然后一直沿用到今天。人们寻找新的feature extractor的速度非常缓慢。我们需要提升抽象层级，更普世的寻找下一个高效算子。

软件1.0 vs 2.0

上一篇文章介绍了软件1.0和2.0。我们这里举一个简单例子（如图一所示），比较两者的不同。

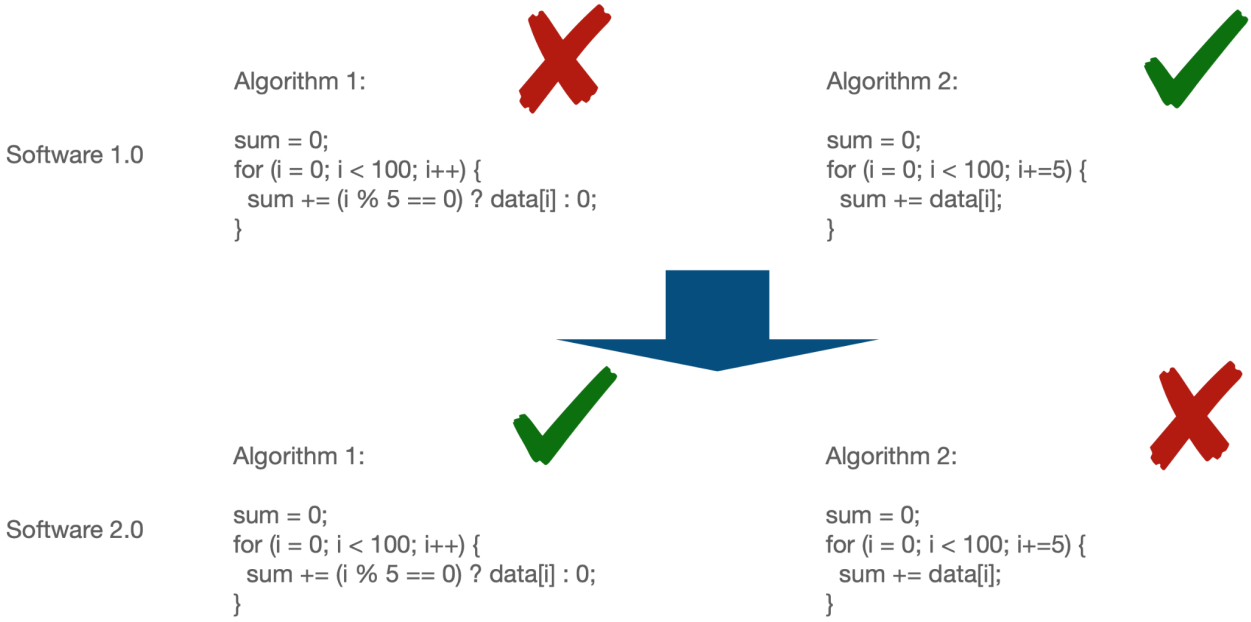
假如你在面试软件1.0的程序员，题目很简单：有一个100个元素的数组，写一个程序得到每隔五个元素的值的总和。

这里有两种写法：算法1，写一个循环查看每个元素，当发现是第5个元素的倍数时，将数字加起来。算法2，同样写一个循环来跳着隔五个元素读一次，将读出的数字加起来。虽然这两种写法都能够得到正确的结果，但如果你在面试时写了第一种算法，很遗憾你过不了这个面试。因为算法1比算法2要多读5倍的数据。

如果你在面试软件2.0的程序员，你不需要写程序，而是编一个模型来实现这个结果，这个模型是不是要把每一个元素都看一下，然后有选择性的让结果反映一部分数字的计算？模型并没有跳跃的读取元素来计算。软件2.0和1.0为什么有这么多的差别呢？

这和深度学习的特性相关，在深度学习中，没有绝对的对错，只有好和更好。相关性没有完全不相关和完全相关，而是一个渐进的过程。为了能够确保结果更好，需要把每个元素都看一下。这样的话，就没有办法在看元素之前就将元素跳过了。

