KNOSSOS: COMPILING AI WITH AI

Anonymous authors
Paper under double-blind review

ABSTRACT

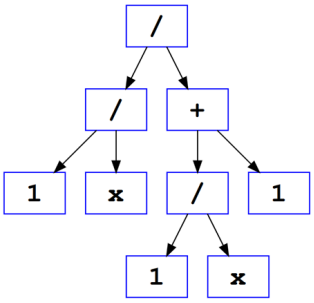
Deep learning frameworks have enabled huge productivity gains by providing a set of abstractions for common tasks, such automatic differentiation and tensor operations. However, the current generation of tools such as PyTorch and TensorFlow relies on custom back-ends in order to achieve efficiency. These back-ends are both architecture-dependent and only work fast on data represented as tensors. Since deep learning workloads are notoriously compute-intensive, this makes it impractical to use them on less common hardware where no such back-ends exist. The Knossos complier achieves efficiency in an architecture-agnostic way by directly optimising the computation graph. Since our optimisation objective is driven by an arbitrary cost model, we produce code tailored to any architecture. The Knossos compiler has minimal dependencies and can be used on any architecture that supports a C++ toolchain. We demonstrate that Knossos outperforms best-effort human code optimisation for both training and inference on a suite of representative ML tasks, including training convolutional networks and probabilistic models.

1 INTRODUCTION

The development of machine learning (ML) software places demands on programming tools, which are unique in two ways. First, since machine learning workloads can take weeks to train (OpenAI, 2018), they need to make maximum use of modern hardware. Second, they have to support the definition of differentiable programs, which is the cornerstone of modern ML methods. Frameworks such as PyTorch (Paszke et al., 2017) and TensorFlow (Abadi et al., 2016) have enabled huge productivity gains by providing abstractions for automatic differentiation and common tensor operations. However, these tools have three major shortcomings.

First, major existing frameworks (Paszke et al., 2017; Abadi et al., 2016) require hardware-specific back-ends such as XLA (XLA authors, 2016) to achieve reasonable performance, making it difficult to use them on new or embedded hardware, or in other settings where no such back-ends exist. This contradicts the drive to make ML ubiquitous, which requires support of very diverse hardware, ranging from low-powered microcontrollers to powerful GPUs. While recent versions of existing frameworks allow the export of models into a format consumable by a pure C++ toolchain, fast training of large models still requires proprietary back-ends. Second, established frameworks require the user to specify the machine learning algorithm by defining a computation graph. This is currently done in one of two ways. In the explicit meta-programming approach (Abadi et al., 2016), the computation graph is represented as an object in a high-level language but does not share its semantics. In the operator overloading approach (Paszke et al., 2017), graph assembly is automated for a subset of the constructs of the high-level language. Both of them are problematic from a software engineering standpoint because the programmer has to simultaneously track the semantics of the host language and the specified computation graph, which do not fully overlap. Third, specifying models that train fast requires careful manual tuning. Among many possible computation graphs with the same functional behaviour, the programmer has manually find the one that allows for the quickest execution on her hardware. This again contradicts modern software-engineering practice, which stresses a high-level declarative programming style over hand-coded optimisations.

In order to address these issues, a new generation of tools has recently appeared that treat the transformation of computation graphs as a compilation task (van Merrienboer et al., 2018; Wei et al., 2018; Sotoudeh et al., 2019; Rotem et al., 2018). Akin to traditional compliers, these tools use an intermediate representation to perform optimisations. In addition, program optimisation is being



(a) Sample arithmetic expression tree (MDP state). The tree corresponds to $\frac{1/x}{1/x+1}$

```
(rule (+ 0 e) e)
(rule (* 1 e) e)
(rule (/ (* a b) c)
      (* (/ a c) b))
(rule (* (exp a) (exp b))
      (exp (+ a b)))
(rule (if p a a) a))
(rule
   (apply (lam x body) arg)
   (let x arg body))
```

(b) Example re-write rules. Each rule defines a class of MDP transitions. See section 3.1 for a discussion of guards and side conditions.

Figure 1: The Knossos MDP.

modelled as a machine learning task itself, with the complier learning how to perform re-writes (Chen et al., 2018b;a). In a parallel line of work, support for automatic differentiation is being integrated into as a first-class feature of programming languages (Innes et al., 2019; Frostig et al., 2018).

Knossos expands on this line of work. The Knossos system includes a compiler which combines efficient program optimisation with an intermediate representation (IR) designed with machine learning in mind. We formalize program optimisation as a finite-horizon Markov Decision Process (MDP), with the reward signal determined by the cost of executing a program. By solving this MDP, we are able to produce code tailor-made for any given task and architecture, without relying on backends. In addition, akin to JAX (Frostig et al., 2018) and Zygote (Innes et al., 2019) Knossos avoids the syntactic awkwardness that arises from embedding a differentiable program in a host language. Instead, the programmer writes her code as usual in a language such as F# or Julia without worrying about differentiation. The code can then be transpiled near-losslessly to the Knossos IR, where *all* functions are potentially differentiable. This enables faster prototyping and enhances developer productivity. However, since our framework works on any platform that supports a C++ toolchain, Knossos code can be seamlessly deployed on specialized or embedded hardware without the need of manual tuning, both for training and for deployment of models, enabling a much broader user base than competing approaches.

To our knowledge, Knossos is the first compiler that combines RL-based program optimisation, first-class support for deep learning primitives and the ability to target any architecture supporting the C++ toolchain. We defer detailed scope comparisons with prior work to Section 7. We empirically demonstrate the benefits of our program optimisation in Section 8.

2 Code Optimisation as a Reinforcement Learning Problem

We model code optimisation as a finite-horizon Markov Decision Process (MDP). An MDP is defined (Puterman, 2014; Sutton & Barto, 2018) as a tuple (S, A, T, R, H, p_0), where S denotes the state space, A denotes the action space, T denotes the transition dynamics, R denotes the rewards, H is the maximum time budget allowed to solve the problem (the horizon) and p_0 is a fixed probability distribution over initial states. We provide a detailed description of the states, transitions and rewards later on this section.

