Neural Packet Classification

Eric Liang[1], Hang Zhu[2], Xin Jin[2], Ion Stoica[1]

[1]UC Berkeley, [2]Johns Hopkins University

ekl@berkeley.edu, hzhu@jhu.edu, xinjin@cs.jhu.edu, istoica@cs.berkeley.edu

ABSTRACT

Packet classification is a fundamental problem in computer networking. This problem exposes a hard tradeoff between the computation and state complexity, which makes it particularly challenging. To navigate this tradeoff, existing solutions rely on complex hand-tuned heuristics, which are brittle and hard to optimize.

In this paper, we propose a deep reinforcement learning (RL) approach to solve the packet classification problem. There are several characteristics that make this problem a good fit for Deep RL. First, many existing solutions iteratively build a decision tree by splitting nodes in the tree. Second, the effects of these actions (e.g., splitting nodes) can only be evaluated once the entire tree is built. These two characteristics are naturally captured by the ability of RL to take actions that have sparse and delayed rewards. Third, it is computationally efficient to generate data traces and evaluate decision trees, which alleviate the notoriously high sample complexity problem of Deep RL algorithms. Our solution, NeuroCuts, uses succinct representations to encode state and action space, and efficiently explore candidate decision trees to optimize for a global objective. It produces compact decision trees optimized for a specific set of rules and a given performance metric, such as classification time, memory footprint, or a combination of the two. Evaluation on Class-Bench shows that NeuroCuts outperforms existing hand-crafted algorithms in classification time by 18% at the median, and reduces both classification time and memory footprint by up to 3×.

CCS CONCEPTS

• **Networks** → **Packet classification**; • **Theory of computation** → **Reinforcement Learning**.

KEYWORDS

Packet classification, Reinforcement learning

1 INTRODUCTION

Packet classification is one of the fundamental problems in computer networking. The goal of packet classification is to match a given packet to a rule from a set of rules, and to do so while optimizing the classification time and/or memory footprint. Packet classification is a key building block for many network functionalities, including firewalls, access control, traffic engineering, and network measurements [13, 29, 55]. As such, packet classifiers are widely deployed by enterprises, cloud providers, ISPs, and IXPs [1, 29, 48].

Existing solutions for packet classification can be divided into two broad categories. Solutions in the first category are hardware-based. They leverage Ternary Content-Addressable Memories (TCAMs) to store all rules in an associative memory, and then match a packet to all these rules in parallel [23]. As a result, TCAMs provide constant classification time, but come with significant limitations. TCAMs are inherently complex, and this complexity leads to high cost and power consumption. This makes TCAM-based solutions prohibitive for implementing large classifiers [55].

The solutions in the second category are software based. These solutions build sophisticated in-memory data structures—typically decision trees—to efficiently perform packet classification [29]. While these solutions are far more scalable than TCAM-based solutions, they are slower, as the classification operation needs to traverse the decision tree from the root to the matching leaf.

Building efficient decision trees is difficult. Over the past two decades, researchers have proposed a large number of decision tree based solutions for packet classification [13, 29, 40, 47, 55]. However, despite the many years of research, these solutions have two major limitations. First, they rely on hand-tuned heuristics to build the tree. Examples include maximizing split entropy [13], balancing splits with custom space measures [13], special handling for wildcard rules [47], and so on. This makes them hard to understand and optimize over different sets of rules. If a heuristic is too general, it cannot take advantage of the characteristics of a particular set of rules. If a heuristic is designed for a specific set of rules, it typically does not achieve good results on another set of rules with different characteristics.

Second, these heuristics do not explicitly optimize for a given objective (e.g., tree depth). They make decisions based on information (e.g., the difference between the number of rules in the children, the number of distinct ranges in each dimension) that is only *loosely* related to the global objective. As such, their performance can be far from optimal.

In this paper, we propose a learning approach to packet classification. Our approach has the potential to address the limitations of the existing hand-tuned heuristics. In particular, our approach *learns* to optimize packet classification for a given set of rules and objective, can easily incorporate pre-engineered heuristics to leverage their domain knowledge, and does so with little human involvement. The recent successes of deep learning in solving notoriously hard problems, such as image recognition [22] and language translation [51], have inspired many practitioners and researchers to apply deep learning, in particular, and machine learning, in general,

to systems and networking problems [4, 6, 16, 33, 34, 54, 62, 64, 65]. While in some of these cases there are legitimate concerns about whether machine learning is the right solution for the problem at hand, we believe that deep learning is a good fit for our problem. This is notable since, when an efficient formulation is found, learning-based solutions have often outperformed hand-crafted alternatives [21, 36, 44].

There are two general approaches to apply learning to packet classification. The first is to replace the decision tree with a neural network, which given a packet will output the rule matching that packet. Unfortunately, while appealing, this end-to-end solution has a major drawback: it does not guarantee the correct rule is always matched. While this might be acceptable for some applications such as traffic engineering, it is not acceptable for others, such as access control. Another issue is that large rule sets will require correspondingly large neural network models, which can be expensive to evaluate without accelerators such as GPUs. The second approach, and the one we take in this paper, is to use deep learning to build a decision tree. Recent work has applied deep learning to optimize decision trees for machine learning problems [20, 38, 59]. These solutions, however, are designed for machine learning settings that are different than packet classification, and aim to maximize accuracy. In contrast, decision trees for packet classification provide perfect accuracy by construction, and the goal is to minimize classification time and memory footprint.

Our solution uses deep reinforcement learning (RL) to build efficient decision trees. There are three characteristics that makes RL a particularly good fit for packet classification. First, the natural solution to build a decision tree is to start with one node and recursively split (cut) it. Unfortunately, this kind of approach does not have a greedy solution. When making a decision to cut a node, we do not know whether that decision was a good one (i.e., whether it leads to an efficient tree) before we finish building the actual tree. RL naturally captures this characteristic as it does not assume that the impact of a given decision on the performance objective is known immediately. Second, unlike existing heuristics which take actions that are only loosely related to the performance objective, the explicit goal of an RL algorithm is to directly maximize the performance objective. Third, unlike other RL domains such as as robotics, for our problem it is possible to evaluate an RL model quickly (i.e., a few seconds of CPU time). This alleviates one of the main drawbacks of RL algorithms: the non-trivial learning time due to the need to evaluate a large number of models to find a good solution. By being able to evaluate each model quickly (and, as we will see, in parallel) we significantly reduce the learning time.

To this end, we design NeuroCuts, a deep RL solution for packet classification that learns to build efficient decision trees. There are three technical challenges to formulate this problem as an RL problem. First, the tree is growing during the execution of the algorithm, as existing nodes are split. This makes it very difficult to encode the decision tree, as RL algorithms require a fixed size input. We address this problem by noting that the decision of how to split a node in the tree depends only on the node itself; it does not depend on the rest of the tree. As such, we do not need to encode the entire tree; we only need to encode the current node. The second challenge is in reducing the sparsity of rewards to accelerate the learning process; here we exploit the branching structure of the

Priority	Src IP	Dst IP	Src Port	Dst Port	Protocol
2	10.0.0.0	10.0.0.0/16	*	*	*
1	*	*	[0, 1023]	[0, 1023]	TCP
0	*	*	*	*	*

Figure 1: A packet classifier example. Real-world classifiers can have 100K rules or more.

problem to provide denser feedback for tree size and depth. The final challenge is that training for very large sets of rules can take a long time. To address this, we leverage RLlib [30], a distributed RL library.

In summary, we make the following contributions.

- We show that the packet classification problem is a good fit for reinforcement learning (RL).
- We present NeuroCuts, a deep RL solution for packet classification that learns to build efficient decision trees.
- We show that NeuroCuts outperforms state-of-the-art solutions, improving classification time by 18% at the median and reducing both time and memory usage by up to 3×.

The code for NeuroCuts is open source and is available at https://github.com/neurocuts/neurocuts.

2 BACKGROUND

In this section, we provide background on the packet classification problem, and summarize the key ideas behind the decision tree based solutions to solve this problem.

2.1 Packet Classification

A packet classifier contains a list of rules. Each rule specifies a pattern on multiple fields in the packet header. Typically, these fields include source and destination IP addresses, source and destination port numbers, and protocol type. The rule's pattern specifies which packets match the rule. Matching conditions include prefix based matching (e.g., for IP addresses), range based matching (e.g., for port numbers), and exact matching (e.g., for protocol type). A packet matches a rule if each field in the packet header satisfies the matching condition of the corresponding field in the rule, e.g., the packet's source/destination IP address matches the prefix of the source/destination address in the rule, the packet's source/destination port number is contained in the source/destination range specified in the rule, and the packet's protocol type matches the rule's protocol type.