这从某种程度看，这也显示出现阶段深度学习的局限性，而稀疏的目的就是对输入进行区分，只处理最相关数据。从而论证稀疏的重要性。



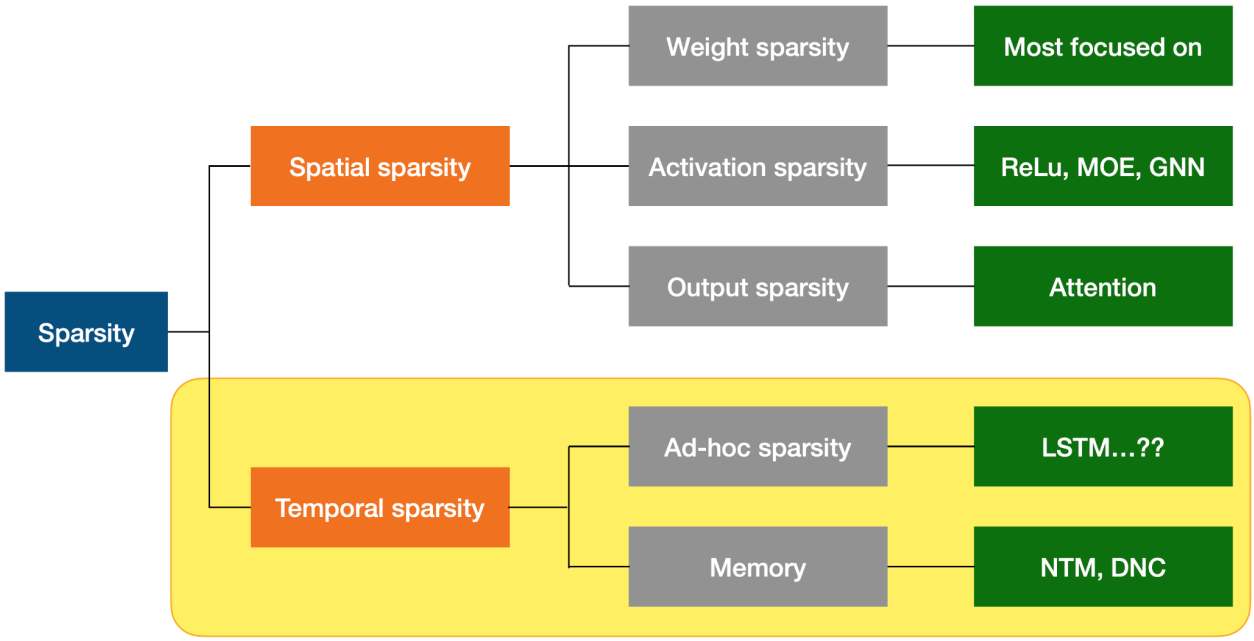
图一：软件1.0和2.0比较示例

稀疏的种类

传统上稀疏可以分为权重稀疏，激励稀疏，和输出稀疏，如图二所示。权重稀疏指的是模型的权重中有很多零，所以不需要计算，这也是人们研究最多的稀疏类型。激励稀疏是指利用输入在模型中一层层传递过程中的激励中会产生很多零，也不需要计算。这种零可以是细粒度的，如ReLu产生的，也可以是非常粗粒度的，如一些gating network会阻止激励往模型某一部分传递，像MOE。模型的输入也可以是稀疏的，如GNN。输出稀疏和激励稀疏通常同时存在，激励稀疏指的是激励中的零不需要参加下一层的计算，而输出稀疏指的是知道输出是零的话那些产生零的计算也不需要计算，像attention里K/Q矩阵相乘的结果经过softmax是稀疏的话那些零的值也不需要计算了。这些分类在[1]中有比较详细的论述。这里所有稀疏方式都是对同一输入产生的稀疏，这里我称之为空间稀疏（spatial sparsity）。

同时，稀疏也可以在时间上产生（temporal sparsity）。现在的模型里并没有单独针对时间的稀疏，而是将时间的信息嵌入在模型中 ，这样和空间稀疏就没有办法区分，例如RNN模型（LSTM）。如

果模型需要长期记忆过去发生的事情，将这些事件无序的存在模型中是不可取的。需要对记忆单独处理，比如将过去发生的事件都放到一个memory bank里，新的事件发生后，在memory bank里寻找对新事件最有帮助的过去的事件，如NTM[2]，DNC[3]。这个过程就是一个稀疏的过程。我们是没有办法每来一个新事件就将所有过去事件都扫一遍的。



图二：稀疏的分类

稀疏发展的挑战

在现有神经网络发展的基础上力推稀疏有很多挑战。这里我总结出有以下几点。

发展稀疏的动机。现在很多人看见模型运行中有很多零产生，自然而然的就想不去算这些零。这样运算的效率就会更高，功耗更低。这导致了稀疏的方法完全是为了计算效率的提升（computation efficiency）。在算法上稀疏的模型从稠密模型剪枝下来（这样模型不能太大，因为先要有一个稠密模型）。而稀疏的效果要和其他提高计算效率的方法比较，例如量化（quantization）。另外，稀疏模型的效果都是和稠密的大模型相比，这种方式限制了人们的思维，限制了稀疏研究的范畴。

稀疏算法的现状。因为稀疏发展的动机决定了稀疏算法的取舍。从稠密模型剪枝下来的算法并不是最适合稀疏场景的。（我认为）更适合的是训练稠密模型太大而无法训练的网络。而大多数研究都没有往这方面努力。另外，因为算法发展的不充分，在从稠密模型剪枝下来的过程中有很多效果损失，减少稀疏的吸引力。

稠密模型的发展。因为硬件对稠密模型的性能较好，深度模型都是稠密模型。而随着深度模型的广泛应用，硬件都针对稠密模型做了种种优化。这样就起了一个正反馈的作用。而稀疏跳出了这个正反馈的圈圈。但是由于稀疏算法和硬件的发展都很薄弱，基础很差。还需要和高度优化了很多年的稠密模型算法和硬件相比，处于一个劣势状态。这里的突破口是要找到一个稠密计算不能胜任的方向，集中力量站稳桥头堡，再拓展到其他领域。

模型大小。中小模型是稠密计算的sweet spot。稀疏在这个领域和稠密模型比拼是很困难的。稀疏模型能大放光彩的是超级大模型。但是由于人们的惯性思维，现在仍在拼命扩大稠密模型。而对稀疏的思考不足。这导致稀疏的优势不能体现。

稀疏发展的趋势

最近一两年，神经网络取得了长足进步。2017年发现Transformer架构[4]，其中self-attention对sequence length的计算复杂度以平方级别提高。最近这种类型的模型在各个领域的应用越来越广泛，而sequence length也越来越长，对现有机器的计算和存储产生了很强的挑战。基于这一架构研究出一个又一个超级大模型。GPT-3有150B个参数[5]，switch transformers有1.6T参数[6]。而训练的开销也急剧增长，例如，GPT-3训练一次需要4.6M美元[7]。

这些种种迹象都表明现有的硬件发展不能够跟上模型扩展的速度。需要后退一步，用另一种方式思考整个神经网络从算法到硬件的所有层级。稀疏的机会即将到来。

类比Dennard scaling

现阶段神经网络的发展有可能是稠密模型快到尽头，稀疏模型快要腾飞的时候。让我想起十几年前CPU的单核向多核演变的过程。

Dennard scaling[8] 指的是集成电路每一个工艺节点晶体管线宽下降30%，面积下降50%，频率上升40%，功耗降低50%。在80-90年代，Intel CPU都是单核，每代产品的性能都能提高百分之几十。那时软件工程师很高兴，软件写好后什么都不需要改，每代性能可以提高好多。因为硬件是单核，软件想改也没有用。那时所有人的思维都是怎么提高频率来提高性能。因为惯性思维，绝大多数的架 构改变都是针对单核的，怎么提高IPC来提高性能。

这种好日子一直持续到2000年出头，那时Dennard scaling broke了，频率上不去，性能提高不了了。那时Intel的小弟AMD率先推出了多核的芯片，一下子占领了很多服务器市场。软件迅速跟上。然后Intel奋起直追。整个industry快速往多核推动，虽然频率没提高，但芯片性能成倍提高的幅度远远超过之前单核提高频率带来的好处。

