Neural Packet Classification 论文报告

杨广超　202028013229114
谈昊　2020E8013282037

2020 年 12 月 9 日

1　背景介绍

1.1　问题描述

数据包分类一直是计算机网络中的一个基本问题，其目标是在一组规则中，为给定的数据包匹配其中一个规则。数据包分类是许多网络功能的关键组成部分，包括防火墙、访问控制、流量工程和网络测量。因此，数据包分类器的应用十分广泛。在数据包分类过程中，往往需要权衡计算复杂性和状态复杂性。如何缩短分类时间，减少内存占用正是数据包分类问题所面临的难点。

1.2　研究现状

现有的数据包分类解决方案可以分为两大类。第一类中的解决方案是基于硬件的，通常利用三元内容寻址存储器 (Ternary Content-Addressable Memories，TCAMs) 存储所有规则，然后将收到的数据包与这些规则进行匹配。这种方式的分类时间是恒定不变的，但其由于其本身比较复杂，导致了高成本和高功耗。因此，基于 TCAM 的分类方法很少被用来实现大型分类器。第二类解决方案是基于软件的，一般会构建一个复杂的内存数据结构（通常是决策树）进行数据包分类。此类方案虽然相较比基于 TCAM 的分类方法具有更大的可伸缩性，但由于分类操作需要从根到匹配的叶遍历决策树，导致其速度更慢。建立高效的决策树是困难的，经过多年的研究，人们提出了大量基于决策树的数据包分类解决方案但仍然存在两个主要的问题。首先，它们依赖手工调优的启发式来构建树使得它们很难在不同的规则集上理解和优化。其次，这些启发式并没有显式地优化一个给定的目标，而是根据与全局目标联系并不密切的部分信息做决策，这导致其性能可能远远不是最佳的。目前，将机器学习应用于数据包分类一般有两种方法。第一种方法是用神经网络代替决策树，给定一个数据包，神经网络将输出与该数据包匹配的规则。这种方法的缺点是不能保证总是匹配正确而且评估花费过高。第二种方法，也就是文中作者所采用的方法，即使用深度强化学习 (deep reinforcement learning，RL) 来建立有效的决策树。

2 NeuroCuts 方法

2.1 总体思想

文章作者设计了一种名为 NeuroCuts 的深度 RL 数据包分类方法。给定一个规则集和一个目标函数 (例如，分类时间，内存占用，或者两者的组合)，通过 NeuroCuts 学习建立一个决策树，使目标最小化。希望用这样一种基于机器学习的方法来进行数据包分类，从而解决现有手工调优启发式的局限性。有三个特征使得 RL 特别适合于数据包分类。首先，构建决策树的解决方案是从一个节点开始，然后反复地对其进行分割。当我们做出割开一个节点的决策时，在我们完成实际树的构建之前，我们不知道这个决策是否是一个好的决策。RL 很自然地捕获了这个特性，因为它不假定给定决策对性能目标的影响是立即知道的。其次，不同于现有的启发式，RL 算法的明确目标是直接最大化性能目标。第三，能够快速地评估每个模型，显著地减少学习时间。为了实现这样的设计，作者解决了三个关键的挑战。第一，如何编码变长决策树状态 st 作为神经网络策略的输入。考虑到到如何分割树中的节点只取决于节点本身，不依赖于树的其他部分，因此，不需要对整个树进行编码，只需要对当前节点进行编码。第二，如何处理在逐个节点构建决策树过程中产生的稀疏和延迟奖励。在这里，作者利用问题的分支结构，为树的大小和深度提供更密集的反馈。第三，如何将解决方案扩展到大型数据包分类器。训练非常大的规则集可能需要很长时间，为了解决这个问题，作者利用了 RLlib 这个分布式的 RL 库。