Figure 1 shows a packet classifier with three rules. The first rule matches all packets with source address 10.0.0.1 and the destination addresses sharing prefix 10.0.0.0/16. Other fields are unspecified (i.e., they are ⋆) meaning that the rule matches any value in these fields. The second rule matches all TCP packets with source and destination ports in the range [0, 1023], irrespective of IP addresses (as they are ⋆). Finally, the third rule is a default rule that matches all packets. This guarantees that any packet matches at least one rule.

Since rules can overlap, it is possible for a packet to match multiple rules. To resolve this ambiguity, each rule is assigned a priority. A packet is then matched to the highest priority rule. For example, packet (10.0.0.0, 10.0.0.1, 0, 0, 6) matches all the three rules of the
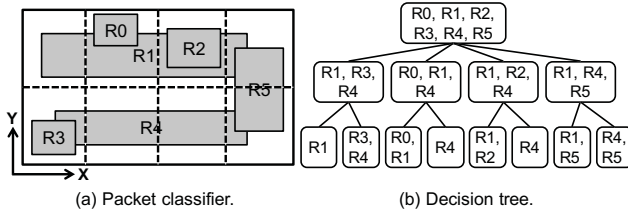
Figure 2: Node cutting.

packet classifier in Figure 1. However, since the first rule has the highest priority, we match the packet to the first rule only.

2.2 Decision Tree Algorithms

Packet classification is similar to the point location problem in a multi-dimensional geometric space: the fields in the packet header we are doing classification on (e.g., source and destination IP addresses, source and destination port numbers, and protocol number) represent the dimensions in the geometric space, a packet is represented as a point in this space, and a rule as a hypercube. Unfortunately, the point location problem exhibits a hard tradeoff between time and space complexities [14].

The packet classification problem is then equivalent to finding all hypercubes that contains the point corresponding to a given packet. In particular, in a d-dimensional geometric space with n non-overlapping hypercubes and when $d > 3$, this problem has either (i) a lower bound of $O(\log n)$ time and $O(n^d)$ space, or (ii) a lower bound of $O(\log^{d-1} n)$ time and $O(n)$ space [14]. The packet classification problem allows the hypercubes (i.e., rules) to overlap, and thus is at least as hard as the point location problem [14]. In other words, if we want logarithmic computation time, we need space that is exponential in the number of dimensions (fields), and if we want linear space, the computation time will be exponential in the logarithm of the number of rules. Given that for packet classification $d = 5$, neither of these choices is attractive.

Next, we discuss two common techniques employed by existing solutions to build decision trees for packet classification: *node cutting* and *rule partition*.

Node cutting. Most existing solutions for packet classification aim to build a decision tree that exhibits low classification time (i.e., time complexity) and memory footprint (i.e., space complexity) [55]. The main idea is to split nodes in the decision tree by "cutting" them along one or more dimensions. Starting from the root which contains all rules, these algorithms iteratively split/cut the nodes until each leaf contains fewer than a predefined number of rules. Given a decision tree, classifying a packet reduces to walk the tree from the root to a leaf, and then chose the highest priority rule associated with that leaf.

Figure 2 illustrates this technique. The packet classifier contains six rules (R0 to R5) in a two-dimensional space. Figure 2(a) shows each rule as a rectangle in the space, and represents the cuts as dashed lines. Figure 2(b) shows the corresponding decision tree for this packet classifier. The root of the tree contains all the six rules. First, we cut the entire space (which represents the root) into four chunks along dimension x. This leads to the creation of four children. If a rule intersects a child's chunk, it is added to that child.
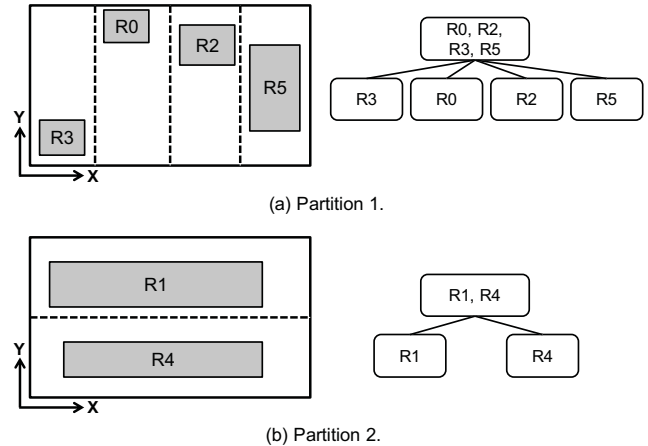

Figure 3: Rule partition.

For example, R1, R3 and R4 all intersect the first chunk (i.e., the first quarter in this space), and thus they are all added to the first root's child. If a rule intersects multiple chunks it is added to each corresponding child, e.g., R1 is added to all the four children. Next, we cut the chunk corresponding to each of the four children along dimension y. As a result, each of the nodes at the first level will end up with two children.

Rule partition. One challenge with "blindly" cutting a node is that we might end up with a rule being replicated to a large number of nodes [55]. In particular, if a rule has a large size along one dimension, cutting along that dimension will result in that rule being added to many nodes. For example, rule R1 in Figure 2(a) has a large size in dimension x. Thus, when cutting along dimension x, R1 will end up being replicated at every node created by the cut. Rule replication can lead to decision trees with larger depths and sizes, which translate to higher classification time and memory footprint.

One solution to address this challenge is to first partition rules based on their "shapes". Broadly speaking, rules with large sizes in a particular dimension are put in the same set. Then, we can build a separate decision tree for each of these partitions. Figure 3 illustrates this technique. The six rules in Figure 2 are grouped into two partitions. One partition consists of rules R1 and R4, as both these rules have large sizes in dimension x. The other partition consists of the other four rules, as these rules have small sizes in dimension x. Figure 3(a) and Figure 3(b) show the corresponding decision trees for each partition. Note that the resulting trees have lower depth, and smaller number of rules per node as compared to the original decision tree in Figure 2(b). To classify a packet, we classify it against every decision tree, and then choose the highest priority rule among all rules the packet matches in all decision trees.

Summary. Existing solutions build decision trees by employing two types of actions: node cutting and rule partition. These solutions mainly differ in the way they decide (i) at which node to apply the action, (ii) which action to apply, and (iii) how to apply it (e.g., along which dimension(s) to partition).

3 A LEARNING-BASED APPROACH

In this section, we describe a learning-based approach for packet classification. We motivate our approach, discuss the formulation of classification as a learning problem, and then present our solution.

3.1 Why Learn?

The existing solutions for packet classification rely on hand-tuned heuristics to build decision trees. Unfortunately, this leads to two major limitations.

First, these heuristics often face a difficult *trade-off* between *performance* and *cost*. Tuning such a heuristic for a given set of rules is an expensive proposition, requiring considerable human efforts and expertise. Worse yet, when given a different rule set, one might have to do this all over again. Addressing this challenge has been the main driver of a long line of research over the past two decades [13, 29, 40, 47, 55]. Of course, one could build a general heuristic for a large variety of rule sets. Unfortunately, such a solution would not provide the best performance for a given set of rules.

Second, existing algorithms do not directly optimize for a global objective. Ideally, a good packet classification solution should optimize for (*i*) classification time, (*ii*) memory footprint, or (*iii*) a combination between the two. Unfortunately, the existing heuristics do not *directly* optimize for any of these objectives. At their core, these heuristics make greedy decisions to build decision trees. At every step, they decide on whether to cut a node or partition the rules based on simple statistics (e.g., the size of the rules in each dimension, number of unique ranges in each dimension), which are poorly correlated with the desired objective. As such, the resulting decision trees are often far from being optimal.

As we will see, a learning-based approach can address these limitations. Such an approach can learn to generate an efficient decision tree for a specific set of rules without the need to rely on hand-tuned heuristics. This is not to say these heuristics do not have value; in fact they often contain key domain knowledge that we show can be leveraged and improved on by the learning algorithm.

3.2 What to Learn?

Classification is a central task in machine learning literature. The recent success of using deep neural networks (DNNs) for image recognition, speech recognition and language translation has been single-handedly responsible for the recent AI "revolution" [10, 22, 51].

As such, one natural solution for packet classification would be to replace a decision tree with a DNN. In particular, such DNN will take as input the fields of a packet header and output the rule matching that packet. Related to our problem, prior work has shown that DNN models can be effectively used to replace B-Trees for indexing [21].