正是因为Dennard scaling的成功，限制了人们的想象力，多核的推出一直推迟到Dennard scaling不能持续的时候。而那时Intel作为Dennard scaling的既得利益者，没有动力预判做出革命性的改变，需要其他小弟（AMD）来趟这条路。

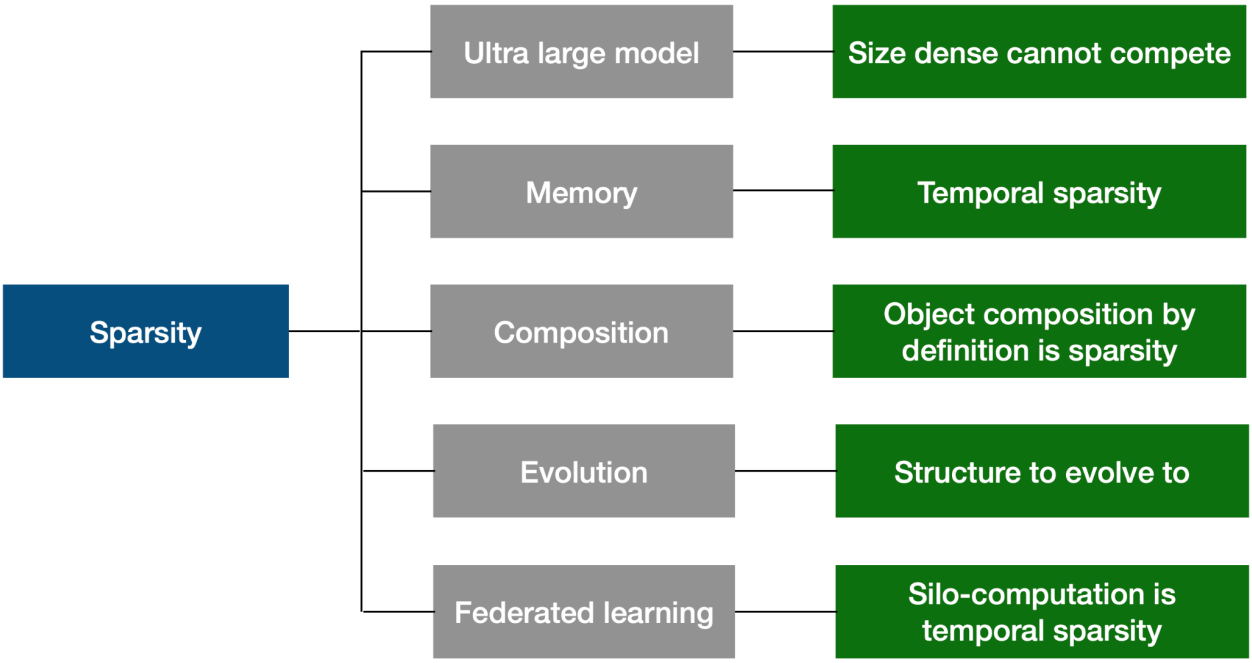
我们现在是不是也处在这种转折关头呢？模型训练性能提高全靠NVIDIA每年新产品发布。GPU性能提高多少，我们单卡训练就能提高多少倍。我们还可以用很多GPU并行训练，用金钱换取性能。但这些方式依然不能跟上模型大小增大的速度。

NVIDIA作为既得利益者，也不一定有动力预判，打破自己的金饭碗。这是是不是需要其他小弟来趟一趟新的计算方式呢？

稀疏对AGI的重要性

实现AGI的路上有很多技术壁垒需要攻克，而稀疏对解决很多技术难点都起到了关键的作用，如图三所示

- 超级大模型。AGI的模型大小应该是现在难以想象的。而对模型所有权重和激励都同等对待的稠密模型一定不能够拓展到那个模型大小。从人脑的超级稀疏性我们也可以猜测超级大模型一定是稀疏的。
- Memory。记忆是时间上稀疏的访问过去关联的事件。
- Composition。Object composition从定义上讲意味着object内部的联系要紧于外部联系。这就是一种稀疏表现。
- 进化。进化需要一种很好的结构比较容易进化。稠密方式虽然简单，但过于离散，进化的难度有点高。而稀疏就是一种结构，可以很细粒度的控制进化方向。
- 联邦学习。联邦学习是分散的，稀疏也是分散的。稀疏有很多特性适合用联邦学习的方法进化模型结构。



图三：稀疏是很多研究的基础

下面我们仔细讨论以下稀疏对其他技术难点的重要性。

稀疏超级大模型

随着模型越来越大而硬件的计算能力增长显著滞后，我们也许需要被迫寻找一种更有效的方式增加模型表征能力。而稀疏提供了一种能显著增加模型表征能力而不相应增加计算的方法。稀疏有希望训练出大幅超出现在稠密模型上限的模型。[1]对之有比较详细的论述。

如图三所示，以稀疏作为基础的技术难点有很多，但我觉的首先需要研究的是稀疏的超级大模型。原因有三点：

- 从必要性来说，现阶段很多人已经意识到传统的稠密这条路块走到尽头了，必须要做一些改变。
- 从难易程度来说，现阶段已经有不少关于稀疏的研究，对大模型训练也有了很多基础，将两者结合起来的难度就减少很多。
- 从重要性来说，稀疏大模型是很多其他研究的基础，为其他研究提供方法以及好的出发点。

稀疏超级大模型在算法和硬件两方面都有非常大的挑战。

在算法方面，现今流行的稀疏剪枝算法是需要先训练一个稠密模型，然后剪枝出一个稀疏模型。这种方法只适用于一些中小型用现有硬件能够训练稠密模型的稀疏模型。而对于包含稀疏模型的稠密模型是现有算力几十倍的情况下，这种算法是无能为力的。因此,我们需要在没有稠密模型的情况下训练出高效的稀疏模型。现在算法没有对此做深入研究。这是首先需要加强的地方。

