KNOSSOS: COMPILING AI WITH AI

Anonymous authors
Paper under double-blind review

ABSTRACT

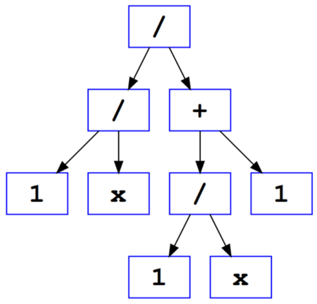
Machine learning workloads are often expensive to train, taking weeks to converge. The current generation of frameworks relies on custom back-ends in order to achieve efficiency, making it impractical to to train models on less common hardware where no such back-ends exist. Knossos builds on recent work that avoids the need for frameworks, instead compiling machine learning models in much the same way one would compile other kinds of software. In order to make the resulting code efficient, the Knossos complier directly optimises the abstract syntax tree of the program. 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 can automatically learn optimisations that past compliers had to implement by hand. Moreover, Knossos outperformed a traditional compiler on a suite of machine learning programs, including convolutional networks and performing inference in probabilistic models.

1 INTRODUCTION

While the development of any kind of software can benefit from compliers able to produce fast code, runtime efficiency is particularity important for modern machine learning. In particular, because modern models they can take weeks to train (OpenAI, 2018), complier optimisations that lead to execution speed-ups are of huge value. In parallel, machine learning is being deployed on a variety of diverse devices ranging from wearables to huge clusters clusters of powerful GPUs. Since each architecture has different performance profile and requires different code optimisations, it is difficult to provide tooling that works fast on all of them.

Traditionally, the tension between performance and interoperability is resolved by machine learning frameworks (Paszke et al., 2017; Abadi et al., 2016). In these frameworks, while code execution is outsourced to hardware-specific back-ends such as XLA (XLA authors, 2016). While this approach has seen huge initial success, the cost of providing customised back-ends for each target architecture is prohibitive. Moreover, the frameworks also custom front-ends that require the programmer to specify the model being trained as a compute graph. Since the compute graph has semantics separate from the host programming language, this process is often error-prone and time-consuming. In order to address these obstacles, a new generation of tools has recently appeared that transform machine learning code using the same techniques that have been used for compiling traditional software. The need for a separate front-end API for machine learning operations is eliminated by including automatic differentiation as a first-class feature of the complied language (Innes et al., 2019; Frostig et al., 2018). Instead of custom back-ends, modern machine learning compliers use an intermediate representation and perform extensive code optimisations (Innes et al., 2019; Frostig et al., 2018; van Merrienboer et al., 2018; Wei et al., 2018; Sotoudeh et al., 2019; Rotem et al., 2018). In addition, program optimisation is being modelled as a machine learning task itself, with the complier learning how to perform rewrites (Chen et al., 2018b;a).

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 fast code tailor-made for any given task and architecture, without relying on backend-specific hand-written libraries. Knossos works by re-writing programs written in an intermediate representation (IR). Akin to JAX (Frostig et al., 2018) and Zygote (Innes et al., 2019), all



(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 rewrite rules. Each rule defines a class of MDP transitions.

Figure 1: The Knossos MDP.

Knossos functions are potentially differentiable, avoiding the syntactic awkwardness that arises from embedding a differentiable program in a host language. The IR can then be transpiled, allowing it to run on any platform that supports a C++ toolchain. This allows Knossos code to 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 4. We empirically demonstrate the benefits of our program optimisation in Section 5, showing that Knossos was able to automatically learn *loop fusion*, a type of compiler optimisation that previously had to be applied manually.

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.

States and transitions An MDP state $s = (e_s, t_s)$ consists of a Knossos program (or expression) $e \in E$ and the remaining time budget $t \in [0, 1, \ldots, H]$ (i.e., the number of remaining steps), where H is the maximum budget. Any state with $t = 0$ is terminating. The initial state distribution p_0 models the expressions that the RL agent is likely to be asked to optimize. A sample Knossos expression is shown in Fig. 1a. The action set A corresponds to different possible ways of rewriting the same expression (see Fig. 1b). The transition function $T : S \times A \to S$ returns the next state after taking an action. For example, the first rule in Fig. 1b says that adding zero to any expression can be simplified to the expression itself. Once the action is chosen, the transition is deterministic. Because rewrite rules can be applied to different subexpressions, we specify A using generic rewrite rules, which are applied by pattern matching. There are over 50 rules like this – we provide the details in Appendix C. An essential feature of the rewrites is that they do not change the meaning of the program, i.e. by simplifying from one expression to another we also implicitly generate a proof that the expressions are equivalent.

Policies and Value Functions The RL agent maintains a policy $\pi(a|s)$, 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 $\tau_\pi = (s_1, a_1, r_1, s_2, a_2, r_2, \ldots s_H, r_H)$. 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 $J_\pi = \mathrm{E}_{\tau_\pi}\left[\sum_{t=0}^{t=H-1} R(s_t, s_{t+1})\right]$. Given a policy and the number of timesteps t remaining till the end of episode, we define a value function $V(s) = \mathrm{E}_\tau\left[\sum_{i=0}^{t_s-1} R(s_i, s_{i+1}) \;\middle|\; s_0 = s\right]$, where t_s denotes the remaining time budget at state s. The optimal value function $V^\star$ is defined as the value function of an optimal policy $\pi^\star$.

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 e_s, the expression represented by state $s = (e_s, t_s)$, on representative inputs. While developing a perfect 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, the state that is reachable from s_0 and minimizes the surrogate cost function c agrees with that for the ideal cost function c_B, that is,

$$\arg\min_{s\sim s_0} c(s) = \arg\min_{s\sim s_0} c_B(s), \tag{1}$$

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 minimum over programs. We show experimentally in Section 5 that it is indeed possible to reduce the wall clock time of running a program by optimising such a proxy cost model. 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 rewriting expression e_{s_1} into e_{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.

$$V^\star((e,t')) \geq V^\star((e,t)) \quad \text{for any} \quad e,\ t' \geq t. \tag{2}$$

3 Training the RL Agent

Hard and Easy Aspects of Rewriting There are two main ways in which the task of rewriting expressions is *more challenging* than typical RL benchmarks. First, 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 rewritten, are represented as graphs, whose size and topology varies as optimisation progresses. This is unlike traditional deep Reinforcement Learning (Mnih et al., 2013), which learns 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 for three reasons. First, MDP transitions are completely deterministic. Second, the task has a large degree of locality in the sense that the performance of a program can often be substantially improved by optimising its parts separately. Third, we can generate state transitions in any order convenient to us, as opposed to the traditional RL setting, where we are constrained by the order imposed by the environment. Overall, we have a problem similar to traditional planning, but which requires us to generalise well in order to obtain competitive solutions. To do this, 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 3.2, and the search algorithm in Section 3.1.