However, this solution has two drawbacks. First, a DNN-based classifier does not guarantee 100% accuracy. This is because training a DNN is fundamentally a stochastic process. Second, given a DNN packet classification result, it is expensive to verify whether the result is correct or not. Unlike the recently proposed learned index solution to replace B-Trees [21], the rules in packet classification are
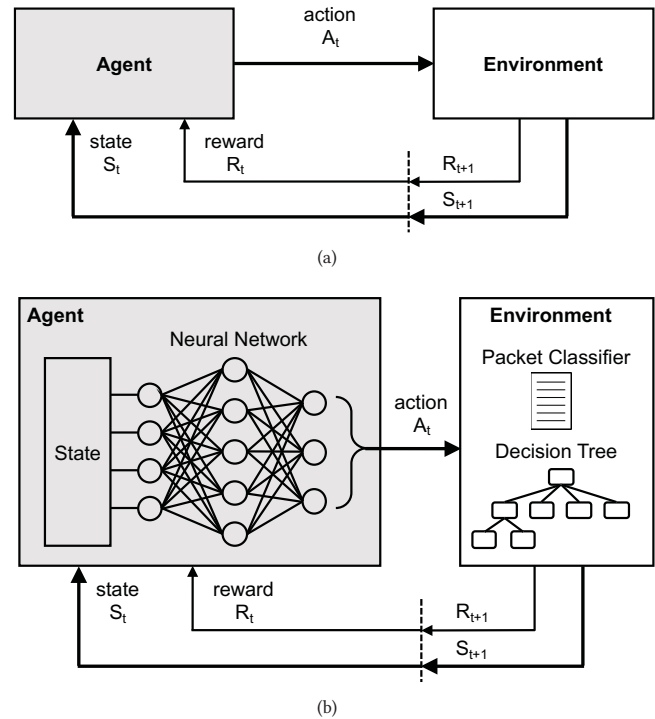

(a)

(b)

Figure 4: (a) Classic RL system. An agent takes an action, A_t, based on the current state of the environment, S_t, and applies it to the environment. This leads to a change in the environment state (S_{t+1}) and a reward (R_{t+1}). (b) NeuroCuts as an RL system.

multi-dimensional and overlap with each other. If a rule matches a packet, we still need to check other rules to see if this rule has the highest priority among all matched rules.

To avoid these drawbacks, in this paper we propose to learn building decision trees for a given set of rules. Since the result is still a decision tree, we can guarantee correctness, and it will be easy to deploy the classifier with existing systems (hardware and software) compared to a DNN.

3.3 How to Learn?

In this section, we show that the problem of building decision trees maps naturally to RL. As illustrated in Figure 4(a), an RL system consists of an agent that repeatedly interacts with an environment. The agent observes the state of the environment, and then takes an action that might change the environment's state. The goal of the agent is to compute a *policy* that maps the environment's state to an action in order to optimize a reward. As an example, consider an agent playing chess. In this case, the environment is the board, the state is the position of the pieces on the board, an action is moving a piece on the board, and the reward could be 1 if the game is won, and −1, if the game is lost.

This simple example illustrates two characteristics of RL that are a particularly good fit to our problem. First, rewards are *sparse*, i.e., not every state has associated a reward. For instance, when moving a piece we do not necessary know whether that move will result

in a win or loss. Second, the rewards are *delayed*; we need to wait until the end of the game to see whether the game was won or lost.

To deal with large state and action spaces, recent RL solutions have employed DNNs to implement their policies. These solutions, called Deep RL, have achieved remarkable results matching humans at playing Atari games [36], and beating the Go world champion [46]. These results have encouraged researchers to apply Deep RL to networking and systems problems, from routing, to congestion control, to video streaming, and to job scheduling [4, 6, 16, 33, 34, 54, 62, 64, 65]. Building a decision tree can be easily cast as an RL problem: the environment's state is the current decision tree, an action is either cutting a node or partitioning a set of rules, and the reward is either the classification time, memory footprint, or a combination of the two. While in some cases there are legitimate concerns about whether Deep RL is the right solution for the problem at hand, we identify several characteristics that make packet classification a particularly good fit for Deep RL.

First, when we take an action, we do not know for sure whether it will lead to a good decision tree or not; we only know this once the tree is built. As a result, the rewards in our problem are both *sparse* and *delayed*. This is naturally captured by the RL formulation.

Second, the explicit goal of RL is to maximize the reward. Thus, unlike existing heuristics, our RL solution aims to explicitly optimize the performance objective, rather than using local statistics whose correlation to the performance objective can be tenuous.

Third, one potential concern with Deep RL algorithms is sample complexity. In general, these algorithms require a huge number of samples (i.e., input examples) to learn a good policy. Fortunately, in the case of packet classification we can generate such samples cheaply. A sample, or rollout, is a sequence of actions that builds a decision tree with the associated reward(s) by using a given policy. The reason we can generate these rollouts cheaply is because we can build all these trees in software, and do so in parallel. Contrast this with other RL-domains, such as robotics, where generating each rollout can take a long time and requires expensive equipment (i.e., robots).

4 NEUROCUTS DESIGN

4.1 NeuroCuts Overview

We introduce the design for NeuroCuts, a new Deep RL formulation of the packet classification problem. Given a rule set and an objective function (i.e., classification time, memory footprint, or a combination of both), NeuroCuts learns to build a decision tree that minimizes the objective.

Figure 4(b) illustrates the framing of NeuroCuts as an RL system: the environment consists of the set of rules and the current decision tree, while the *agent* uses a model (implemented by a DNN) that aims to select the best cut or partition action to incrementally build the tree. A cut action divides a node along a chosen dimension (i.e., one of SrcIP, DstIP, SrcPort, DstPort, and Protocol) into a number of sub-ranges (i.e., 2, 4, 8, 16, or 32 ranges), and creates that many child nodes in the tree. A partition action on the other hand divides the rules of a node into disjoint subsets (e.g., based on the coverage fraction of a dimension), and creates a new child node for each subset. The available actions for the current node are advertised by the environment at each step, the agent chooses

among them to generate the tree, and over time the agent learns to optimize its decisions to maximize the reward from the environment. Figure 5 visualizes the learning process of NeuroCuts.

4.2 NeuroCuts Training Algorithm

Recall that the goal of an RL algorithm is to compute a policy to maximize rewards from the environment. Referring again to Figure 4, the environment defines the action space A and state space S. The agent starts with an initial policy, evaluates it using multiple rollouts, and then updates it based on the results (rewards) of these rollouts. Then, it repeats this process until satisfied with the reward.

We first consider a strawman formulation of decision tree generation as a single Markov Decision Process (MDP). In this framing, a rollout begins with a tree consisting of a single node. This is the initial state, $s_0 \in S$. At each step t, the agent executes an action $a_t \in A$ and receives a reward r_t; the environment transitions from the current state $s_t \in S$ to the next state $s_{t+1} \in S$ (i.e., the updated tree and next node to process). The goal is to maximize the total reward received by the agent, i.e., $\sum_t \gamma^t r_t$ where γ is a discounting factor used to prioritize more recent rewards.

Design challenges. While at a high level this RL formulation seems straightforward, there are three key challenges we need to address before we have a realizable implementation. The first is how to encode the variable-length decision tree state s_t as an input to the neural network policy. While it is possible to flatten the tree, say, into an 1-dimensional vector, the size of such a vector would be very large (i.e., hundreds of thousands of units). This will require both a very large network model to process such input, and a prohibitively large number of samples.

While recent work has proposed leveraging recurrent neural networks (RNNs) and graph embedding techniques [58, 60, 61] to reduce the input size, these solutions are brittle in the face of large or dynamically growing graph structures [66]. Rather than attempting to solve the state representation problem to deal with large inputs, in NeuroCuts we instead take advantage of the underlying structure of packet classification trees to design a simple and compact state representation. This means that when the agent is deciding how to split a node, it only observes a fixed-length representation of the node. All needed state is encoded in the representation; no other information about the rest of the tree is observed.

The second challenge is how to deal with the sparse and delayed rewards incurred by the node-by-node process of building the decision tree. While we could in principle return a single reward to the agent when the tree is complete, it would be very difficult to train an agent in such an environment. Due to the long length of tree rollouts (i.e., many thousands of steps), learning is only practical if we can compute *meaningful dense rewards*.[1] Such a dense reward for an action would be based on the statistics of the subtree it leads to (i.e., its depth or size).[2] This effectively reduces the delay of the rewards from O(tree size) to $O(log$(tree size)). Unfortunately, it is not possible to compute this until the subtree is complete. To

---

[1] Note that just returning -1 or $-cutSize$ for each step would be a dense reward but not particularly useful.

[2] The rewards for NeuroCuts correspond to the true problem objective; we do not do "reward engineering" since that would bias the solution.