2.2 主要模块设计

图 1 显示了 NeuroCuts 作为 RL 系统的结构：环境由规则集和当前决策树组成，而代理使用的模型 (由 DNN 实现) 旨在选择最佳的切割或分区操作，以此来增量地构建树。剪切操作将节点按照选定的维度 (即 SrcIP、DstIP、SrcPort、DstPort 和 Protocol 中的一个维度) 划分为若干个子范围 (即 2,4,8,16 或 32 个范围)，并在树中创建这么多的子节点。另一方面，分区操作将一个节点的规则划分为不相交的子集 (例如，基于维度的覆盖率)，并为每个子集创建一个新的子节点。当前节点的可用操作在每一步都由环境告知，代理在其中选择生成树，随着时间的推移，代理学会优化其决策，以最大限度地从环境中获得回报。图 2 显示了 NeuroCuts 的学习过程，其中，x 轴表示树的级别，y 轴表示该级别上的节点数。树的每一层的切割维度的分布以颜色显示。

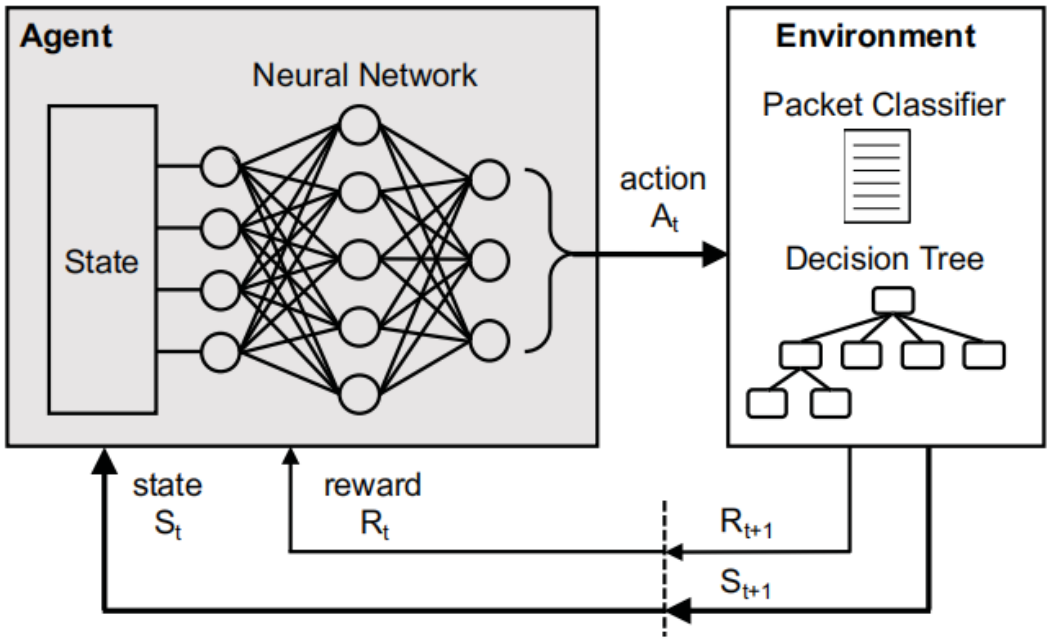
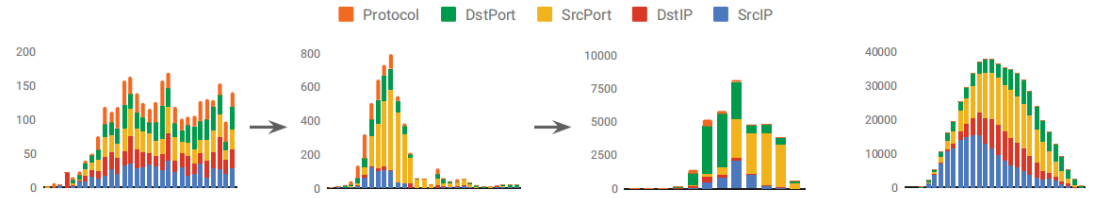


图 1 NeuroCuts 作为 RL 系统的结构



(a) NeuroCuts starts with a randomly initialized policy that generates poorly shaped trees (left, truncated). Over time, it learns to reduce the tree depth and develops a more coherent strategy (center). The policy converges to a compact depth-12 tree (right) that specializes in cutting `SrcIP`, `SrcPort`, and `DstPort`.