3.1 Searching the space of rewrites with $A^\star$

We use the $A^\star$ algorithm (Hart et al., 1968) both to train the compiler and to deploy it. $A^\star$ maintains two priority queues. One queue (O) stores the frontier, i.e. states from which transitions have not

Algorithm 1 Knossos

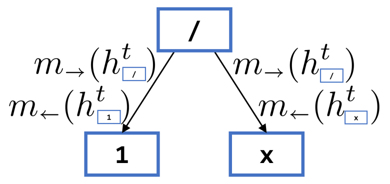
```
function TRAIN($E_{\text{train}}, H, M$)
    Input: $E_{\text{train}}$: Set of expressions to train
           $H$: Maximum depth of the search
           $M$: Number of epochs
    Initialize a value function $\hat{V}$
    for $M$ iterations do
        for $e \in E_{\text{train}}$ do
            $S, V^{\text{target}} \leftarrow A^\star(e, \hat{V}, H)$        ▷ Optimize $e$ and store the learned values $V^{\text{target}}$
            $\hat{V} \leftarrow \text{FIT}(S, V^{\text{target}}, \hat{V})$        ▷ Fit the value function - see Section 3.2
        end for
    end for
end function

function OPTIMIZE($e, \hat{V}, H$)
    Input: $e$: Expression to optimize
           $\hat{V}$: Learned value function
           $H$: Maximum depth of the search
    $S, \_ \leftarrow A^\star(e, \hat{V}, H)$        ▷ Optimize $e$
    return $\arg\min_{s \in S} c(s)$        ▷ Return the best expression found during the process
end function
```

Algorithm 2 Deep $A^\star$ search

```
function $A^\star(s_0, \hat{V}, H)$
    Input: $s_0$: Expression to optimize
           $\hat{V}$: Learned value function
           $H$: Maximum depth of the search
    $s_0 \leftarrow (e_0, H)$
    $O \leftarrow \{s_0\}$        ▷ $O$ is an open list storing all unexplored states
    $C \leftarrow \{s_0\}$ ▷ $C$ is a closed list storing all states ▷ $t$ represents the step budget left for the state.
    while not TERM_CONDITION() and $O$ is not empty do
        $s \leftarrow \arg\max_{s \in O} f(s)$.        ▷ $f$ is the heuristic function defined in equation 3.
        $O \leftarrow O \setminus \{s\}$
        for all $a \in A(s)$ do
            $n \leftarrow T(s, a)$        ▷ Obtain the next state $n = (e_n, t_n)$
            if $(e_n, t) \notin C (\forall t)$ or $(e_n, t) \in C$ and $t_n > t$ then
                if $t_n > 0$ then
                    $O \leftarrow O \cup \{n\}$
                end if
                if $n \notin C$ then
                    $C \leftarrow C \cup \{n\}$
                end if
            end if
        end for
    end while
    for all $s \in C$ do
        $V^{\text{target}}(s) \leftarrow \max_{s' \in C \cup \text{DISTANCE}(s,s') \leq t_s} (c(s) - c(s'))$
        ▷ Empirical estimate of maximum cost reduction achievable from $s$
    end for
    return $C$, $t$, $V^{\text{target}}$
end function
```

been explored yet. The other one (C) stores the states visited so far and is used to avoid exploring the same path twice. The states are explored in the order induced by the $A^\star$ heuristic, which in our case corresponds to the learned value function $\hat{V}$, obtained from previous iterations. In particular,


/
$m_{\rightarrow}(h^t_{/})$
$m_{\leftarrow}(h^t_{1})$
$m_{\rightarrow}(h^t_{/})$
$m_{\leftarrow}(h^t_{x})$
1
x

(a) Message passing on the expression graph.

Edge type	Directionality
first child	directed
second child	directed
third child	directed
tuple child	directed
is-identical	undirected

(b) List of edge types

Figure 2: Graph neural network

node priority is set as follows:

$$f(s) = f(\underbrace{(e,t)}_{s}) = \hat{V}(s) + \underbrace{\textstyle\sum_{i=0}^{H-t-1} R(s_i, s_{i+1})}_{c(s_0)-c(s)}. \tag{3}$$

Here, $\hat{V}(s)$ is the estimated future cost reduction obtained from state s within t remaining time-steps. The quantity $c(s_0) - c(s)$ corresponds to the cost reduction that has already been achieved by time t, measured against the cost of the initial expression. Thus, $f(s)$ is an estimate of the maximum possible cost improvement from a trajectory passing through state s at time t.

After the search, we compute the empirical estimate of the maximum cost reduction achievable ($V^{\text{target}}(s)$) for each visited state. 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. The algorithm stops after the value function was evaluated a set number of times. In the code this is represented with the function TERM-CONDITION.

$A^\star$ is well-suited for the rewriting task because it exploits its characteristic features. In particular, it exploits determinism by assuming that a cost reduction achievable once can always be achieved again. It exploits the availability of reset by considering nodes in the order defined by the heuristic function. It exploits locality by preferring re-writes that need a small number of rule applications. Before deciding on $A^\star$, we also performed experiments with Monte Carlo Tree Search (MCTS). MCTS does not make use of reset and had worse empirical performance (see Appendix D for details).

3.2 Learning Value Functions with Graph Neural Networks

States in the Knossos MDP 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.

Graph representation We represent a Knossos expression as a graph. The graph nodes correspond to subexpressions (see Fig. 1a). The graph edges are of two kinds. The first kind of edges connects the nodes with their parents. In addition, we use another kind of edges, which is used to explicitly provide the information that two subexpressions are identical. See Table 2b for a list of all edge types. Edges can be directed or undirected, with the directed edges going in opposite ways considered different.