Policies and Value Functions The RL agent maintains a policy $\pi(a|s,t)$, which defines the probability of taking an action in state s given there are t steps remaining till the total time budget is exhausted. A policy π generates rollouts τ_π. A rollout τ_π is defined as a sequence of states, actions and rewards obtained from the MDP $\tau_\pi = (s_1, a_1, r_1, s_2, a_2, r_2, \dots s_H, r_H)$. Since the policy π can be stochastic, it is modelled as a random variable. The goal of RL agent is to find an optimal policy $\pi^\star = \arg\max_\pi J_\pi$, which attains the best possible per-step return. The return is defined

as $J_\pi = \frac{1}{H} \mathrm{E}_{\tau_\pi} \left[\sum_{t=1}^{t=H} R(s_t, s_{t+1}) \right]$. Given a policy and the number of timesteps t remaining till the end of episode, we define a value function $V_t(s) = \frac{1}{H-t} \mathrm{E}_\tau \left[\sum_{i=t}^{H-1} R(s_i, s_{i+1}) \;\middle|\; s_t = s \right]$. The optimal value function $V^\star$ is defined as the value function of an optimal policy $\pi^\star$.

States and transitions The state space is infinite and corresponds to the set of possible Knossos programs. The initial state distribution p_0 models the programs that the RL agent is likely to be asked to optimize. A sample MDP state is shown in Figure 1a. The action set A and the set of possible transitions T correspond to different possible ways of re-writing the same program. Once the action is chosen, the re-write step is deterministic. Because re-write rules can be applied to different sub-expressions, we specify A and T using generic re-write rules, which are applied by pattern matching. We show subset of simplifying rules in Figure 1b. For example, the first rule says that adding zero to any number results in the same number. There are over TODO rules like this we provide the details in Appendix A. An essential feature of the re-writes is that they do not change the mathematical meaning of the expression, i.e. by simplifying from one expression to another we also implicitly generate a proof that the expressions are equivalent (see also Section 3.2 for special treatment of floating-point numbers).

Rewards and the Cost Model We assume access to a function $c(s)$, which provides the *cost model*, i.e. the computational cost of running the program s on representative inputs. While developing perfect is cost models is theoretically impossible due to the intractability of the halting problem Turing (1937), very good cost models exist for the particular subset of programs that compliers are asked to optimise. The ideal cost model c_B would correspond to the run-time of the program on typical inputs, but evaluating costs by benchmarking is very computationally intensive. In practice, one can often find a surrogate cost function such that for most initial programs s_0, we have

$$\arg\max_s c(s_0) - c(s_H) = \arg\max_s c_B(s_0) - c_B(s), \tag{1}$$

which is much easier to acquire. In other words, the cost function c does not have to produce the same run-time but the same maximum over programs. Knossos has a modular architecture, making it easy to change the cost function. This makes it possible to quickly re-tune Knossos programs for any target hardware. We stress that the formalism allows us to find optimisations even in case getting to the optimized version of the code requires using intermediate programs of higher cost.

Our reward function is based on this cost model. The rewards $R(s_1, s_2) = c(s_2) - c(s_1)$ correspond to the attained reduction in cost when transforming expression s_1 into s_2. This formulation ensures that return J_π equals the total cost reduction attained along the length of the rollout τ. Similarly, the value function corresponds to the average reduction in cost per time-step achieved by the current policy. Since our MDP includes a 'no-op' rewrite rule that allows us to keep the current expression and hence the cost, the optimal value function is monotonic in t i.e.

$$t' V^\star_{t'}(s) \geq t V^\star_t(s) \quad \text{for any} \quad t' \geq t. \tag{2}$$

3 Re-write Rules

3.1 Guards and Side Conditions

TODO.

3.2 Floating point numbers

TODO.

4 Training the RL Agent

The task of re-writing expressions is distinct from typical RL benchmarks in several ways. The allowed set of actions not only changes from state to state, but grows with the size of the expression. This makes exploration hard. Second, the states of the MDP, which correspond to the expressions

being re-written, are represented as graphs, whose size and topology varies as optimisation progresses. This is unlike traditional deep Reinforcement Learning (Mnih et al., 2013), which learn either from pixels or from data of fixed shape.

While the rewriting task has many features that make it difficult, it is also easier than many traditional RL tasks in two important ways. First, MDP transitions are completely deterministic. The only randomness in the generation of roll-outs is caused by the stochastic policy. This means that we have control over the variance of the obtained roll-outs. Second, the task has a large degree of locality in the sense that the performance a program can often be substantially improved by optimising its parts separately. In order to exploit these properties, Knossos uses a custom RL algorithm, based on $A^\star$ search supported by value function learned with a graph neural networks (Algorithm 1). We describe how to obtain the heuristic in Section 4.2, the search algorithm in Section 4.1 and the used curriculum in 4.3.

Algorithm 1 Knossos

function TRAIN($E, d, epochs$)
  Initialize a value function $\hat{V}$
  for $\_ \in [1..epochs]$ **do**
    for $e \in E$ **do**
      $S, t, V^{\text{target}} \leftarrow A^\star(e, \hat{V}, d)$
      $\hat{V} \leftarrow Fit(S, t, V^{\text{target}}, \hat{V})$
    end for
  end for
end function
function OPTIMIZE($e, \hat{V}, d$)
  $S, \_, \_ \leftarrow A^\star(e, \hat{V}, d)$
  return $\arg\min_{s \in S} cost(s)$
end function

4.1 SEARCHING THE SPACE OF RE-WRITES WITH $A^\star$

(TODO: detail)

We use the the $A^\star$ algorithm both to train the compiler and to deploy it Hart et al. (1968). In both cases, the $A^\star$ heuristic function corresponds to the learned value function $\hat{V}$, obtained from the previous iterations.