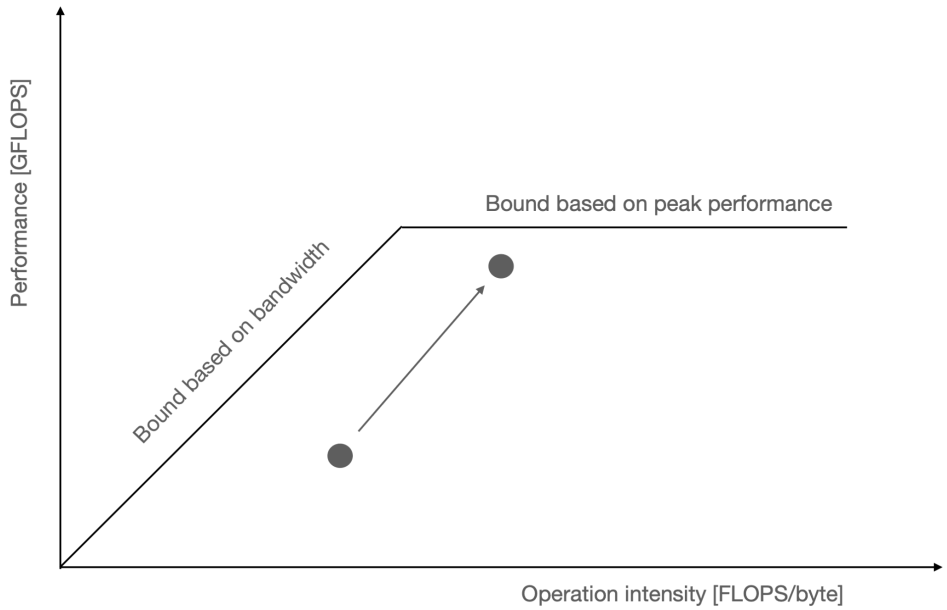
其次，现在稀疏还是主要针对权重的稀疏，我们已经取得很好的结果[9]，但也发现有更多的问题需要解决。然而，只是权重稀疏是不够的，我们也需要加强对激励稀疏的研究，特别是few shot training的研究。只有权重和激励两方面能够同时前进，我们才能找出高效的稀疏超级大模型的方法。

从硬件方面，硬件需要能够高效同时支持权重和激励稀疏。同时，硬件需要支持single batch training。这是因为激励稀疏意味着同一个batch的不同数据走不同路径，大batch节省计算的优势不明显。现阶段对此的改进（主要是MOE的改进）为同一mini-batch内部不同路径保留计算资源，会产生不必要的浪费。

下面详细介绍我认为针对稀疏计算需要进行怎样的优化。这些优化和现有的软硬件系统有很大不同。

数据复用

数据复用是现今针对神经网络计算优化最重要的考虑因素。从roofline的角度来说，我们努力从算法、软件、硬件所有方面将计算往右上角推。在compute bound的情况下靠近roofline。如图四所示。



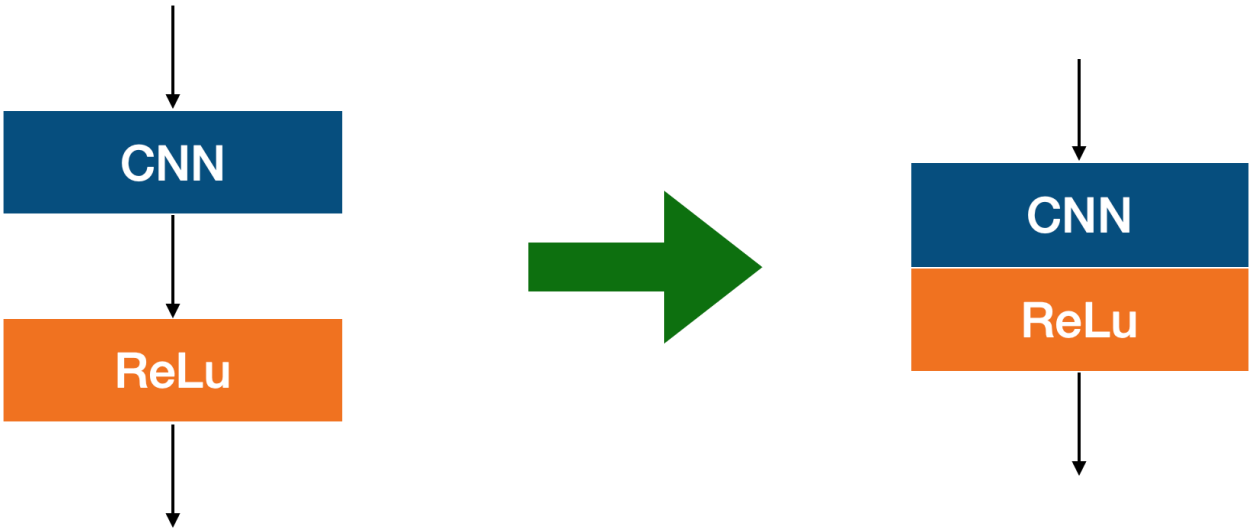
图四：roofline模型

现今稠密模型的数据复用

对于稠密模型来说，数据复用比较简单。因为我们要无差别读取所有权重和激励，所以我们可以读取相邻的数值，利用locality，提高数据复用。这时，我们采用的方法主要是要tile权重、激励和输出，使数据能够更长时间的保存在硬件中的cache中，减少数据的反复读取。

在这种模式下，最常用的选择是在模型的同一层内最大限度的复用数据。这也是现在大多数ML框架所实现的方式，框架负责graph，层与层之间的联系，而算子负责一层内部的实现。也符合现在商业分工，框架由FB（PyTorch）或Google（Tensorflow）实现，而算子由NVIDIA（cuBLAS）或Intel（MKL）负责。

但这种分工完全没有利用层与层之间的数据复用。在某些场景下对性能有很大损失。比如，一个CNN层之后紧跟着一个element wise操作的层，如ReLu。ReLu的操作需要的计算很少，但执行这两层时需要把所有的激励从global memory里读写两次，占用很多时间。这时一个常用的方法就是算子融合（fusion），将这两层并为一层，写一个算子做CNN+ReLu。这样激励只需要读写一次就好了，对性能有很大的提升。如图五所示。