Graph neural network To compute the value function for an expression e and time budget t, we start from computing the initial node embedding $h_v^0 \in \mathbb{R}^d$. The initial node embedding consists of a one-hot encoding of the node type (constant, variable, etc) followed by zero padding.

This embedding is then fed into the following recurrent computation (see Fig. 2a):

$$h_v^{t+1} = f\left(h_v^t, \bigoplus_{\substack{v' \in A_p(v), \\ p \in P}} m_p\left(h_{v'}^t\right)\right) \quad (t = 0, \ldots, T-1), \tag{4}$$

where $p \in P$ indexes different edge types, $A_p(v)$ is the set of neighbors of node v with respect to the pth edge type. We choose the message function m_p to be a single dense layer for each edge type p and the aggregation operator $\oplus$ as the sum of all incoming messages. We use the GRU cell (Cho et al., 2014) as the recurrent unit f. The final node embedding h_v^T is computed by unrolling equation 4 for T time steps.

Finally the value of expression e is computed by taking a weighted sum of the final node embedding h_v^T and passing through a dense layer as follows:

$$\boldsymbol{V}(e) = O\left(\textstyle\sum_{v \in N(e)} \sigma\left(g(h_v^T, h_v^0)\right) \cdot r(h_v^T)\right) \tag{5}$$

where $g : \mathbb{R}^d \to \mathbb{R}^d$, $r : \mathbb{R}^d \to \mathbb{R}^d$, $O : \mathbb{R}^d \to \mathbb{R}^H$ are all one-layer dense networks, σ denotes the sigmoid function, and $\boldsymbol{V}(e) = [V((e, 0)), V((e, 1)), \ldots, V((e, H-1))]^\top \in \mathbb{R}^H$, $N(e)$ is the set of vertices in expression e.

We train the above GNN to approximate the optimal value function $V^\star$. Let $\bar{V}(s) = \boldsymbol{V}(e_s)[t_s]$ the value function $\boldsymbol{V}$ computed for expression e_s and time budget t_s. To track an approximate lower bound of the optimal value function $V^\star$, we minimize the loss $l(\bar{V}(s) - V^{\text{target}}(s)/t_s)$. The target value $V^{\text{target}}(s)$ is defined in Algorithm 2 and corresponds to the best cost improvement obtained with the current policy in t_s steps. Normalization by t_s is introduced to ease optimisation by ensuring that target values for all outputs of $\boldsymbol{V}$ are in a similar magnitude. Thus the value function estimate $\hat{V}(s)$ can be obtained from per-step value estimate $\bar{V}(s)$ as $\hat{V}(s) = t_s \cdot \bar{V}(s)$. For the loss function l we use the Huber loss. Details about the optimiser used to minimize the loss l are given in Appendix B. In the pseudocode in Algorithm 1, this optimisation is represented with the function FIT.

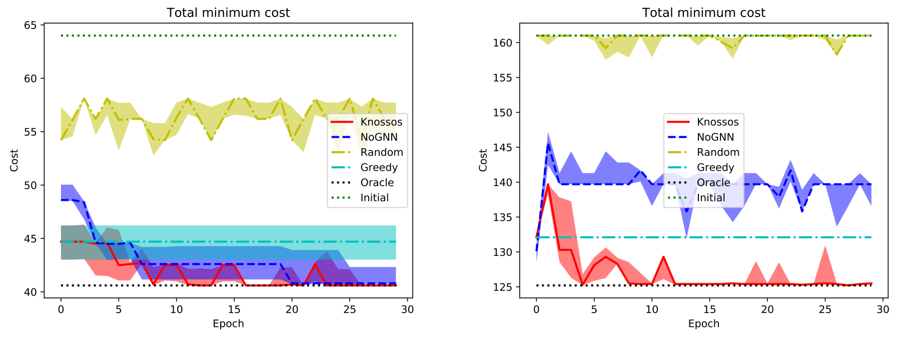
4 Related Work

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 (Steuwer et al., 2017) defines an intermediate language suited for expressing numerical computation, but focuses on providing the right set of rewrite rules rather than on the program optimisation process itself. In Section 5, 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



(a) Standard Compiler Mode (b) Generalization to Unseen Expressions

Figure 3: Performance of Knossos on a set of arithmetic expressions. Horizontal axis shows epochs, vertical axis shows cost. Shaded area spans the 20% and 80% percentile over 10 repetitions.

manually-prepared back-ends. Moreover, using them 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 in the most optimal way possible.

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. 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. Reinforcement Learning has also been applied to expression rewriting and scheduling problems (Chen & Tian, 2019). However, since this approach used actor-critic RL that does not exploit reset, it less well-suited for compilation tasks as described in Section 3.

5 Benchmarks

We evaluated Knossos in four settings. First, to understand how close and reliably we can achieve the best optimisation, we applied Knossos to a manually curated set of arithmetic expressions, where we know the best available sequence of rewrites. Second, we applied Knossos to a set of linear algebraic operations, which are representative of typical workloads in numerical computing. Third, we implemented a Gaussian Mixture Model (GMM) to asses how the code optimizer scales to more realistic tasks, including differentiation. Fourth, in order to reflect common modern use-cases, we evaluated Knossos on a computer vision task that uses a convolutional neural network. Since Knossos is as an alternative to traditional compliers, we compare it to a hand-written rule-based transpiler of the Knossos IL, which we call `ksc`. Both Knossos and `ksc` output C++ , which is compiled to binary using `gcc` *with optimisation enabled*, ensuring a fair comparison. We describe the results below.