$A^\star$ maintains a priority queue called open list to store all the leaf nodes (nodes which child nodes are not explored yet) and a table called closed list to store all the nodes visited so far [1].

Each iteration of $A^\star$ is as follows (Algorithm 2). First, an agent picks a node which has the highest priority. Then, it generates all the child nodes of the node. For each generated child node, it queries the closed list to check if we already visited that state. If the child state c is not visited ($c \notin Closed$), or visited only by longer number of steps ($t(s) - 1 > t(c)$), then it is stored in the open list. We repeat this procedure until some termination condition satisties.

$A^\star$ picks a node which has the highest priority f in the open list. We set the priority of a node n to be the estimated total cost reduction from the initial expression.

$$f(s) = t\hat{V}_{t(s)}(s) + \sum_{k'=k}^{H} r_{k'} \tag{3}$$

$$= t\hat{V}_{t(s)}(s) + c(s_0) - c(s) \tag{4}$$

$t\hat{V}_t(s)$ corresponds to the estimated cost reduction it can achieve from state s, t is the number of steps allowed from state s, and $cost(s_0) - cost(s)$ corresponds to the cost reduction achieved

[1] It is called open/closed list for historical reasons but not necessarily implemented as a list.

Algorithm 2 $A^\star$ search

function $A^\star(s_0, \hat{V}, H)$
  $f(s) \leftarrow t\hat{V}_{t(s)}(s) + \sum_{k'=k}^{H} r'_k$
  $Open \leftarrow \{s_0\}$
  $Closed \leftarrow \emptyset$
  $t(s_0) \leftarrow H$ ▷ t represents the step budget left for the state.
  while not term_condition() and $Open$ is not empty **do**
    $s \leftarrow \arg\max_{s \in Open} f(s)$.
    $Open \leftarrow Open \setminus \{s\}$
    for all $a \in A(s)$ **do**
      $c \leftarrow a \circ s$
      if $c \notin Closed$ **or** $t(s) - 1 > t(c)$ **then**
        $t(c) \leftarrow t(s) - 1$
        if $t(c) > 0$ **then**
          $Open \leftarrow Open \cup \{c\}$
        end if
        if $c \notin Closed$ **then**
          $Closed \leftarrow Closed \cup \{c\}$
        end if
      end if
    end for
  end while
  for all $s \in Closed$ **do**
    $V^{\text{target}}_{t(s)}(s) \leftarrow \frac{1}{t(s)} \max_{s' \in Closed \cup distance(s,s') \leq t(s)} cost(s) - cost(s')$
    ▷ Empirical estimate of maximum cost reduction achievable from s
  end for
  return $Closed$, t, V^{target}
end function

at s compared to the initial expression. Thus, $f(s)$ represents the estimate of the maximum cost improvement from a trajectory passing through state s. Note that $cost(s_0) - cost(s)$ corresponds to the accumulated reward to reach s from s_0 ($\sum_{k'=k}^{H} r'_k$).

While typical model-free reinforcement learning algorithms select next state to explore subject to a constraint that it has to be a neighbor of the state it jused explored, $A^\star$ has no such constraint. Such constraint is required for typical reinforcement learning domains where an agent is not allowed to change the state of the environment or the cost of copying states is huge or impossible (e.g. robotics task). However, for a graph search problem where such constrants doesn't exist, $A^\star$ is a powerful algorithm as it has less constraint on which state to explore next.

After the search, we compute the empirical estimate of maximum cost reduction achievable ($V^{\text{target}}_{t(s)}(s)$) from each state it visited during the search. The estimated value of s with $t(s)$ timesteps is the maximum cost reduction found from s within $t(s)$ steps. $distance(s, s')$ in Algorithm 2 is the number of steps required to reach s' from s.

4.2 Learning Value Functions with Graph Neural Networks

States in the Knossos MDP are expressions and correspond to computation graphs. In order to apply deep RL to these graphs, we need to be able to construct differentiable embeddings of them. To do this, we employ Graph Neural Networks based on Gated Recurrent Units (Li et al., 2016). During the forward pass, the GNN begins with an initial embedding of the graph nodes. It then iteratively applies a diffusion process to the graph. At each step, the obtained representation is fed into a gated recurrent unit (GRU). The process implicitly encodes the edge structure of the graph in the obtained representation.

(TODO: detail about GNN).

(TODO: Both value function and vertices are usually denoted V. What's the best notation?)

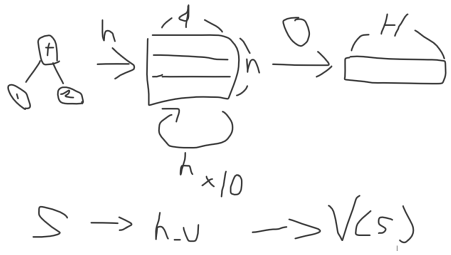


Figure 2: Value function with graph neural network

Our value function consists of two modules, $h : S \to \mathbb{R}^d$ and $O : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^H$:

$$V(s) = O(\sum_{v \in N(s)} h_v, \sum_{v \in N(s)} x_v), \tag{5}$$

where $V(s) = [V_0(s), V_1(s), ...V_H(s)]$, d is the dimension of the node feature, $N(S)$ is a set of vertices in the input graph and x_v is the feature of node v and h_v is the embedding of node v (Figure 2). An embedding h_v is a represented by a function of the embedding of its neighbor nodes:

$$h_v = f(x_v, x_{co(v)}, h_{ne(v)}, x_{ne(v)}), \tag{6}$$

where $x_{co(v)}$ is the feature of the edges connected to v, $h_{ne(v)}$ is the embedding of the neigbor nodes, and $x_{ne(v)}$ is the feature of the neighbor nodes. h_v is the fixed point for the following iterative procedure:

$$h_v^{t+1} = f(x_v, x_{co(v)}, h_{ne(v)}^t, x_{ne(v)}), \tag{7}$$

where h_v^t denotes the t-th iteration of h_v. We use h_v^t as an approximation of h_v and feed it into O to estimate the value of s.