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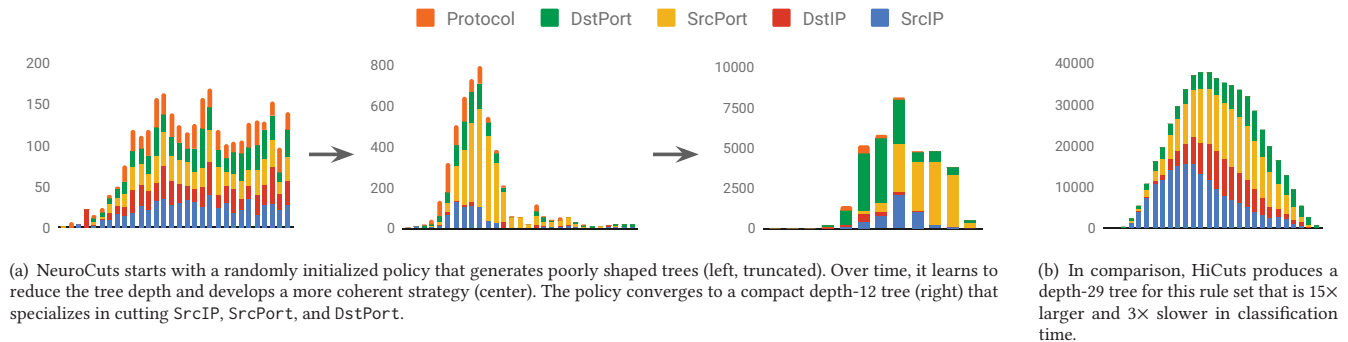
Figure 5: Visualization of NeuroCuts learning to split the fw5_1k ClassBench rule set. The x-axis denotes the tree level, and the y-axis the number of nodes at the level. The distribution of cut dimensions per level of the tree is shown in color.

handle this, we take the somewhat unusual step of only computing rewards for the rollout when the tree is completed, and setting $\gamma = 0$, effectively creating a series of 1-step decision problems similar to contextual bandits [24]. However, unlike the bandit setting, where an agent only makes a decision once per environment, these 1-step decisions are connected through the dynamics of the tree building process. For instance, this makes NeuroCuts amenable to techniques from the Deep RL literature such as GAE [42].

Another way of looking at the dense reward problem is that the process of building a decision tree is not really sequential but *tree-structured* (i.e., it is more accurately modeled as a branching decision process [8, 18, 39]), and we need to account for the reward calculations accordingly. In such a "branching" formulation, $\gamma > 0$, but the rewards of an action are computed as an aggregation over *multiple child states* produced by an action. For example, cutting a node produces multiple child sub-nodes, and the reward calculation may involve a sum or a min over each child's future rewards, depending on whether we are optimizing for tree size or depth. The 1-step decision problem and branching decision process formulations of NeuroCuts are roughly equivalent; in the implementation section we describe how we adapt standard RL algorithms to run NeuroCuts.

The final challenge is how to scale the solution to large packet classifiers. The decision tree for a packet classifier with 100K rules can have hundreds of thousands of nodes. The size of the tree impedes training along several dimensions. Not only does it take more steps to finish building a tree, but the execution time of each action increases as there are more rules to process. The space of trees to explore is also larger, requiring the use of larger network models and generating more rollouts to train.

State representation. One key observation is that the action on a tree node only depends on the node itself, so it is not necessary to encode the entire decision tree in the environment state. Our goal to optimize a global performance objective over the entire tree suggests that we would need to make decisions based on the global state. However, this does not mean that the state representation needs to encode the entire decision tree. Given a tree node, the action on that node only needs to make the best decision to optimize the sub-tree rooted at that node. It does not need to consider other tree nodes in the decision tree.
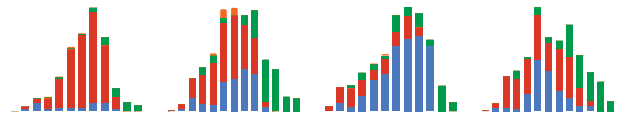

Figure 6: The NeuroCuts policy is stochastic, which enables it to effectively explore many different tree variations during training. Here we visualize four random tree variations drawn from a single policy trained on the acl4_1k Class-Bench rule set.

Formally, given tree node n, let t_n and s_n denote n's classification time and memory footprint, respectively, and T_n and S_n be the classification time and memory footprint of the entire sub-tree rooted at node n, respectively. Then, for a cut action, we have the following equations:

$$T_n = t_n + \max_{i \in children(n)} T_i \qquad (1)$$

$$S_n = s_n + \text{sum}_{i \in children(n)} S_i \qquad (2)$$

Similarly, for a partition action, we have as an upper bound on cost, assuming serial execution:

$$T_n = t_n + \text{sum}_{i \in children(n)} T_i \qquad (3)$$

$$S_n = s_n + \text{sum}_{i \in children(n)} S_i \qquad (4)$$

An action, a, taken on node n only needs to optimize the sub-tree rooted at n according to the following expression,

$$V_n = \text{argmax}_{a \in A} - (c \cdot T_n + (1 - c) \cdot S_n), \qquad (5)$$

where c is a coefficient capturing the tradeoff between classification time and memory footprint. The negation is needed since we want to *minimize* time and space complexities. We note that these values can be computed after the tree is fully built, regardless of the traversal order taken building the tree.

When $c \in \{0, 1\}$, it is easy to see that if at every tree node n we take the action that optimizes V_n, then, by induction, we end up optimizing V_r, where r is the root of the tree. In other words, we end up optimizing the global objective (reward) for the entire decision tree. For $0 < c < 1$ this optimization becomes approximate, but we find empirically that c can still be used to interpolate between the two objectives. It is important to note here that while the state

representation only encodes current node n, action a taken for node n is *not local*, as it optimizes the *entire sub-tree* rooted at n.

In summary, we only need to encode the current node as the input state of the agent. This is because the environment builds the tree node-by-node, node actions need only consider their own state, and each node contains a subset of the rules of its parent (i.e., rules contained in some subspace of its parent space). Therefore, nodes in the tree can be completely defined by the ranges they occupy in each dimension. Given d dimensions, we use $2d$ numbers to encode a tree node, which indicate the left and right boundaries of each dimension for this node. The state also needs to describe the partitioning at the node, which can be handled in a similar way. We note that the set of rules for the packet classifier are *not* present in the observation space. NeuroCuts learns to account for packet classifier rules implicitly through the rewards it gets from the environment. A full description of the NeuroCuts state and action representations can be found in Table 1.

Training algorithm. We use an actor-critic algorithm to train the agent's policy [19]. This class of algorithms have been shown to provide state-of-the-art results in many use cases [5, 35, 43], and can be easily scaled to the distributed setting [7]. We also experimented with Q-learning [37] based approaches, but found they did not perform as well.

Algorithm 1 shows the pseudocode of the NeuroCuts algorithm, which executes as follows. NeuroCuts starts with the root node of the decision tree, s^*. The end goal is to learn an optimized stochastic policy function $\pi(a|\mathbf{s}; \theta)$ (i.e., the actor). NeuroCuts first initializes all the parameters (line 1-6), and then runs for N rollouts to train the policy and the value function (line 7-23). After each rollout, it reinitializes the decision tree to the root node (line 9). It then incrementally builds the tree by repeatedly selecting and applying an action on each non-terminal leaf node (line 11-13) according to the current policy. A terminal leaf node is a node in which the number of rules is below a given threshold.

More specifically, NeuroCuts traverses the tree nodes in depth-first-search (DFS) order (line 13), i.e., it recursively cuts the child of the current node until the node becomes a terminal leaf. Note that the DFS order is not essential. It is used to give a way for the agent to find a tree node to cut. Other orders, such as the breadth-first-search (BFS), can be used as well. After the decision tree is built, the gradients are reset (line 14), and then the algorithm iterates over all the tree nodes to aggregate the gradients (line 15-21). Finally, NeuroCuts uses the gradients to update the parameters of the actor and critic networks (line 22), and proceeds to the next rollout (line 23).

The first gradient computation (line 19) corresponds to that for the *policy gradient loss*. This loss defines the direction to update θ to improve the expected reward. An estimation of the state value $V(s; \theta_v)$ is subtracted from the rollout reward R to reduce the gradient variance [19]. V is trained concurrently to minimize its prediction error (line 21). Figure 5 visualizes the learning process of NeuroCuts to build a decision tree. The NeuroCuts policy is stochastic, enabling it to effectively explore many different tree variations during training, as illustrated in Figure 6.

