arXiv:2305.05531v1 [cs.SE] 8 May 2023

Imperial College London

MIT

MEng Individual Project

Imperial College of Science, Technology and Medicine

Massachusetts Institute of Technology

Department of Computing

使用机器学习建模并发错误

作者：Teodor-Rares Begu

导师：Shashank Srikant

学院导师：Una-May 博士 O'Reilly

第二评阅人：Sergio 博士 MaffeisAnandha 博士 Gopalan

2020年5月29日

摘要

近年来，人工智能日益受到关注，机器学习也开始在众多领域得到更多应用，涵盖医疗保健 [1], 金融 [2] 和自动驾驶汽车 [3]。

我们感兴趣的一项机器学习应用，是软件安全性与安全防护，尤其是在并行程序这一背景下。如何自动检测并发错误这一问题长期以来一直吸引着程序员 [4], [5], ，因为额外增加的复杂性使并发程序更容易发生故障。开发此类自动检测工具能为程序员带来显著益处，既能节省调试时间，也能减少意外错误的数量。

我们认为，机器学习有望通过在工具整体准确率和编程语言灵活性方面优于现有方法，助力实现这一目标。然而，由于机器学习方法面临诸多特有挑战（例如，正确标注足够大的数据集、找出最佳模型类型／架构等），我们必须分别处理开发此类工具过程中的各项问题。

因此，本项目的重点是比较常见的机器学习方法与近年来出现的方法。我们将获取并发程序带标签的数据集这一复杂过程抽象为一个合成数据集，并定义和生成该数据集，以模拟现实生活中的（并发）程序。我们围绕各种机器学习模型类型的基本局限性提出假设，并通过在合成数据集上运行大量测试来验证这些假设。我们希望研究结果能够进一步阐明使用机器学习对（并发）程序进行建模时，各种模型类型的优势与劣势，以及其他相关领域（例如自然语言处理）中的类似问题。

致谢

首先，我要感谢我的父母，他们在我大学求学期间始终支持我，并鼓励我无论自己的热情所在是什么，都要勇于追求。

我要特别感谢我的导师 Shashank Srikant 和教职导师 Una-May O'Reilly博士，感谢他们为我提供了这次绝佳的研究机会，并在整个项目过程中给予我指导。

我还要感谢Sergio Maffeis博士协调帝国理工学院与麻省理工学院之间的这一年交流项目。若没有他的帮助，我不仅没有机会参与这个项目，也无法获得在麻省理工学院学习的宝贵经历。

目录

图目录 7

表目录 8

List 列表目录 9

1 I引言 10
1.1 研究动机.111.2 贡献.11

2 背景 13
2.1 并发.132.1.1 同步机制.132.1.2 信号量.132.1.3 并发错误.142.1.4 当前并发工具.162.2 人工神经网络.162.2.1 卷积神经网络.162.2.2 长短期记忆神经网络.172.2.3 DeepSets.172.3 机器学习模型可解释性.17

3 Sy合成数据集 18
3.1 Python.183.2 合成数据生成语法.193.2.1 函数1（空操作函数）.193.2.2 函数2（写入函数）.193.2.3 函数3（读取函数）.203.2.4 语法示例.203.3 底层并发模式.223.4 选择语法和规则背后的推理.233.4.1 规则复杂度.233.5 数据集参数.24

4 数据集筛选器 25
4.1 筛选器 #1 - 随机子集.254.2 筛选器 #2 - 仅前半部分.254.3 筛选器 #3 - 每隔 n 个样本跳过一个.264.4 筛选器 #4 - 可变操作间距.274.5 组合过滤器.274.5.1 筛选器顺序.274.6 验证集.27

5 机器学习模型 285.1 前馈网络.295.2 DeepSets.305.2.1 DeepSets 变体 #1.305.2.2 DeepSets 变体 #2.315.3 卷积神经网络.325.3.1 卷积滤波器.335.3.2 卷积神经网络 - 'wu' 对比 'uw'.345.3.3 卷积神经网络 - 位置方差.355.3.4 卷积神经网络 - 操作间距方差.365.4 长短期记忆网络.375.5 Keras 对比 PyTorch.385.6 训练.38

6 评估 39
6.1 模型性质形式化.396.1.1 对代码位置变化的鲁棒性.396.1.2 对操作间距方差的鲁棒性.406.1.3 区分操作的相对顺序的能力.406.2 测试流程.416.3 迭代实验.416.4 结果.426.4.1 检查模型性质.426.4.2 按筛选器和模式划分的结果.436.4.3 补充说明.44

7 Co结论 45
7.1 总结.457.2 未来工作.45

<escape>8 参考文献 47</escape>

9 附录 51
9.1 F完整结果 51
9.1.1 筛选器 #1.519.1.2 筛选器 #2.519.1.3 筛选器 #3.529.1.4 筛选器 #3 - 位置方差.529.1.5 筛选器 #4.529.1.6 筛选器 #2 +#4.52

图目录

1.1 摘自 CO572 课程讲义 [7], 这个示例展示了一种特定的并发错误（丢失更新）。由于两个线程以特定的交错执行顺序运行，线程 1 对账户 34 的写入被遗忘，因此账户 34 的余额比应有余额少 10000.102.1 原子性违例可能导致不良行为的另外两个示例 . 152.2 并发错误的可视化概览. 163.1 流程的可视化总结——通过反复进行实验来改进我们对任务和上下文的理解至关重要. 185.1 前馈网络类型的模型架构——仅使用全连接层 . . . . . 295.2 DeepSets 变体 #1 架构，与前馈网络非常相似 5.1. 单个深蓝色层表示对之前 3 个浅蓝色层进行逐元素求和 . . . 305.3 DeepSets 变体 #2 架构，与前馈网络相似5.1 .315.4 微调后的卷积神经网络架构概览，与 VGGNet 类似 [8] 其特点是逐步缩小输入维度，同时增加卷积滤波器维度.325.5 对大小为 3 的字符串 ”.wu” 进行独热编码后，会得到一个维度为$3x3$ 的二进制数组（也可以展平为单个维度 9）.335.6 将大小为 1 的单个卷积神经网络滤波器应用于输入.335.7 卷积神经网络需要最大池化才能区分 ’wu’ 和 ’uw’.345.8 卷积神经网络需要最大池化才能在筛选器 2 上取得良好表现. 355.9 由于掩码维度固定，卷积神经网络无法将 ’wu’ 泛化到 ’w..u’ . . . 365.10 长短期记忆网络架构.37

6.1 筛选器1下卷积神经网络的收敛.446.2 筛选器1下长短期记忆网络的收敛.44

表目录

3.1 总结伪代码指令与对其建模的标记之间对应关系的表格.20 3.2 2个伪代码程序及其模拟对应程序之间的对应关系示例.20 3.3 我们的合成数据集的形式化BNF语法.21 3.4 总结函数2与函数3之间的交互如何决定一个“程序”的标签的表格（有效程序用绿色表示（✓），有缺陷/无效程序用红色表示（✗））.22 3.5 简单规则.23 3.6 复杂规则.23 4.1 展示各个筛选器如何将完整数据集划分为训练集和测试集的示例表格.25 4.2 考虑两个函数中操作位置的筛选器 2 划分.26 6.1 按模型类型和属性细分的分数——分数越高，模型性能越好.42 6.2 按模型类型和筛选器细分的分数.43 6.3 按模型类型和模式细分的分数.43 9.1 筛选器1简单版本——训练子集占完整数据集的25%.51 9.2 筛选器1困难版本——训练子集占完整数据集的5%.51 9.3 筛选器 2 简单版本——训练集包含出现在程序前半部分的操作，测试集包含出现在后半部分的操作.51 9.4 筛选器 2 困难版本——训练集包含出现在程序前三分之一的操作，测试集包含出现在后三分之二的操作.51 9.5 筛选器3简单版本——每第4个样本分配到训练集.52 9.6 筛选器3困难版本——每第19个样本分配到训练集.52 9.7 筛选器3（含位置方差）简单版本——每第5个样本分配到训练集.52 9.8 筛选器3（含位置方差）困难版本——每第20个样本分配到训练集.52 9.9 筛选器 4 简单版本——训练集中有意义的操作之间的距离为0和2，测试集中为1.52 9.10 筛选器 4 困难版本——训练集中有意义的操作之间的距离为1，测试集中为0和2.52 9.11 筛选器 2 和 4 简单版本——结合筛选器 2 的简单版本与筛选器 4 的简单版本 .52 9.12 筛选器 2 和 4 困难版本 —— 结合筛选器 2 的简单版本与筛选器 4 的困难版本.52

代码清单目录

2.1 互斥信号量模式.142.2 生产者-消费者信号量访问模式.142.3 在这个更简单的示例中，当两个线程都在第 6 行等待时，就会发生死锁。如果两个线程都按照相同的顺序执行 $down$ 和 up，就可以轻松避免这种情况。153.1 示例：20 个随机样本，以字符串数组的形式呈现.214.1 应用第三个筛选器后训练集中的前 10 个样本 26

第 1 章

引言

所有形式的并发错误都特别难以发现，因为程序中的任何程度的并行性都会导致已执行指令的序列变得不确定。由于并发线程可以在运行时以任意方式交错执行（下方展示了一种有问题的交错执行方式 1.1），可能的线程交错数量会呈指数级增长。因此，分析工具难以预测代码中的错误，因为它们无法穷举搜索所有交错执行的可能性。

当前方法分为静态分析器 [9, 10]和动态分析器 [11, 4], ，其中大多数方法（无论是静态还是动态）都基于规则。事实上，其中一些方法 [9, 11] 是在先前工作的基础上，进一步探讨并发程序类型系统的形式化定义 [12, 13, 14]。

近期也有一些将机器学习应用于这一特定问题的研究 [15] 。不过，我们相信，机器学习模型能够识别出比其中所展示的模式更复杂的模式。此外，人工智能或统计模型也已被应用于软件领域，其中最值得注意的是对程序属性（例如变量名或类型）的预测 [16, 17, 18]。另一篇较新的论文 [19] 全面考察了人工智能与编程语言交叉领域的所有相关工作。

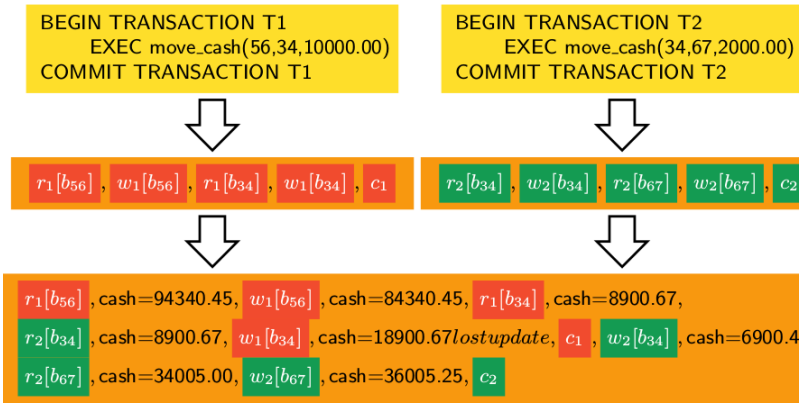


图 1.1：摘自 CO572 课程讲义 [7], ，此示例展示了一种特定的并发错误（丢失更新）。由于两个线程的一种特定交错执行，线程 1 对账户 34 的写入被遗忘，因此账户 34 的余额比应有金额少 10000。

1.1 研究动机

基于规则的工具常见的一个缺陷是缺乏灵活性。开发此类工具需要具备针对特定编程语言的技术专长，而且工具开发完成后，相关工作无法快速迁移到另一种编程语言上。对于 Solidity 这类较新的语言而言，这尤其棘手 [20] 因为这类语言可能没有现成的并发检测工具。

我们认为，可以通过采用机器学习方法来解决这一问题，因为模型能够学习所有编程语言通用的并发模式。然而，这就要求使用这些语言的通用中间表示来训练模型（例如 LLVM 中间表示 [21], ，它是一种较低层级的代码表示，位于源代码与汇编代码之间）。不过，在这种情况下，每种语言都需要进行特定的转换，将源代码转换为中间表示。尽管如此，我们认为这比从头开发新的并发检测工具更可行。此外，每种新语言都能受益于在其他语言的数据集中发现的并发模式。

这让我们看到机器学习方法可能具备的另一项优势：准确率。理想情况下，我们设想，只要拥有足够大的、涵盖多种编程语言的数据集，机器学习模型就能够胜过基于规则的工具，因为它们可以检测出那些过于出乎意料或过于复杂，以至于人类难以针对其编写规则的缺陷模式（当前方法便是如此）。

与大多数机器学习任务一样，数据往往是问题中最关键的方面。然而，在我们的研究情境中，这也恰恰是最具挑战性的方面。要正确判定一个并发程序是否存在错误，需要付出大量人工劳动（对于非常复杂的程序，甚至可能无法完成标注），因此很难获得并发程序的标注数据集。此外，相较于能否获得带标签的数据集，「是否存在能够成功从代码中学习模式的机器学习模型」这一问题的不确定性更大。前一个问题的不确定性与后一个问题所需的繁重工作相结合，构成了本 SuperUROP 项目的目标。因此，我们希望帮助澄清前一个问题，同时深入了解模型在输入性质与此任务相似的任务中的学习能力。

在本项目中，我们分析不同的机器学习模型类型和架构，提出关于模型学习能力的假设，并在合成数据集上开展实验，以验证最初的推理。合成数据集旨在模拟真实世界中的（并发）程序，同时避免处理实际程序时的复杂性。使用生成的合成数据集有两大优势，首先是能够获得训练数据。另一方面，对数据集的完全掌控也使我们能够避开其他与将代码作为神经网络输入进行建模相关的问题（例如输入大小可变、确定使用哪种中间表示等）。因此，我们不仅可以逐步解决每个代码建模问题，还可以精细控制模型将接受训练和测试的样本，从而全面压力测试模型的局限性和能力。

1.2 贡献

我们通常测试四种类型的人工神经网络，即仅包含全连接层的简单前馈网络、卷积神经网络（CNN）、循环神经网络（RNN）（采用长短期记忆网络（LSTM）的形式），以及一种名为 DeepSets 的最新神经网络架构 [6]。我们的结果证实了这一假设：迄今为止，只有卷积神经网络和长短期记忆网络在仅获得全部可用数据集中的特定子集时，才能实现泛化。Across

在我们所有的实验变体中，卷积神经网络和长短期记忆网络模型类型的准确率始终超过 90%（所有实验的总体准确率分别为 98.4% 和 98.6%），而前馈网络和 DeepSets 的准确率分别仅为 78.2% 和 78.1%。未来工作还可包括测试更多循环神经网络变体，例如门控循环单元网络（GRU）[22]或图神经网络 [23]。

因此，我们的主要贡献总结如下：

- 生成合成的带标签数据集，用于模拟真实（并发）程序中发现的模式。我们使用 BNF语法定义一种形式化的玩具编程语言，然后用它生成模拟程序。接着，我们定义底层规则，根据这些规则，将使用该语言生成的模拟程序分类为有缺陷或无缺陷。尽管这些规则对人类而言简单易懂、容易解释，模型也必须展现出这种能力。</final

- 用于将数据集划分为训练集和测试集的筛选器：通过谨慎划分，我们可以测试模型的特定属性。与所有机器学习任务类似，在训练过程中仅接触某些类型的样本，可能会导致模型过拟合，并在测试过程中表现出较低的准确率。通过使用这些筛选器，我们可以选择性地确定哪些样本只能在训练过程中被模型接触，哪些样本只能在测试过程中被模型接触，从而测试模型的泛化能力以及抵御过拟合的鲁棒性。