The GNN is used to obtain an approximation $\hat{V}$, which tracks an approximate lower bound of the optimal value function $V^\star$. To do this, we minimize the loss $l(\hat{V}_k(s) - V_k^{\text{target}}(s))$. The target value $V_k^{\text{target}}(s) = \frac{1}{H-k} \sum_{t=k}^{H} r_t$ corresponds to the best cost improvement per step obtained with the current policy in $H - k$ steps.

We stress that our GNN works seamlessly with programs which include recursive function calls. Naively, it may seem that allowing recursion in Knossos expressions would lead to an infinite expansion, making GNN inference impossible. Fortunately, this is not the case. Knossos represents calls to the same recursive function by graph nodes with shared labels, rewriting the abstract syntax tree of a program rather than its stack trace. In other words, using recursion does not lead to explosion in the size of the computation graph for the same reason that recursive programs do not take an infinite number of lines of code to write.

4.3 Curriculum Learning

Due to the scale of the search problem in Knossos, it is difficult to obtain a good agent by immediately training on the hardest-to-optimize programs. In order to address this problem, we follow a curriculum (Elman, 1993; Bengio et al., 2009) when training the agent. In each training epoch, we associate a *success rate* p_{success} with each expression. The success rate p_{success} counts, the proportion of rollouts which reached the current best-known value for each expression. During training, we only consider expressions with success rate $p_L < p_{\text{success}} < p_H$, where p_L and p_H are hyperparameters. By excluding expressions which are either trivial or intractable from training, we allow the agent to make faster progress. Our results in section 8 show that this curriculum is very effective.

(TODO: more detail).

Listing 1: F# code fed into Knossos

```
let rec embedding phrase =
  match phrase with
    | Leaf word ->
       weights.embedding.[word]
    | Node (l,r) ->
       let sl = embedding l
       let sr = embedding r
       tanh (weights.Wl * sl +
           weights.Wr * sr)
```

Listing 2: TensorFlow code

```
def __loop_body(tensor_array, i)
  is_leaf = tf.gather(self.
      is_leaf_placeholder, i)
  word_index = tf.gather(self.
    word_indices_placeholder, i)
  left_child = tf.gather(self.
      left_children_placeholder, i)
  right_child = tf.gather(self.
      right_children_placeholder,i)
  node_tensor = tf.cond(is_leaf,
   lambda:__embed_word(word_index),
   lambda:__combine_children(
       tensor_array.read(
           left_child),
  tensor_array.read(right_child)))
  tensor_array = tensor_array.write
      (i, node_tensor)
  i= tfadd(i, 1)
return tensor_array, i
```

Figure 3: Comparison of equivalent code Knossos/F# vs TensorFlow.

5 THE KNOSSOS INTERMEDIATE REPRESENTATION

(TODO: this is likely to need shortening)

As part of the Knossos complier, we also introduce the Knossos intermediate representation (IR) with native support for differentiation. We provide a transpiler for F#, effectively giving it automatic differentiation capability. Our system has a modular architecture, meaning that support for additional languages can be added easily. We are already working on a module for Julia. Moreover, we provide a LISP-like surface syntax for the IR which can be useful on its own for users familiar with the LISP family of languages.

The benefits of our approach are best seen by example. In Figure 3, we compare the same algorithm written in in Knossos with a version written using Python with the TensorFlow library. The code implements a Tree Neural Network and is representative of modern ML workloads. The TensorFlow code, shown on the right-hand side of Figure 3, follows the *meta-programming* paradigm, where the programmer uses Python to construct a Abstract Syntax Tree (AST), which is then processed with automatic differentiation. Because Python lacks compile-time type safety, logical errors in building up the AST can often be only detected at runtime, making debugging difficult. In fact, some kinds of these errors are so common that special projects have been created to work around them (Rush). Some of these errors persist even in Swift for TensorFlow, which is strongly-typed. For example, tensor dimension is not part of the type in Swift for TensorFlow (TODO: does the Knossos type system enforce more constraints, like matrix sizes?).

The F# code, shown on the left, follows a more straightforward convention. The programmer simply writes out tree traversal in the normal way. Because of the type systems used in Knossos and F#, once the code complies, the abstract syntax can be assumed to be free of a broad class of logical errors. Since differentiation in Knossos is a first-class citizen, the programmer can use a short language primitive to perform optimisation with respect to any variable (see Section 6 for details). This makes writing differentiable programs as easy as writing traditional software while providing all the power of modern machine learning. This technology also makes it possible to use refactoring and other IDE automation tools that already exist for F#, for differentiable programs, further enhancing programmer productivity.

Superficially, our approach resembles PyTorch, where the meta-programming is obscured with syntactic sugar such as operator overloading. However, underneath, PyTorch still works by constructing an Abstract Syntax Tree of PyTorch primitives that are fundamentally distinct from Python ones. For

example, it means that one cannot import a non-PyTorch Python function and use it in a differentiable PyTorch program. Moreover, because Knossos code is completely decoupled from the source language, it can be easily optimized, allowing for much better performance.

6 AUTOMATIC DIFFERENTIATION

Generalized Transpose (TODO: this section may need to be shortened in the final paper). Modern machine learning crucially depends on the capability to compute gradients of a broad range of expressions. Historically, algorithms for gradient computation have been divided into forward-mode and backward-mode, where the forward mode was more efficient for functions with more outputs than inputs and the backward mode is more efficient for functions with more inputs than outputs. Recently, a new, unified view of automatic differentiation has emerged (Elliott, 2018). By modelling automatic differentiation as a generalized transposition, it unifies previous approaches and allows for an implementation of automatic differentiation as a transformation of syntax trees. The Knossos system exploits this view. Rather than having an explicit distinction between forward mode and reverse mode AD, Knossos uses uses a type system together with a set of consistent re-write rules. Whenever the gradient operator is used as part of a Knossos algorithm, the complier first generates a syntax tree corresponding to the differentiated program and then applies re-writes to optimize the cost of its execution. This means that the resulting AD algorithm is tailor-made and optimized with that exact use case in mind. This is in stark contrast to systems such as PyTorch, which have hard-coded routines for backward-mode AD.