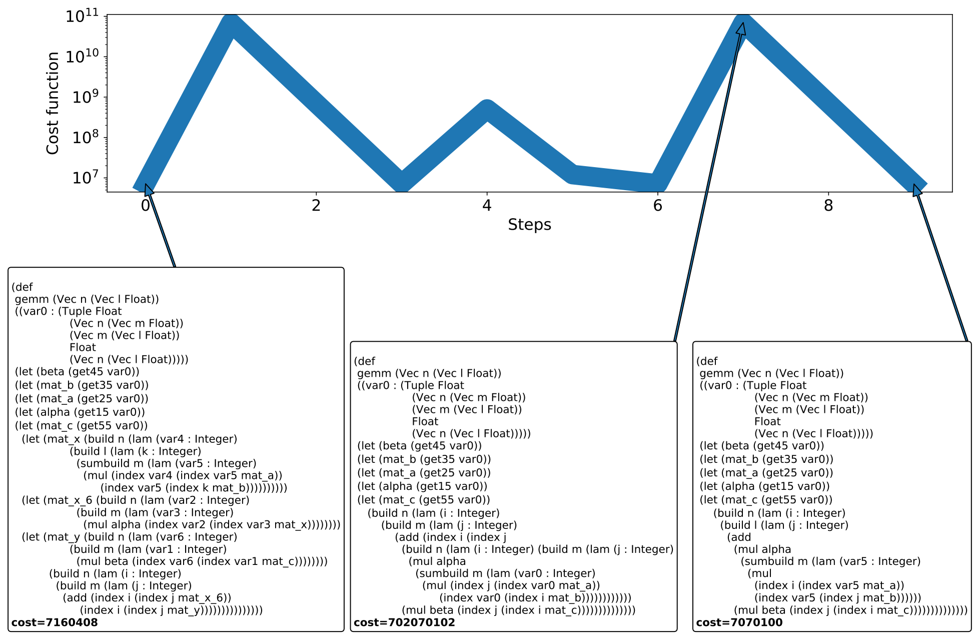
Optimality and Standard Complier Mode While optimising arithmetic expressions is a relatively simple task, it is attractive as a benchmark for evaluating optimality. Since we know the *best possible cost* for each expression, we can evaluate how fast Knossos is from this oracle value. Similarly to a traditional complier, where we are given a concrete program to optimize, the expressions used to evaluate Knossos in this benchmark were the same ones that we used used during training. Even in this setup, Knossos still generalises, but it does it across sub-expressions of the expressions in the training set. We tested that on 8 expressions, training for 40 epochs and running 10 repetitions for each experiment with different random seeds. Search depth was limited to 10 and the termination condition in $A^\star$ was set to 5000 evaluations of the value function. See Appendix B for the full details including network parameters. Figure 3a. It can be seen from the figure that Knossos achieved the best possible cost for all expressions. We also performed an ablation, comparing Knossos to $A^\star$ algorithm (shown as NoGNN) does not uses a simpler architecture for the value function, that does not perform the GNN recurrence in equation 4. As a baseline, we compared to greedy best-first search, which picks a next state to explore greedily without using the value function $f(s) := c(s_0) - c(s)$. We also show a comparison to random search and the initial cost of the expression, before any optimisation.

Generalization to Unseen Data While generalisation to unseen expressions isn't strictly required in a compiler, it is desirable because it means we can avoid retraining the complier for each new program. To test Knossos in this setting, we used a training set of 36 arithmetic expressions and a test set of 12 different ones. The details of the experimental setup are given in Appendix B. In this setting, we pick 6 expressions randomly from a training set to train in each epoch. We show the results in Figure 3b. It can be seen that Knossos (red line) outperforms the baselines and attained the oracle value. The minimum cost found for each expressions is shown in the appendix. (TODO: put performance of each expression)

Linear Algebra Primitives Numerical linear algebra is fundamental to most calculations in scientific computing and machine learning. Primitives such as vector multiplication, plane rotation, matrix multiplications and similar primitives often represent the most time-consuming part of the given computation. To evaluate the performance of Knossos on in this setting, we used a set of 15 such linear algebra primitives. We trained for 20 epochs, each of which included optimisation of cost of 6 primitives. Search depth was limited to 30 and the termination condition in $A^\star$ was set to 5000 evaluations of the value function. Figure 5a shows the total cost for 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 one standard deviation across the runs of Knossos. Results show that Knossos produced code of lower cost than the output of the traditional `ksc` complier according to our cost model. We also performed a benchmark using wall clock time, shown in Figure 6a, again showing an improvement. In addition, we performed a qualitative evaluation of the output. In Figure 4, we compare the difference in the intermediate representation of a program obtained by Knossos (right-hand listing) vs the `ksc` complier (left-hand listing). It can be seen by looking at the code that Knossos has discovered a form of loop fusion – the type of optimisation that previously had to be built into a complier by a laborious manual process.

Gaussian Mixture Model (GMM) 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 deal with the complex expressions produced by automatic differentiation. For this experiment, we fed Knossos with a source file containing an implementation of GMM split into 20 functions. We ran the training for 20 epochs. In each of them, we optimised the cost of one function. We fixed the search depth to 30. The termination condition in $A^\star$ was set to 30000 evaluations of the value function. We used an augmented set of training rules, which included vector operations. Because of the complexity of the task, we split the search into two phases of 15 steps each. The training phases differ in the set of allowed rules. In the first phase, we only allow rules that result in large changes to the cost. In the second phase, we allow all rules. Figure 5b shows the total cost over all the functions measured with our cost model, while figure 6b shows the reduction as measured by wall clock time. The Knossos optimizer produced code that runs faster than the `ksc` baseline.

Convolutional Network In order to evaluate Knossos on workloads characteristic of modern machine learning pipelines, we also evaluated Knossos on a computer vision task. We optimize a code for training a convolutional deep network on the MNIST dataset (LeCun, 1998). The source code represents a typical implementation of a deep learning algorithm and contains primitives such as