图五：算子融合示例

一个复杂的算子后面跟着一个简单算子（element wise），做算子融合还是很容易的。但是这种类型的融合有很多组合，手工一个一个写非常消耗人力，而且融合算子的适用性会降低。一般只会写一些常用的融合算子。而ML的编译器，如TVM[10]，可以自动融合element wise的算子。降低手工劳动。

但是，就算有编译器的支持，融合也仅限于element wise的简单算子，而对于复杂算子，例如融合两个相邻的CNN层就会非常复杂，现在是没有办法完成的。所以，层与层之间的数据复用在现今的架构下面没有很好的利用。

这对于稠密运算不是一个很大的问题，因为同一层之间有很多的数据复用可以利用。人们也可以加大batch size来增加层内的数据复用。这也是我们现在训练追求超大batch size的主要原因之一。

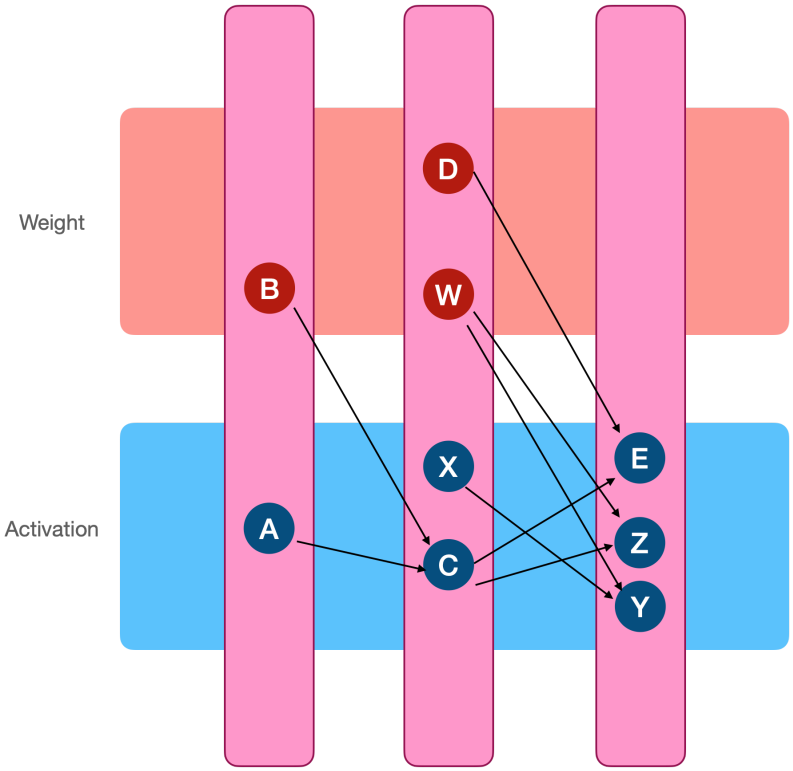
稀疏模型的数据复用

如前节所示，稠密模型计算分为一层一层，只利用层内的数据复用。这对稠密计算还是可以接受的。

但是这种分层方法对于稀疏计算是灾难性的。稀疏的定义是随机存取，在同一层内不能利用locality来复用数据。就算有很大的batch size，因为有激励稀疏，同一batch的不同数据走的路线是不一样的，这样大batch不能解决问题。

那，需要怎样利用稀疏的数据复用呢？

和稠密模型一样，我们还要打层与层之间数据复用的主意。



图六：稀疏计算数据复用示例

如图六所示，有一个稀疏模型，激励A和权重B生成激励C，在下一层中，激励C和权重D可以生成激励E。在这两层中，C生成以后可以立刻reuse来生成E。这就是层与层之间的复用。同时，C和W可以生成Z，这需要C和W同时在cache里，所以在生成C同时我们也需要预读取W。然而W和X要生成Y，所以X也要和C差不多时候生成。

由于这些权重和激励可能在不同时间生成，或者在不同的memory hierarchy而导致读取时间不定，这需要我们有一个强大的scheduler，可以最大程度的复用数据。

异步计算

由于稀疏计算中数据的复用一定比稠密计算低，稀疏运算绝大多数是memory bound。而且由于稀疏读取的随机性，数据的带宽也不能有效利用。在这种情况下数据流是最重要的，需要针对IO优化整个系统设计。

我们可以设想稀疏计算中有非常多的随机数据读取并且进行计算，其中有一部分计算有先后关系，但还有很多计算没有先后关系。在这种情况下，我们不希望读取一个数据的延迟影响到其他数据的计算。这种计算模式非常适用异步计算（asychronous programming）[11]。

下面，我们针对稀疏的异步计算从编程模型，软件，和硬件三方面做一些论述。

编程模型

异步计算在UI领域非常流行，包括在客户端的Javascript，以及在服务器端的NodeJS都是采用异步计算。作为实现异步计算的方法，它们使用event driven programming model[12]。在UI领域，event来自终端用户，是不可知的。但是在稀疏运算中，event来自如下两个方面：

- 静态：执行路径决定于权重数值和graph的结构。
- 动态：执行路径决定于动态激励数值。

这些大多数event是可以预判的，但是由于这些event数量非常巨大，为了降低编程复杂度，我们也采用相同的编程模型。在这种情况下，一个好的scheduler可以利用event中的相关性，最大程度复用数据，降低计算开销。同时，这种编程模型并不需要很大的batch size。Single batch就可以得到很好的结果。

软件影响

Event driven programming model可以很好的提高计算效率和软件工程师的编程效率。写程序时并不需要explicit协调读取数据和计算数据。例如，现在CUDA编程时需要预先把数据从global memory把数据搬运到shared memory里，而且要预估搬运时间。这种做法在event driven programming里就包含在编程模型中了，程序员就不需要考虑了。但是，这种编程方式和现在大多数程序员熟悉的同步计算编程有很大的不同，程序员还需要一个适应的过程。

这种编程方式可以encapsulate在框架里，这样算法工程师可以不需要考虑新的编程模型，而集中精力找出效果更好的稀疏模型。

但是采用这种方式的框架的设计变得更为复杂了，框架的两层结构（graph和kernel）对于细粒度的稀疏就不适用了。这两层结构是针对同步计算设计的。在异步中我们要打破层与层间的隔阂。手动设计框架和算子不会成功，我们需要依靠编译器的力量给runtime足够的暗示优化整体计算时间。