From the perspective of the user, this process is completely transparent. For example, to perform supervised learning, one can use the F# code below.

```
weights_trained = adam (w -> loss w examples) weights_init
```

This calls the adam optimizer, which adjusts the network weights by differentiating the loss. Crucially, `loss` can be an arbitrary F# function defined in the usual way – the programmer does not have to know anything about gradients.

Arbitrary Tangent Spaces (TODO: this section is aspirational and may not appear in the final paper). Modern deep learning algorithm often involve objects of different types. For example, algorithms that perform probabilistic modelling use probability distributions, while algorithms that manipulate 3D rotations might use quaternions. Even though both are ultimately represented as vectors of real numbers, comparing them by simply computing the Euclidean distance between these vectors does not reflect their properties. Instead, one should use a notion of similarity that is tailor-made for a specific type of object (Deza & Deza, 2009). For example, the natural way to compare probability distributions is to use an α-divergence while the natural way to compare quaternions is to use an elliptic distance. This notion of similarity becomes crucial when defining gradients. Since a gradient quantifies the behaviour of a function in a small neighbourhood of its argument, it is critical that this neighbourhood is defined using the right notion of similarity. Each type of similarity corresponds to a specific way of computing the gradients, known as the tangent space.

The Knossos type system supports arbitrary tangent spaces natively. For example, in Knossos, all that is needed to turn a regular policy gradient algorithm (which compares policies using their parameters) into a natural policy gradient algorithm (which compares policies using the KL-divergence) is to declare that the policy object is of type *probability distribution*. For example,

`pi : Vector<double>` becomes `pi : Multinomial .`

Since Knossos compares probability distributions using the KL divergence by default, using natural gradients is trivial.

7 RELATED WORK

Knossos builds on a long tradition of compiler technology. Similarly to traditional compliers (Santos & Peyton-Jones, 1992; Lattner & Adve, 2004) and the more recent deep learning compliers such as Myia (van Merrienboer et al., 2018), DLVM (Wei et al., 2018), ISAM (Sotoudeh et al., 2019) and GLOW (Rotem et al., 2018), Knossos uses an intermediate representation to optimize programs. However, while these approaches rely on layers of hand-coded optimisation heuristics, Knossos *learns* the algorithm used to optimize its programs.

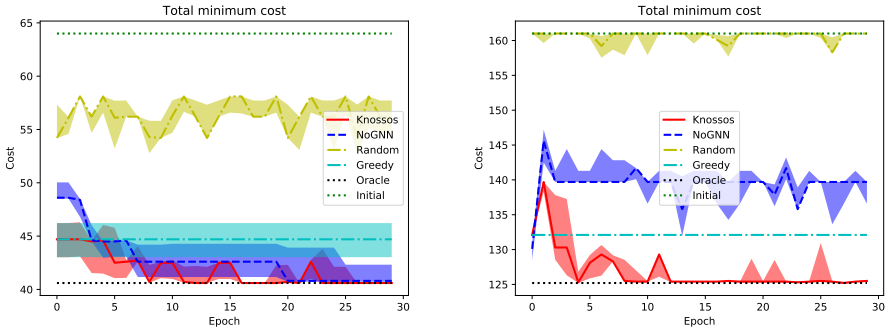
In this respect, Knossos is a spiritual successor of benchmark-driven hardware-agnostic optimisation approaches in computational linear algebra (Padua, 2011) and signal processing (Frigo & Johnson, 1998). However, unlike these approaches, Knossos is a fully-fledged complier, and can optimize arbitrary programs. Moreover, thanks to its Reinforcement Learning-driven optimizer, Knossos has an advantage over existing approaches that attempt to learn how to optimize arbitrary code. For example, Bunel et al. (2017) learns parameters of a code optimizer with a hard-coded hierarchy. REGAL (Paliwal et al., 2019) only learns the hyper-parameters for a fixed genetic algorithm that preforms the actual optimisation. The TVM compiler (Chen et al., 2018a) learns a cost model over programs, but uses simple simulated annealing to perform the optimisation. Similarly, Chen et al. (2018b) handles only index summation expressions and again relies on simulated annealing. LIFT (Mogers et al., 2019) optimises programs by re-writing expressions, but does not describe its search algorithm (TODO: is this right?). In Section 8, we demonstrate that the RL optimizer used by Knossos outperforms this approach by a large margin.

Knossos is also related to JAX (Frostig et al., 2018), which performs just-in-time compilation of Python code using the XLA backend (XLA authors, 2016). Knossos differs from JAX in two ways. First, it uses efficient RL code optimisation, which is architecture-agnostic. In fact, since Knossos generates C++ code, it supports a much broader variety of target architectures. Also, unlike JAX, it makes use of the benefits of a statically typed languages. In terms of scope, Knossos is also similar to Zygote for Julia Innes et al. (2019). However, unlike these compliers, Knossos makes use of an RL-driven code optimizer.

Since Knossos provides first class support for automatic differentiation, it is also related to established deep learning frameworks (Maclaurin et al., 2015; Abadi et al., 2016; Paszke et al., 2017). However, unlike Knossos, these frameworks do not learn how to optimize code, instead relying on manually-prepared back-ends. Moreover, using these frameworks either requires meta-programming, where the user has to use a high-level language to specify the desired computation graph using constructions external to the language (Abadi et al., 2016), or is constrained to a restricted subset of the language (Paszke et al., 2017). In contrast, the Knossos language can be used directly, without manually specifying computation graph constructs or restricting oneself to an allowed subset of the language.

In parallel, the idea of automated rewriting to achieve a given objective was explored in the context of automated theorem provers. This is conceptually related to our approach since finding an equivalence between formulae is the same as finding a proof that they are equal. However, recent work in this space has substantial differences in scope. In particular, state-of-the-art work that searches for refutational proofs in first-order logic (Zombori et al., 2019; Kaliszyk et al., 2018) uses hard-coded features and cannot learn any new ones. Also, the optimal objective is very different. While a mathematical proof is only correct when completely reduced to a tautology, we are satisfied with simplifying an expression by a certain margin, not necessarily on finding the most optimal way to do it.