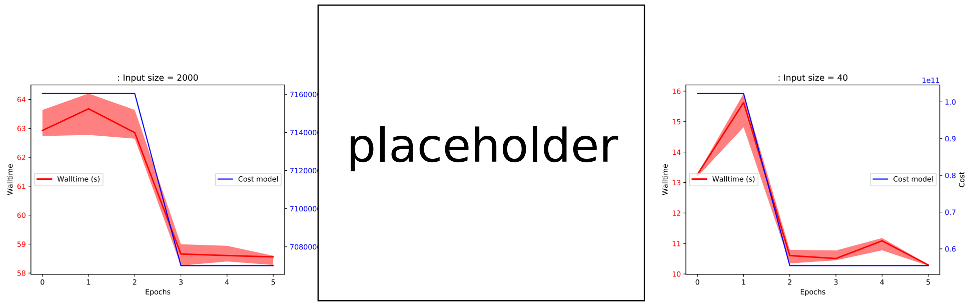


```
(def
 gemm (Vec n (Vec l Float))
 ((var0 : (Tuple Float
                 (Vec n (Vec m Float))
                 (Vec m (Vec l Float))
                 Float
                 (Vec n (Vec l Float)))))
 (let (beta (get45 var0))
 (let (mat_b (get35 var0))
 (let (mat_a (get25 var0))
 (let (alpha (get15 var0))
 (let (mat_c (get55 var0))
   (let (mat_x (build n (lam (var4 : Integer)
                 (build l (lam (k : Integer)
                   (sumbuild m (lam (var5 : Integer)
                     (mul (index var4 (index var5 mat_a))
                          (index var5 (index k mat_b))))))))))
   (let (mat_x_6 (build n (lam (var2 : Integer)
                   (build m (lam (var3 : Integer)
                     (mul alpha (index var2 (index var3 mat_x))))))))
   (let (mat_y (build n (lam (var6 : Integer)
                 (build m (lam (var1 : Integer)
                   (mul beta (index var6 (index var1 mat_c))))))))
         (build n (lam (i : Integer)
           (build m (lam (j : Integer)
             (add (index i (index j mat_x_6))
                  (index i (index j mat_y)))))))))))))))
cost=7160408
```

```
(def
 gemm (Vec n (Vec l Float))
 ((var0 : (Tuple Float
                 (Vec n (Vec m Float))
                 (Vec m (Vec l Float))
                 Float
                 (Vec n (Vec l Float)))))
 (let (beta (get45 var0))
 (let (mat_b (get35 var0))
 (let (mat_a (get25 var0))
 (let (alpha (get15 var0))
 (let (mat_c (get55 var0))
   (build n (lam (i : Integer)
     (build m (lam (j : Integer)
       (add (index i (index j
         (build n (lam (i : Integer) (build m (lam (j : Integer)
           (mul alpha
             (sumbuild m (lam (var0 : Integer)
               (mul (index j (index var0 mat_a))
                    (index var0 (index i mat_b)))))))))))))
         (mul beta (index j (index i mat_c))))))))))))))
cost=702070102
```

```
(def
 gemm (Vec n (Vec l Float))
 ((var0 : (Tuple Float
                 (Vec n (Vec m Float))
                 (Vec m (Vec l Float))
                 Float
                 (Vec n (Vec l Float)))))
 (let (beta (get45 var0))
 (let (mat_b (get35 var0))
 (let (mat_a (get25 var0))
 (let (alpha (get15 var0))
 (let (mat_c (get55 var0))
    (build n (lam (i : Integer)
      (build l (lam (j : Integer)
        (add
          (mul alpha
            (sumbuild m (lam (var5 : Integer)
              (mul
                (index i (index var5 mat_a))
                (index var5 (index j mat_b))))))
          (mul beta (index j (index i mat_c))))))))))))))
cost=7070100
```

Figure 4: General Matrix Multiply (GEMM) program rewrite sequence obtained by Knossos. The initial expression was obtained from our rule-based `ksc` compiler and shown in the bottom left. The final expression was obtained after 10 rewriting steps and shown in the bottom right. We can see that Knossos has achieved a form of loop fusion. The expression in the bottom middle corresponds to the highest point in the cost sequence.

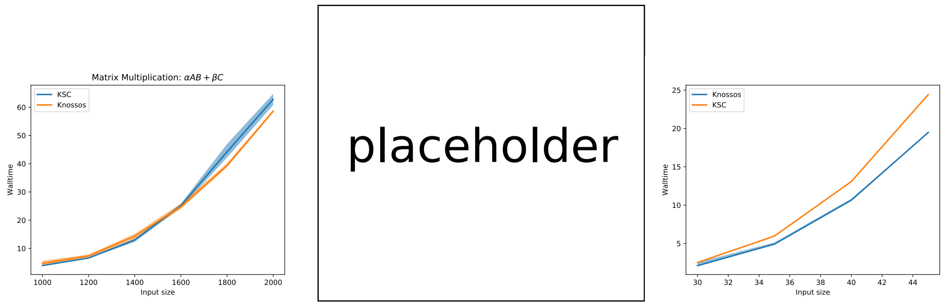


(a) Basic Linear Algebra (b) Gaussian Mixture Model (c) Convolutional Neural Network

Figure 5: Performance of Knossos on basic linear algebra, Gaussian Mixture Model and convolutional network. Shaded area indicates one standard deviation. **TODO: title of the figures.**

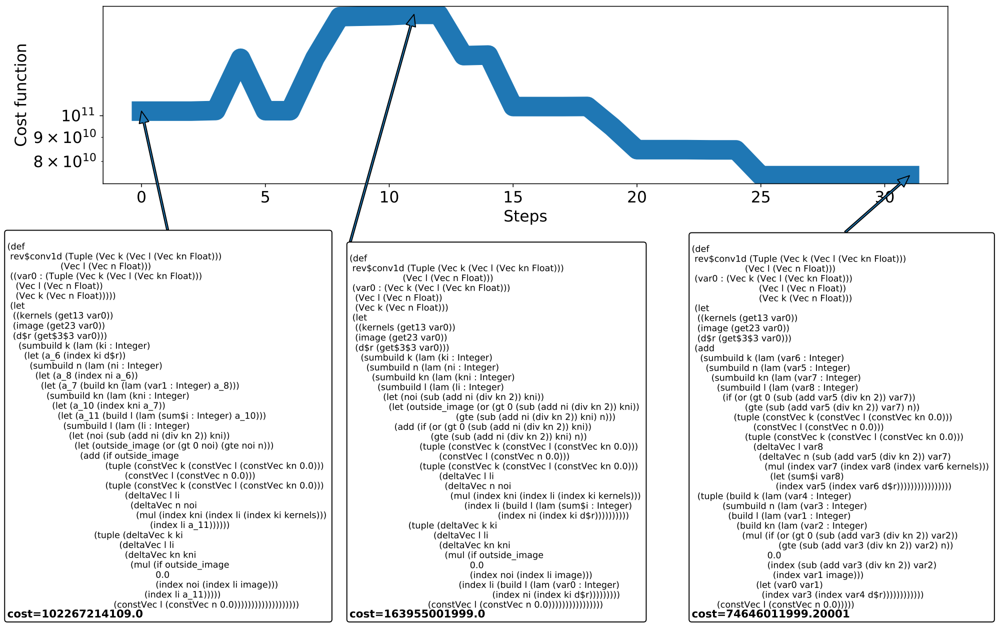
dense layers, convolutional layers, pooling layers, and so on. While MNIST is a basic benchmark, we stress that the goal of Knossos was *code optimisation* as opposed to the computer vision task itself. For optimisation, we used the same two phase strategy used for optimizing GMM. Results are shown in Figure 5c for the cost model and Figure 6c for the wall clock time. The shaded area represents the standard deviation across the runs of Knossos and the resulting binary. As above, the Knossos optimizer produced code that outperformed the baseline.