架构影响

异步计算的核心是non-blocking。现在的异步计算都是软件实现，是基于现有硬件架构的前提下的。这种方式能处理一些粗粒度的event，而对非常细粒度的event就无能为力了。

我们先调研一下现有硬件架构对异步计算的支持吧。

- In order processors：这是纯同步计算，没有任何non-blocking计算能力。
- Out-of-order processors：在硬件层面用有限状态机（FSM）方式设计有限的non-blocking操作，如scoreboarding, tomosulo, non-blocking cache。但是，由于ISA没有异步支持，在commit之前都强制为同步（当然也有exception和其他考量）。
- Stream processors：这种以数据流为主导的处理器对静态图有非常好的支持，所有的操作都是确定的[13]。但是，如何处理动态event呢？
- Graph accelerators：graph是稀疏的一种，这方面的研究对稀疏有着很好的借鉴意义。其中GraphPulse[14]就是用event-driven方式加速计算的一种尝试。

从中我们可以看出，现在软件可以在同步的硬件架构上实现non-blocking，而硬件也可以在微架构上实现有限的non-blocking，但是ISA的设计并不是non-blocking的，这对实现大范围细粒度的event driven architecture有所阻碍。

那，我们是不是需要设计一种异步架构将细粒度non-blocking access通过ISA传递到软件，这样软件可以实现更好的global scheduling？（另一个问题是，我们需要这么做吗？）

异步电路

既然谈到异步架构了，我们就更进一步聊一聊异步电路（asynchronous circuit）吧[15]。既然上面整个stack都是异步的，我们在电路实现上还需要同步的时序电路（sequential circuit）吗？

异步电路有很多好处，比如说性能很高，功耗很低等等，这些都是成倍甚至数量级的提升啊。它的劣势主要是非常难设计，没有EDA支持。

过去很多异步电路的尝试都无疾而终了。最近一个倒下的公司是用异步电路做神经网络加速器的Wave computing[16]。曾经风光无限的独角兽被异步电路坑惨了。

Wave computing用异步电路做传统稠密网络加速失败了。但是用异步电路做新兴的异步稀疏网络加速是不是有一点点希望呢？

我觉得现在还没有到这个阶段，这可以留作一个非常长期的研究项目吧。

异步计算的思考

异步计算虽然有一个event queue，有一个scheduler，但是执行的task可以相关性非常大，也可以相关性非常弱。从概念上讲和分布式计算（distributed computing）有种种联系。

因为分布式计算由于不同机器之间的带宽非常有限，在很多场景下都是机器间做异步计算，而同一机器内做同步计算。但是，稀疏训练场景还是以同步训练为多，就算是机器间也是同步训练。这导致了很多系统优化挑战。也是现在模型训练不能scale到很多台机器的原因。

当一个超级大网络分配到很多台机器上，每台机器对分配到自己部分的网络实行异步计算，意味着机器和机器之间也是异步计算。这样不需要专门为机群做优化。但是，这也意味着我们在算法上要研究asynchronous SGD。我认为，模型扩展到一定程度（成千上万台机器），模型训练一定要用asynchronous SGD了。

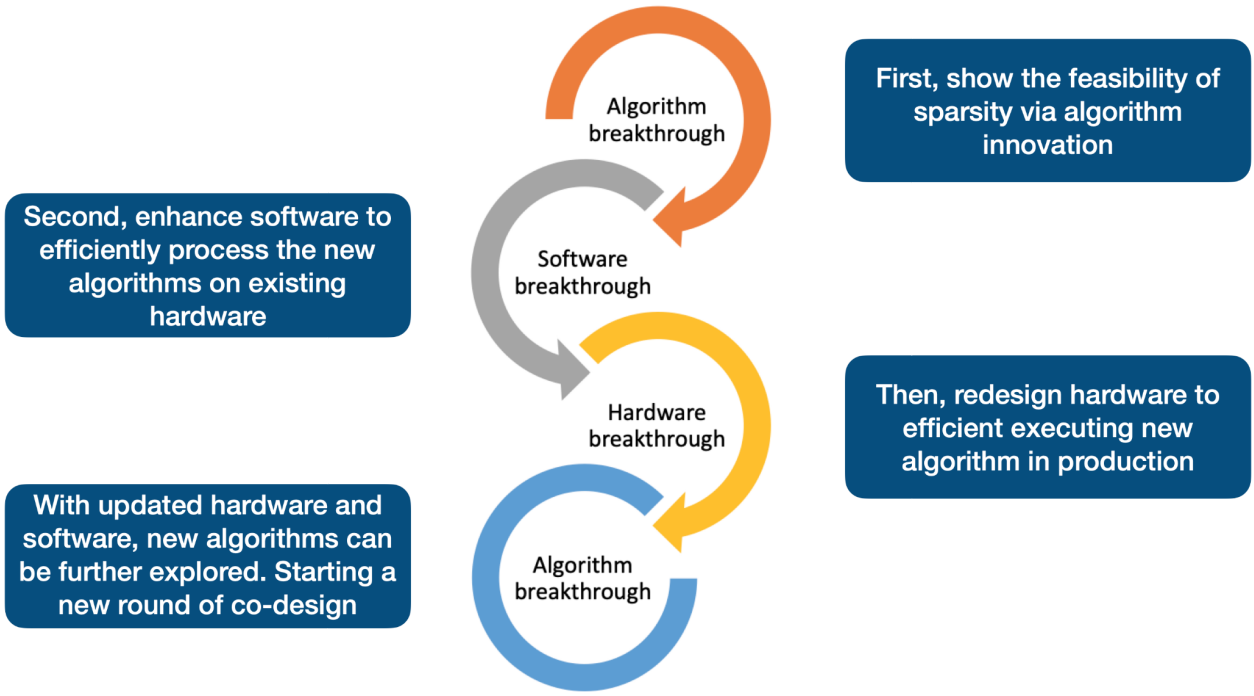
我们再思考一下，我们训练算法是异步的，机器内部是异步的，机器间是异步的。这种方法可以更加容易的拓展到非常非常多的机器。那我们是在训练一个超级大模型吗？还是有很多个小模型在协同训练？这两者之间的界限变得模糊了。