For the Reinforcement Learning part, our algorithm differs from standard techniques in that it has a much larger action space and a state space that consists of graphs, which makes the application of traditional RL algorithms like DQN (Mnih et al., 2013), A2C (Mnih et al., 2016) and PPO (Schulman et al., 2017) ineffective. The closest existing agent is AlphaGo (Silver et al., 2016; 2017), which also performs a search over a large state space, but differs from Knossos in that it learns for pixel observations and uses an action space of bounded size.



(a) Combined

(b) Generalization

Figure 4: Performance of Knossos on a set of arithmetic expressions. (TODO: Remove Initial costs?)

8 BENCHMARKS

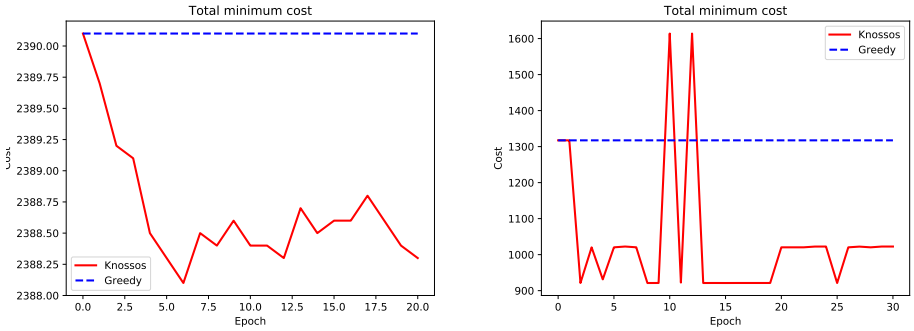
We evaluated Knossos in three settings. First, we wanted to measure how optimal our RL optimizer is. To do this, we applied Knossos to a manually curated set of arithmetic expressions, where the cost of the best re-write possible is known exactly. Second, we implemented a Gaussian Mixture Model (GMM) to asses how the code optimizer scales to more realistic tasks, including differentiation. Third, in order to reflect common modern use-cases, we evaluated Knossos on a computer vision task that uses a convolutional neural network. We compare our RL optimizer to best-effort code optimisation by humans. This comparison reflects the reality of tooling choice on platforms where no back-ends for mainstream tensor frameworks exist. We describe the results below.

Arithmetic Expressions We evaluate the system on the following scenarios: Combined and Generalization. (TODO: Scenario names?) The rewrite rule set for Combined consits of arithmetic rules and binding rules (TODO: show rule set). The training set of Combined consists of 8 expressions and the test set is the same as the training set. We use binding rules as a rule set for Generalization. The training set of Generalization consists of 36 expressions and the test set consists of 12 expressions. The training and test set are mutually exclusive. Thus, the value function V has estimate the value of previously unseen expressions. Because the training set is large, we pick 6 expressions randomly from a training set to train instead of training all the expressions in the training set for each epoch.We ran 10 repetitions for each experiment with different random seeds. We ran the training for 40 epochs. We terminate the search on each epoch for each expression when the agent queried $V(s)$ for 5000 times. We fixed the maximum depth of the search to be 10.

Hyperparameters of the neural network is present in appendix. (TODO)

Figure 6 shows the performance of the Knossos code optimizer on a set of arithmetic expressions as a function of training epochs. Horizontal axis shows the number of training epochs. Vertical axis shows the minimum cost found by an agent trained for that number of epochs, summed over the test set. The minimum cost found for each expressions is shown in appendix. (TODO: put performance of each expression) Red line shows the performance of Knossos. Purple dashed line shows the performance of greedy best-first search which picks a next state to explore greedily without using the value function $f(s) := c(s_0) - c(s)$. The yellow dash dotted line at the bottom shows the total minimum cost achievable and the blue dash dotted line at the top shows the total cost of the initial expressions. Shaded areas in the plots shows the range of the 20% and 80% percentile over 10 repetitions (i.e. second best and eighth best performance are shown).

We also evaluated the performance of Knossos using Monte Carlo Tree Search as a search engine. See appendix for the comparison of the performance of search engines. (TODO)



(a) Gaussian Mixture Model

(b) Convolutional Neural Network

Figure 5: Performance of Knossos on a Gaussian Mixture Model and a convolutional network. **TODO: Update results.**

Gaussian Mixture Model GMMs are a standard tool in probabilistic modelling and capture the essential features of many real-world ML workloads. In particular, when processing the GMM code, the Knossos optimizer needs to both support fast linear algebra and be able to compute gradients.

The input source file to the reinforcement learning agent is 20 expressions representing functions used in GMM code (e.g. matrix multiplications, objective function, vector allocation, etc.). Because the training set is large, we pick 6 expressions randomly to train instead of training all the expressions in the training set for each epoch. We ran the training for 20 epochs. We terminate the search on each epoch for each expression when the agent queried $V(s)$ for 30000 times.

(TODO: Details of the rewrite rules. In appendix?)

In addition to the rewrite rules introduced for arithmetic expressions, we add a set of rules to optimize vector operations. Unfortunately, with all the rewrite rules added, the branching factor scales super-linearly to the size of the expression. We observed that the agent often stuck at local optimum for a long time because there are many zero-cost actions available at the local optimum. In order to support all the rewrite rules to optimize broad set of expressions but still keep the tasks tractable, we take a two-phase strategy to optimize the code. At the first phase the agent seeks for large improvement. We restrict the rewrite rules to those which make a big change in the cost of the expressions in this phase. At the second phase we no longer have such restriction and all the rewrite rules are available to the agent. We fixed the maximum depth of the search to be 30. We restrict the agent to high-rewarding rewrite rules (first phase) for the first 15 steps and allow all rules (second phase) for the last 15 steps.

