Model-based Reinforcement Learning in Computer Systems

Sean J. Parker
Clare Hall

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: sjp240@cam.ac.uk

May 9, 2021

Declaration

I, Sean J. Parker of Clare Hall, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 0

Signed:

Date:

Acknowledgements

Abstract

This project investigates the use of model-based reinforcement learning (RL) in the domain of computer systems, specifically, that of optimising deep learning models by applying transformations to the computation graph to minimise the runtime cost on hardware devices. Recent work has aimed to apply reinforcement learning to computer systems with some success, especially with using model-free RL techniques. However, more recently, model-based methods has seen an increased focus of research as model-based reinforcement learning can learn a model of the environment, such that an agent can take actions inside the learned world-model to train more efficiently; environment rollouts can occur safely in parallel and, especially in systems environments, it circumvents the possible latency impact of stepping a system environment that can take orders of magnitude longer to perform an action compared to a video game emulator for example. This dissertation examines both the prior work for optimising deep learning models and the applicability of reinforcement learning to the problem.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertations key contributions are:

- Applied modern reinforcement learning approaches to tackle a recent problem common in all machine learning frameworks to reduce the significant engineering time required to manually design, implement and test optimisation rules for computation graphs.
- Implemented a model-based RL agent and environment that are used to directly optimising the structure of computation graphs to reduce real-world runtime and compare performance to current baseline results.
- This work, to the best of our knowledge, is the first that has applied model-based reinforcement learning to optimising the structure of computation graphs.

The rest of the dissertation is structured as follows. Chapter 2 provides a background for computation graphs and the representation of deep learning models, reinforcement learning—both model-free and model-based—in the context of computer systems. Chapter 3 concretely introduces the problem we are trying to solve and provides baselines produced by prior works and formulate the problem in the context of reinforcement learning. Chapter 4 describes our approach to applying reinforcement learning to choose substitutions to optimise the computation graphs as well as learning an accurate

model of the environment. Chapter 5 covers the evaluation setup, our experiments and results for different methodologies. Finally, in chapter 6 we conclude the dissertation with a summary of our findings and discuss potential future work.

Chapter 2

Background and Related Work

2.1 Introduction to Deep Learning Models

This section discusses the way in which machine learning models are represented for efficient execution on physical hardware devices. First, we discuss how the mapping of tensor operations to computation graphs is performed followed by an overview of recent approaches that optimise computation graphs to minimise execution time.

Over the past decade, there has been a rapid development of various deep learning architectures that aim to solve a specific task. Common examples include convolutional networks (popularised by AlexNet then ResNets, etc), transformer networks that have seen use in the modelling and generation of language. Recurrent networks that have shown to excel at learning long and short trends in data.

The fundamental building blocks of deep learning models have remained largely unchanged. As the networks become more complex, it becomes tedious to manually optimise the networks to reduce the execution time on hardware. Therefore, there is extensive work in ways to both automatically optimise the models, or, alternatively apply a set of hand-crafted optimisations.

Computation graphs are a way to graphically represent both the individual tensor operations in a model, and the connections (or data-flow) along the edges between nodes in the graph. Figure 2.1 shows how the expression, $y = \texttt{ReLU}(\mathbf{w} \cdot \mathbf{x} + b)$, can be represented graphically in a computation graph.

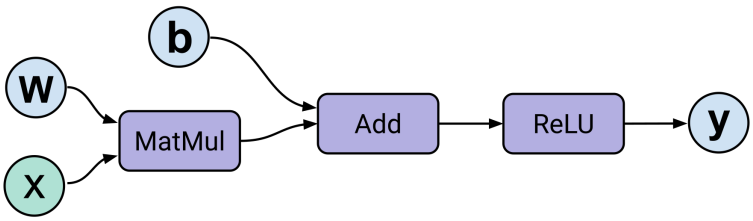


Figure 2.1: The operations shown in purple are the nodes of the computation graph which take an arbitrary number of inputs, performs a computation at the node and produces an output. The blue nodes represent the input nodes for tensors. The directed edges show the flow of tensors through the graph.

Similarly, the whole model can be converted into a stateful dataflow (computation) graph in this manner. Using a computation graph as an intermediate representation it provides two key benefits compared to using a raw model definition. First, we can execute the model on any hardware device as the models have a single, uniform representation that can be modified as required. Secondly, it allows for pre-execution optimisations based on the host device, for example, we may perform different optimisations for executing on a GPU compared to a TPU requires different data layouts and optimisations.

2.1.1 Current approaches to optimising deep learning models

Due to the prevalence and importance of machine learning, especially deep networks, there is a focus on finding ways decrease the inference runtime and by extension, increasing the model throughput. All major frameworks such as TensorFlow [1], PyTorch [2], MXNet [3], and Caffe [4] have some level of support for performing pre-execution optimisations. However, the process of performing such optimisations is often time-consuming and cannot

be completed in real-time. Rather, it is common to use a deep learning optimisation library such as cuDNN [5] or cuBLAS [6] that instead directly optimise individual tensor operations.