(b) In comparison, HiCuts produces a depth-29 tree for this rule set that is 15× larger and 3× slower in classification time.

Figure 5: Visualization of NeuroCuts learning to split the fw5_1k ClassBench rule set. The x-axis denotes the tree level, and the y-axis the number of nodes at the level. The distribution of cut dimensions per level of the tree is shown in color.

图 2 NeuroCuts 学习分割 w5_1k ClassBench 规则集的可视化

2.2.1 训练算法

我们使用一个 actor-critic 算法来训练代理的策略。此算法在许多用例中都提供了较好的结果，并且容易地缩放到分布式环境下。算法 1 显示了 NeuroCuts 算法的伪代码，其执行如算法所示。

Algorithm 1 Learning a tree-generation policy using an actor-critic algorithm.

Input: The root node $\mathbf{s}^*$ where a tree always grows from.

Output: A stochastic policy function $\pi(a|\mathbf{s};\theta)$ that outputs a branching action $a \in \mathcal{A}$ given a node state $\mathbf{s}$, and a value function $V(\mathbf{s};\theta_v)$ that outputs a value estimate for a node state.

Main routine:

// Initialization
Randomly initialize the model parameters θ, θ_v
Maximum number of rollouts N
Coefficient $c \in [0, 1]$ that trades off classification time vs. space
Reward scaling function $f(x) \in \{x, \textsc{log}(x)\}$
$n \leftarrow 0$
// Training
while $n < N$ **do**
    $s \leftarrow \textsc{Reset}(\mathbf{s}^*)$
    // Build a tree using the current policy
    while $s \neq \textsc{Null}$ **do**
        $a \leftarrow \pi(a|s;\theta)$
        $s \leftarrow \textsc{GrowTreeDFS}(s, a)$
    Reset gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    for $(s, a) \in \textsc{TreeIterator}(\mathbf{s}^*)$ **do**
        // Compute the future rewards for the given action
        $R \leftarrow -(c \cdot f(\textsc{Time}(s)) + (1 - c) \cdot f(\textsc{Space}(s)))$
        // Accumulate gradients wrt. policy gradient loss
        $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a|s;\theta)(R - V(s;\theta_v))$
        // Accumulate gradients wrt. value function loss
        $d\theta_v \leftarrow d\theta_v + \partial(R - V(s;\theta_v))^2 / \partial\theta_v$
    Perform update of θ using d_θ and θ_v using $d\theta_v$.
    $n \leftarrow n + 1$

Subroutines:

- $\textsc{Reset}$(s): Reset the tree s to its initial state.
- $\textsc{GrowTreeDFS}$(s, a): Apply action a to tree node s, and return the next non-terminal leaf node in the tree in depth-first traversal order.
- $\textsc{TreeIterator}$(s): Non-terminal tree nodes of the subtree s and their taken action.
- $\textsc{Time}$(s): Upper-bound on classification time to query the subtree s. In non-partitioned trees this is simply the depth of the tree.
- $\textsc{Space}$(s): Memory consumption of the subtree s.

2.2.2 整合现有的启发式

NeuroCuts 很容易加入额外的启发式来改进它学习的决策树，一个例子是添加规则划分动作。除了切割动作，在我们的 NeuroCuts 实现中，我们还允许两种类型的分割动作:Simple 和 EffiCuts。

2.2.3 处理大型包分类器

小型分类器可以采用 NeuroCuts 的单线程实现，但对于拥有数万甚至数十万条规则的大型分类器，并行性可以显著提高训练速度。图 3展示了如何调整算法来并行地构建多个决策树。