Figure 5a shows the sum of the cost over all the expressions in the input file. The plot shows results for 10 independent runs of the Knossos code optimizer on the same input source file. The shaded area represents the standard deviation across the runs of Knossos. Results show that Knossos produced code that ran faster than the best-effort result obtained by manual optimisation.

Convolutional Network We use Knossos on a computer vision task. We optimize a code for training a convolutional deep network on a MNIST dataset (LeCun, 1998). The source code consists of XXX functions such as dense layer, convolutional layer, max pooling layer, etc. We deploy the same two phase strategy used for optimizing GMM. The results are shown in Figure 5b. Again, the shaded area represents the standard deviation across the runs of Knossos and the resulting binary. As above, the Knossos optimizer produced code that runs faster than the human baseline.

Summary of Experiments We have demonstrated that Knossos is capable of producing code that is more performant than best-effort human optimisation by margins of up to TODO%. While established tensor frameworks outperform Knossos on the hardware they support, Knossos code is much more portable. In fact, the code generated by Knossos in our experiments can perform both training and inference and can be run on any hardware supporting the C++ toolchain, including inexpensive embedded devices. On these devices, where no specialised back-ends exist, Knossos is

capable of obtaining state-of-the-art performance. Since our architecture is generic, we hope that future versions of Knossos, equipped with an augmented set of rules and a cost-model tailored to the target hardware, will be able to beat existing frameworks on powerful GPUs.

(TODO: finalize list of experiments. Other experiments suggested by Andrew: CSAT, bootstrapping i.e. Knossos complies Knossos, TreeNN)

9 CONCLUSIONS

We have introduced Knossos, a new complier targetting machine learning software. Thanks to its automatic code optimisation, Knossos produces binaries that achieve significantly better run-times than obtained with best-effort manual tuning of the code for many days. Thanks to its type system with first-class support for automatic differentiation, it is easier to write Knossos code then to use existing deep learning libraries based on the meta-programming paradigm. Moreover, Knossos code is very portable, allowing both inference and training on a variety of embedded devices which were not previously supported by deep learning frameworks.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In Kimberly Keeton and Timothy Roscoe (eds.), *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283. USENIX Association, 2016.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48. ACM, 2009.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. Learning to superoptimize programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker (eds.), *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pp. 578–594. USENIX Association, 2018a.

Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to Optimize Tensor Programs. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pp. 3393–3404, 2018b.

Michel Marie Deza and Elena Deza. Encyclopedia of distances. In *Encyclopedia of distances*, pp. 1–583. Springer, 2009.

Conal Elliott. The simple essence of automatic differentiation. *PACMPL*, 2(ICFP):70:1–70:29, 2018. doi: 10.1145/3236765.

Jeffrey L Elman. Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99, 1993.

Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98, Seattle, Washington, USA, May 12-15, 1998*, pp. 1381–1384. IEEE, 1998. ISBN 978-0-7803-4428-0. doi: 10.1109/ICASSP.1998.681704.

Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. 2018. URL http://www.sysml.cc/doc/2018/146.pdf.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckus, Elliot Saba, Viral B Shah, and Will Tebbutt. Zygote: A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*, 2019.

Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement Learning of Theorem Proving. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pp. 8836–8847, 2018.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pp. 75–88. IEEE Computer Society, 2004. ISBN 978-0-7695-2102-2. doi: 10.1109/CGO.2004.1281665.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05493.

Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

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

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Naums Mogers, Aaron Smith, Dimitrios Vytiniotis, Michel Steuwer, Christophe Dubach, and Ryota Tomioka. Towards Mapping Lift to Deep Neural Network Accelerators. In *Workshop on Emerging Deep Learning Accelerators (EDLA) 2019 at HiPEAC*, 2019.

OpenAI. AI and Compute. https://openai.com/blog/ai-and-compute/, 2018. Blog post.

David A. Padua. Automatically Tuned Linear Algebra Software (ATLAS). In *Encyclopedia of Parallel Computing*, pp. 101. Springer, 2011. ISBN 978-0-387-09765-7. doi: 10.1007/978-0-387-09766-4_2061.

Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. REGAL: Transfer Learning For Fast Optimization of Computation Graphs. *CoRR*, abs/1905.02494, 2019.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

Martin L Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR*, abs/1805.00907, 2018.

Alexander Rush. Tensor considered harmful. http://nlp.seas.harvard.edu/NamedTensor. Blog post.

André L. M. Santos and Simon L. Peyton-Jones. On Program Transformation in the Glasgow Haskell Compiler. In John Launchbury and Patrick M. Sansom (eds.), *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992*, Workshops in Computing, pp. 240–251. Springer, 1992. ISBN 978-3-540-19820-8.

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

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Matthew Sotoudeh, Anand Venkat, Michael J. Anderson, Evangelos Georganas, Alexander Heinecke, and Jason Knight. ISA mapper: A compute and hardware agnostic deep learning compiler. In Francesca Palumbo, Michela Becchi, Martin Schulz, and Kento Sato (eds.), *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*, pp. 164–173. ACM, 2019. ISBN 978-1-4503-6685-4. doi: 10.1145/3310273.3321559.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

Bart van Merrienboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. Automatic differentiation in ML: Where we are and where we should be going. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pp. 8771–8781, 2018.

Richard Wei, Lane Schwartz, and Vikram S. Adve. DLVM: A modern compiler infrastructure for deep learning systems. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018.

XLA authors. TensorFlow XLA (Accelerated Linear Algebra). https://www.tensorflow.org/xla/overview, 2016.

Zsolt Zombori, Adrián Csiszárik, Henryk Michalewski, Cezary Kaliszyk, and Josef Urban. Towards Finding Longer Proofs. *CoRR*, abs/1905.13100, 2019.

APPENDICES

APPENDIX A SPECIFICATION OF THE RE-WRITE RULES.

TODO.

APPENDIX B FULL EXAMPLE PROGRAM IN THE KNOSSOS LANGUAGE.