- 推理每种机器学习模型类型应具备哪些属性，并通过在合成数据集上评估模型进行实证验证。我们提出一组理想属性，并推理哪些模型可能具备这些属性，以及原因。我们使用特定筛选器为这些属性定义测试集，然后考察模型在这些筛选器上的准确率。

第 2 章

背景

2.1 并发

并发是指程序能够让多个代码片段（线程）同时并行执行 [24]。例如，这可以提高计算密集型程序的执行速度，也能为带有图形用户界面组件的应用程序提供更流畅的用户体验。与非并发程序不同，并发程序没有确定性的顺序执行过程，因为不同次执行中，线程的交错执行几乎不可能完全相同。这可能由多种因素造成，例如线程执行的相对速度、操作系统的线程调度 [25]，或对外部资源或输入的依赖。

这种非确定性特性极大地影响了并发程序自动化测试的可复现性 [26] ，进而影响并发程序的可靠性。并发程序可能通过所有自动化测试，但在生产环境中，由于线程交错执行方式不同，程序仍可能出现错误。因此，编写自动化测试并不那么可靠，这也说明了需要能够准确检测并发问题的工具。

2.1.1 同步机制

同步机制是指程序员用来协调程序中多个线程访问共享内存的方法，以确保同一时刻最多只有一个线程能够进入所谓的临界区 [27]。每当同步方法能够正确确保对临界区的互斥访问时，代表该临界区的代码段便成为原子性的，从而确保不受其他线程的干扰 [12]。然而，滥用同步机制或完全不使用同步机制，都可能导致下一节所述的并发问题。

有大量同步方法可供选择，具体取决于源语言所提供的机制，从 Java 中的高级同步方法 [28] 到 C 中的底层锁++ [29]。

2.1.2 信号量

在本节中，我们将介绍信号量同步方法的机制，因为我们的合成数据集复现了其功能。信号量由一个正整数变量（也称为计数值）以及两个方法表示，我们将它们称为 up 和 $down$。每当线程对信号量执行 up 操作时，其计数值就会加 1。每当线程对信号量执行 $down$操作时，根据信号量计数值的不同，可能发生两种情况。如果该值大于零， $down$ 方法只需将计数值减 1，程序便继续执行。如果信号量为零，调用 $down$ 方法的线程会暂时暂停执行，直到另一个线程调用 up 方法，使计数器递增为非零值。在

每次操作最多只能有一个线程恢复执行 up 。信号量的一个重要属性是， up 和 $down$操作都是原子性的，这意味着任意时刻都不能有两个线程同时访问共享计数器。下面展示了使用信号量确保临界区互斥的正确模式2.1。

清单 2.1：互斥信号量模式

```
P操作( 信号量 ) 临 界 区 .
. . V 操作 ( 信号量 )
```

信号量还可用于其他同步场景，其中一种常见场景称为生产者-消费者问题 [30]。在这种情况下，一个线程称为生产者，它会向缓冲区中添加一个项目；该缓冲区与第二个线程共享，第二个线程称为消费者，它会从缓冲区中取出项目并进行处理。如下所示，可以使用信号量来同步这两个线程 2.2。

清单 2.2：生产者-消费者信号量访问模式

```
将信号量计数初始化为 0
. . .
    线程 1                          | 线程 2
    . . .                           |  . . .
    添加项目（缓冲区，     项目）   | P操作（信号量）
    V操作（信号量）                 | 项目  =移除缓冲区中的项目
    . . .                           |  进 程 ( 项 目 )
    . . .                           |  . . .
```

这里假设缓冲区的容量无限。我们将说明如何使用信号量，确保消费者总是先等待有项目放入缓冲区，然后才能将其取出并进行处理（如果从空缓冲区中取出项目，应视为错误）。

2.1.3 并发错误

并发错误可归因于同步机制使用不当或缺失。虽然使用不当的原因可能复杂多样，但缺少同步机制可能是因为程序员对编程语言或程序本身作出了某些错误假设。下面根据导致这些错误的原因、它们的表现形式以及对程序造成的负面影响，总结并发错误的四个主要类别。

- 原子性违例当多个线程同时访问共享内存时，就会发生这类情况，从而违反原子性 [11]并导致意料之外的行为。这里总结了此类意外行为中的一种1.1 ，但根据操作和线程交错执行的情况，也可能出现许多其他异常，如图 2.1所示。

- 竞态条件
  竞态条件的负面影响与原子性违例相似1.1 但我们会区分导致它们的原因。我们认为，原子性违例的主要原因是对原子性的错误假设或对同步机制的误用；而竞态条件的一个主要原因，则是缺少同步机制，这可能是因为忘记使用，也可能是错误地假设线程执行速度。举例来说，如果从上面的生产者-消费者示例中删除 down(semaphore) 这一行2.2，就可能出现竞态条件：线程 1 抢先向缓冲区添加项目，而线程 2 随后尝试从空缓冲区中移除项目。

- 死锁
  当两个或更多线程暂停执行，等待彼此释放各自的资源时，就会发生这种情况，从而形成循环依赖 [31]。死锁通常会导致程序无法取得进展，此时操作系统往往不得不介入并打破这个循环。

- 活锁
  活锁 [31] 与死锁非常相似。两者的区别在于，发生活锁时，线程不会暂停执行，而是会重置到进入循环依赖之前的某个状态。不过，如果线程总是重新进入循环依赖，就会形成活锁。因此，表面上看程序似乎在取得进展，实际上却并非如此。

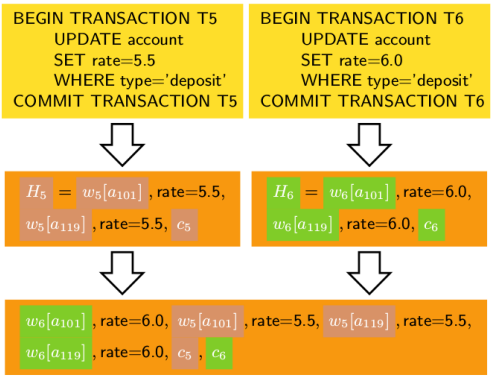
Excerpt from `java.lang.StringBuffer`

```
public final class StringBuffer {

    public synchronized
          StringBuffer append(StringBuffer sb) {
      int len = sb.length();
      ... // other threads may change sb.length(),
      ... // so len does not reflect the length of sb
      sb.getChars(0, len, value, count);
      ...
    }

    public synchronized int length() { ... }
    public synchronized void getChars(...) { ... }
    ...
}
```

(a) 摘自 [11], , 此示例展示了一种原子性违例，其原因是错误地认为 append() 函数具有原子性



（b）脏写异常 [7]：这种特定的交错执行会使银行处于不一致状态，其中两种利率之间存在非零差值账户

Fi图 2.1：原子性违例可能导致不良行为的另外两个示例

代码清单 2.3：在这个更简单的示例中，当两个线程都在第 6 行等待时，就会发生死锁。如果两个线程都按照相同的顺序处理 *down* 和 *up*，就可以轻松避免这种情况。

```
0: 信号量 1 =_ 1
1: 信号量 2 =_ 1
2: ...
3:      线程 1                    |  线程 2
4:      ...                       |  ...
5:      P操作( 信号量 1 ) _       |  P操作( 信号量 2 ) _
6:      P操作( 信号量 2 ) _       |  P操作( 信号量 1 ) _
7:      关键的      章.....       |  关键的      section...
8:      V操作( 信号量 2 )         |  V操作( 信号量 1 ) _
9:      V操作( 信号量 1 )         |  V操作( 信号量 2 ) _
10:     ...                       |  ...
```

这些并发问题、它们的成因，以及它们可能导致的非预期行为，都可以在图 2.2 中直观地看到；不过，该图并未涵盖所有可能的并发问题。

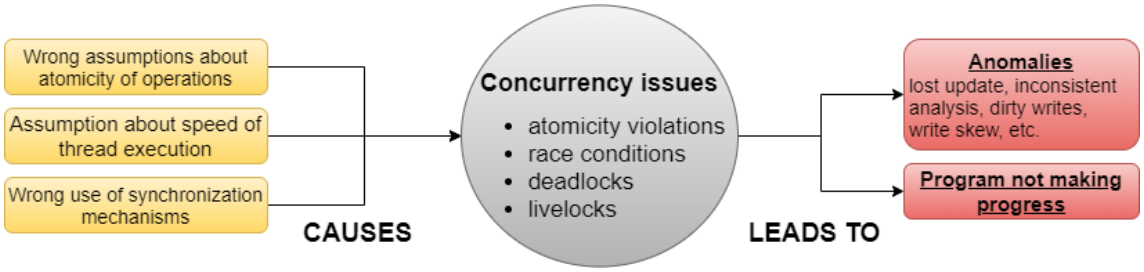


图 2.2：并发错误的可视化概览

2.1.4 当前并发工具

如前所述，现有的并发漏洞检测研究并未考虑并发执行的所有可能轨迹，因为这些轨迹会呈指数级增长。相反，这些方法采用基于规则的方式，其中已知方法包括 [11, 9]进一步扩展并发类型系统，例如此处所示的类型系统 [12]。这些方法利用归约定理，将大型代码块简化为单个原子操作，从而简化分析过程。这些定理建立在类型系统之上，不同的操作类型表现出不同的归约语义。例如，由于正确使用信号量作为同步机制，2.1 中的整个代码片段可以归约为单个原子操作。

其他方法不使用类型系统，而是像本文 [4]那样，对程序的共享内存以及用于访问它的同步机制进行监视。所有上述方法在性能和准确率方面都各有取舍。然而，随着并发程序的复杂性不断增加，人类要推理缺陷模式并针对它们编写工具，会变得困难得多。

2.2 人工神经网络

尽管神经网络背后的理论早已存在， [32], 但计算能力的近期提升与大数据革命相结合，意味着神经网络终于开始能够胜过以往的人工智能方法。 [33]我们不会全面介绍神经网络的发展历程，因为这是一个常见的话题，网上有大量资源可供参考。 [34, 35], 不过，我们会介绍本项目所关注的神经网络不同变体中的重要方面。

2.2.1 卷积神经网络

卷积神经网络是神经网络的一个子类，理论上于 1980 [36] 提出，并在 2012 [37]的 ImageNet 竞赛中取得了最佳性能。与标准的密集型对应结构相比，卷积层具有多项优势，其中最广为人知的是其计算效率显著更高。不过，另一项与我们尤其相关的优势是局部性。标准前馈网络能够像拥有全局视野一样处理输入特征，而卷积层则以局部视野处理特征。在具有空间成分的问题中（例如计算机视觉、自然语言处理或我们所研究的问题），这使它们能够学习与输入特征向量中的绝对位置无关的特征。

2.2.2 长短期记忆神经网络

长短期记忆网络 [38]是循环神经网络（循环神经网络）的一种变体，我们之所以对其感兴趣，是因为它能够成功学习序列数据，例如语音识别 [39], 金融序列预测 [40] 和自然语言处理 [41]。事实上，从代码中预测属性的问题，其输入特征结构可能与自然语言处理问题非常相似，因此我们对循环神经网络很感兴趣。长短期记忆网络还通过改进基本循环神经网络架构中的一个重要缺陷——长序列中的梯度消失问题——进一步提升了性能 [42]。与卷积神经网络非常相似，循环神经网络也能够学习基本上与输入特征向量中的绝对位置无关的特征，这正是它们能够成功学习上述任务的原因。

2.2.3 DeepSets

另一种我们感兴趣的神经网络架构是较新的 DeepSets 架构[6]，它将输入特征建模为置换不变，即如果输入向量中特征的顺序发生变化，机器学习模型仍会作出相同的预测。他们从形式上证明，只有对输入特征求和才能实现置换不变，并在神经网络层中实现了这一性质。他们展示了该架构在多个应用中的成功，其中与我们的问题最为接近的是数字求和预测 [43],，在该任务中，他们取得了比长短期记忆网络更好的收敛效果。鉴于我们的问题具有并发性，对特征进行置换不变建模尤其值得关注。考虑以下情况：如果多个线程并行运行，那么在运行时，所有操作之间可能存在数量呈指数级增长的全局顺序。然而，我们希望单个模型预测能够涵盖所有可能的顺序，因为操作的实际顺序只能在运行时观察到，而且如前所述，探索所有顺序并不可行。理想情况下，如果不存在任何可能的顺序／交错执行会导致非预期的程序行为，或者等价地说，如果所有可能的顺序在程序输出方面实际上都等价，那么模型就应将程序分类为存在错误。

2.3 机器学习模型的可解释性

如今，机器学习面临的最大问题之一是某些模型的黑箱特性：更复杂的模型能够实现更高的准确率，但可解释性却较低 [44] （神经网络是其中最显著的例子）。在所有解释方法中，我们可以区分出两种重要且相互兼容的理解模型学习机制的方法。

第一种方法旨在了解输入特征如何影响模型预测，包括它们的重要程度，以及影响预测的方向。这种可解释性方法围绕每个预测在局部范围内发挥作用，首先通过改变原始样本的特征，为每个样本构建一个次级数据集。随后，训练一个非常简单的模型，使其在该次级数据集上的预测结果与原始模型的预测结果相匹配。简单模型的权重表明各特征的重要性，以及它们影响预测的方向 [45]。借助 LIME [46]和 SHAP（SHapley 加性解释） [44]等工具，可以轻松完成此类分析。

第二种可解释性方法正是我们在整个项目中采用的方法。这种方法尝试通过有选择地过滤训练集和测试分布，来衡量模型的泛化能力。采用这种解释方法的研究动机是，当底层训练集和测试分布不同时 [47], 、使用对抗网络时 [48, 49]，或者训练样本不足时 [50]，机器学习模型的准确率可能会受到显著影响。尽管已有用于探查模型的工具 [51], ，我们仍开发了自己的数据集和过滤方法，以实现细粒度控制。事实上，使用合成数据集来测试／验证神经网络属性的方法并不鲜见 [52]。

第3章

合成数据集

尽管一开始我们就对想要测试的神经网络属性有了一些设想，但我们的实验过程无疑是迭代式的。每进行一组实验，我们都会加深对特定模型类型在某些输入上为何比其他类型表现更好的整体理解。进而，这让我们能够思考更多需要测试的场景，并据此增加合成数据集的复杂度。

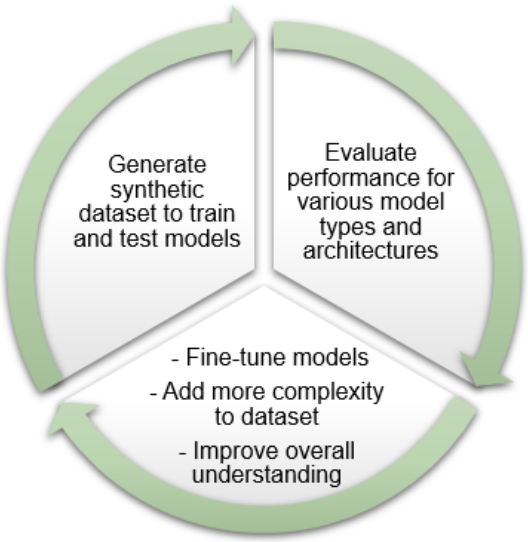


图 3.1：我们流程的可视化总结——反复进行实验对于加深我们对任务和上下文的理解至关重要

3.1 Python

我们选择使用 Python 生成和处理合成数据集，并开发机器学习模型 [53]。Python 不仅易于使用、开发速度快，非常适合用于实验，还支持种类繁多的机器学习库，包括但不限于 Keras [54], PyTorch [55], Scikit-learn [56]。为此，我们还使用了 Jupyter Notebook [57],，进一步简化开发。