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:
1: // **Initialization**
2: Randomly initialize the model parameters θ, θ_v
3: Maximum number of rollouts N
4: Coefficient $c \in [0, 1]$ that trades off classification time vs. space
5: Reward scaling function $f(x) \in \{x, \text{LOG}(x)\}$
6: $n \leftarrow 0$
7: // **Training**
8: **while** $n < N$ **do**
9:     $s \leftarrow \text{RESET}(s^*)$
10:     // **Build a tree using the current policy**
11:     **while** $s \neq \text{NULL}$ **do**
12:         $a \leftarrow \pi(a|s; \theta)$
13:         $s \leftarrow \text{GROWTREEDFS}(s, a)$
14:     Reset gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
15:     **for** $(s, a) \in \text{TREEITERATOR}(s^*)$ **do**
16:         // **Compute the future rewards for the given action**
17:         $R \leftarrow -(c \cdot f(\text{TIME}(s)) + (1 - c) \cdot f(\text{SPACE}(s)))$
18:         // **Accumulate gradients wrt. policy gradient loss**
19:         $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a|s; \theta)(R - V(s; \theta_v))$
20:         // **Accumulate gradients wrt. value function loss**
21:         $d\theta_v \leftarrow d\theta_v + \partial(R - V(s; \theta_v))^2/\partial\theta_v$
22:     Perform update of θ using d_θ and θ_v using $d\theta_v$.
23:     $n \leftarrow n + 1$

Subroutines:
- RESET(s): Reset the tree s to its initial state.
- 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.
- TREEITERATOR(s): Non-terminal tree nodes of the subtree s and their taken action.
- TIME(s): Upper-bound on classification time to query the subtree s. In non-partitioned trees this is simply the depth of the tree.
- SPACE(s): Memory consumption of the subtree s.

Incorporating existing heuristics. NeuroCuts can easily incorporate additional heuristics to improve the decision trees it learns. One example is adding rule partition actions. In addition to the cut action, in our NeuroCuts implementation we also allow two types of partition actions:

  (1) **Simple**: the current node is partitioned along a single dimension using a learned threshold.
  (2) **EffiCuts**: the current node is partitioned using the EffiCuts partition heuristic [55].

Scaling out to handle large packet classifiers. The pseudocode in Algorithm 1 is for a single-threaded implementation of NeuroCuts. This is sufficient for small classifiers. But for large classifiers with tens or hundreds of thousands of rules, parallelism can significantly improve the speed of training. In Figure 7 we show how Algorithm 1 can be adapted to build multiple decision trees in parallel.

Handling classifier updates. Packet classifiers are often updated by network operators based on application requirements, e.g., adding access control rules for new devices. For small updates of only a few rules, NeuroCuts modifies the existing decision tree to reflect the changes. New rules are added to the decision tree according to the

Action Space	Tuple(Discrete(NumDims), Discrete(NumCutActions + NumPartitionActions))
Observation Space	Box(low=0, high=1, shape=(278,))
Observation Components	$\big[$BinaryString($Range_{min}^{dim}$) + BinaryString($Range_{max}^{dim}$) + OneHot($Partition_{min}^{dim}$) + OneHot($Partition_{max}^{dim}$)$\big]$ $\forall dim \in \{SrcIP, DstIP, SrcPort, DstPort, Protocol\}$ + OneHot(EffiCutsPartitionID) + ActionMask

Table 1: NeuroCuts action and observation spaces described in OpenAI Gym format [2]. Actions are sampled from two categorical distributions that select the dimension and action to perform on the dimension respectively. Observations are encoded in a one-hot bit vector (278 bits in total) that describes the node ranges, partitioning info, and action mask (i.e., for prohibiting partitioning actions at lower levels). When not using the EffiCuts partitioner, the $Partition^{dim}$ rule dimension coverage thresholds are set to one of the following discrete levels: 0%, 2%, 4%, 8%, 16%, 32%, 64%, and 100%.

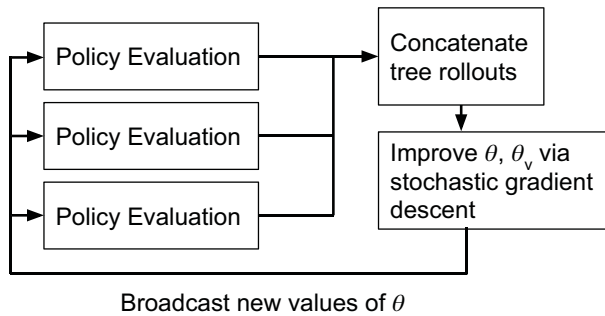Broadcast new values of θ

Figure 7: NeuroCuts can be parallelized by generating decision trees in parallel from the current policy.

existing structure; deleted rules are removed from the terminal leaf nodes. When enough small updates accumulate or a large update is made to the classifier, NeuroCuts re-runs training.

5 IMPLEMENTATION

Deep RL algorithms are notoriously difficult to reproduce [15]. For a practical implementation, we prioritize the ability to (*i*) leverage off-the-shelf RL algorithms, and (*ii*) easily scale NeuroCuts to enable parallel training of policies.

Decision tree implementation. We implement the decision tree data structure for NeuroCuts in Python for ease of development. To ensure minor implementation differences do not bias our results, we use this same data structure to implement each baseline algorithm (e.g., HiCuts, EffiCuts, etc.), as well as to implement NeuroCuts.

Branching decision process environment. As discussed in Section 4, the branching structure of the NeuroCuts environment poses a challenge due to its mismatch with the MDP formulation assumed by many RL algorithms. A typical RL environment defines a transition function $P_a(s_{t+1}|s_t)$ and a reward function $R_a(s, s')$. The first difference is that the state transition function in NeuroCuts returns multiple child states, instead of a single state., i.e., $(s_t, a_t) \rightarrow \{s_{t+1}^0, ..., s_{t+1}^k\}$. Second, the final reward for NeuroCuts is computed by aggregating across the rewards of child states. More precisely, for the cut action we use max aggregation for classification time and sum aggregation for memory footprint. For the partition action, we use sum aggregation for both metrics.

The recursive dependence of the NeuroCuts reward calculation on all descendent state actions means that it is difficult to flatten

the tree structure of the environment into a single MDP, which is required by existing off-the-shelf RL algorithms. Rather than attempting to flatten the NeuroCuts environment, our solution is to instead treat the NeuroCuts environment as a series of *independent* 1-step decision problems, each of which yields an "immediate" reward. The actual reward for these 1-step decisions is calculated once the relevant sub-tree rollout is complete.

For example, consider a NeuroCuts tree rollout from a root node s_1. Based on π_θ the agent decides to take action a_1 to split s_1 into s_2, s_3, and s_4. Of these child nodes, only s_4 needs to be further split (via a_2), into s_5 and s_6, which finishes the tree. The experiences collected from this rollout consist of two independent 1-step rollouts: (s_1, a_1) and (s_4, a_2). Taking the time-space coefficient $c = 1$ and discount factor $\gamma = 1$ for simplicity, the total reward R for each rollout would be $R = 2$ and $R = 1$ respectively.

Multi-agent implementation. Since these 1-step decisions are logically independent of each other, NeuroCuts execution can be realized as a multi-agent environment, where each node's 1-decision problem is taken by an independent "agent" in the environment. Since we want to learn a single policy, π_θ, for all states, the agents must be configured to share the same underlying stochastic neural network policy. This ensures all experiences go towards optimizing the single shared policy π_θ. When using an actor-critic algorithm to optimize the policies of such agents, the relevant loss calculations induced by this multi-agent realization are identical to those presented in Algorithm 1.

There are several ways to implement the 1-step formulation of NeuroCuts while leveraging off-the-shelf RL libraries. In Algorithm 1 we show standalone single-threaded pseudocode assuming a simple actor-critic algorithm is used. In our experiments, we use the multi-agent API provided by Ray RLlib [30], which implements parallel simulation and optimization of such RL environments.

Performance. We found that NeuroCuts often converges to its optimal solution within just a few hundred rollouts. The size of the rule set does not significantly affect the number of rollouts needed for convergence, but affects the running time of each rollout. For smaller problems (e.g., 1000 rules), this may be within a few minutes of CPU time. The computational overhead for larger problem scales with the size of the classifier, i.e., linearly with the number of rules that must be scanned per action taken to grow the tree. The bulk of time in NeuroCuts is spent executing tree cut actions. This is largely an artifact of our Python implementation, which iterates

over each rule present in a node on each cut action. An optimized C++ implementation of the decision tree would further reduce the training time.

5.1 Optimizations

Rollout truncation. During the initial phase of learning, the unoptimized policy will create excessively large trees. Since NeuroCuts does not start learning until a tree is complete, it is necessary to truncate rollouts to speed up the initial phase of training. For larger classifiers, we found it necessary to allow rollouts of up to 15000 actions in length.

Depth truncation. Since valid solutions never involve trees of depth greater than a few hundred, we also truncate trees once they reach a certain depth. In our experience, depth truncation is only a factor early on in learning; NeuroCuts quickly learns to avoid creating very deep trees.

Proximal Policy Optimization. For better stability and more sample-efficient learning, in our experiments we choose to use Proximal Policy Optimization (PPO) [43]. PPO implements an actor-critic style loss with entropy regularization and a clipped surrogate objective, which enables improved exploration and sample efficiency. We report the PPO hyperparameters we used in Table 2. It is important to note however that this particular choice of RL algorithm is not fundamental to NeuroCuts.