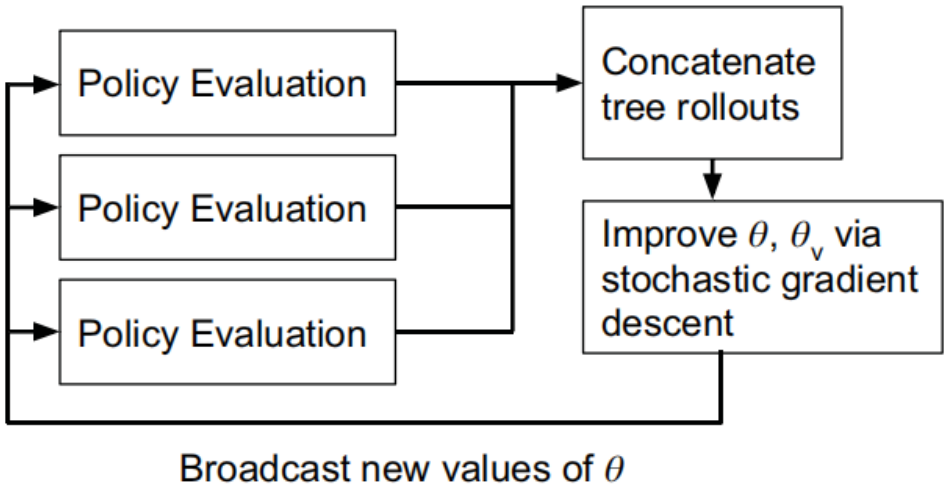


图 3 NeuroCuts 可以通过从当前策略并行生成决策树来并行化

2.2.4 处理分类器更新

实际使用过程中经常根据应用需求来更新数据包分类器。对于少数规则的小更新，NeuroCuts 修改现有的决策树来反映这些变化，包括添加和删除；当变化较大时 NeuroCuts 会重新运行训练。

3 实验评估

3.1 实验环境

在 m4.16xl AWS 机器上运行 NeuroCuts，每个实例使用四个 CPU 内核加速，使用 Python 实现决策树的生成，每个 NeuroCuts 实例最多运行一千万个时间步长，或者直到收

敛为止。

3.2 实验评估

作者使用标准的 ClassBench 来生成具有不同特征和大小的数据包分类器，将 NeuroCuts 与 HiCuts、HyperCuts、EfiCuts 和 CutSplit 四种手动调优算法进行比较。基准度量标准包括分类时间和内存占用。结果表明，NeuroCuts 在分类时间上显著提高了所有基线，同时也生成了更紧凑的树。在优化内存占用方面，NeuroCuts 在不影响时间的情况下，比 EffiCuts 提高了 25% 的中位空间。在分类时间方面，通过对基于 ClassBench 分类器的最佳时间优化树的生成时长进行对比，NeuroCuts 分别比 HiCuts, HyperCuts, EffiCuts 和 CutSplit 提供了 20%，38%，52% 和 56% 的中值改善。

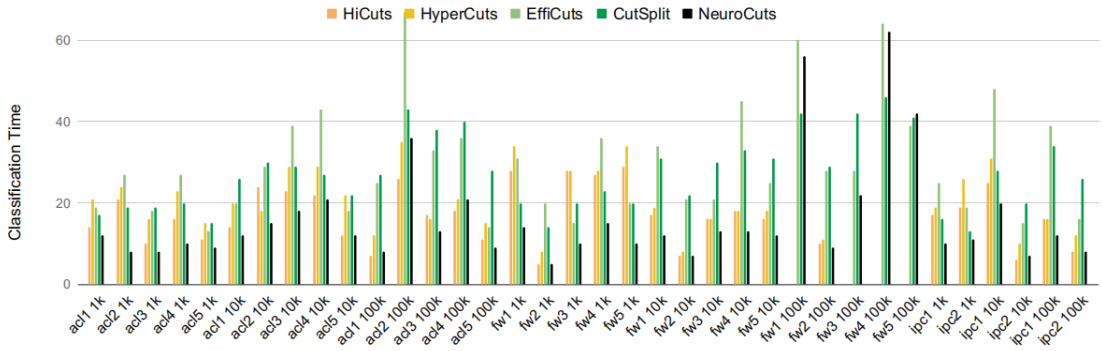


图 4 HiCuts、HyperCuts、EffiCuts 和 NeuroCuts 的分类时间 (树的深度)(时间优化)。我们省略了 4 个在超过 24 小时后没有完成的 HiCuts 和 HyperCuts 条目。