3.2 用于合成数据生成的语法

具体而言，合成数据集中的每个样本都由字符数组（即字符串）构成，其中每个字符都可以视为代表某项程序操作的词元。该数据集旨在模拟我们可能在（并发）程序中看到的简单模式，以及由于根本性限制而可能难以被机器学习模型捕捉的某些模式。我们定义了可访问共享内存或对共享内存的访问进行同步的“并发”操作。我们还定义了不会以任何方式影响共享内存、而是充当噪声的“空操作”。此外，还有两个模拟对象，分别是共享变量和同步机制。这些操作的使用方式及其相互作用，决定了根据我们定义的一些底层规则，生成的样本是否被视为存在缺陷。

为了全面测试机器学习模型和架构的不同变体，我们生成程序中所有可能位置上的操作的所有可能组合。同时，我们只使用其中一个特定子集来训练模型。如何选择这个子集确实非常重要，因为我们可以通过选择模型在训练阶段能够看到哪些样本，以及在验证/测试阶段能够看到哪些样本，来测试模型的泛化能力（有关筛选器的更多详情，请参阅第 4 章 4）。

我们生成 3 个字符字符串，用于模拟并行运行的 3 个并发函数。第一种函数不会访问共享内存，因此不会影响样本是否被标记为存在错误；第二种函数尝试写入或初始化一个变量，而第三种函数随后会读取该变量。如果存在任何可能性，使得第三种函数中的读取发生在第二种函数中的初始化之前，我们就认为该程序存在错误。

在接下来的章节中，我们将介绍每种操作、函数及其底层规则，以及它们与样本真值标签的关联。我们还将在 3.1、3.2、3.3 中以表格形式直观概述我们的语法，并在3.1中展示生成样本的实际示例。接下来的章节还将说明我们选择这些操作、函数及其底层规则的原因。目前，可以将其与我们之前的生产者-消费者示例2.2进行具体比较：在我们的场景中，有缺陷的样本（即先读后写）类似于尝试从空缓冲区中移除元素。在生产者-消费者示例中，这种错误行为通过使用同步机制（通常是信号量）得以避免；我们也在玩具用例中采用了相同的机制。

3.2.1 函数1（空操作函数）

第一类函数完全不会影响全局状态。它是一个形式为 ’(,.)+’ 的随机正则表达式，其中每个符号 (, .) 都应被解释为一个不会影响程序全局状态的操作。例如，.,. ,... . . . 可能就是这样一个生成的字符串，它由一系列字符组成，对应于某个特定函数，其执行过程是先执行 ’.’，接着执行 ’,’，然后执行 ’ ’，依此类推。与其包含的操作非常相似，这类函数的目的是充当噪声，使模型学会忽略它。

3.2.2 Function2（写入函数）

第二种类型的函数通过写入/初始化共享变量来影响程序的全局状态。它可以执行以下 3 种不同操作中的任意一种，也可以完全不执行任何操作：

- 仅写入/初始化变量（我们用“w”表示）
- 写入/初始化变量，并在之后对信号量执行 V 操作。这表示正确使用同步机制。我们将其记为 ’wu’

- 执行信号量的V操作，然后写入/初始化变量。这表示同步操作使用不当，因为操作顺序错误。我们用“uw”表示这一情况。

函数2也可以在影响状态的操作之前、之间和之后执行任意数量的不影响状态的操作。一种这样的模式可能是 ’. ,w u.,..’。

3.2.3 函数 3（读取函数）

最后，第三类函数可以访问第二个函数写入的全局变量。然而，如果第二个函数尚未向该变量写入内容就访问它，则会构成错误程序。第三类函数包含以下操作：

- 仅读取变量，记作 ’r’
- 检查该值此前是否已写入。如果是，则读取共享变量；如果否，则读取默认值。我们将其记作 ’cr’。
- 对信号量执行 P操作，然后读取变量。这表示正确使用了同步机制。我们将其记作 ’dr’。

因此，这个函数生成的一个可能字符串可以是 ’.., r.’。

3.2.4 语法示例

	Function 2	Function 3	Function 2	Function 3	Function 3	All function types
Pseudocode instruction	**w**rite to shared variable	**r**ead from shared variable	**u**p semaphore (synch mechanism)	**d**own semaphore (synch mechanism)	**c**heck if shared variable has been written to before yes ⇒ **r**ead shared variable no ⇒ **r**ead default value	other operations not affecting shared state
Corresponding token	**w**	**r**	**u**	**d**	**cr**	Any of , . _

表 3.1：总结伪代码指令与对其进行建模的标记之间对应关系的表格。

	Function 2	Function 3	Function 2	Function 3
Pseudocode program	… **w**rite to shared variable **u**p semaphore …	… **d**own semaphore **r**ead shared variable ….	… **w**rite to shared variable …	… **c**heck shared var assigned **r**ead shared var / default value ….
Corresponding token string	…**wu**…	…**dr**…	…**w**…	…**cr**…

表 3.2：两个伪代码程序与其模拟对应项之间的示例对应关系

<no-op> := ‘.’ | ‘,’ | ‘_’
<no-ops> := <no-op> <no-ops> | <no-op> | ε

<f2-op> := ‘w’ <no-ops> ‘u’ | ‘u’ <no-ops> ‘w’ | ‘w’ | <no-op>
<f3-op> := ‘d’ <no-ops> ‘r’ | ‘c’ <no-ops> ‘r’ | ‘r’ | <no-op>

<F1> := <no-ops>
<F2> := <no-ops> <f2-op> <no-ops>
<F3> := <no-ops> <f3-op> <no-ops>

Program := <F1> <F2> <F3>

表 3.3：我们合成数据集的形式化 BNF语法

代码清单 3.1：示例：20 个随机样本，采用字符串数组的形式

```
#    函数1                    函数2                    函数3                    标 签
[[’’’°,__._,.,._.,’,  ’_____.,_.uw._.,’,  ’___..,__.__dr_._’,  True],
 [’,°___,,._,,.__.,.’,  ’__._,__w__._,,,,’,  ’_,.,,,.,_.dr_...’,  True],
 [’,...__..,.,,,_..’,  ’.,___.__.wu___.,’,  ’._cr._..__,,,.,.’,  False],
 [’.,,._..._.,_,,,..’,  ’_..__,,,.,.__.,,’,  ’,_,._._,_,.r,,__’,  True],
 [’.__.,,.,,_.,___,’,  ’_._,,,._...._,_,,’,  ’._.,_dr.,,..._,__’,  True],
 [’,,__,,_.___...,_’,  ’_,__,_,_,,_.___,’,  ’_,.,,cr.._,.._,.’,  False],
 [’.._...,.,._,,.,,’,  ’,._,,.,,,.w_._.,’,  ’,,.,,__.,.__,__,’,  False],
 [’._.,...._.._....._’,  ’_,_...__.,,.wu,_’,  ’_..._.,.._._,_,._’,  False],
 [’.._._,,_._,....,’,  ’_....__,.,uw,_._’,  ’,.,_,,.._,__r,,,’,  T真],
 [’.,_..,,._,__.,,.’,  ’,___,_..._,_,w.,_’,  ’,,._,,.__,cr__.’,  False],
 [’,_._,___.,.,,,.,’,  ’,.,_.__.__,,,...’,  ’.,,.._____.__,,.cr’,  False],
 [’_______,.,_.,._,’,  ’.,_._,,__,.,.wu,’,  ’,_.__._,,,.,__,,’,  Fa否则],
 [’__..,.._.,__,.__’,  ’.___,_.w_._,_...’,  ’,_,_,.,_..,,____’,  False],
 [’.._.,.,,._..._..,’,  ’,..,_w..,.,,.,_.’,  ’,_..,,._r__,,_,_’,  True],
 [’_____.___._,..,,’,  ’.,___,._,.,w_,._’,  ’,_.,r..,_,._,_,,’,  T真],
 [’,.,,_.,,,__.,__.’,  ’,.,,,,.,u,,w.._,’,  ’d,_r_,_....,,._,’,  T真],
 [’._.,,_,,._,_,.’,  ’.__.__._uw..,,,,’,  ’_.__,,_.dr_,,..._’,  T真],
 [’.,__,.,.....,.,.’,  ’..__uw_.___,___,’,  ’__,...._,.r._,.,_’,  T真],
 [’_._____.._..._,,,’,  ’.,__wu._,,,.____’,  ’_,___cr,_.,_.__.’,  F假],
 [’....__..,_.,..._,,’,  ’._w..u_.__..,.,.’,  ’__._,_.,._,,d.,r’,  Fals e]]
```

3.3 底层并发模式

一旦我们定义了与简单并发程序相对应的语法，就可以定义一些规则，用来判断随机生成的三个函数的组合是否对应有缺陷的（✗）程序（✓）。

因此，与我们的生产者-消费者示例一样，如果函数3在没有任何同步机制或检查的情况下尝试读取共享变量，程序就可能存在竞争条件。因此，任何函数组合中，只要第三个函数仅包含“r”操作，就是有缺陷的（✗）（见下表第2列 3.4）。

不读取共享变量（即函数3中没有字符“r”），或在读取共享变量之前进行检查（即函数3中出现“cr”），这两种情况都构成无缺陷程序（✓）（见下表第1列和第3列 3.4）。

只有在函数3正确使用同步机制时，函数2的操作才会影响程序的真值标签（即下表3.4第4列中，函数3出现 'dr'）。在这种情况下，只有函数2也必须正确使用信号量，先写入共享变量，再对信号量执行 V 操作（对应 'wu'），程序才有效（✓）。这里假设信号量计数初始化为 0，因此函数3只有在函数2增加信号量计数后才能继续读取共享变量。如果未使用信号量，则程序被认为存在错误（✗），因为函数3会卡在对信号量执行 P 操作上。如果信号量使用不当，例如在写入变量之前就对信号量执行 V 操作（'uw'），程序同样被认为存在错误（✗）。这是因为第三个函数中的 'dr' 可能交错出现在 'u' 和 'w' 之间（即 $u \to d \to r \to w$ 的全局执行轨迹）。

Function 2 \ Function 3	no-op	‘r’	‘c’ ‘r’	‘d’ ‘r’
no-op	✓	✗	✓	✗
‘w’	✓	✗	✓	✗
’u’ ‘w’	✓	✗	✓	✗
‘w’ ‘u’	✓	✗	✓	✓

✓ = valid/bug free program　　✗ = invalid/buggy program

表 3.4：总结函数2与函数3之间的交互如何决定“程序”的标签（绿色表示有效程序（✓），红色表示存在错误/无效程序（✗））

3.4 选择语法和规则的推理依据

如前所述，我们的语法和规则在一定程度上受到生产者-消费者示例2.2的启发。然而，生产者-消费者模式并不像其他一些并发模式（例如互斥示例2.1）那样常见。尽管如此，我们认为，由于两个函数的操作明确分离，我们的语法具有更好的模型可解释性。如果两个函数具有相同或相似的操作集合（例如，在对互斥2.1进行建模时），我们就无法像评估章节中将要展示的那样，充分测试模型性质。

此外，根据函数2和函数3中操作的组合，还可以将其与背景部分讨论的标准并发错误进行比较2.1.3。函数3不使用任何同步机制会导致竞态条件，函数2不使用任何同步机制会在函数3对信号量执行P操作时导致死锁，而误用函数2中的同步机制则构成原子性违例。

3.4.1 规则复杂度

我们设计底层规则时，使模式学习具有两个复杂度级别。第一个级别最为简单，因为模型只需考虑函数3的操作。然而，第二个复杂度级别要求模型还要考虑函数2的操作，以及函数2与函数3之间的交互。只有先成功学习较简单的规则，模型才能成功学习复杂规则。

Func 2 \ Func 3	no-op	‘r’	‘c’ ‘r’
(*)	✓	✗	✓

表 3.5：简单规则

- 在 16 种可能情况中（函数2和函数3各有 4 种），仅考虑函数3即可正确分类其中 12 种。
- 我们仅测试模型学习是否包含操作 ’c’ 和 ’r’ 的能力。

Func 2 \ Func 3	‘d’ ‘ r’
no-op	✗
‘w’	✗
’u’ ‘ w’	✗
‘w’ ‘ u’	✓

表 3.6：复杂规则

- 在 16 种情况中的 4 种里，模型必须考虑两个函数的操作。
- 模型不仅要学会同时包含 ’u’ 和 ’w’，还必须考虑二者的相对顺序，因为顺序实际上很重要：’uw’ 被视为有缺陷（✗），而 ’wu’ 被视为有效（✓）。
- 无法学习这些复杂模式的模型，在函数3中出现 ’dr’ 时，会将每个样本都预测为有缺陷（✗），因为这是最佳折中方案
- 我们还通过实证观察到6.3，’wu ∣ dr’ 案例给我们的模型带来了最大的挑战

3.5 数据集参数

通过使用灵活的 API，我们可以控制合成数据生成的各种参数，包括：

- **控制每个样本中可出现的操作数量**
  我们可以控制每个样本中每个函数的操作数量。例如，3.1中的样本均以每个样本包含 10 个操作的方式生成。改变每个函数的操作数量，也能让我们控制可生成的样本总数，因为可能的操作组合数量会随函数规模呈二次增长。通过改变同一数据集中各样本的操作数量，我们还可以测试程序对规模变化的鲁棒性。

- **影响状态的操作之间会出现多少个空操作**
  根据我们的规则，样本之间空操作数量的变化（例如，“wu”、“w.u”和“w..u”的操作间距分别为 0、1 和 2）不应影响样本的真值标签。因此，模型也应能据此进行泛化。此参数与下一章所述的筛选器 #4 密切相关。

- **样本在数据集中出现的特定顺序，按其包含的模式及其出现的位置排列**
  按特定顺序生成模式，不仅有助于可视化，也有助于根据操作的位置清晰地区分样本（下一章 4.2 中展示了这种区分方式）。此参数与下一章所述的筛选器 #2 密切相关。

第 4 章

数据集筛选器

如 2.3 所述，解读模型能力的一种方法是仔细区分训练集和测试集样本，这也将是本节采用的方法和探讨的主题。最初，区分训练集和测试集的唯一方法是对完整数据集进行随机子采样。然而，随着我们对问题的理解不断加深，我们可以添加更多筛选器，并测试更多模型性质。迄今为止，我们已开发出四种此类筛选器，未来工作还包括开发其他筛选器。我们将在以下各节讨论这些筛选器，但参考示例表 4.1 始终会有所帮助。

Dataset \ Filter	Random subset (#1)	First half in train (#2)	Adding every 2nd op (#3)	Variable ops distance (#4)
Full dataset	[w....] [.w...] [..w..] [...w.] [....w]	[w...] [.w..] [..w.] [...w]	[w....] [.w...] [..w..] [...w.] [....w]	[wu..] [.wu.] [..wu] [w.u.] [.w.u] [w..u]
Train set	[.w...] [....w]	[w...] [.w..]	[w....] [..w..] [....w]	[w.u.] [.w.u]
Test set	[w....] [..w..] [...w.]	[..w.] [...w]	[.w...] [...w.]	[wu..] [.wu.] [..wu] [w..u]

表 4.1：示例表，展示各个筛选器如何将完整数据集划分为训练集和测试集

4.1 筛选器 #1 -随机子集

Ran根据给定的训练测试比参数对完整数据集进行域分离 **r**

这是最基础的筛选器，旨在反映程序在现实生活中的随机性，同时也作为一个简单的基线，用于比较其他筛选器的性能。

4.2 筛选器 #2 -仅前半部分

有意义的操作仅出现在程序前半部分的样本，与有意义的操作仅出现在程序后半部分的样本之间的区分

这用于测试模型对操作位置变化的鲁棒性。即使模型在训练期间能够正确分类仅出现在程序前半部分的模式，在测试期间，如果相同的模式出现在程序后半部分，模型也应该能够泛化并正确分类。

表 4.1简化了此过滤器的效果，因为它只展示了单个函数的分隔情况。实际上，需要分隔两个函数。完整数据集中的样本按照 4.2 所示的顺序生成。它们也经过了划分，因此只有函数2和函数3的操作都位于前半部分的样本才能出现在训练集

（测试数据集则相应地采用后半部分）。这导致一部分样本被舍弃，如表 4.2 所示。测试集中舍弃这些样本的原因是，这样可以更清晰地了解模型在测试期间对位置方差的鲁棒性。

Dataset \ Filter	First half only in train, second half only in test (#2)
Full dataset	[w... | r...] [w... | .r..] [.w.. | r...] [.w.. | .r..] [w... | ..r.] [w... | ...r] [.w.. | ..r.] [.w.. | ...r] [..w. | r...] [..w. | .r..] [...w | r...] [...w | .r..] [..w. | ..r.] [..w. | ...r] [...w | ..r.] [...w | ...r]
Train set	[w... | r...] [w... | .r..] [.w.. | r...] [.w.. | .r..]
Test set	[..w. | ..r.] [..w. | ...r] [...w | ..r.] [...w | ...r]
Discarded samples	[w... | ..r.] [w... | ...r] [.w.. | ..r.] [.w.. | ...r] [..w. | r...] [..w. | .r..] [...w | r...] [...w | .r..]