6 EVALUATION

In the evaluation, we seek to answer the following questions:

(1) How does NeuroCuts compare to the state-of-the-art approaches in terms of classification time and memory footprint? (Section 6.1 and 6.2)

(2) Beyond *tabula rasa* learning, can NeuroCuts effectively incorporate and improve upon pre-engineered heuristics? (Section 6.3)

(3) How sensitive is NeuroCuts to the hyperparameters of the neural network architecture (Section 6.4), and the time-space coefficient c (Section 6.5)?

For the results presented in the next sections, we evaluated NeuroCuts within the space of hyperparameters shown in Table 2. We did not otherwise perform extensive hyperparameter tuning; in fact we use close to the default hyperparameter configuration of the PPO algorithm. The notable hyperparameters we swept over include:

- Allowed top-node partitioning (none, simple, and the EffiCuts heuristic), which strongly biases NeuroCuts towards learning trees optimized for time (none) vs space (EffiCuts), or somewhere in the middle (simple).

- The max number of timesteps allowed per rollout before truncation. It must be large enough to enable solving the problem, but not so large that it slows down the initial phase of training.

- We also experimented with values for the time-space tradeoff coefficient $c \in \{0, 0.1, 0.5, 1\}$. When $c < 1$, we used LOG(x) as the reward scaling function to simplify the combining of the time and space rewards.

We ran NeuroCuts on m4.16xl AWS machines, with four CPU cores used per NeuroCuts instance to speed up the experiment. Because the neural network model and data sizes produced by NeuroCuts are quite small (e.g., in contrast to image observations from Atari games), the use of GPUs is not necessary. Our main training bottleneck was the Python implementation of the decision tree. We ran each NeuroCuts instance for up to 10 million timesteps (i.e., up to a couple thousand generated trees in total), or until convergence.

We compare NeuroCuts with four hand-tuned algorithms: HiCuts [13], HyperCuts [47], EffiCuts [55], and CutSplit [29]. We use the standard benchmark, ClassBench [52], to generate packet classifiers with different characteristics and sizes. The benchmark metrics are those from prior work: classification time (tree depth) and memory footprint (bytes per rule). Since we use the same underlying tree data structure for all algorithms, a lesser depth virtually guarantees more efficient traversal, and the same is true for memory footprint.

We find that NeuroCuts significantly improves over all baselines in classification time while also generating significantly more compact trees. NeuroCuts is also competitive when optimizing for memory, with a 25% median space improvement over EffiCuts without compromising in time.

6.1 Time-Optimized NeuroCuts

In Figure 8, we compare the best time-optimized trees generated by NeuroCuts against HiCuts, HyperCuts, EffiCuts, and CutSplit in the ClassBench classifiers. NeuroCuts provides a 20%, 38%, 52% and 56% median improvement over HiCuts, HyperCuts, EffiCuts, and CutSplit respectively. NeuroCuts also does better than the minimum of all baselines in 70% of the cases, with a median all-baseline improvement of 18%, average improvement of 12%, and best-case improvement of 58%. These time-optimized trees generally correspond to NeuroCuts runs with either no partitioning action or the simple top-node partitioning action.

6.2 Space-Optimized NeuroCuts

We again compare NeuroCuts against the baselines in Figure 9, this time selecting the most space-optimized trees and comparing the memory footprint (bytes per rule). As expected, NeuroCuts does significantly better than HiCuts and HyperCuts since it can learn to leverage the partition action. NeuroCut's space-optimized trees show a 40% median and 44% mean improvement over EffiCuts. In our experiments NeuroCuts does not usually outperform CutSplit in memory footprint, with a 26% higher median memory usage compared to CutSplit, though the best case improvement is still 3× (66%) *over all baselines*.

Separately, we also note that the memory footprints of the best *time-optimized trees* generated by NeuroCuts are significantly lower than those generated by HiCuts and HyperCuts, with a >100× *median* space improvement along with the better classification times reported in Section 6.1. However, these time-optimized trees are not competitive in space with the space-optimized NeuroCuts, EffiCuts and CutSplit trees.
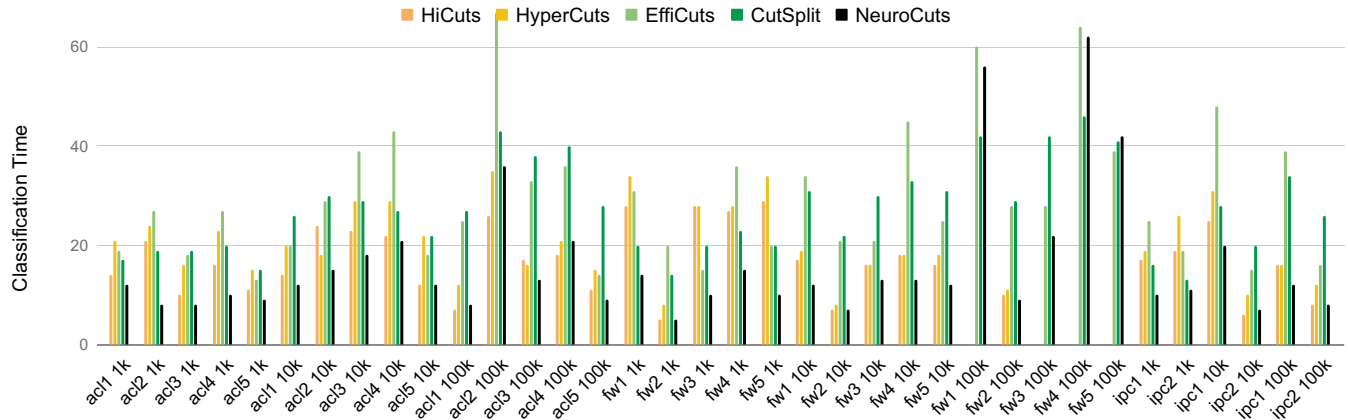
Figure 8: Classification time (tree depth) for HiCuts, HyperCuts, EffiCuts, and NeuroCuts (time-optimized). We omit four entries for HiCuts and HyperCuts that did not complete after more than 24 hours.
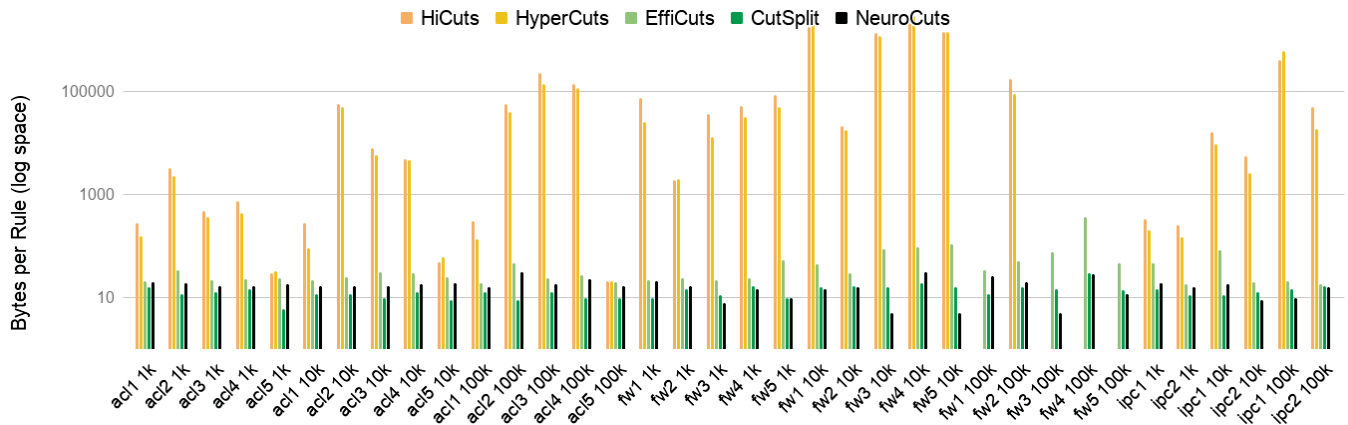

Figure 9: Memory footprint (bytes per rule) used for HiCuts, HyperCuts, EffiCuts, and NeuroCuts (space-optimized). We omit four entries for HiCuts and HyperCuts that did not complete after more than 24 hours.

6.3 Improving on EffiCuts

In Figure 10 we examine a set of 36 NeuroCuts trees (one tree for each ClassBench classifier) generated by NeuroCuts with the Effi-Cuts partition action. This is in contrast with the prior experiments that selected trees optimized for either space or time alone. On this 36-tree set, there is a median space improvement of 29% relative to EffiCuts; median classification time is about the same. This shows that NeuroCuts is able to effectively incorporate and improve on pre-engineered heuristics such as the EffiCuts top-level partition function.