TODO.

APPENDIX C ADDITIONAL ABLATIONS.

In addition to $A\star$ search, we compare the performance of Monte Carlo Tree Search (Browne et al., 2012) using the UCT formula (Auer et al., 2002; Kocsis & Szepesvári, 2006).

Each iteration of MCTS consists of four steps (Algorithm 3).

1. Selection: Starting from the root, a tree policy is recursively applied to descend through the tree until it reaches a leaf node.
2. Expansion: One child node is added to expand the tree.
3. Simulation: A rollout policy is applied from the new node until the end of the episode (maximum depth).
4. Backpropagation: The simulation result is backuped through the selected nodes to update their statistic.

The tree policy π_t and rollout policy π_r are as follows:

$$\pi_t(a|s) = \arg\max_{a \in A}\left(\frac{X(s')}{n(s')} + c\sqrt{\frac{\ln n(s)}{n(s')+1}}\right), \tag{8}$$

$$\pi_r(a|s) = \text{softmax}_{a \in A}(R(s,a) + V(s'), \alpha), \tag{9}$$

where $n(s)$ is a visitation count of state s, and c is a constant to control the exploration bonus. $X(s')/n(s')$ is the average cost reduction achieved by a set of trajectories which passed through s' in H: $X(s') = \sum_{T \in H: s' \in T} \max_{s_d \in T: d(s_d) \geq d(s')} cost(s_0) - cost(s_d)$, and H is a set of trajectories sampled at current epoch for the current expression so far. The more the state s' is visited, the exploration bonus $(c\sqrt{\frac{\ln n(s)}{n(s')+1}})$ is reduced. In this way it encourages the agent to try diverse set of actions.

We evaluate the performance of $A\star$ search and MCTS for both training and test algorithm. We evaluate on Generalization scenario. The experimental setup is the same as Section 8 except we evaluate using different search algorithms. We set $\alpha = 5.0$ and $c = 0.5$ for MCTS for both training and test.

First we compare the performance of search algorithms for training with test algorithm fixed to $A\star$. That is, we use $A\star$ or MCTS to explore the expressions for training set and train the value function using the sampled states. Then, we optimize the expressions in test set with $A\star$. Figure 6a shows the total minimum cost achieved on test set, using different algorithms for training. $A\star$ significantly outperformed MCTS as a training algorithm. Next, we compare the performance of search algorithms for test with training algorithm fixed to $A\star$. Figure 6b shows the total minimum cost achieved on test set. $A\star$ significantly outperformed MCTS as a test algorithm. Figure 6c shows the performance of all the combination of search algorithms for training and test. Overall, using $A\star$ for both training and test achieved the best performance.

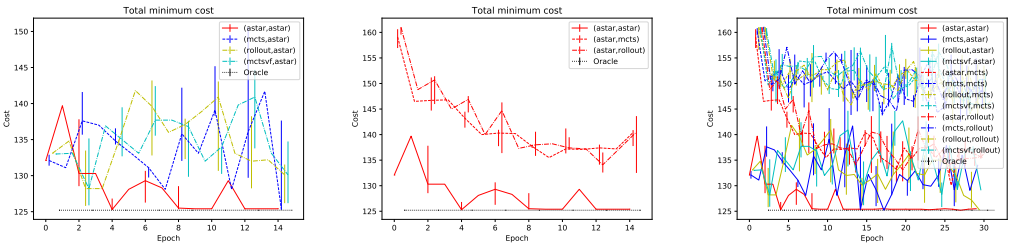
APPENDIX D REPRODUCIBILITY AND DETAILS OF EXPERIMENTAL SETUP

The Knossos system consists of three main components: the transpiler that priduces Knossos files from F#, the RL Knossos optimizer and the Knossos complier that produces C++ code. The complier

Algorithm 3 Monte Carlo Tree Search (MCTS)

function MCTS(s_0, V, d, α, c)
  $\pi_t(a|s) \leftarrow \arg\max_{a \in A}(X(s')/n(s') + c\sqrt{\frac{\ln n(s)}{n(s')+1}})$
  $\pi_r(a|s) \leftarrow \text{softmax}_{a \in A}(R(s,a) + V(s'), \alpha)$
  $H \leftarrow \emptyset$
  $\forall s \in S : n(s) \leftarrow 0$ ▷ Initialize a visitation count
  while not term_condition() **do**
    $T_t \leftarrow \text{TREESEARCH}(s_0, d, \pi_t)$
    $s_l \leftarrow \text{tail}(T_t)$
    $T_r \leftarrow \text{ROLLOUTSEARCH}(s_l, d - \text{length}(T_t), \pi_r)$
    $T \leftarrow \text{concat}([s_0], T_t, T_r)$
    $x \leftarrow cost(s_0) - \min_{s \in T_r} cost(s)$
    ▷ x is the max cost reduction found by rollout
    for all $s \in T_t$ **do** ▷ Backpropagation
      $X(s) \leftarrow X(s) + x$
      $n(s) \leftarrow n(s) + 1$
    end for
    $H \leftarrow H \cup \{T\}$
  end while
  return H
end function

function TREESEARCH(s, d, π)
  $T \leftarrow []$
  while length$(T) < d$ **and** $n(s) > 0$ **do**
    $a \leftarrow \pi(a|s)$ ▷ Tree policy π is deterministic.
    $s \leftarrow a \circ s$
    $T \leftarrow \text{concat}(T, [s])$
  end while
  return T
end function



(a) Generalization (Train) (b) Generalization (Test) (c) Generalization

Figure 6: Comparison of $A\star$ search and Monte Carlo Tree Search. (TODO: Which other algorithms should I compare against?)

and transpiler are entirely deterministic and do not have any hyper-parameters. We describe the RL code optimizer below.

We used the following hyper-parameters for the RL code optimizer.

Parameter	**Value**
Learning rate	TODO
Number of diffusions in the GNN	TODO
Number of MC rollouts	TODO
Temperature setting α^{-1}	TODO