表 4.2：考虑两个函数中操作位置的筛选器 2 分离 s

本质上，两个函数的分离方法是：将前四分之一的样本分配到训练集，将最后四分之一分配到测试集，并丢弃其余部分。

此筛选器也可以泛化为将前 $1/n$ 个有序样本划分到训练集中，并将最后的 $(n-1)/n$ 个区间划分到测试集中，不过需要格外小心，尤其是在将其与其他筛选器结合使用时。

4.3 筛选器 #3——每隔 n 个样本跳过一次

仅将每第 n 个有序样本纳入训练过程 = 每次先跳过 n-1 个样本，再将一个样本加入训练集

其目的与第一个筛选器类似，但这次采用确定性的分离方式，以反映训练集与测试集之间的偏差。根据 n 的实际取值，可能会观察到一种有趣的效果，如下所示。

清单 4.1：应用第三个筛选器后训练集中的前 10 个样本

```
过滤器 跳过 大小 = 5                  |   筛选器 跳过 大小 = 6
函数 2          函数 3                |   函数 2          函数 3
['w___.__.,,_'  'cr,.,_,__.']         |   ['w___.__.,,_'  'cr,.,_,__.']
['w,__.__,,.,'  '.,.__cr_,_']         |   ['w__.'''.,,'  '.._..._cr,_']
['_w..,,...._'  '.cr,.,__.,']         |   [',w_.__,,.__'  ',,.cr...__']
['.w_.,__,,,'  '_,.__.cr._']         |   ['_,w,.__,..,'  'cr,,.__,__.']
['..w,___.,_'  '.,cr_,__,.']         |   ['_.w__,___,'  '.,...__cr__']
['_,w_.,,.__,'  ',....,.cr_']         |   ['..._w,..,_.'  '..,cr..,._']
['..._w,..,_.'  '..,cr..,._']         |   ['.,__w_,,.,'  'cr_,,.__.__.']
[',.,w__..__,'  ',,,.__,_.cr']        |   [',..,w,.,.__'  ',,,.__cr._']
['___.w...__.'  '.__,cr,.,,']         |   ['__...w___,'  '...cr.,,._']
```

根据所使用的跳跃大小不同，'cr' 操作在 Function 3 代码中的可能位置也有所不同，这一点很有意思。在左侧示例中，'cr' 可以出现在程序中的任意位置。然而，在右侧示例中，'cr' 只能出现在某些位置，即 0、6 和 3。因此，将跳跃大小参数设为 6，意味着第 3 个过滤器也会测试代码对位置方差的鲁棒性（因为在训练期间，模型只能在 3 个位置看到 'cr'，而可能的位置还要多得多）。

4.4 筛选器 #4——变量操作间距

改变有意义的操作之间空操作的数量 = 根据有意义操作之间的距离，将训练集和测试集中的样本分开

例如，训练集可以包含操作间距为 1 的样本（如 'w.u'），而测试集可以包含操作间距为 2 或 0 的样本（分别如 'w..u' 和 'wu'）。添加此过滤器的原因是，我们预期卷积神经网络和长短期记忆网络会有很大差异，后者的表现会优于前者。我们在结果部分展示了这一预期确实得到了验证。

4.5 组合过滤器

我们不仅可以单独使用这 4 种筛选器，而且得益于 API 的灵活设计，只要遵守筛选器的应用顺序，就可以任意组合这 4 种筛选器。这很有用，因为我们可以通过组合不同的筛选器（例如，分别组合筛选器 2 和 4），来测试多种期望的属性（例如，对操作位置变化的鲁棒性、程序大小，以及两个有意义的操作之间的相对位置）。在给出的筛选器 2 和 4 组合示例中，这种组合在经验上也有意义，因为组合情况下模型的准确率是各个筛选器下准确率的平均值 6.2。

4.5.1 筛选器顺序

若要使用这 4 个筛选器的任意组合，必须遵循以下顺序：

$$Filter2 \rightarrow Filter3 \rightarrow Filter1 \iff Filter4$$

筛选器 2 必须首先应用，因为它高度依赖生成样本时的原始顺序（参见 4.2）。筛选器1和滤波器 4 都不受原始数据集顺序的影响，因此可以将其中任意一个放在最后应用。不过，筛选器3必须在筛选器1之前应用，因为筛选器1会进行随机打乱，从而破坏筛选器3的确定性。

4.6 验证集

我们将验证集视为测试集的一部分，因为我们使用验证准确率指标进行早停，以防止过拟合。因此，我们从测试集中随机划分出一个子集作为验证集（验证集与测试集的划分比例通常为 1:3）。

第 5 章

机器学习模型

由于这三种函数类型旨在对并行运行的并发代码进行建模，因此我们还需要为每个函数分别设置输入特征向量。如下方最基础架构的概览图所示 5.1，三个独立的输入分别连接到三组独立的隐藏层。为每个函数分别设置输入特征向量，有助于模型专注于学习各个函数的逻辑，而不受其他函数的影响。随后，三条数据流会在一个全连接层中合并，之后再连接一个或多个全连接层。这样，模型还可以学习函数之间的全局交互。

我们认识到，由于每个程序的并行代码段数量差异很大（而我们的并行函数数量固定为三个），这种方法可能不适用于现实程序。不过，对这类输入进行建模并非易事，超出了本项目的范围。未来工作可以考虑采用图神经网络来绕过这一问题，或使用填充或其他机制来规范化输入维度。

不同模型类型的架构非常相似，大多数差异来自特殊层（即卷积神经网络/长短期记忆网络/DeepSets），以及全连接层的数量或每个全连接层、特殊层中的神经元数量。我们提供了经过微调的模型架构示意图，这些架构用于取得我们的实验结果，图示既具说明性又较为全面。图中仅省略了对输入进行独热编码的过程。

5.1 前馈网络

这种模型架构仅由全连接层组成，旨在作为基线，供更复杂的模型在此基础上进一步提升。我们最初在数据集上进行的学习实验使用了这种模型类型；大约在同一时期，我们决定将输入特征向量拆分为三部分，因为这样可以提高准确率。

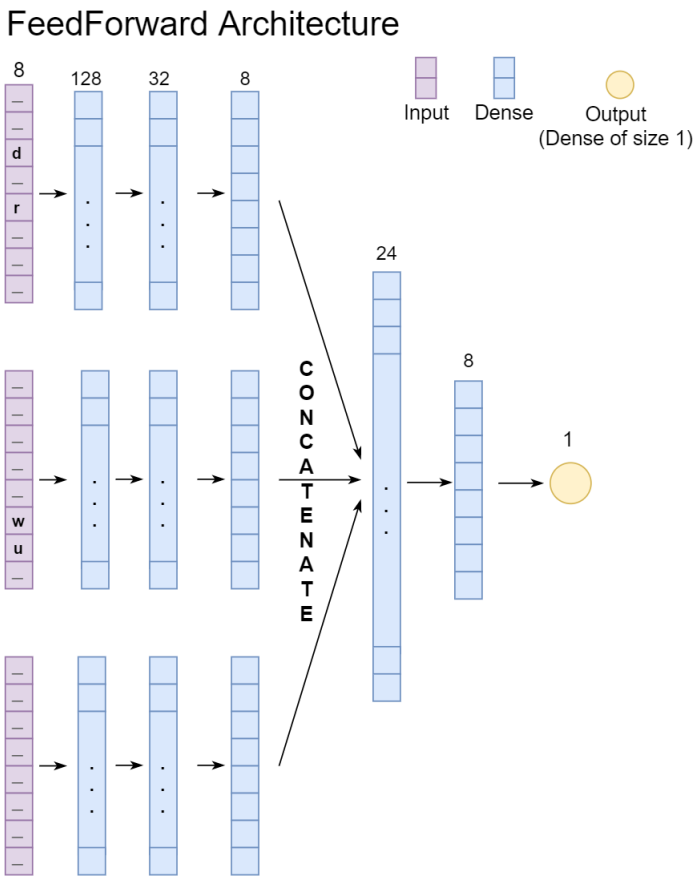


图 5.1：前馈网络类型的模型架构——仅使用全连接层

5.2 DeepSets

我们对 DeepSets 架构的两种变体进行了实验 [6]，，每种变体都旨在采用论文中的结论：只有求和函数才能真正实现置换不变。

5.2.1 DeepSets 变体 #1

这是对 DeepSets DigitSum 实验所用架构的近似改编 [43]。DigitSum 的任务是预测最多 10 个数字的总和，其中每个数字由一个 28×28 的像素值矩阵表示（与 MNIST 数据集类似 [58]）。输入按数字分别拆分，随后为每个数字输入矩阵连接 10 组全连接层，这与我们的架构非常相似。最后 10 个全连接层的大小均为 30，然后通过对这 10 个全连接层进行逐元素求和将它们合并，得到一个大小为 30 的向量。我们采用了非常相似的架构，如下所示5.2。

不过，一个区别是，我们还在与 DeepSets 层并行的位置添加了另一个全连接层。在早期实验中，我们发现，额外的全连接层有助于增强后的模型至少达到未增强模型的表现。为求全面，未来工作还将包括获取未增强模型的实验结果。

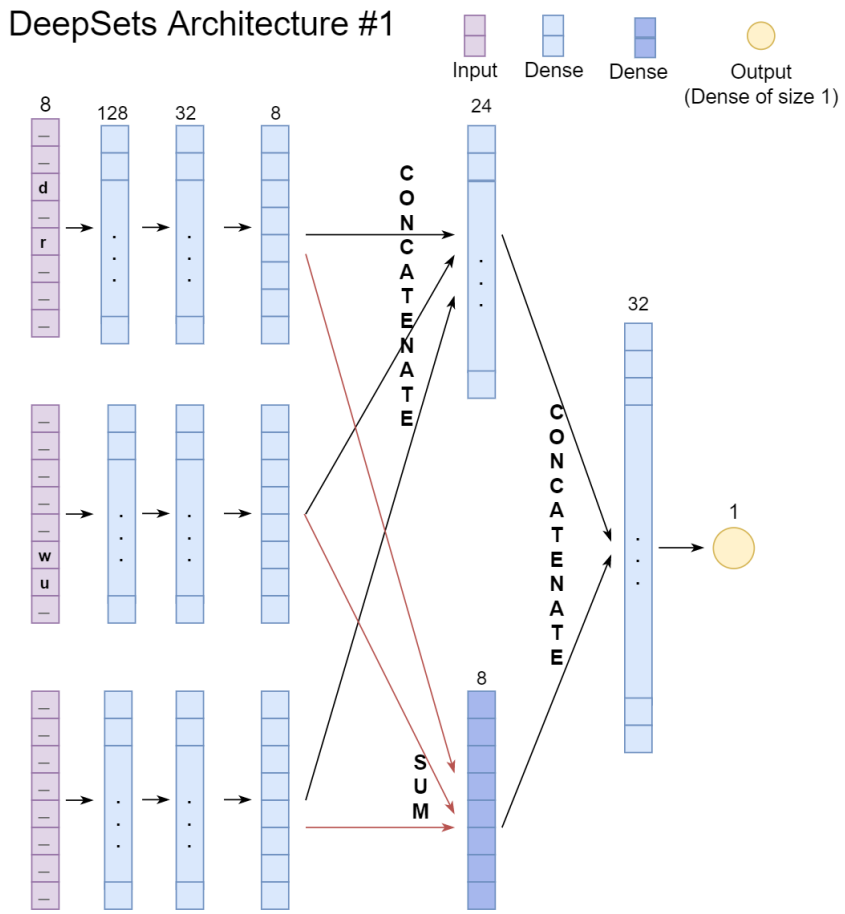


图 5.2：DeepSets 变体 #1 架构，与前馈网络5.1非常相似。单个深蓝色层表示对前面 3 个浅蓝色层进行逐元素求和

5.2.2 DeepSets 变体#2

DeepSets 的第二种变体旨在一开始就使用置换不变求和层。与简单的线性回归模型类似，它会对输入进行加权求和。这种变体最初旨在展示模型对代码位置方差的鲁棒性，同时我们预期操作的相对顺序会带来问题。然而，在运行实验时，我们发现这种 DeepSets 变体无法学到任何东西，即使是在训练集上也是如此。

未来工作包括以不同方式呈现输入（例如词袋），从而使第一层的权重本身不依赖于操作的绝对位置。虽然我们在讨论结果时排除了这种特定的模型类型，但其原始性能仍可在附录中查阅。

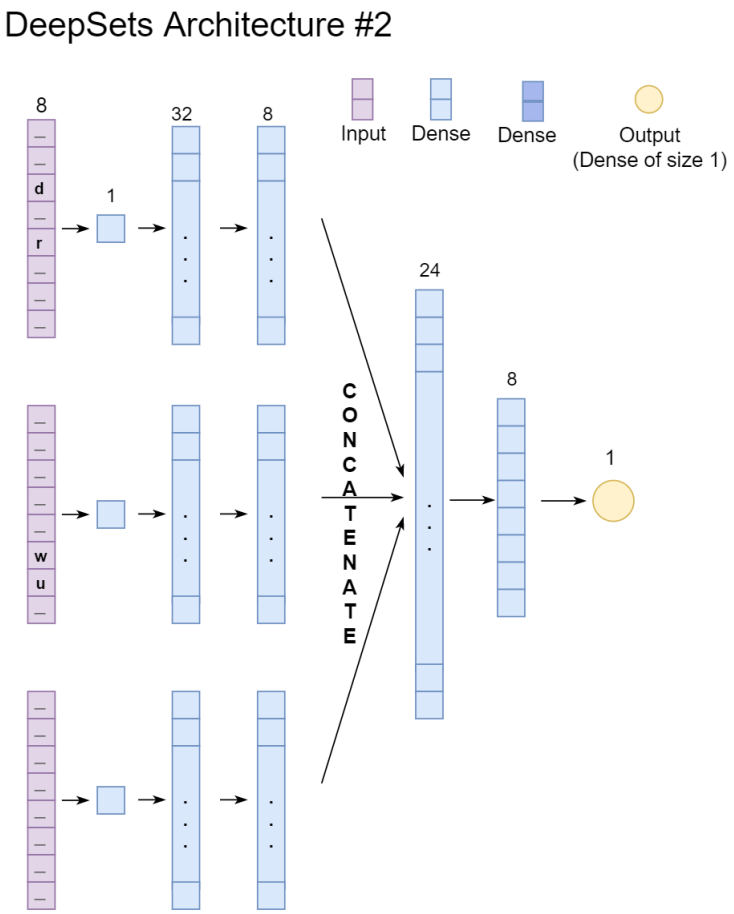


图 5.3：DeepSets 变体 #2 架构，与前馈网络相似 5.1

5.3 卷积神经网络

按时间顺序，卷积神经网络是我们实现的第三种模型类型，排在前馈网络和 DeepSets 之后。它也是第一种仅使用完整数据集的一个小子集（10%）进行训练，就能达到近乎完美准确率的模型类型。下面我们将展示卷积神经网络的架构图，并在接下来的几页中深入探讨卷积滤波器的能力与局限，以及期望卷积神经网络正确分类各种模式是否合理。