Summary of Benchmarks We have demonstrated that Knossos is capable of producing code that is faster than the output of a traditional complier. Moreover, unlike traditional compliers, Knossos does not rely on hand-crafted optimisation strategies that are very laborious to implement. In fact, in our benchmark of linear algebra primitives, Knossos was able to automatically discover *loop fusion*, an optimisation strategy long known to complier designers. Knossos code in our experiments can perform both training and inference and can be run on any hardware supporting the C++ toolchain, including inexpensive embedded devices.



(a) Basic Linear Algebra (b) Gaussian Mixture Model (c) Convolutional Neural Network

Figure 6: Comparison of wall time. Shaded area indicates one standard deviation.**TODO: Update results.**



```
(def
 rev$conv1d (Tuple (Vec k (Vec l (Vec kn Float)))
                  (Vec l (Vec n Float)))
 ((var0 : (Tuple (Vec k (Vec l (Vec kn Float)))
   (Vec l (Vec n Float))
   (Vec k (Vec n Float)))))
 (let
  ((kernels (get13 var0))
  (image (get23 var0))
  (d$r (get$3$3 var0)))
    (sumbuild k (lam (ki : Integer)
      (let (a_6 (index ki d$r))
        (sumbuild n (lam (ni : Integer)
          (let (a_8 (index ni a_6))
            (let (a_7 (build kn (lam (var1 : Integer) a_8)))
              (sumbuild kn (lam (kni : Integer)
                (let (a_10 (index kni a_7))
                  (let (a_11 (build l (lam (sum$i : Integer) a_10)))
                    (sumbuild l (lam (li : Integer)
                      (let (noi (sub (add ni (div kn 2)) kni))
                        (let (outside_image (or (gt 0 noi) (gte noi n)))
                          (add (if outside_image
                                  (tuple (constVec k (constVec l (constVec kn 0.0)))
                                         (constVec l (constVec n 0.0)))
                                  (tuple (constVec k (constVec l (constVec kn 0.0)))
                                         (deltaVec l li
                                           (deltaVec n noi
                                             (mul (index kni (index li (index ki kernels)))
                                                  (index li a_11))))))
                            (tuple (deltaVec k ki
                                     (deltaVec l li
                                       (deltaVec kn kni
                                         (mul (if outside_image
                                                  0.0
                                                  (index noi (index li image)))
                                              (index li a_11)))))
                                   (constVec l (constVec n 0.0))))))))))))))))))))
cost=102267214109.0
```

```
(def
 rev$conv1d (Tuple (Vec k (Vec l (Vec kn Float)))
                  (Vec l (Vec n Float)))
 (var0 : (Vec k (Vec l (Vec kn Float)))
  (Vec l (Vec n Float))
  (Vec k (Vec n Float)))
 (let
  ((kernels (get13 var0))
  (image (get23 var0))
  (d$r (get$3$3 var0)))
    (sumbuild k (lam (ki : Integer)
      (sumbuild n (lam (ni : Integer)
        (sumbuild kn (lam (kni : Integer)
          (sumbuild l (lam (li : Integer)
            (let (noi (sub (add ni (div kn 2)) kni))
              (let (outside_image (or (gt 0 (sub (add ni (div kn 2)) kni))
                                      (gte (sub (add ni (div kn 2)) kni) n)))
                (add (if (or (gt 0 (sub (add ni (div kn 2)) kni))
                             (gte (sub (add ni (div kn 2)) kni) n))
                         (tuple (constVec k (constVec l (constVec kn 0.0)))
                                (constVec l (constVec n 0.0)))
                         (tuple (constVec k (constVec l (constVec kn 0.0)))
                                (deltaVec l li
                                  (deltaVec n noi
                                    (mul (index kni (index li (index ki kernels)))
                                         (index li (build l (lam (sum$i : Integer)
                                                         (index ni (index ki d$r))))))))))
                     (tuple (deltaVec k ki
                              (deltaVec l li
                                (deltaVec kn kni
                                  (mul (if outside_image
                                           0.0
                                           (index noi (index li image)))
                                       (index li (build l (lam (var0 : Integer)
                                                       (index ni (index ki d$r))))))))
                            (constVec l (constVec n 0.0))))))))))))))))
cost=163955001999.0
```

```
(def
 rev$conv1d (Tuple (Vec k (Vec l (Vec kn Float)))
                  (Vec l (Vec n Float)))
 (var0 : (Vec k (Vec l (Vec kn Float)))
                       (Vec l (Vec n Float))
                       (Vec k (Vec n Float)))
 (let
  ((kernels (get13 var0))
  (image (get23 var0))
  (d$r (get$3$3 var0)))
 (add
   (sumbuild k (lam (var6 : Integer)
     (sumbuild n (lam (var5 : Integer)
       (sumbuild kn (lam (var7 : Integer)
         (sumbuild l (lam (var8 : Integer)
           (if (or (gt 0 (sub (add var5 (div kn 2)) var7))
                   (gte (sub (add var5 (div kn 2)) var7) n))
               (tuple (constVec k (constVec l (constVec kn 0.0)))
                      (constVec l (constVec n 0.0)))
                   (tuple (constVec k (constVec l (constVec kn 0.0)))
                      (deltaVec l var8
                        (deltaVec n (sub (add var5 (div kn 2)) var7)
                          (mul (index var7 (index var8 (index var6 kernels)))
                            (let (sum$i var8)
                              (index var5 (index var6 d$r)))))))))))))))))
  (tuple (build k (lam (var4 : Integer)
           (sumbuild n (lam (var3 : Integer)
             (build l (lam (var1 : Integer)
                (build kn (lam (var2 : Integer)
                  (mul (if (or (gt 0 (sub (add var3 (div kn 2)) var2))
                               (gte (sub (add var3 (div kn 2)) var2) n))
                           0.0
                           (index (sub (add var3 (div kn 2)) var2)
                             (index var1 image)))
                       (let (var0 var1)
                         (index var3 (index var4 d$r))))))))))))
         (constVec l (constVec n 0.0)))))
cost=74646011999.20001
```