在内存占用方面，NeuroCuts 远远好于 HiCuts 和 HvperCuts，对比 EffiCuts，其空间优化树的中值提高了 40%，平均提高了 44%。不过，NeuroCuts 与 CutSplit 相比，中值内存使用量高出 26%，尽管在所有基准上最佳情况下的改进仍然是 3 倍（66%）。

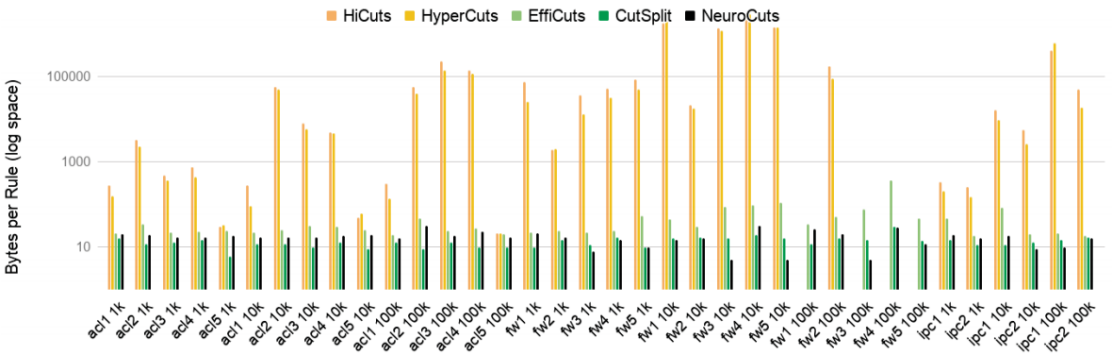
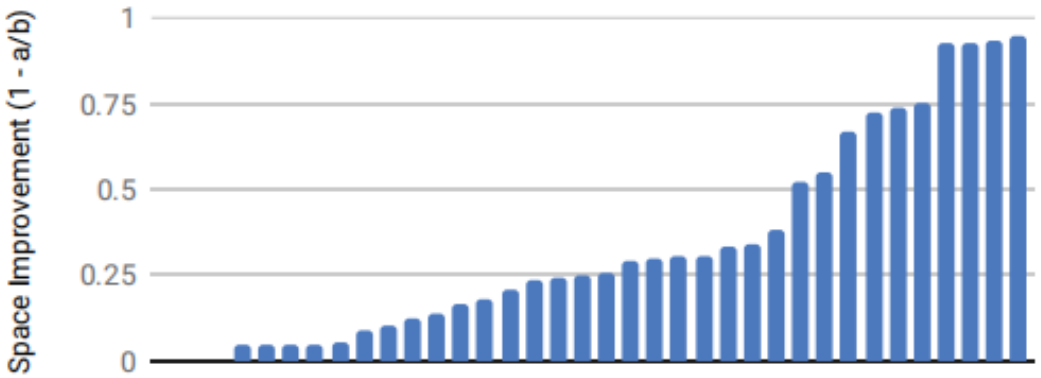
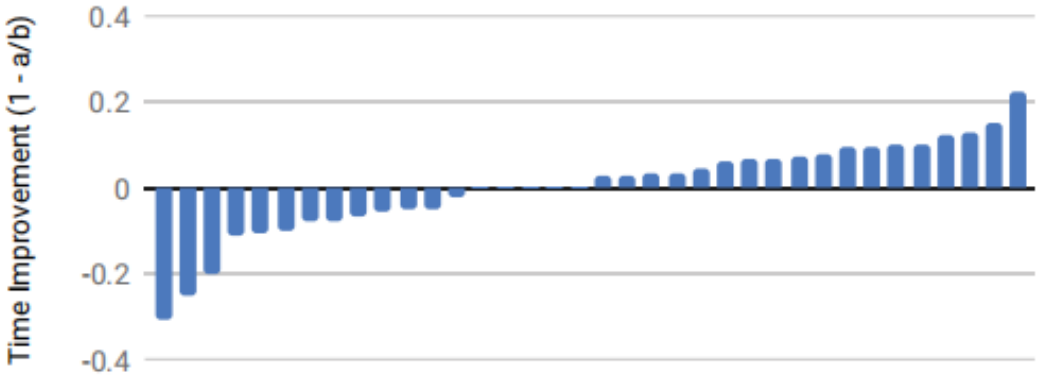