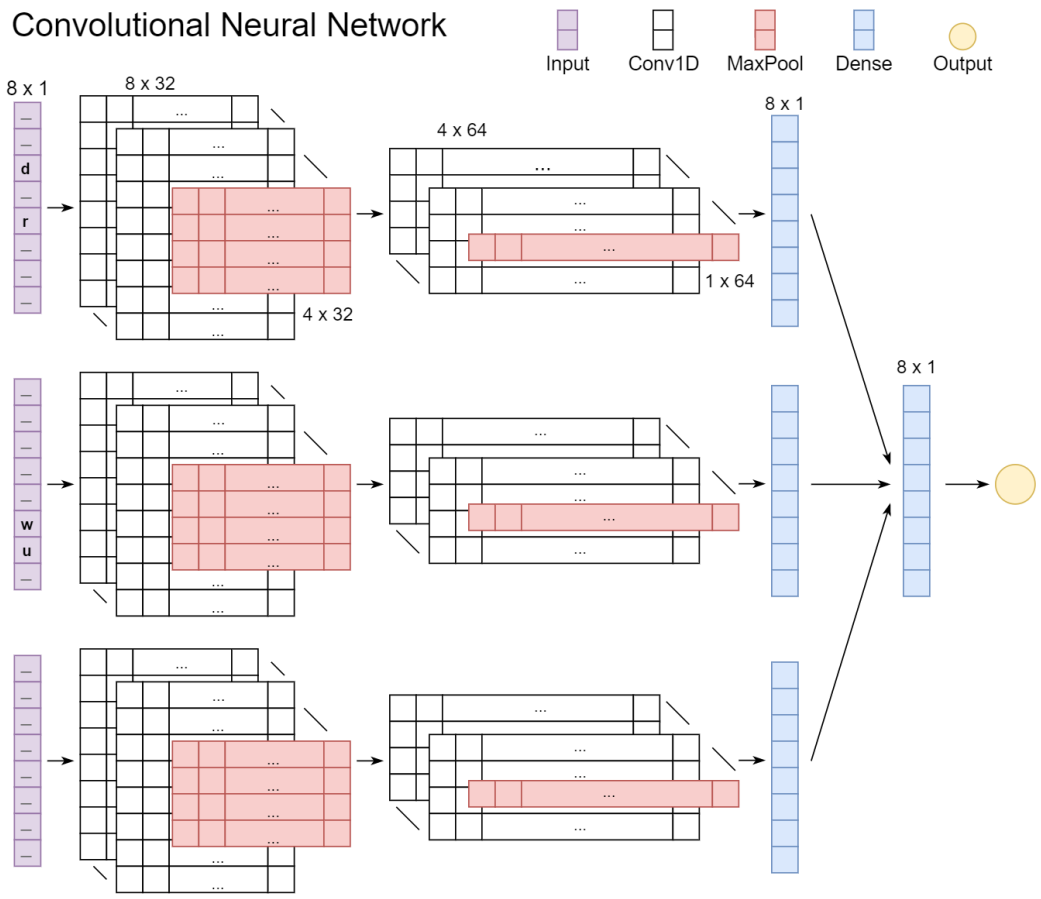


图 5.4：我们微调后的卷积神经网络架构概览，与 VGGNet 类似 [8]，其特点是逐步缩小输入维度，同时增大卷积滤波器维度

由于我们认为，要推理卷积神经网络在我们数据集上的表现尤其困难，因此在接下来的四个小节中，我们将探讨卷积神经网络在幕后可能如何运作，从而成功分类模式。我们举例说明卷积神经网络可能如何训练其权重，以正确分类样本。我们并不声称卷积神经网络在实际中会像示例所示那样训练权重。不过，我们将展示这种做法完全在其能力范围之内；而且由于示例很简单，我们认为实现这一点也是合理的。未来工作包括更详细地考察实际的过滤器权重，以验证我们的示例，或为卷积神经网络的过滤器权重如何影响模式分类提供新的理论见解。

还值得一提的是，理论上，卷积神经网络能够用少量筛选器和层成功学习所有模式。不过在实践中，更多的筛选器和层可以为模型提供更大的潜在有用权重池，从而帮助其收敛到这一状态。

5.3.1 卷积滤波器

在本小节中，我们定义用于说明卷积神经网络能力的符号。首先，我们给出一个示例，以提醒读者独热编码输入的格式。

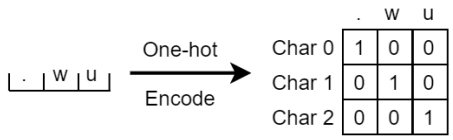


图 5.5：对长度为 3 的“.wu”字符串进行独热编码，会得到一个维度为 $3x3$ 的二进制数组（也可以展平为单个维度 9）

在左侧的数组中，只要第 j 列对应的字符出现在原始输入的第 i 个位置，我们就在位置 i, j 处填入 1。否则，其余所有值都设为 0。独热编码数组也可以展平，并且可以在我们的代码中展平（在此示例中，它会展平成向量 $[100010001]$）

下面展示如何将卷积滤波器应用于输入，以生成指示是否包含特定操作的二进制输出。每当卷积滤波器与输入“匹配”时，输出为 1，否则输出为 0。由于独热编码输入可能难以理解且不便于可视化，因此我们将其抽象处理。

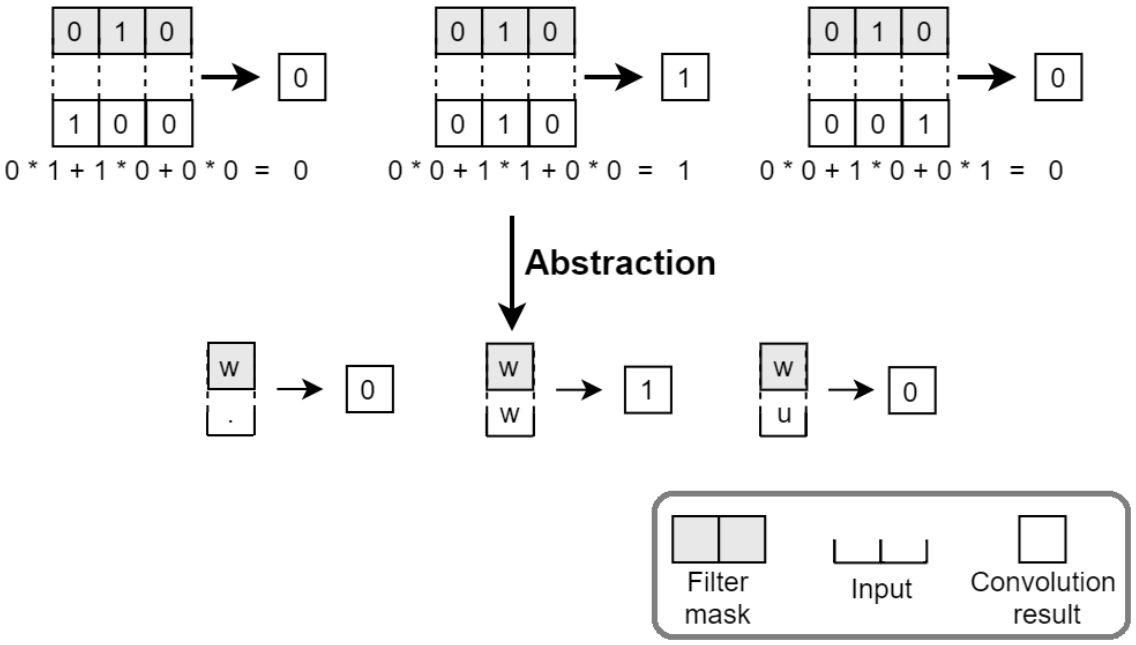


图 5.6：将大小为 1 的单个卷积神经网络过滤器应用于输入

在上面的示例中，卷积滤波器学会了检测字符 ’w’：当它位于该字符上方时，输出为 1；当它位于其他任何字符上方时，输出为 0。虽然这个示例使用的是大小为 1 的卷积滤波器，但我们可以轻松地将这种表示法扩展到任意滤波器大小，具体如下文各小节所示。

5.3.2 卷积神经网络——“wu”与“uw”

问题：我们的卷积神经网络架构，或者说一般的 *CNN* 模型类型，是否应该能够区分 *'wu'* 模式和 *'uw'* 模式？

下面通过一个简单示例说明，卷积神经网络模型如何训练出特定的掩码过滤器，从而成功区分这两种模式。我们还指出，使用最大池化层有助于成功区分这两种模式，甚至是必不可少的。

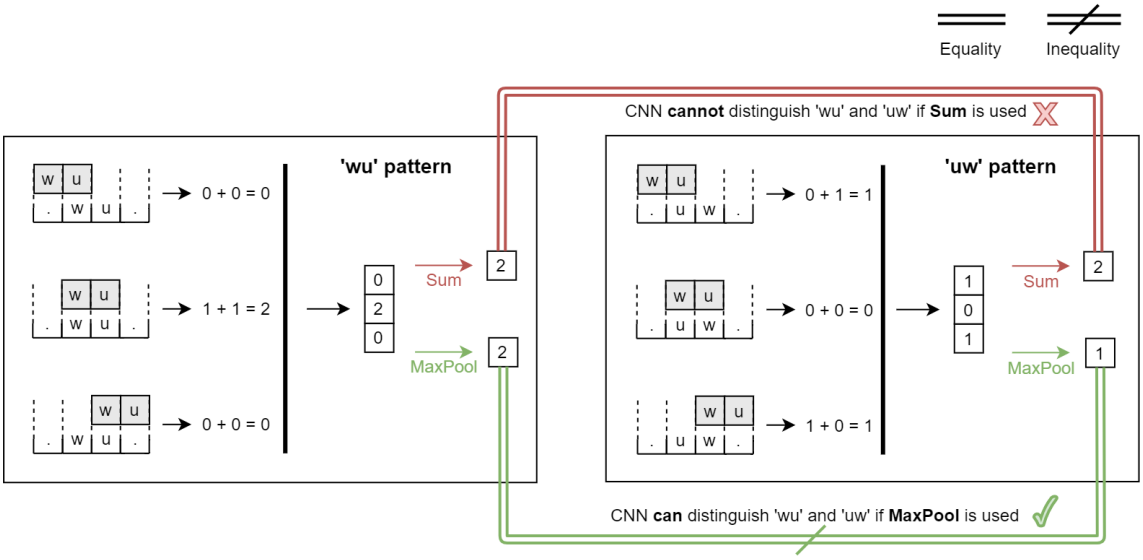


图 5.7：卷积神经网络需要最大池化才能区分“wu”和“uw”

为简化说明并便于观察，我们大幅缩小了输入维度。我们还省略了填充操作，以便输入与第一个卷积层的维度保持一致。

这个示意图可能会引出一个问题：是否有必要对卷积结果求和（图中红色部分），以及为什么不能直接将结果连接到后续的全连接层。事实上，在这个最小示例中，模型无需最大池化层也能成功学习这种模式。然而，对于较大的输入，我们必须确保模型对于操作位置方差具有鲁棒性。这可以通过对卷积结果求和来实现（求和是一个置换不变函数，参见 DeepSets [6]），也可以通过最大池化层来实现。

事实上，最大池化层历来用于提取最显著的特征[45]，，这也正是我们的卷积神经网络能够成功区分这两种模式的原因。尽管本报告中具体讨论的所有结果都只涉及包含多个最大池化层的卷积神经网络，但根据我们的经验，添加最大池化层确实提高了卷积神经网络的整体准确率。

5.3.3 卷积神经网络——位置方差

Question：我们的卷积神经网络架构，或者一般的 *CNN*模型类型，是否应该在第 *2* 个过滤器上表现良好 r?

不难看出，卷积滤波器固有的局部性赋予了卷积神经网络对代码位置方差的鲁棒性。事实上，由于卷积滤波器只考虑输入中的局部邻域，因此操作的绝对位置并没有那么重要。然而，在极端情况下会怎样？例如筛选器 2 中，所有有意义的操作在训练集的程序中只能出现在前半部分，而在测试集的程序中只能出现在后半部分。事实上，我们表明，简单的卷积层可能还不够，因此仍然需要最大池化层。

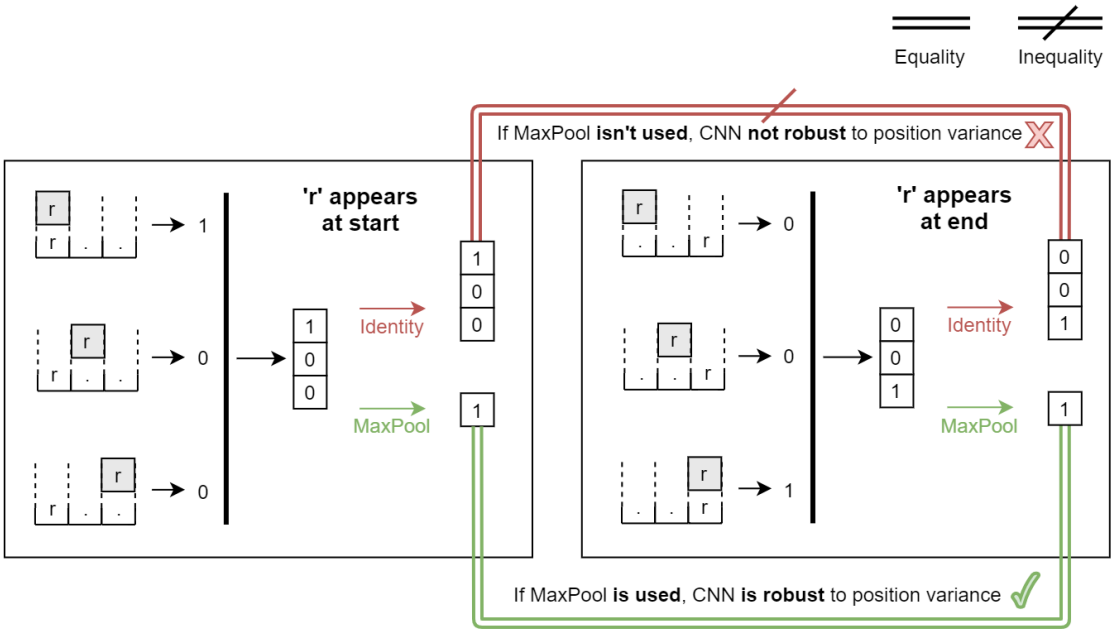


图 5.8：筛选器 2 要取得良好效果，需要使用最大池化

如上所示，最大池化层有助于应对位置方差，这一点并不令人意外。值得注意的是，必须沿第一维的整个范围执行最大池化，才能完全实现位置不变性。在我们的架构中，这一功能的实现方式与 VGGNet [8], 非常相似，即连续串联多个卷积层和最大池化层。

尽管我们没有使用本报告中的结果对此进行验证，但我们的经验还表明，在整个第一维度上执行最大池化，能大幅提升卷积神经网络在筛选器 2 上的性能。

5.3.4 卷积神经网络 - 操作间距方差

Ques：我们的卷积神经网络架构，或一般的 *CNN* 模型类型，是否应该在第 *4* 个过滤器上表现良好 *r?*

在前两个小节中，我们给出了一些合理的示例，说明卷积神经网络能够学习哪些有用的权重。不过，我们预计，滤波器 4 中的模式超出了卷积神经网络的学习能力。

假设卷积神经网络学会了第一个示例 5.3.2中所示的“wu”滤波器掩码，那么当同一滤波器掩码应用于它从未见过的“w..u”时，会发生什么?