Figure 7: Reverse mode of convolutional layer rewrite sequence obtained by Knossos. The initial expression was obtained from our rule-based `ksc` compiler and shown in the bottom left. The final expression was obtained after 32 rewriting steps and shown in the bottom right. The expression in the bottom middle corresponds to the highest point in the cost sequence.

6 Conclusions

We have introduced Knossos, a new complier targetting machine learning and numerical computation. Thanks to its automatic code optimisation, Knossos produces binaries that achieve better run-times than a traditional, rule-based complier. Knossos can deal with complex code generated by automatic differentiation and automatically discover optimisations that previously required careful complier design. We believe that Knossos will pave the way towards a new generation of future compliers, which will crucially rely on automatically inferring the correct optimisations.

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.

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.

Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. *NeurIPS*, abs/1810.00337, 2019. URL `http://arxiv.org/abs/1810.00337`.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

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

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.

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.

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.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pp. 74–85, 2017. URL `http://dl.acm.org/citation.cfm?id=3049841`.

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 KNOSSOS INTERMEDIATE REPRESENTATION

For background, we give a brief overview of the intermediate representation (IR) used by the Knossos complier. the Knossos IR provides a convenient symbolic form for the reinforcement learning optimizer. It also has a LISP-like surface syntax, which we used to implement our programs. In the future, we plan to provide transpilers, allowing for the compilation of code written in other languages into Knossos. We provide a sample Knossos program in Figure 4.In order to facilitate Machine Learning workloads, the Knossos IL has native support for automatic differentiation. We use a new unified view of automatic differentiation as generalised transposition (Elliott, 2018). 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 rewrite 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 rewrites 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 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 in the sense that taking gradients can be applied to any piece of Knossos code.

While the details of this process are beyond the scope of this paper, from the perspective of this work, the important feature of AD is that it corresponds to a transformation of the abstract syntax tree. The resulting AST can then be optimised in the same way as any other code.

APPENDIX B REPRODUCIBILITY AND DETAILS OF EXPERIMENTAL SETUP

We now describe the parameters used to perform the experiments reported on in the paper. The parameters used by $A^{\star}$ in the four tasks described in Sec. 5 are listed in Tab. 1. The hyper-parameters for the value network training are given in Tab. 2.

In the Graph Neural Network, initial node features are one-hot vectors that represent the node types. The used node types are: constant, variable, let, if, tuple, select, +, -, *, /, exp, log, ==, >, >=, or, build, apply, lam, sum, sumbuild, constVec, and deltaVec. Edge types are listed in Tab. 2b. The auxiliary edge type"is-identical" is inserted to identify identical subexpressions. It was added so that it is easier to learn re-writes that rely on matching expressions.

TODO: cite paper using matching-edge

. The GNN was implemented using a sparse adjacency matrix instead of dense matrix in order to conserve GPU memory in settings where some expressions grow beyond > 10000 nodes during training. We ran the GNN recursion 10 times. For optimization we used the Adam optimizer with learning rate 0.0001 and set the dropout rate zero for GNN and 0.2 for the MLP.

APPENDIX C REWRITE RULES

We list the basic rule set used in the arithmetic expressions benchmark in Tab. 3. The additional rewrite rules used in basic linear algebra, Gaussian mixture model, and convolutional neural network are given in Tab. 4.

APPENDIX D 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). In order to disambiguate across subtly different versions of the algorithm, we describe it below.

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

1. Selection: Starting from the root, a tree policy is recursively descends through the tree until it reaches a leaf node.
2. Expansion: A child node is added to expand the tree.

Table 1: List of Experimental Settings

Arithmetic Expressions	
Parameter	**Value**
Number of training expressions	36
Number of test expressions	12
Rewrite rules	Simple rule set
Number of evaluation budget per epoch	5000
Minimum batch size for evaluation	16
Maximum Search Depth	10
Basic Linear Algebra	
Number of training expressions	TODO
Number of test expressions	4
Rewrite rules	Extended rule set
Number of evaluation budget per epoch	30000
Minimum batch size for evaluation	16
Maximum Search Depth	30
Gaussian Mixture Model	
Number of training expressions	TODO
Number of test expressions	TODO
Rewrite rules	Extended rule set
Number of evaluation budget per epoch	30000
Minimum batch size for evaluation	16
Maximum Search Depth	30
Convolutional Neural Network	
Number of training expressions	TODO
Number of test expressions	TODO
Rewrite rules	Extended rule set
Number of evaluation budget per epoch	30000
Minimum batch size for evaluation	16
Maximum Search Depth	30

Table 2: List of hyperparameters

Value Function	
Model Parameter	
Number of features per node	200
Number of propagation steps	10
Activation function for GNN	tanh
Dropout rate for GNN	0
Number of hidden layers in MLP	2
Number of dimensions of hidden layers in MLP	200
Activation function for MLP	tanh
Dropout rate for MLP	0.2
Training Parameter	
Loss function	Huber loss
Optimizer	Adam optimizer
Learning rate	0.0001
Batch size	16

Table 3: List of rewrite rules used for arithmetic expressions.