Surprisingly, NeuroCuts is able to outperform EffiCuts despite the fact that NeuroCuts does not use multi-dimensional cut actions. When we evaluate EffiCuts with these cut types disabled, the memory advantage of NeuroCuts widens to 67% at the median. This suggests that NeuroCuts could further improve its performance if we also incorporate multi-dimensional cut actions via parametric action encoding techniques [9]. It would also be interesting to, besides adding actions to NeuroCuts, consider postprocessing

steps such as resampling that can be used to further improve the stochastic policy output.

6.4 Neural Network Architecture

To better understand the influence of the neural network architecture on NeuroCuts performance, we conduct an ablation study where the network size is reduced from 512x512 (hundreds of thousands of parameters) all the way down to 16x16 (a couple hundred parameters). We also consider the case where the network is trivial and does not process the observation at all, similar to a non-contextual bandit. For this study we run a single sweep across only these architecture hyperparameters, keeping all the others fixed, and use the simple partition method.

The results are shown in Figure 11. We observe that while the larger 64x64 network consistently outperforms 16x16, at 512x512 performance starts to be impacted due to the larger number of
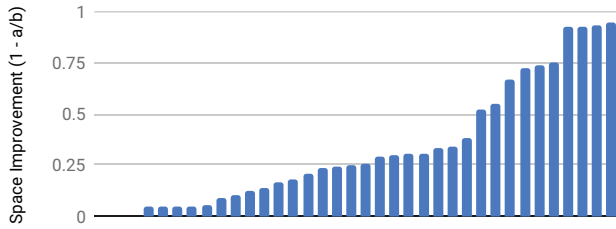
(a) NeuroCuts can build on the EffiCuts partitioner to generate trees up to 10× (90%) more space efficient than EffiCuts. In this experiment NeuroCuts did as well or better than EffiCuts on all 36 rule sets.
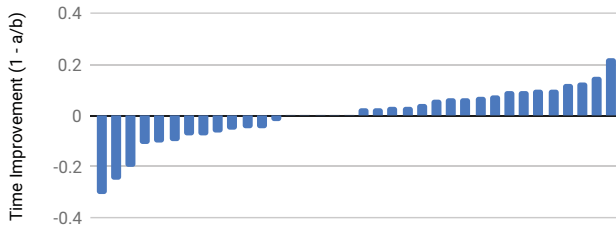
(b) NeuroCuts with the EffiCuts partitioner generates trees with about the same time efficiency as EffiCuts.

Figure 10: Sorted rankings of NeuroCuts' improvement over EffiCuts in the ClassBench benchmark. Here NeuroCuts is run with only the EffiCuts partition method allowed. Positive values indicate improvements.

learnable parameters.[3] Interestingly, while the bias-only network did the worst, it still was able to generate reasonably compact trees in many cases. This suggests that NeuroCuts may operate by first learning a random distribution of actions that leads to a basic solution, and then leveraging the capacity of its neural network to specialize the action distribution to different portions of the rule space.

6.5 Tuning Time vs Space

Finally, in Figure 12 we sweep across a range of values of c for NeuroCuts with the simple partition method and LOG(x) reward scaling. We plot the ClassBench median of the best classification times and bytes per rule found for each classifier. We find that classification time improves by 2× as $c \to 1$, while the number of bytes per rule improves 2× as $c \to 0$. This shows that c is effective in controlling the tradeoff between space and time.

7 RELATED WORK

Packet classification. Packet classification is a long-standing problem in computer networking. Decision-tree based algorithms are a major class of algorithmic solutions. Existing solutions rely on hand-tuned heuristics to build decision trees. HiCuts [13] is a pioneering work in this space. It cuts the space of each node in one dimension to create multiple equal-sized subspaces to separate rules. HyperCuts [47] extends HiCuts by allowing cutting in multiple dimensions at each node. HyperSplit [40] combines the advantages

---

[3]We note that these results might not hold for different hyperparameters, e.g., if allowed longer training periods, larger networks may dominate.
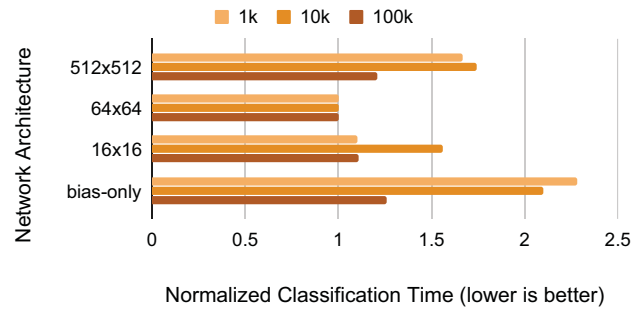

Figure 11: Comparison of the mean best classification time achieved by NeuroCuts across different network architectures and groups of classifiers. The *bias-only* architecture refers to a trivial neural network that does not process the observation at all and emits a fixed action probability distribution (i.e., a pure bandit). Results are normalized within classifier groups so that the best tree has a normalized time of 1. Rulesets that did not converge to a valid tree were assigned a time of 100 prior to normalization.
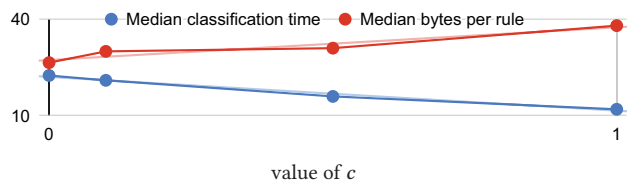

Figure 12: The classification time improves by 2× as the time-space coefficient $c \to 1$, and conversely, number of bytes per rule improves 2× as $c \to 0$.

of rule-based space decomposition and local-optimized recursion to guarantee worst-case classification time and reduce memory footprint. EffiCuts [55] introduces four heuristics, including separable trees, tree merging, equal-dense cuts and node co-location, to reduce rule replication and imbalance cutting. CutSplit [29] integrates equal-sized cutting and equal-dense cutting to optimize decision trees. Besides decision-tree based algorithms, there are also other algorithms proposed for packet classification, such as tuple space search [49], RFC [12] and DCFL [53]. These algorithms are not as popular as decision-tree based algorithms, because they are either too slow or consume too much memory. There are also solutions that exploit specialized hardware such as TCAMs, GPUs and FPGAs to support packet classification [17, 23, 31, 32, 41, 48, 50, 57]. Compared to existing work, NeuroCuts is an algorithmic solution that applies Deep RL to generate efficient decision trees, with the capability to incorporate and improve on existing heuristics as needed.

Decision trees for machine learning. There have been several proposals to use deep learning to optimize the performance of decision trees for machine learning problems [20, 38, 59]. In these settings, the objective is maximizing test accuracy. In contrast, packet classification decision trees provide perfect accuracy by

construction, and the objective is minimizing classification time and memory usage.

Structured data in deep learning. There have many recent proposals towards applying deep learning to process and generate tree and graph data structures [11, 58, 60, 61, 63, 66]. NeuroCuts sidesteps the need to explicitly process graphs, instead exploiting the structure of the problem to encode agent state into a compact fixed-length representation.

Deep reinforcement learning. Deep RL leverages the modeling capacity of deep neural networks to extend classical RL to domains with large, high-dimensional state and action spaces. DQN [36, 37, 56] is one of the earliest successes of Deep RL, and shows how to learn control policies from high-dimensional sensory inputs and achieve human-level performance in Atari 2600 games. A3C, PPO, and IMPALA [7, 35, 43] scale actor-critic algorithms to leverage many parallel workers. AlphaGo [44], AlphaGo Zero [46] and AlphaZero [45] show that Deep RL algorithms can achieve superhuman performance in many challenging games like Go, chess and shogi. Deep RL has also been applied to many other domains like natural language processing [28] and robotics [25–27]. NeuroCuts works in a discrete environment and applies Deep RL to learn decision trees for packet classification.

Deep learning for networking and systems. Recently there has been an uptake in applying deep learning to networking and systems problems [4, 6, 16, 33, 34, 54, 62, 64, 65]. NAS [62] utilizes client computation and deep neural networks to improve the video quality independent to the available bandwidth. Pensieve [34] generates adaptive bitrate algorithms using Deep RL without relying on pre-programmed models or assumptions about the environment. Valadarsky *et al.* [54] applies Deep RL to learn network routing. Chinchali *et al.* [4] uses Deep RL for traffic scheduling in cellular networks. AuTO [3] scales Deep RL for datacenter-scale traffic optimization. There are also many solutions that apply deep reinforcement learning to congestion control [6, 16, 64] and resource management [33]. We explore the application of Deep RL to packet classification, and propose a new algorithm to learn decision trees with succinct encoding and scalable training mechanisms.