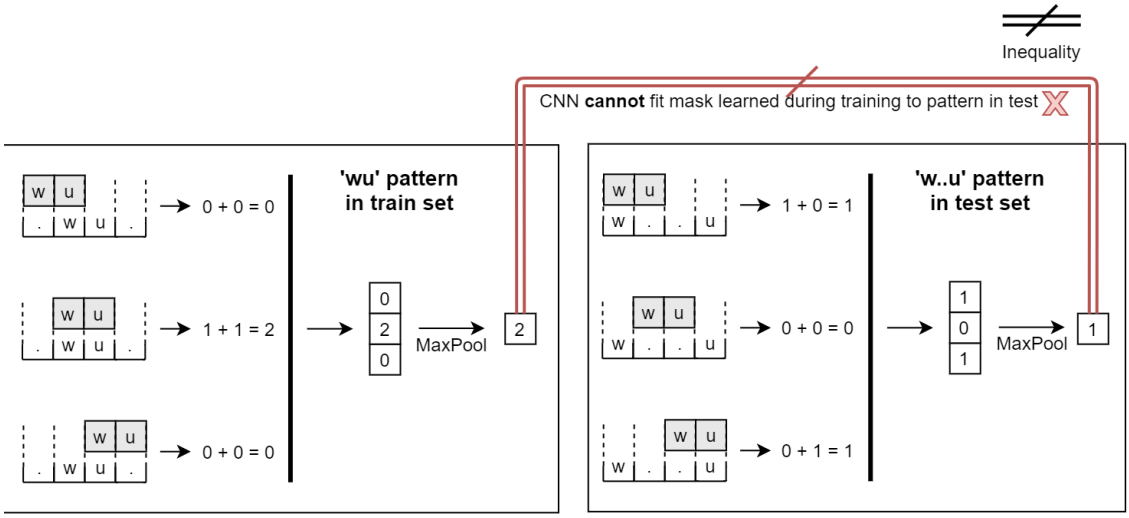


图 5.9：由于掩码维度固定，卷积神经网络无法从“wu”泛化到“w..u” s

如上所示，将“wu”滤波器掩码应用于“wu”输入，与将其应用于“w..u”输入，最终结果有所不同。如果我们将所有包含“wu”的样本划分到训练集，将所有包含“w..u”的样本划分到测试集，那么我们不认为卷积神经网络也会学会“w..u”滤波器掩码。但如果移除最大池化层，只对卷积结果求和呢？那么模型将难以区分“w..u”和“u..w” 5.3.2 ，因为最大池化在这种情况下至关重要。

权重正则化似乎在滤波器 4 的情况下也发挥着重要作用。根据实验结果，在第4个过滤器实验中加入 0.01 L2 正则化，有助于卷积神经网络实现近乎完美的准确率，唯独 ’wu ｜ dr’ 这种情况例外。对于函数3中除 ’dr’ 以外的任何模式，只需观察单个操作是否出现，就能学会。因此，无论操作间距的方差如何，卷积神经网络仍能达到相对较高的准确率。

5.4 长短期记忆网络

长短期记忆网络模型类型是最后加入我们阵容的。我们推测它会表现最佳，在第4个过滤器的情况下取得比卷积神经网络更高的准确率。结果表明确实如此；不过，卷积神经网络在第2个滤波器上的表现或许出人意料地更好。

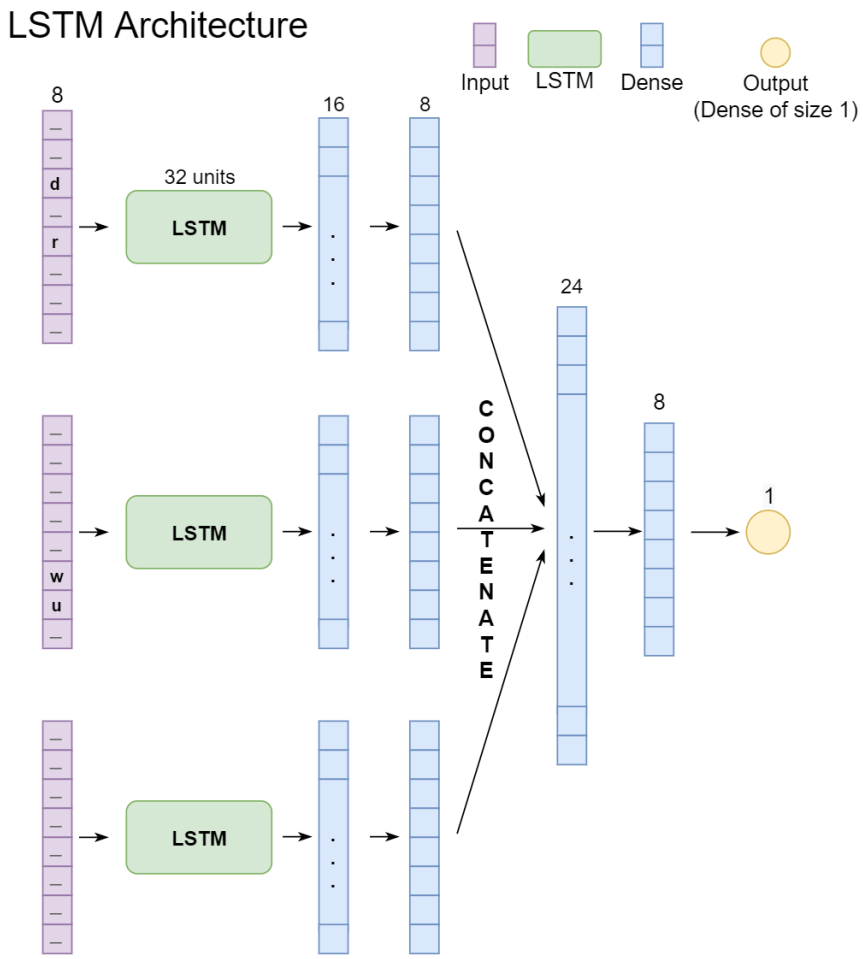


图 5.10：长短期记忆网络架构

5.5 Keras 与 PyTorch

机器学习模型最初使用 Keras 库开发， [54], 因为它提供了一个非常易用的 API。不过，随着模型的架构变得越来越复杂（例如 DeepSets 层），我们认为最好将所有模型永久迁移到 PyTorch 库 [55] ，因为它能更精细地控制模型架构。为确保从 Keras 迁移到 PyTorch 不会影响模型的准确率，我们在相同的数据集上并排比较了模型在各训练轮次中的准确率变化。我们发现，Keras 默认的权重初始化方法（“glorot uniform”）比 PyTorch 默认的权重初始化方法收敛更快、效果更好。幸运的是，我们能够调整 PyTorch 的权重初始化方法，使其与 Keras 的一致，从而让模型的性能几乎完全相同（权重初始化中的随机性除外）。

5.6 训练

我们生成的字符数组在输入模型之前会进行独热编码。目前，我们只尝试过让所有字符串使用相同的固定长度。不过，我们也为数据集提供了填充功能，以处理大小可变的输入。

我们将模型训练大量训练轮数（通常为 100-200 轮），并在每个训练轮次记录训练集准确率和损失，以及验证准确率和损失。我们使用两个检查点：第一个检查点保存以验证准确率衡量的最佳训练轮次表现所对应的模型权重。第二个检查点会在验证准确率经过大量训练轮数后仍未提升时执行早停。我们选择 Adam [59], 作为优化器，因为根据经验，它提供了最佳性能。我们还使用类别权重来平衡正类与负类的比例（因为有 7 个存在错误的案例和 9 个不存在错误的案例）。

第6章

评估

评估的核心体现在我们如何生成数据，以及如何将某些模式从训练集和测试集中分离出来。这使我们能够测试关于每种模型类型能力和基本局限性的假设。

6.1 模型属性的形式化

为了全面测试模型的学习能力，我们需要正式定义要检查哪些属性，以及需要开展哪些相关实验。这些属性的灵感来自对现实程序建模时可能遇到的模式。我们会说明为何预期某些模型的表现优于其他模型。我们已在专门的章节 5.3.1 中讨论了 CNN 模型类型的理论能力。下面我们简要推理其他 3 种模型类型的表现，因为无需深入分析便可论证它们的行为。

6.1.1 对代码位置变化的鲁棒性

如果模型在训练阶段学会将样本 [r...], [.r..] 分类为正类，那么它能否泛化，从而在测试阶段也将 [..r.], [...r] 分类为正类?

- 在实际程序中，一条指令可能出现在程序中的任意位置，但它出现的绝对位置可能并不重要。此外，我们在训练期间可能只会看到指令出现在有限的位置集合中，但仍希望在测试时能够泛化到其他位置。

- 我们通过统计第2个过滤器实验中的整体准确率来衡量模型对位置方差的鲁棒性，因为该过滤器会根据操作位置区分训练集和测试集。

- 由于它们天生具有局部性，我们预计只有长短期记忆网络和卷积神经网络能够对此保持鲁棒。我们此前已经论证过卷积神经网络的鲁棒性 5.3.3。我们认为，长短期记忆网络也应具有鲁棒性，因为学习特定的隐藏状态权重是合理的，且在其层的能力范围内：这些权重可确保噪声操作（',. '）不会影响当前隐藏状态，从而只有有意义的操作才能改变隐藏状态。输入端连接了全连接层的模型对此不应具有鲁棒性。这是因为，在训练集期间，没有被有意义的操作占据的输入位置所对应的权重不会更新（例如，在我们的例子中，这对应于 [..r.] 中'r'位于位置2, [...r] 中'r'位于位置 3）

6.1.2 对操作间距方差的鲁棒性

如果模型学会将样本 [wu], [w..u] 分类为负类，它能否实现泛化，从而也将 [w.u]分类为负类?

- 我们通过考察第4个过滤器实验中“wu-dr”案例的准确率，来衡量模型对位置方差的鲁棒性（除“wu-dr”之外的所有案例，即使没有这种鲁棒性也能学会，因为函数3中只需一个操作即可做出决策）

- 就像现实生活中一样，两个共同影响程序共享状态的操作之间的距离可能会发生变化，因此训练期间可能只出现某些距离，但我们仍希望模型能够泛化到任何合理的距离

- 我们不认为前馈网络和DeepSets对此具有鲁棒性，因为训练阶段学到的权重非常具体。因此，与联合操作相关的权重由联合操作在程序中的位置以及两个操作之间的距离共同唯一确定。因此，测试阶段具有不同间距的操作将无法匹配训练期间学到的特定权重。另一方面，基于与前一项属性相同的原因，我们预计长短期记忆网络能够对此保持鲁棒性。然而，正如之前所述，我们预计卷积神经网络会失效5.3.4

6.1.3 区分操作相对顺序的能力

模型能否区分将 [w.u] 判定为负类、但将 [u.w] 判定为正类?

- 我们通过以下公式衡量模型区分相对顺序的能力：$score = min(1 - r,\ r)/0.5$其中 $r = dr\ wu/(dr\ wu + dr\ uw)$，$dr\ wu$ 表示过滤器1和3中“dr-wu”情形的模型准确率，$dr\ uw$ 表示过滤器1和3中“dr-uw”情形的模型准确率。注意，r 和 $score$ 均处于 $[0, 1]$ 范围内

- 如果模型正确地将“dr-uw”分类为有缺陷，但错误地将“dr-wu”也分类为有缺陷，这意味着它无法区分 [u.w] 和 [w.u]，因此 r 会最小化（请记住，“dr-uw”和“dr-wu”的真实标签不同）。反之亦然，类似的推理也适用，此时 $1 - r$ 会最小化

- 另一方面，当 $dr\ wu$ 和 $dr\ uw$ 的准确率相同时，$min(1 - r, r)$ 达到最大值，因此 $1 - r = r = 0.5$（所以通过除以 0.5 进行归一化，以使分数保持在 $[0, 1]$ 范围内）。因此，该模型能够正确区分：’dr-uw’ 存在错误，而 ’dr-wu’ 不存在错误。

- 过滤器1和3用于检查此属性，因为它们不测试另外两个属性，因此可以通过筛选器清晰地区分属性检查

- 另一个体现这类属性的简单现实示例，是对变量进行赋值的时机：在初始化**之后**赋值（有效 ✓），而不是在**之前**赋值（错误 ✗）。

- 理论上，除了 DeepSets 变体 #2 之外，所有模型都应该能够学会这一点，因为操作在输入向量中的位置唯一决定了与之关联的权重（因此能够区分两种不同的操作顺序）。然而在实践中，我们认为前馈网络和 DeepSets 变体 #1 无法切实学会这一点，因为可能的组合数量太多（当 [wu] 和 [uw] 出现在程序中的位置 0、位置 1 等位置时，都必须对它们加以区分）。

然而，由于卷积神经网络和长短期记忆网络固有的局部性，组合数量要少得多，不会造成问题，因此我们预期它们能够区分这两种情况。我们已经展示了卷积神经网络如何学会区分相对顺序5.3.2。不难看出长短期记忆网络为何也能学会区分这一点。例如，长短期记忆网络可以通过在遇到第一个 'u' 之前始终保持隐藏状态为零，来学会这种模式。遇到 'w' 时，则将隐藏状态取反。遇到噪声操作时，隐藏状态保持不变。由于将零状态取反后仍然是零状态，'wu' 会产生非零输出状态，而 'uw' 会产生取反后的非零输出状态，因此这两种情况有所不同

6.2 测试流程

为了确保在测试模型能力时不受随机性的影响，我们采取了几个步骤。我们对每个模型进行大量训练轮次（100–200轮），因为有些模型可能会遇到不理想的权重初始化。对于每种模型类型，我们还训练了该类型的多个实例（每种类型20个模型），以确保结果在统计上有意义（这些模型实例之间唯一的区别是初始权重）。最后，我们既考察所有模型的汇总结果，也考察表现最佳的前50%模型的汇总结果（两种情况下的汇总结果均按模型类型分组）。

6.3 迭代实验

如前所述，我们通过在之前的实验基础上不断迭代来加深对问题的理解，同时微调模型，并增加或降低合成数据集及其过滤器的复杂度。在第一系列实验中，我们只测试了前馈网络和DeepSets模型类型，整体流程的计划性也较低。随着我们逐渐加深对问题的理解，我们会加入其他模型类型，并以更有条理的方式开展实验。我们也能够开始预判结果，例如卷积神经网络和长短期记忆网络在筛选器1上的表现会优于各自的对应模型。

在项目后期，我们开始正式定义想要检查的属性以及评估它们的方法。在第一轮正式实验中，除长短期记忆网络在筛选器 2 上的表现以及卷积神经网络在滤波器 4 上的表现外，我们大多得到了预期结果。为改善长短期记忆网络在筛选器 2 上的表现，我们在训练阶段将噪声填充添加到输入左侧，并在测试时添加到右侧。这确保了无论在训练集还是测试集，有意义的操作都位于程序中间。另一方面，我们提高了卷积神经网络在滤波器 4 中的权重正则化，并减少了每个卷积层的筛选器数量，使其表现远胜于前馈网络和 DeepSets。

提高长短期记忆网络和卷积神经网络的性能，促使我们开展了最后一系列实验。对于每个筛选器，我们都分别进行了难度不同的两个版本的实验。由于每个筛选器都有各自的参数，用于控制训练集和测试集中出现的样本数量（例如，筛选器1中的训练测试比、筛选器3中的跳跃大小），因此我们可以调整实验的难度。事实上，尽管简单版本和困难版本的结果彼此一致，但在简单版本的实验中，不同模型类型之间的差异更加明显。这是因为简单版本更侧重于测试模型能力的上限，而非模型在训练过程中达到这些上限的可能性。因此，以下结果基于较简单的筛选器版本。不过，所有结果均列于附录中。