	Simple Rule Set	
LHS	**RHS**	**Side conditions**
	Arithmetic Rules	
(add 0.0 a)	a	
(mul 1.0 a)	a	
(sub a 0.0)	a	
(div a 1.0)	a	
(add a (mul b -1.0))	(sub a b)	
(mul a b)	(mul b a)	
(add a b)	(add b a)	
(sub a a)	0.0	
(add (add a b) c)	(add (add a c) b)	
(sub (add a b) c)	(add (sub a c) b)	
(add (sub a b) c)	(sub (add a c) b)	
(sub (sub a b) c)	(sub (sub a c) b)	
(sub a (sub b c))	(sub (add a c) b)	
(mul (mul a b) c)	(mul (mul a c) b)	
(div (mul a b) c)	(mul (div a c) b)	
(mul (div a b) c)	(div (mul a c) b)	
(div (div a b) c)	(div (div a c) b)	
(div a (div b c))	(div (mul a c) b)	
(div (div a b) c)	(div a (mul b c))	
(mul a (add b c))	(add (mul a b) (mul a c))	
(add (mul a b) (mul a c))	(mul a (add b c))	
(mul a (sub b c))	(sub (mul a b) (mul a c))	
(add a (mul -1.0 b))	(sub a b)	
(sub (mul a b) (mul a c))	(mul a (sub b c))	
(add (div y x) (div z x))	(div (add y z) x)	
(sub (div y x) (div z x))	(div (sub y z) x)	
(div a (mul b c))	(div (div a b) c)	
(mul 0.0 a)	0.0	
	Binding Rules	
e	(let (x e) x)	
(op (let (x e) x))	(let (x e) (op x))	x ∉ op
(let (x e1) (let (x e1) e2))	(let (x e1) e2)	
(let (x e1) e2)	(let (x e1) e2[x/e1])	replace all free x in e2 with e1
(let (x e1) e2)	e2	x ∉ e2

Table 4: List of rewrite rules used in addition to simple rule set (Table 3) for Basic Linear Algebra, Gaussian Mixture Model, and Convolutional Neural Network benchmarks. In the two-phase strategy described in Sec. 5, only the rules with checkmarks in the last column are used for the first 15 epochs.

	Extended Rule Set		
LHS	**RHS**	**Side conditions**	**Include in phase 1**
(size (build n (lam x b)))	n		
(index arg (build sz (lam x body)))	(let (x arg) body)		✓
(select i (tuple e1 e2 e3...))	ei		
(sum (build n (lam x b)))	(sumbuild n (lam x b))		✓
(sumbuild n (lam x (deltaVec m i v)))	(deltaVec m i (sumbuild n (lam x v)))	x∉ m ∩ x ∉ i.	✓
(sumbuild n (lam x (tuple f g)))	(tuple (sumbuild n (lam x f)) (sumbuild n (lam x g)))		✓
(sumbuild n (lam x (if c t f)))	(if c (sumbuild n (lam x t)) (sumbuild n (lam x f)))		✓
(sumbuild n (lam x c))	(mul n c)	x ∉ c	✓
(sumbuild o (lam oi (add e1 e2)))	(add (sumbuild o (lam oi e1)) (sumbuild o (lam oi e2)))		✓
(sumbuild o (lam oi (build n (lam ni e))))	(build n (lam ni (sumbuild o (lam oi e))))		✓
(add (deltaVec o oi e1) (deltaVec o oi e2))	(deltaVec o oi (add e1 e2))		✓
(add (tuple x y) (tuple a b))	(tuple (add x a) (add y b))		✓
(mul (tuple x y) z)	(tuple (mul x z) (mul y z))		✓
(if True a b)	a		
(if False a b)	b		
(if p True False)	p		
(if p a a)	a		
(eq a a)	True		

Algorithm 3 Monte Carlo Tree Search (MCTS)

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

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

3. Simulation: A rollout policy is applied from the new node until the end of the episode.
4. Back-up: The simulation result is backed up through the selected nodes to update their statistic.

The tree policy π_t and rollout policy π_r are defined as follows.

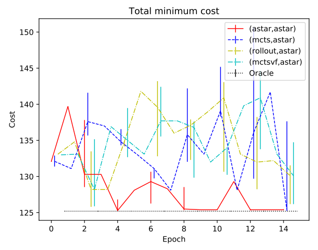
$$\pi_t(a|s) = \arg\max_{a \in A}\left(\frac{X(s')}{n(s')} + \beta\sqrt{\frac{\ln n(s)}{n(s')+1}}\right) \tag{6}$$

$$\pi_r(a|s) = \text{softmax}_{a \in A}(R(s,a) + V(s'), \alpha) \tag{7}$$

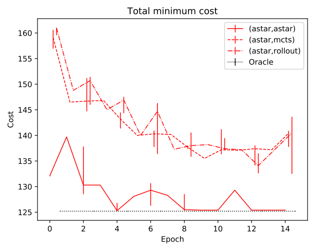
Here, $n(s)$ is a visitation count of state s, and β is a constant to control the exploration bonus. $X(s')/n(s')$ is the average cost reduction achieved by a set of trajectories which passed through s' in M: $X(s') = \sum_{T \in H : s' \in T} \max_{s_d \in T : d(s_d) \geq d(s')} c(s_0) - c(s_d)$, and M 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 ($\beta\sqrt{\frac{\ln n(s)}{n(s')+1}}$) is reduced. This way, the agent is encouraged to try a diverse set of actions.

We evaluated the performance of $A^\star$ search and MCTS for both training and test. The experimental setup is the same as the *Generalisation to Unseen Data* experiment in Section 5 except for the used search algorithm. For MCTS, we used $\alpha = 5.0$ and $\beta = 0.5$ for both training and test.

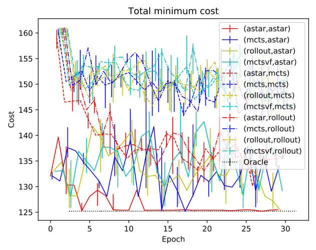
Figure 8a shows the results of running all possible combinations of search algorithms when used for training and test in various configurations. Overall, using $A\star$ for both training and test achieved the best performance. In particular, when we fixed the algorithm used during test to $A^\star$ and varied



(a) Generalization (Train)



(b) Generalization (Test)



(c) Generalization

Figure 8: Comparison of $A\star$ search and Monte Carlo Tree Search.

the training algorithm between $A^\star$ and MCTS, $A\star$ achieved a significantly lower the total minimum cost than MCTS. Similarly, when we fixed the algorithm used for training to $A\star$ and compared the performance during testing $A\star$ achieved significantly lower cost than MCTS again.