图 5 HiCuts、HyperCuts、EffiCuts 和 NeuroCuts(空间优化) 使用的内存占用 (每个规则的字节数)。我们省略了 4 个在超过 24 小时后没有完成的 HiCuts 和 HyperCuts 条目

除此之外，实验还表明，NeuroCuts 能够有效地合并和改进预先设计的启发式，如 EffiCuts 顶层划分函数。NeuroCuts 可能首先学习产生基本解决方案的随机动作分布，然后利用其神经网络的能力，将动作分布专门化到规则空间的不同部分。最后，作者通过使用简单的分割方法和 LOG(X) 奖励缩放来扫描 NeuroCuts 的 c 值范围发现，时空系数 c 可以有效权衡并控制内存占用空间和与分类时间。



(a) NeuroCuts can build on the EffiCuts partitioner to generate trees up to 10× (90%) more space efficient than EffiCuts. In this experiment NeuroCuts did as well or better than EffiCuts on all 36 rule sets.



(b) NeuroCuts with the EffiCuts partitioner generates trees with about the same time efficiency as EffiCuts.

图 6 在 ClassBench 基准测试中，对 NeuroCuts 相对于效率的改进进行排序。这里只允许使用 EffiCuts 分区方法运行 NeuroCuts。正值表示改进

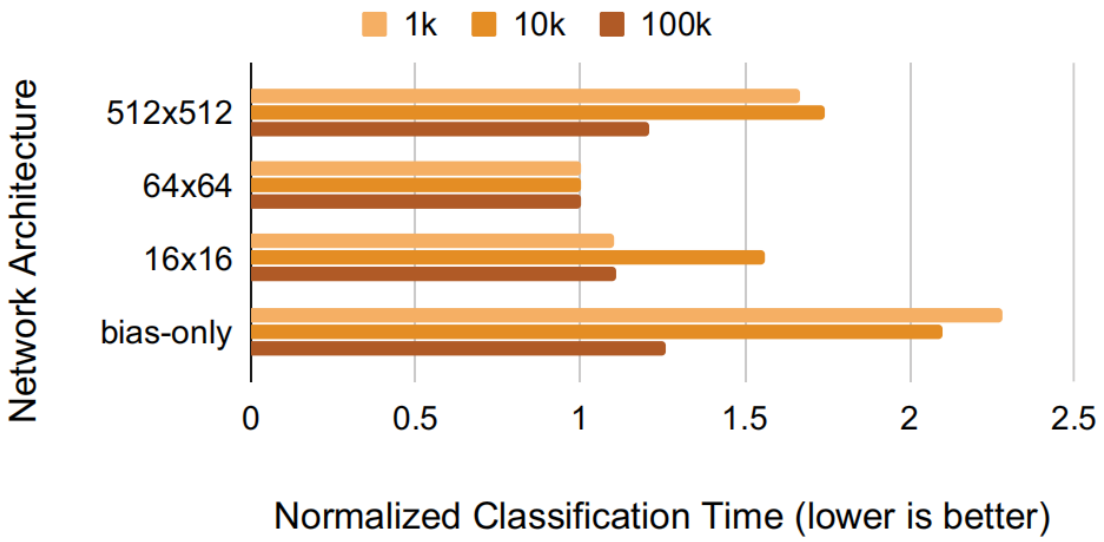


图 7 不同网络架构和不同分类器组的神经切割器平均最佳分类时间的比较仅偏倚架构指的是一个不处理观察结果并发出固定动作概率分布 (即纯强盗) 的普通神经网络。结果在分类器组内进行归一化，使最佳树的归一化时间为 1。未收敛到有效树的规则集在归一化之前被分配 100 的时间