6.4 结果

6.4.1 检查模型性质

如上所述，我们分别计算了每种模型类型在三个属性上的得分。下面的表格展示了这些结果，并证实了我们最初的假设。

Robust to / Model	Operation position variance	Inter-operational distance variance	Relative ordering
CNN	**99.1%**	15.2%	**99.8%**
LSTM	95.0%	**100%**	**99.9%**
FeedForward	56.9%	7.9%	15.2%
DeepSets	56.7%	5.1%	16.2%
Relevant filters/cases	Filter 2 - overall accuracy	Filter 4 - accuracy for 'wu-dr' case	Filters 1 and 3 - ratio between 'wu' and 'uw' cases

* only top 50% best performing models are considered

表 6.1：按模型类型和属性细分的得分——得分越高，模型性能越好

正如我们所见，事实确实如此：卷积神经网络和长短期记忆网络凭借其固有的局部性，在所有这些属性上的表现都优于前馈网络和 DeepSets，唯一的例外是卷积神经网络在第4个滤波器上的表现不佳。不过，在进行相应实验之前，我们就已经推断出会是这种情况5.9。

不过，我们未曾预料到的一个结果是，卷积神经网络和长短期记忆网络在筛选器 2 上的得分存在显著差异。我们认为，长短期记忆网络表现较差，是因为它存在所有循环神经网络共有的问题，即处理较长输入时会出现梯度消失问题 [38]。在筛选器 2 这个更极端的情形下，循环神经网络的这一缺陷尤为突出，因为训练集和测试集数据的分布差异极大。

值得注意的是，前馈网络和 DeepSets 在第一个属性上的得分为 56%，这是因为该得分是根据筛选器 2 上的整体模型准确率衡量的。因此，在这种情况下，56% 的得分并不比随机预测对应的 50% 好多少。

请注意，这些结果来自实验中简单变体里表现最佳的前 50% 模型。完整结果请参见附录。

6.4.2 按筛选器和模式划分的结果

为了支持我们先前的一些论断或提供更多论据，我们还按筛选器或模式细分了模型准确率。

Model \ Filter	Filter 1	Filter 2	Filter 3	Filter 4	Filter 2 + Filter 4
CNN	**99.9%**	**99.1%**	**99.5%**	96.7%	96.8%
LSTM	**100%**	95.0%	**100%**	**100%**	**98.2%**
FeedForward	92.1%	56.9%	84.1%	96.1%	61.8%
DeepSets	92.2%	56.7%	84.2%	96.0%	61.8%
Tests robustness to:	Relative ordering	Position variance	Relative ordering	Inter-operational distance variance	Position and inter-op distance variance

* only top 50% best performing models are considered

表 6.2：按模型类型和筛选器细分的得分

我们此前的一个论断是，从实证角度看，组合筛选器 2 和 4 是合理的 4.5。事实上，正如我们所见，对于所有模型，在过滤器 $2+4$ 的实验中，其准确率始终是筛选器 2 的准确率与滤波器 4 的准确率的加权平均值。这验证了这样一种观点：模型在组合筛选器上的表现不会比最难的筛选器更差，也不会比最简单的筛选器更好。

Model \ Case	F3 'no-op'	F3 'r'	F3 'cr'	F3 'dr' F2 'wu'	F3 'dr' F2 'uw'	F3 'dr' F2 'w'	F3 'dr' F2 'noop'
CNN	**100%**	**98.8%**	**100%**	77.6%	**99.4%**	**99.9%**	**99.9%**
LSTM	**100%**	**98.1%**	**100%**	**89.7%**	98.1%	**99.9%**	**100%**
FeedForward	98.7%	65.6%	98.4%	29.6%	72.7%	74.4%	73.4%
DeepSets	99%	65.8%	98.2%	29.1%	72%	73.3%	72.8%
True label is:	✓	✗	✓	✓	✗	✗	✗

* calculated by average case accuracy for filters 1-4
** only top 50% best performing models are considered

表 6.3：按模型类型和模式细分的得分

此外，我们提到，“wu | dr”模式对于模型而言最难成功分类 3.6。事实上，如上所示，情况似乎确实如此：对于任何模型类型，“wu | dr”模式的平均准确率都低于其他任何模式。

6.4.3 补充说明

在筛选器一节中，我们讨论了某些参数如何使筛选器3也能测试对位置方差的鲁棒性 4.3。事实上，对于对位置方差不具鲁棒性的模型（即前馈网络和 DeepSets），性能存在差异。当训练集和测试分布之间不存在位置方差时，它们的表现更好 9.1.3；而存在一定方差时，表现则更差9.1.4。

此外，我们还监测训练每种模型类型所需的时间，以及每个训练轮次的性能。不同模型类型的训练时间存在差异（例如，在图形处理器上进行第2个滤波器实验时，卷积神经网络和长短期记忆网络分别需要1小时49分钟和1小时42分钟，而前馈网络和DeepSets则分别需要27分钟和26分钟），达到收敛所需的训练轮数也有所不同（例如，在筛选器1下，卷积神经网络略胜于长短期记忆网络6.1 6.2 ）。不过，根据我们的经验，优秀模型之间的差异并不足以证明选用某一种模型而非其他模型是合理的。

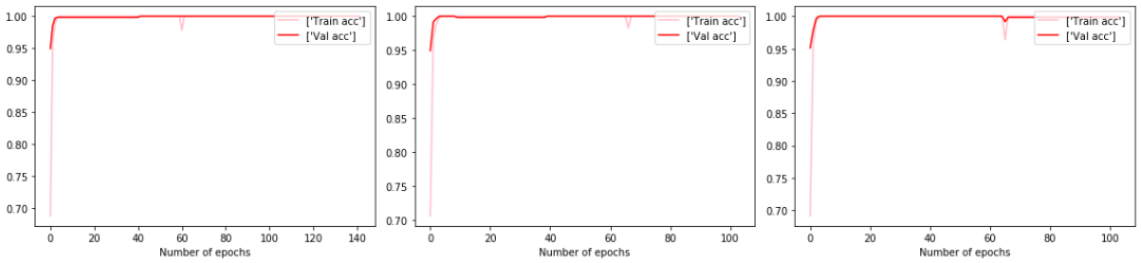


图 6.1：筛选器1下卷积神经网络的收敛情况

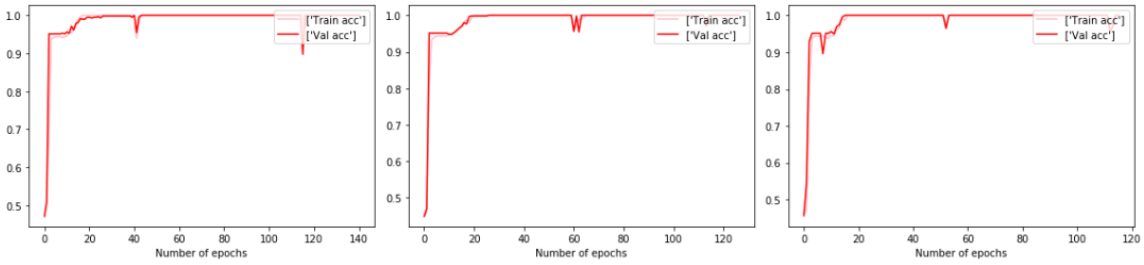


图 6.2：筛选器1下长短期记忆网络的收敛情况

第 7 章

结论

7.1 总结

总而言之，我们提供了一个合成数据集，用于模拟实际生活中的（并发）程序所呈现的模式。我们还为该数据集提供了筛选器，旨在通过特定输入探测模型，从而测试某些属性。我们探讨了每种模型类型的基本局限性和能力，并特别花时间研究卷积神经网络的机制。我们将一些期望具备的属性形式化，以期神经网络应用于实际程序时能够满足这些属性。随后，我们通过实验证明，关于模型学习局限性或能力的初步预期得到了验证，并展示了各种模型类型在这一问题上的优缺点。我们表明，对输入具有局部视野的模型（卷积神经网络、长短期记忆网络）优于不具备这种视野的模型（前馈网络、DeepSets）。我们还表明，传统上用于其他序列问题的模型（长短期记忆网络）也可能优于实力相近的模型（卷积神经网络）。我们希望这些发现能够进一步加深人们对神经网络在序列问题整体表现的认识。

7.2 未来工作

在比较各种模型类型并学习将代码作为输入特征进行建模这一具体背景下（即本 SuperUROP 项目的贡献），我们接下来想要解决几个亟待处理的问题。

- **探索图神经网络（GNN）**
  在对实际代码进行建模时，我们很可能会采用图表示作为输入，而不是在合成数据集中使用的向量方法。这是因为程序大小和并发代码段数量的差异很大，因此本报告所采用的输入建模方法在实际程序上可能并不适用。与循环神经网络类似，图神经网络在处理大小可变的输入时表现出色。此外，它们还可以利用循环神经网络模型类型特有的门控机制，而我们已证明这种机制在合成数据集上效果良好。
- **进一步解读模型**
  在本项目中，我们通过将某些样本集划分为训练集和测试集，来解读模型的学习局限和能力。未来，我们希望更深入地研究模型，特别是输入特征的重要性2.3 和层权重5.3。
- **增加数据集和筛选器的复杂性**
  为了弥合我们的合成问题与实际应用场景之间的差距，我们需要进一步提高模拟程序的复杂性。这可以通过增加语法的复杂性来实现，例如为我们的函数添加更多操作（或许在...

尤其是函数1）。此外，我们还可以添加一个额外的过滤器，根据每个函数中的操作数量进行区分。虽然对于大多数模型类型，我们必须对输入进行填充以规范输入长度，但长短期记忆网络不需要任何进一步的输入预处理，因此提供了另一个潜在优势。

在项目的总体目标（即利用神经网络开发并发工具）中，我们提到了需要实现的一些重要里程碑。

- **精准预测存在缺陷的操作**
  模型仅仅将程序分类为有缺陷或无缺陷是不够的。程序员还需要知道具体是哪些代码行导致了问题。这个问题可能并不简单，而且可能与上文提到的“进一步解释模型”这一要点有一定重叠。

- **确定中间表示**
  我们必须牢记项目的主要研究动机之一：通过使用所有源语言通用的中间表示来训练模型，从而支持较新的语言。需要关注的一个关键因素是将较新的源语言转换为中间表示，这可能需要投入额外的实现工作。

- **获取带标签的并发数据集**
  最具挑战性的里程碑，也是我们使用合成数据集的主要动机。手动为并发程序添加标签非常费力且容易出错，尤其是在考虑多种源语言时。我们可以使用现有工具为数据集添加标签：尽管模型永远无法超越原始工具的表现，但若能取得相当的结果，就意味着我们成功地对并发程序进行了建模。此外，我们甚至可能通过结合多个工具生成的、使用不同源语言的数据集，取得更好的表现。

第 8 章

参考文献

[1] K.-H. Yu、A. L. Beam 和 I. S. Kohane，《医疗保健中的人工智能》，《自然生物医学工程》，第 2 卷，第 10 期，第 719–731 页，2018年。[2]J. Heaton、N. G. Polson 和 J. H. Witte，《金融领域的深度学习》，*arXiv* 预印本 *arXiv:1602.06561*，2016年。[3] M. Bojarski、D. Del Testa、D. Dworakowski、B. Firner、B. Flepp、P. Goyal、L. D. Jackel、M. Monfort、U. Muller、J. Zhang、等，《端到端自动驾驶汽车学习》，*arXiv* 预印本 *arXiv:1604.07316*，2016年。[4] S. Savage、M. Burrows、G. Nelson、P. Sobalvarro 和 T. Anderson，《Eraser：面向多线程程序的动态数据竞争检测器》，《*ACM* 计算机系统汇刊（*TOCS*）》，第 15 卷，第 4 期，第 391–411 页，1997年。[5] A. T. Chamillard、L. A. Clarke 和 G. S. Avrunin，静态并发分析技术的实证比较。博士论文，马萨诸塞大学阿默斯特分校，1996年。[6] M. Zaheer、S. Kottur、S. Ravanbakhsh、B. Poczos、R. R. Salakhutdinov 和 A. J. Smola，《深度集合》，载于《神经信息处理系统进展》，第 3391–3401 页，2017年。[7] D. P. McBrien，并发控制课件，*CO572*。高级数据库课程讲座，伦敦帝国理工学院，2018年。[8] K. Simonyan 和 A. Zisserman，《用于大规模图像识别的超深卷积网络》，*arXiv* 预印本 *arXiv:1409.1556*，2014年。[9] L. Wang 和 S. D. Stoller，《具有非阻塞同步机制的程序的原子性静态分析》，载于第十届 *ACM SIGPLAN* 并行程序设计原理与实践研讨会论文集，第 61–71 页，2005年。[10] M. Vakilian、S. Negara、S. Tasharofi 和 R. E. Johnson，《Keshmesh：一种检测并修复 Java 并发错误模式的工具》，载于 *ACM* 面向对象编程系统、语言及应用国际会议伴随会议论文集，第 39–40 页，2011年。[11] C. Flanagan 和 S. N. Freund，《Atomizer：一种面向多线程程序的动态原子性检查器》，*ACM SIGPLAN Notices*，第 39 卷，第 1 期，第 256–267 页，2004年。[12] C. Flanagan 和 S. Qadeer，《用于原子性的类型与效果系统》，*ACM SIGPLAN Notices*，第 38 卷，第 5 期，第 338–349 页，2003年。[13] C. Boyapati 和 M. Rinard，《用于无数据竞争 Java 程序的参数化类型系统》，载于第 *16* 届 *ACM SIGPLAN* 面向对象编程、系统、语言与应用会议论文集，第 56–69 页，2001年。

[14] C. Boyapati、R. Lee 和 M. Rinard，“用于防止 Java 程序中数据竞争和死锁的类型系统”，[15] A. Tehrani、M. Khaleel、R. Akbari 和 A. Jannesari，“DeepRace：通过深度学习发现数据竞争错误”，arXiv 预印本 arXiv:1907.07110，2019年。[16] V. Raychev、M. Vechev 和 A. Krause，“从‘大规模代码’预测程序属性”，ACM SIGPLAN Notices，第 50 卷，第 1 期，第 111–124 页，2015年。[17] U. Alon、M. Zilberstein、O. Levy 和 E. Yahav，“一种用于预测程序属性的通用基于路径的表示”，ACM SIGPLAN Notices，第 53 卷，第 4 期，第 404–419 页，2018年。[18] M. Allamanis、E. T. Barr、C. Bird 和 C. Sutton，“学习自然的编码惯例”，载于第 22 届 ACM SIGSOFT软件工程基础国际研讨会论文集，第 281–293 页，2014年。[19] M. Allamanis、E. T. Barr、P. Devanbu 和 C. Sutton，“大规模代码与自然性的机器学习综述”，ACM Computing Surveys (CSUR)，第 51 卷，第 4 期，第 1–37 页，2018年。[20]维基百科贡献者，“Solidity——维基百科，自由的百科全书。” https://en. wikipedia.org/w/index.php?title=Solidity&oldid=955799465，2020年。[在线；访问日期：5月19日，2020]。[21] C. Lattner 和 V. Adve，“LLVM：用于程序的终身分析与转换的编译框架”，载于代码生成与优化国际研讨会，2004年。CGO 2004，第 75–86 页，IEEE，2004年。[22] J. Chung、C. Gulcehre、K. Cho 和 Y. Bengio，“门控循环神经网络在序列建模中的实证评估”，arXiv 预印本 arXiv:1412.3555，2014年。[23] J. Zhou、G. Cui、Z. Zhang、C. Yang、Z. Liu、L. Wang、C. Li 和 M. Sun，“图神经网络：方法与应用综述”，arXiv 预印本 arXiv:1812.08434，2018年。[24] 维基百科贡献者，“并发（计算机科学）——维基百科，自由的百科全书。” https://en.wikipedia.org/w/index.php?title=Concurrency_(computer_science) &oldid=954985273，2020年。 [在线；访问日期：5月19日，2020]。[25] 维基百科贡献者，“线程（计算）——维基百科，自由的百科全书。” https: // en.wikipedia.org/w/index.php?title=Thread_(computing)#Scheduling，2020年。 [在线；访问日期：5月19日，2020]。[26] 维基百科贡献者，“可复现性——维基百科，自由的百科全书。” https://en. wikipedia.org/w/index.php?title=Reproducibility&oldid=955741439，2020年。 [在线；访问日期：5月19日，2020]。[27]维基百科贡献者，“同步（计算机科学）——维基百科，自由的百科全书。” https://en.wikipedia.org/w/index.php?title=Synchronization_(computer_science)#Thread_or_process_synchronization，2020年。 [在线；访问日期：5月19日，2020]。[28]甲骨文，“Java 同步方法。” https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html，2020年。 [在线；访问日期：5月19日，2020]。[29] “C ++ 互斥锁。” http://www.cplusplus.com/reference/mutex/mutex/lock/，2020年。 [在线；访问日期：5月19日，2020]。[30]维基百科贡献者，“生产者—消费者问题——维基百科，自由的百科全书。” https://en.wikipedia.org/w/index.php?title=Producer%E2%80%93consumer_problem&oldid=956140578，2020年。 [在线；访问日期：5月19日，2020]。