这是另一个层面的超级超级大模型。

算法、软件、硬件协同设计

稀疏超级大模型是一个非常长期的项目。需要算法、软件、硬件的协同设计。

- 我们首先要在算法方面获得突破，能够展示稀疏这条路是可行的。可是，由于硬件的局限，我们只能探索一些渐进的算法改进。并且，我们能实际获得的好处会比较有限。可能只在某些点上和传统方法相比有优势。
- 然后，我们软件更新要紧紧跟上，充分利用现有硬件得到最大性能。同时也暴露现有硬件瓶颈。
- 因为硬件设计开销巨大，我们需要业务论证需求，算法和软件证明可行，改进现有硬件设计。从而能够落实稀疏带来的性能优势。
- 有了硬件的支持，我们又回到第一步。算法又可以探索更多的未知方向，展示更多的稀疏优势，开启另一轮的协同设计。



图七：稀疏研究需要算法、软件、硬件协同设计

稀疏超级大模型并不是一个独立的项目。可以和下面介绍的其他一些稀疏项目一起探索，让稀疏稀疏研究均衡发展。

Sparsity on memory

一个聪明的agent，经历了很多事，它将这些事都记录下来，那它是以一种无序的方式和perception，reasoning混在一起，记录在模型中呢？还是单独找一个地方，有序的将这些事件记录下来呢？我认为后者更为智能。

记忆就是时间上的稀疏。当过去所有的事件都有序的保存起来的时候，我们在遇到一件新事件，我们并不需要将过去所有的事件都拿出来看一下，而是只是将相关的事件拿出来审查一下。这种从过去事件中随机读取有用事件就是稀疏在时间上的表现形式。

我们从空间稀疏中总结出的经验是不是可以对时间上的稀疏提供一些参考呢？

Sparsity on composition

之前说过，object composition是一种包装方式，将内部的数据和关系包装起来，外部就不得知晓。这意味着内部数据之间的关系要比对外部数据的关系更为紧密。如果我们将这些数据都映射到高维空间离散的点的话，而关系则是这些点的边。那object其实是一种聚类的过程（是不是用Mincut算法来决定object呢？）。

如此看来，object composition从定义上就是稀疏的。既然composition对AGI如此重要，我们也希望稀疏在其他方面的研究可以对composition有所促进。

我们最终希望有一种形式将各种稀疏模式（如权重、激励、输出、记忆等等）都统一起来。Composition是不是就是我们追求的大统一理论呢？

Few shot learning

现今的大模型都需要极其多的数据来训练，主流的研究方向变成了如何获得更多的数据，在数据没有label的情况下训练。这是世界上只有少数的公司有财力训练超级大模型的另一个原因。这种方式也是不可持续的。

我们应该换一个方向，搜寻减少训练数据需求的方法，即，使用few shot learning。现在研究的方向还是用极其多的数据预训练一个大模型，然后在fine-tune的时候用少量数据训练。这种方式不能本质上解决问题。

我们需要寻找完全不需要众多数据训练模型的方式。这只有在我们对记忆有了比较好的解决方案后才能实现。举例来说，我们看见一只狗，我们训练模型并不是只处理这只狗一次，而是我们需要将以前看见的狗从记忆中拿出来，和刚看见的狗比较，这一个新数据需要在模型内处理数十遍甚至上百遍。这种方法可能会显著减少对训练数据的需求。

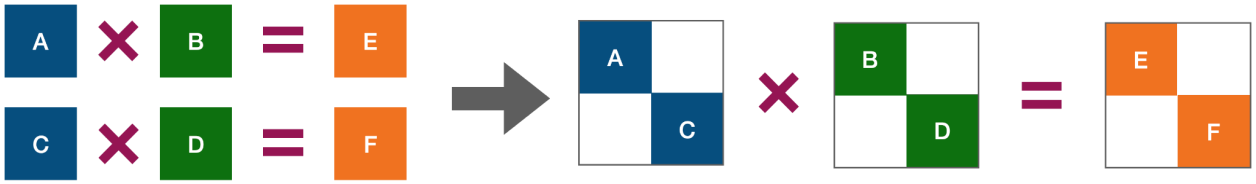
但是，如果我们第一次看见狐狸，那该怎么办呢？我们就需要使用composition的方法，将看见的狐狸能够分解为头，身子，腿，尾巴。头还可以再分解为耳朵、眼睛、嘴巴等等。然后再从记忆中将已知的狗和其他动物也进行分解，比较分解后的部分，将其中不同的部分记录下来，作为狐狸的标签。这时如果我们再看见一只狐狸，我们要进行两次操作，先和前面做的方式一样，这样就可以找到前一次看见的狐狸，再和前一次狐狸做进一步的整体比较，强化印象。