8 CONCLUSION

We present NeuroCuts, a simple and effective Deep RL formulation of the packet classification problem. NeuroCuts provides significant improvements on classification time and memory footprint compared to state-of-the-art algorithms. It can easily incorporate pre-engineered heuristics to leverage their domain knowledge, optimize for flexible objectives, and generates decision trees which are easy to test and deploy in any environment.

We hope NeuroCuts can inspire a new generation of learning-based algorithms for packet classification. As a concrete example, NeuroCuts currently optimizes for the worst-case classification time or memory footprint. By considering a specific traffic pattern, NeuroCuts can be extended to other objectives such as average classification time. This would allow NeuroCuts to not only optimize for a specific classifier but also for a specific traffic pattern in a given deployment.

9 ACKNOWLEDGEMENTS

This work does not raise any ethical issues.

Hyperparameter	Value
Time-space coefficient c	<set by user>
Top-node partitioning	{none, simple, EffiCuts}
Reward scaling function f	{x, LOG(x)}
Max timesteps per rollout	{1000, 5000, 15000}
Max tree depth	{100, 500, inf}
Max timesteps to train	10000000
Max timesteps per batch	60000
Model type	fully-connected
Model nonlinearity	tanh
Model hidden layers	{256x256, 512x512}
Weight sharing between θ, θ_v	{true, false}
Learning rate	0.00005
Discount factor γ	1.0
PPO entropy coefficient	0.01
PPO clip param	0.3
PPO VF clip param	10.0
PPO KL target	0.01
SGD iterations per batch	30
SGD minibatch size	1000

Table 2: NeuroCuts hyperparameters. Values in curly braces denote a space of values searched over during evaluation. We found that the most sensitive hyperparameter is the top-node partitioning, which greatly affects the structure of the search problem. It is also important to ensure that the rollout timestep limit and model used are sufficiently large for the problem.

REFERENCES

[1] Florin Baboescu, Sumeet Singh, and George Varghese. 2003. Packet classification for core routers: Is there an alternative to CAMs?. In *IEEE INFOCOM*.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI gym. *arXiv preprint arXiv:1606.01540* (2016).

[3] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In *ACM SIGCOMM*.

[4] Sandeep Chinchali, Pan Hu, Tianshu Chu, Manu Sharma, Manu Bansal, Rakesh Misra, Marco Pavone, and Sachin Katti. 2018. Cellular network traffic scheduling with deep reinforcement learning. In *AAAI*.

[5] Ekin Dogus Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. 2018. AutoAugment: Learning Augmentation Policies from Data. *CoRR* (2018).

[6] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *USENIX NSDI*.

[7] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Legg Shane, and Kavukcuoglu Koray. 2018. IMPALA: Scalable distributed Deep-RL with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561* (2018).

[8] Kousha Etessami and Mihalis Yannakakis. 2005. Recursive Markov decision processes and recursive stochastic games. In *International Colloquium on Automata, Languages, and Programming*. Springer, 891–903.

[9] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, and Xiaohui Ye. 2018. Horizon: Facebook's Open Source Applied Reinforcement Learning Platform. *arXiv preprint arXiv:1811.00260* (2018).

[10] Alex Graves, Abdel-Rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *ICASSP*.

[11] Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. 2018. Learning to search with MCTSnets. *arXiv preprint arXiv:1802.04697* (2018).

[12] Pankaj Gupta and Nick McKeown. 1999. Packet classification on multiple fields. *SIGCOMM CCR* (1999).

[13] Pankaj Gupta and Nick McKeown. 1999. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects*.

[14] Pankaj Gupta and Nick McKeown. 2001. Algorithms for packet classification. (2001).

[15] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2017. Deep Reinforcement Learning that Matters. *CoRR* (2017).

[16] Nathan Jay, Noga H. Rotman, P. Godfrey, Michael Schapira, and Aviv Tamar. 2018. Internet Congestion Control via Deep Reinforcement Learning. *arXiv preprint arXiv:1810.03259* (2018).

[17] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. 2014. SAX-PAC (scalable and expressive packet classification). In *SIGCOMM CCR*.

[18] AN Kolmogorov and NA Dmitriev. 1947. Stochastic branching processes. In *Doklady Akademi Nauk SSSR*, Vol. 56. 7–10.

[19] Vijay R. Konda and John N. Tsitsiklis. 2000. Actor-critic algorithms. In *Advances in neural information processing systems*.

[20] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo. 2015. Deep Neural Decision Forests. In *The IEEE International Conference on Computer Vision (ICCV)*.

[21] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

[23] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. 2005. Algorithms for advanced packet classification with ternary CAMs. In *SIGCOMM CCR*.

[24] John Langford and Tong Zhang. 2008. The epoch-greedy algorithm for multi-armed bandits with side information. In *Advances in neural information processing systems*. 817–824.

[25] Ian Lenz, Honglak Lee, and Ashutosh Saxena. 2015. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research* (2015).

[26] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research* (2016).

[27] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. 2018. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research* (2018).

[28] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. 2016. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541* (2016).

[29] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. 2018. CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification. In *IEEE INFOCOM*.

[30] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for distributed reinforcement learning. In *ICML*.

[31] Alex X Liu, Chad R Meiners, and Yun Zhou. 2008. All-match based complete redundancy removal for packet classifiers in TCAMs. In *IEEE INFOCOM*.

[32] Yadi Ma and Suman Banerjee. 2012. A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification. In *ACM SIGCOMM*.

[33] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *ACM SIGCOMM HotNets Workshop*.

[34] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM*.

[35] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *ICML*.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[37] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* (2015).

[38] Mohammad Norouzi, Maxwell Collins, Matthew A. Johnson, David J. Fleet, and Pushmeet Kohli. 2015. Efficient non-greedy optimization of decision trees. In *Advances in Neural Information Processing Systems*. 1729–1737.

[39] Stanley R Pliska. 1976. Optimization of multitype branching processes. *Management Science* 23, 2 (1976), 117–124.

[40] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. 2009. Packet classification algorithms: From theory to practice. In *IEEE INFOCOM*.

[41] Yun R. Qu, Hao H. Zhang, Shijie Zhou, and Viktor K. Prasanna. 2015. Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*.

[42] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438* (2015).

[43] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[44] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Graepel Thore, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* (2016).

[45] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* (2018).

[46] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* (2017).

[47] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet classification using multidimensional cutting. In *ACM SIGCOMM*.

[48] Ed Spitznagel, David Taylor, and Jonathan Turner. 2003. Packet classification using extended TCAMs. In *IEEE ICNP*.

[49] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet classification using tuple space search. In *SIGCOMM CCR*.

[50] Weibin Sun and Robert Ricci. 2013. Fast and flexible: parallel packet processing with GPUs and click. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*.

[51] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*.

[52] David E. Taylor and Jonathan S Turner. 2005. ClassBench: A packet classification benchmark. In *IEEE INFOCOM*.

[53] David E. Taylor and Jonathan S Turner. 2005. Scalable packet classification using distributed crossproducing of field labels. In *IEEE INFOCOM*.

[54] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. 2017. Learning to route with Deep RL. In *NIPS Deep Reinforcement Learning Symposium*.

[55] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2010. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *ACM SIGCOMM*.

[56] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *AAAI*.

[57] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and TV Lakshman. 2016. Multi-layer packet classification with graphics processing units. *IEEE/ACM Transactions on Networking* (2016).

[58] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge Graph Embedding by Translating on Hyperplanes.. In *AAAI*.

[59] Zheng Xiong, Wenpeng Zhang, and Wenwu Zhu. 2017. Learning decision trees with reinforcement learning. In *NIPS Workshop on Meta-Learning*.

[60] Shuicheng Yan, Dong Xu, Benyu Zhang, Hong-Jiang Zhang, Qiang Yang, and Stephen Lin. 2007. Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE transactions on pattern analysis and machine intelligence* (2007).

[61] Jianchao Yang, Shuicheng Yang, Yun Fu, Xuelong Li, and Thomas Huang. 2008. Non-negative graph embedding. In *CVPR*.

[62] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. 2018. Neural adaptive content-aware internet video delivery. In *USENIX OSDI*.

[63] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. 2018. Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation. *arXiv preprint arXiv:1806.02473* (2018).

[64] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. 2015. Adaptive congestion control for unpredictable cellular networks. In *SIGCOMM CCR*.

[65] Ying Zheng, Ziyu Liu, Xinyu You, Yuedong Xu, and Junchen Jiang. 2018. Demystifying Deep Learning in Networking. In *ACM SIGCOMM APNet Workshop*.

[66] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2018. Graph Neural Networks: A Review of Methods and Applications. *arXiv preprint arXiv:1812.08434* (2018).