[31]维基百科贡献者，“死锁——维基百科，自由的百科全书。”https://en.wikipedia.org/w/index.php?title=Deadlock&oldid=956745835，2020年。[在线；访问日期：5月19日，2020]。[32] F. Rosenblatt，“神经动力学原理：感知器与脑机制理论”，技术报告，康奈尔航空实验室公司，纽约州布法罗，1961年。[33]“机器学习时间线。”
h
t
t
p
s://
www.researchgate.net/figure/Timeline-of-the-development-of-deep-learning-and-commonly-used-machine-learnin g_
图1_323612444，2020年。[在线；访问日期：2020]年5月19日。[34]“神经网络资源。”
https://cs.stanford.edu/people/eroberts/courses/soco/
projects/neural-networks/Sources/index.html，2020年。[在线；访问日期：2020]年5月19日。[35] Y. C. 3Blue1Brown，“什么是神经网络？”。https://www.youtube.com/watch?v=
aircAruvnKk，2020年。[在线；访问日期：2020]年5月19日。[36] K. Fukushima，“新认知机：一种不受位置移动影响的模式识别机制的自组织神经网络模型”，《生物控制论》，第36卷第4期，第193–202页，1980年。[37] A. Krizhevsky、I. Sutskever 和 G. E. Hinton，“使用深度卷积神经网络进行 ImageNet 分类”，载于《神经信息处理系统进展》，第1097–1105页，2012年。[38] S. Hochreiter 和 J. Schmidhuber，“长短期记忆”，《神经计算》，第9卷第8期，第1735–1780页，1997年。[39] X. Tian、J. Zhang、Z. Ma、Y. He、J. Wei、P. Wu、W. Situ、S. Li 和 Y. Zhang，“用于大词汇量连续语音识别的深度长短期记忆网络”，arXiv 预印本 arXiv:1703.07090，2017年。[40]S. Kim 和 M. Kang，“使用注意力长短期记忆网络进行金融序列预测”，arXiv 预印本 arXiv:1902.10877，2019年。[41] T. Young、D. Hazarika、S. Poria 和 E. Cambria，“基于深度学习的自然语言处理最新趋势”，IEEE《计算智能杂志》，第13卷第3期，第55–75页，2018年。[42]维基百科贡献者，“梯度消失问题——维基百科，自由的百科全书。”
https://en.wikipedia.org/wiki/Vanishing_gradient_problem#Long_short-term_memory，2020年。[在线；访问日期：2020]年5月21日。[43]M. Zaheer，“DeepSets - DigitSum。”
https://github.com/manzilzaheer/DeepSets/tree/ master/DigitSum，2020年。[在线；访问日期：2020]年5月21日。[44] S. M. Lundberg 和 S.-I. Lee，“一种解释模型预测的统一方法”，载于《神经信息处理系统进展》，第4765–4774页，2017年。[45]R. Barzilay 和 T. Jaakkola，“MIT 6883课程——使用机器学习进行建模。”
https://www.eecs.mit.edu/academics-admissions/academic-information/
subject-updates-spring-2019/6883，2020年。[在线；访问日期：2020]年5月21日。[46] M. T. Ribeiro、S. Singh 和 C. Guestrin，“‘我为什么应该相信你？’解释任意分类器的预测”，载于《第22届 ACM SIGKDD 知识发现与数据挖掘国际会议论文集》，第1135–1144页，2016年。[47] B. Lakshminarayanan、A. Pritzel 和 C. Blundell，“使用深度集成进行简单且可扩展的预测不确定性估计”，载于《神经信息处理系统进展》，第6402–6413页，2017年。[48] I. J. Goodfellow、J. Shlens 和 C. Szegedy，“解释并利用对抗样本”，arXiv 预印本 arXiv:1412.6572，2014年。

[49] T. Niven 和 H.-Y. Kao，“探究神经网络对自然语言论证的理解”，arXiv 预印本 arXiv:1907.07355，2019年。[50] M. Wang 和 W. Deng，“深度视觉领域自适应：综述”，《Neurocomputing》，第312卷，第135–153页，2018年。[51] J. Wexler、M. Pushkarna、T. Bolukbasi、M. Wattenberg、F. Vi´egas 和 J. Wilson，“what-if 工具：对机器学习模型的交互式探究”，IEEE《可视化与计算机图形学汇刊》，第26卷第1期，第56–65页，2019年。[52] W. Merrill、G. Weiss、Y. Goldberg、R. Schwartz、N. A. Smith 和 E. Yahav，“RNN 架构的形式层级”，arXiv 预印本 arXiv:2004.08500，2020年。[53] G. Van Rossum 和 F. L. Drake，《Python 3 参考手册》。加利福尼亚州 Scotts Valley：CreateSpace，2009年。[54] F. Chollet 等，“Keras。”https://keras.io，2015年。[55] A. Paszke、S. Gross、S. Chintala、G. Chanan、E. Yang、Z. DeVito、Z. Lin、A. Desmaison、L. Antiga 和 A. Lerer，“PyTorch 中的自动微分”，2017年。[56]F. Pedregosa、G. Varoquaux、A. Gramfort、V. Michel、B. Thirion、O. Grisel、M. Blondel、P. Prettenhofer、R. Weiss、V. Dubourg、J. Vanderplas、A. Passos、D. Cournapeau、M. Brucher、M. Perrot 和 E. Duchesnay，“Scikit-learn：Python 中的机器学习”，《机器学习研究杂志》，第12卷，第2825–2830页，2011年。[57] F. P´erez 和 B. E. Granger，“IPython：一个用于交互式科学计算的系统”，《科学与工程计算》，第9卷，第21–29页，2007年5月。[58] Y. LeCun、C. Cortes 和 C. Burges，“MNIST 手写数字数据库”，ATT 实验室 *[在线]*。可访问：http://yann.lecun.com/exdb/mnist，第2卷，2010年。[59] I. Goodfellow、Y. Bengio 和 A. Courville，《深度学习》。麻省理工学院出版社，2016年。

第9章

附录

9.1 完整结果

这里列出了用于计算结果部分中所显示的汇总准确率和得分的结果。我们提供了所有实验的简单版本（左侧）和困难版本（右侧）。需要注意的是，评估章节中的汇总结果是使用实验的简单版本计算得出的。此外，我们仅记录表现最佳的前50%模型的结果。

9.1.1 筛选器 #1

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	99.9%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	99.7%
LSTM	100.0%	100.0%	100.0%	100.0%	99.9%	100.0%	100.0%	100.0%
DeepSet(like in paper)	92.2%	92.8%	93.5%	95.6%	4.7%	98.6%	99.5%	97.6%
DeepSet(sum at start)	42.7%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	92.1%	96.6%	95.3%	98.0%	5.6%	99.5%	99.4%	94.3%
Sample count	2382	160	133	138	140	577	648	586

表9.1：筛选器1简单版本——训练子集占完整数据集的25%

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	98.3%	100.0%	99.4%	98.4%	84.6%	100.0%	100.0%	97.4%
LSTM	95.3%	100.0%	93.3%	99.9%	37.8%	100.0%	100.0%	97.1%
DeepSet(like in paper)	60.4%	58.3%	57.6%	61.1%	48.9%	69.2%	69.1%	46.4%
DeepSet(sum at start)	43.3%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	60.1%	51.4%	50.9%	50.5%	56.1%	76.6%	72.9%	38.4%
Sample count	3017	190	170	184	176	714	821	762

表 9.2：过滤器1 困难版本——训练子集占完整数据集的 5%

9.1.2 筛选器 #2

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	99.1%	99.8%	99.4%	99.8%	95.5%	100.0%	100.0%	97.7%
LSTM	95.0%	100.0%	92.7%	99.6%	59.0%	100.0%	100.0%	92.4%
DeepSet(like in paper)	56.7%	5.2%	5.2%	5.2%	94.7%	94.7%	97.3%	3.5%
DeepSet(sum at start)	42.8%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	56.9%	2.1%	5.4%	6.7%	95.2%	96.4%	96.7%	3.8%
Sample count	872	58	48	54	58	219	222	213

表 9.3：筛选器 2 简单版本——训练集包含出现在程序前半部分的操作，测试集则包含出现在后半部分的操作。

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	99.0%	99.3%	95.6%	98.5%	97.7%	100.0%	100.0%	98.2%
LSTM	86.5%	91.8%	85.2%	90.8%	20.5%	99.8%	99.3%	72.0%
DeepSet(like in paper)	54.6%	5.4%	4.6%	4.9%	92.9%	93.8%	94.3%	6.1%
DeepSet(sum at start)	44.6%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	54.7%	3.8%	3.1%	2.6%	93.7%	95.3%	95.5%	4.9%
Sample count	1456	97	85	88	73	318	416	379

表 9.4：滤波器 2 困难版本——训练集包含程序前三分之一中出现的操作，测试集包含最后三分之二中的操作

9.1.3 筛选器 #3

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	99.5%	100.0%	100.0%	100.0%	99.9%	100.0%	100.0%	97.9%
LSTM	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
DeepSet(like in paper)	84.1%	94.4%	91.5%	94.0%	11.9%	99.8%	99.4%	62.6%
DeepSet(sum at start)	43.8%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	84.2%	95.6%	92.4%	94.0%	9.9%	99.2%	97.8%	65.3%
Sample count	2381	153	138	151	136	574	629	600

表 9.5：筛选器3 简单版本——每 4 个样本中有 1 个被分配到训练集

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	96.5%	99.3%	97.8%	97.0%	53.9%	100.0%	100.0%	98.2%
LSTM	94.8%	99.9%	94.2%	99.7%	21.1%	100.0%	100.0%	99.2%
DeepSet(like in paper)	61.3%	59.0%	55.7%	50.8%	48.0%	71.2%	73.1%	47.1%
DeepSet(sum at start)	43.9%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	62.1%	60.4%	56.4%	53.4%	46.7%	73.7%	73.2%	47.0%
Sample count	3008	197	177	179	175	731	780	769

表 9.6：筛选器3困难版本——每第 19 个样本分配到训练集

9.1.4 过滤器#3- 位置方差

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	100.0%	100.0%	100.0%	99.9%	99.7%	100.0%	100.0%	99.9%
LSTM	100.0%	100.0%	100.0%	100.0%	99.8%	100.0%	100.0%	100.0%
DeepSet(like in paper)	76.0%	100.0%	100.0%	97.9%	0.2%	24.9%	99.5%	98.2%
DeepSet(sum at start)	43.5%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	76.0%	99.9%	99.9%	98.3%	0.0%	25.1%	99.6%	98.0%
Sample count	2540	163	150	164	143	598	693	629

表 9.7：筛选器3（带位置方差）简单版本——每第 5 个样本分配到训练集中

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	97.4%	99.8%	99.4%	98.5%	75.3%	100.0%	100.0%	95.8%
LSTM	94.2%	100.0%	99.4%	100.0%	1.1%	100.0%	99.9%	99.5%
DeepSet(like in paper)	62.2%	58.9%	68.7%	62.9%	37.7%	61.3%	81.4%	47.7%
DeepSet(sum at start)	43.7%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	61.2%	55.2%	63.0%	58.3%	45.4%	64.2%	80.2%	43.7%
Sample count	3016	193	172	180	171	721	805	774

表 9.8：筛选器3（含位置方差）困难版本——每 20 个样本分配一个到训练集中

9.1.5 筛选器 #4

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	96.7%	100.0%	98.3%	100.0%	15.2%	100.0%	100.0%	100.0%
LSTM	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
DeepSet(like in paper)	96.0%	99.0%	98.0%	98.6%	5.1%	99.8%	100.0%	99.5%
DeepSet(sum at start)	44.8%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	96.1%	99.5%	98.0%	98.9%	7.9%	99.9%	100.0%	99.2%
Sample count	3765	176	161	168	142	666	1272	1180

表 9.9：滤波器 4 简单版本——训练集中的有意义的操作之间的距离为 0 和 2，测试集中的距离为 1

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	94.6%	100.0%	99.9%	100.0%	0.3%	98.8%	100.0%	100.0%
LSTM	99.6%	100.0%	97.6%	100.0%	96.8%	100.0%	100.0%	99.7%
DeepSet(like in paper)	88.0%	89.2%	87.5%	88.7%	10.6%	90.1%	98.5%	89.4%
DeepSet(sum at start)	43.9%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	87.9%	88.0%	85.8%	87.3%	12.5%	90.4%	98.8%	88.9%
Sample count	6132	365	309	326	315	1349	1779	1689

表 9.10：滤波器 4 困难版本——训练集中的有意义的操作之间的距离为 1，测试集中的距离为 0 和 2

9.1.6 筛选器2 + #4

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	96.8%	100.0%	100.0%	100.0%	2.7%	100.0%	100.0%	100.0%
LSTM	98.2%	100.0%	91.1%	100.0%	91.1%	100.0%	100.0%	96.6%
DeepSet(like in paper)	61.8%	0.7%	0.5%	1.1%	99.7%	99.6%	99.9%	21.5%
DeepSet(sum at start)	45.6%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	61.8%	1.0%	0.5%	1.3%	99.2%	99.1%	99.9%	21.6%
Sample count	1132	42	37	45	37	166	413	392

表 9.11：筛选器 2 和 4 简单版本——将筛选器 2 的简单版本与筛选器 4 的简单版本结合

	Overall	dr_noop	dr_uw	dr_w	dr_wu	f3_cr	f3_noop	f3_r
CNN	82.3%	100.0%	100.0%	100.0%	0.1%	44.1%	99.7%	98.4%
LSTM	95.8%	95.7%	89.8%	95.6%	70.8%	99.9%	100.0%	93.5%
DeepSet(like in paper)	59.7%	6.9%	6.4%	7.9%	93.0%	99.7%	100.0%	7.8%
DeepSet(sum at start)	43.2%	100.0%	100.0%	100.0%	0.0%	0.0%	0.0%	100.0%
Feedforward	59.8%	8.8%	6.0%	9.0%	91.9%	98.8%	99.9%	8.6%
Sample count	1749	91	88	90	84	386	524	486

表 9.12：筛选器 2 和 4 困难版本——将筛选器 2 的简单版本与筛选器 4 的困难版本相结合