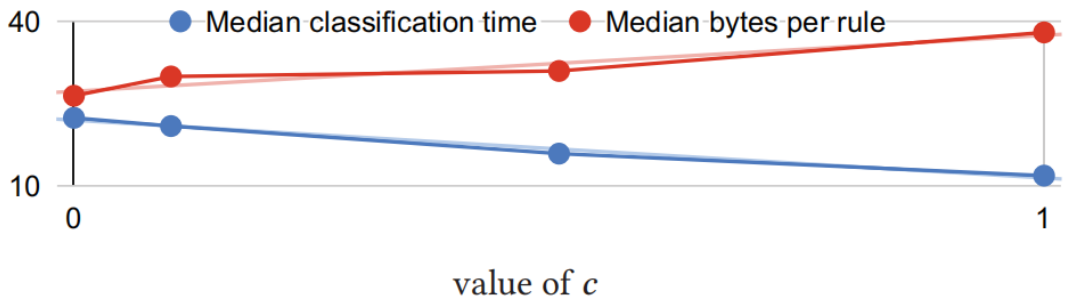


图 8 当 timespace 系数 c = 1 时，分类时间提高了 2x，反之，当 c = 0 时，每个规则的字节数提高了 2x

4 读后感

数据包分类应用十分广泛，其难点就在于如何权衡计算复杂性和状态复杂性，尽可能地缩短分类时间，减少内存占用。在这篇论文中，作者基于深度强化学习 (RL) 方法的三个重要特征，创造性地建立了 NeuroCuts 来解决数据包分类问题。实验评估表明，相较目前几种手动调优算法，这一系统在分类时间和内存占用空间等方面都取得了非常不错的表现。

强化学习一般用于描述和解决智能体在与环境的交互过程中通过学习策略以达成回报最大化或实现特定目标的问题，在信息论、博弈论、自动控制等领域都得到了广泛的讨论。

这篇文章的作者将其应用于数据包分类领域并取得了良好的效果，为解决包分类问题提供了一个新的思路。

通过阅读这篇文章，我们全面地了解了数据包分类问题的研究现状以及难点，明确了该领域当前面临的困境及应重点突破的方向。此外，我们还进一步加深了对深度强化学习的理解，尤其是在研究 RL 适用于数据包分类的三个特征以及 NeuroCuts 设计过程中的三个挑战和解决方案的过程中，我和我的队友就 RL 能解决什么问题以及如何使用 RL 解决问题进行了深入的思考与讨论。进一步认识 RL 奖励的延迟和稀疏两个特点以及学习如何将可变长度决策树状态编码为神经网络策略的输入将会对我们将来的学习与科研有所启发。

5 作者背景介绍

本文作者来自伯克利的 RISElab。RISELab 代表着实时智能安全执行（RISE），其定位是在分布式计算中解决下一个阶段，根据 Databricks 博客的说法：“Storica 曾表示，这个新阶段是为了通过两个项目——Drizzle 和 Opaque，改进 Spark 并实现创新，其致力于构建开源框架、工具、算法，以便能够以更高的安全性，根据实时数据，决定要构建哪些实时应用。”

RISELab 的初期目标是为了增强 Spark 的安全性与实时能力，因此，根据 Databricks 的信息，Drizzle 项目的目标是将 Spark Streaming 的延迟降低一个数量级，同时提高其容错性。Opaque 项目是为了增强 Spark 的动态与静态数据的加密功能。