所以，在few shot learning成功之前，我们需要对记忆和composition有很深的理解，而其中稀疏起了至关重要的作用。

Sparsity on evolution

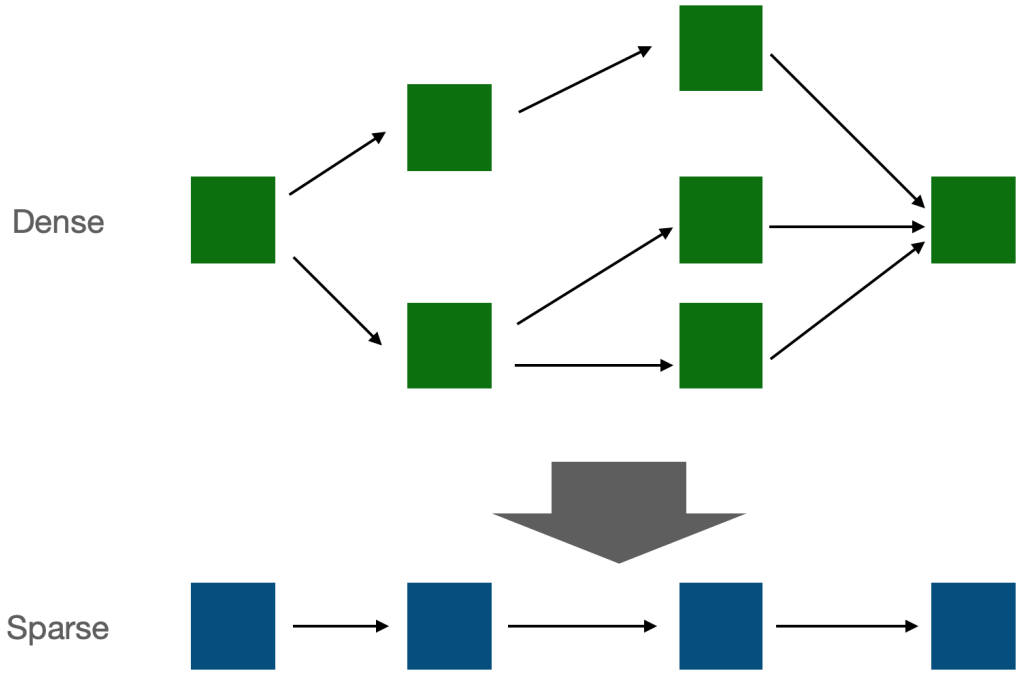
进化指的是从一种结构进化为另一种结构。这需要我们找到一种合适进化的结构模式。如果我们采用稠密模型作为结构的话，模型的多样性比较有限，可以表达模型特征的就是模型深度、每一层宽度、算子大小等等。这些方式对模型多样性的改变很有限。

稀疏也是一种结构，而且是非常细粒度的结构。这样模型结构的改变可以转换为对稀疏格式的改变。同时，稀疏也可以表示一些链接的改变。如图八所示。



图八：用稀疏合并不同路径计算

如上一篇文章介绍，现阶段neural architecture search（NAS）有两大局限，寻找稀疏格式间接的寻找模型结构，寻找稀疏方式可以成为一种factory的映射。另一方面，稀疏作为细粒度的模型结构可以较好的解决搜索building block由人指定的问题。因此，很多NAS问题可以转化为寻找稀疏格式的问题。



图九：转化NAS为寻找稀疏格式

如图九所示，现阶段搜索的方式比较粗。可能有很多枝枝丫丫的。但这种多样的宏观结构，可以转换为一个很简单的一条线的结构，里面每一个block都是稀疏的。因为我们有权重和激励的稀疏，有粗力度和细粒度的稀疏，这样NAS的问题可以转换为寻找稀疏方式的问题，而且比现在NAS的搜索空间更大但是搜索方式更加高效。

稀疏是一种普适的结构，可以是细粒度也可以是粗粒度，有着多样的模型结构选择。稀疏格式可以很好的作为进化的baseline。

Sparsity on federated learning

联邦学习（federated learning）所要解决的问题是：

> Train a deep learning model weights across multiple decentralized devices holding local data samples without exchanging them.

在训练过程中，所有端上的模型是相同的，只是权重的数值不同，我将这种联邦学习称之为联邦学习1.0。

这一问题在算法上的难点是解决训练数据不是独立同分布的（independent and identically distributed），也叫做non-iid。在系统上的难点是降低在端上训练的计算开销以及权重梯度从端到服务器传输的通信开销。

如果我们更进一步，可以在联邦学习的条件下来改变模型结构。我称之为联邦学习2.0。

> Evolve a deep learning model structure across multiple decentralized devices holding local data samples without exchanging them.

这在现在还很少有人研究。而稀疏有可能是研究联邦学习2.0的突破口。

- 从联邦学习1.0的角度来说，在端上更适合稀疏训练
  - 稀疏数据需要的计算量小，计算上更加高效。
  - 稀疏权重的梯度数量也比较少，从端上传到服务器所需通信的数量也较少。
- 从联邦学习2.0的角度来说，稀疏也更适合
  - 稀疏就是一种结构，在每一个端上训练数据不一样的情况下训练出来的稀疏格式就不一样。要改变模型结构只要将稀疏格式传给服务器做选择就好了。如果模型是稠密的话，这种方式改变模型结构是不可想像的。
  - 因为只需要传递稀疏格式，所以不需要在每个minibatch和服务器通信。和联邦学习1.0相比对数据的传输有着极大的减少。

我们将稀疏运用于联邦学习有些low hanging fruits可以摘。另外，我们也可以探寻一下这种方式和超级大网络异步训练的差别。我觉的当模型达到一定程度，这两者要解决的是同一个问题。

总结

这篇文章里我们采用自下而上的方式讨论了为了实现AGI的长远目标，第一个需要着重研究的方向。稀疏计算可以为其他方面研究提供一个很好的基础。在稀疏计算中找出训练稀疏的超级大网络的方法尤其重要。但是，这并不是意味着其他方面不需要研究。对AGI其他方面也研究可以和稀疏研究起到相辅相成的作用，也使得稀疏的研究成果更容易辐射到各个方面。

References

[1] https://arxiv.org/pdf/2004.11946.pdf
[2] https://arxiv.org/pdf/1410.5401.pdf
[3] https://deepmind.com/blog/article/differentiable-neural-computers
[4] https://arxiv.org/pdf/1706.03762.pdf
[5] https://arxiv.org/pdf/2005.14165.pdf
[6] https://arxiv.org/pdf/2101.03961.pdf
[7] https://bdtechtalks.com/2020/08/17/openai-gpt-3-commercial-ai/

[8] https://en.wikipedia.org/wiki/Dennard_scaling
[9] https://arxiv.org/abs/2106.09857
[10] https://tvm.apache.org/
[11] https://medium.com/swlh/synchronous-vs-asynchronous-programming-1bfef19f032c
[12] https://en.wikipedia.org/wiki/Event-driven_programming
[13] https://en.wikipedia.org/wiki/Stream_Processors,_Inc
[14] https://www.cs.ucr.edu/~nael/pubs/micro20-graphpulse.pdf
[15] https://en.wikipedia.org/wiki/Asynchronous_circuit