Alternatively, TVM [7] and TensorRT [8] can be used to optimise deep learning models and offer greater performance gains compared to the more commonly used frameworks such as TensorFlow and PyTorch. They also use greedy rule-based optimisation approaches. TODO - either expand or remove

Rather than using a rule-based optimisation approach, it is possible to use more sophisticated algorithms to optimise deep learning models at the expense of computation time. Jia et al. used a cost-based backtracking search to iteratively search through the state space of possible graphs that are provably equivalent [9]. As opposed to using rule-based optimisation that applied hand-crafted optimisations, TASO generates the candidate subgraphs automatically and formally proves the transformations are equivalent using an automated theorem prover. Furthermore, Jia et al. showed that by jointly optimising both the data layout of the subgraph transformation, and the transformation itself, TASO achieves a speedup compared to performing the operations sequentially.

A key benefit of using a cost-based approach is that the search can take into account far more complex interactions between the transformed kernels. For example, if we apply a series of transformations $T_1, \cdots, T_i$, the runtime may increase, and, due to the first set of transformations, we can now apply $T_{i+1}, \cdots, T_j$, after all transformations have been applied, it is possible that we see an overall decrease in runtime. By increasing the search space of transformations in this way, Jia et al. showed that it is possible to increase the runtime of deep learning models by up to 3x [9, 10] compared to baseline measurements using various deep learning compilers [5, 6, 8].

2.2 Reinforcement Learning

Reinforcement learning (RL) is a sub-field in machine learning, broadly, it aims to compute a control policy such that an agent can maximise its cumulative reward from the environment. It has powerful applications in environments where a model that describes the semantics of the system are not available and the agent must itself discover the optimal strategy via a reward signal.

Formally, RL is a class of learning problem that can be framed as a Markov decision processes (MDP) when the MDP that describes the system is not known [11]; they are represented as a 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_a, \mathcal{R}_a, \rho_0 \rangle$ where:

- $\mathcal{S}$, is a finite set of valid states
- $\mathcal{A}$, is a finite set of valid actions
- $\mathcal{P}_a$, is the transition probability function that an action a in state s_t leads to a state s'_{t+1}
- $\mathcal{R}_a$, is the reward function, it returns the reward from the environment after taking an action a between state s_t and s'_{t+1}
- ρ_0, is the starting state distribution

We aim to compute a policy, denoted by π, that when given a state $s \in \mathcal{S}$, returns an action $a \in \mathcal{A}$ with the optimisation objective being to find a control policy π^* that maximises the *expected reward* from the environment defined by 2.1. Importantly, we can control the 'far-sightedness' of the policy by tuning the discount factor $\gamma \in [0, 1)$. As γ tends to 1, the policy will consider the rewards further in the future but with a lower weight as the distant expected reward may be an imperfect prediction.

$$\pi^* = \arg\max_{\pi} \ \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \ \mathcal{R}_t\right] \tag{2.1}$$

Classic RL problems are formulated as MDPs in which we have a finite state

space, however, such methods quickly become inefficient with large state spaces that we consider with applications such as Atari and Go. Therefore, we take advantage of modern deep learning function approximators, such as neural networks, that makes learning the solutions far more efficient in practise. We have seen many successfully applications in a wide range of fields, for example, robotic control tasks [12], datacenter power management, device placement, and, playing both perfect and imperfect information games to a super-human level. Reinforcement learning excels when applied to environments in which actions may have long-term, inter-connected dependencies that are difficult to learn or model with traditional machine learning techniques.

In the following sections we discuss the two key paradigms that exist in reinforcement learning and the current research in both areas and the application to systems tasks.

2.2.1 Model-Free and Model-Based RL

Model-free and model-based are the two main approaches to reinforcement learning, however, with recent work such as [13, 14, 15], the distinction between the two is becoming somewhat nebulous; it is possible to use a hybrid approach that aims to improve the sample efficiency of the agent by training model-free agents directly in the imagined environment.

The major branching point that distinguishes between model-free and model-based approaches is in what the agent learns during training. A model-free agent, in general, could learn a governing policy, action-value function, or, environment model. On the other hand, model-based agents commonly either learn an explicit representation of the parameterised policy π_θ using planning, such as AlphaZero [16] or ExIt [17]. Alternatively, we can use use data augmentation methods to learn a representation of the underlying environment behaviour, and either only use fictitious model, or augment real experiences to train an agent in the domain [14, 18, 19].

Understandably, a relevant question is why one would prefer a model-free over model-based approach and what are the benefits of the respective methods. The primary benefit of model-based RL is that is has far greater sample efficiency, meaning, the agent requires in total, less interactions with the real environment than the model-free counterparts. If we can either provide, or learn, a model of the environment it allows the agent to plan ahead, choosing from a range of possible trajectories its actions to maximise its reward. The agent that acts in this "*imagined*" or "*hallucinogenic*" environment can be a simple MLP [20] to a model-free agent trained using modern algorithms such as PPO, A2C or Q-learning. Further, training an agent in the world model is comparatively cheap, especially in the case of complex systems environments where a single episode an be on the order of hundreds of milliseconds.

Unfortunately, learning a model of the environment is not trivial. The most challenging problem that must be overcome is that if the model is imperfect, the agent may learn to exploit the models deficiencies, thus the agent fails to achieve a higher performance in the real environment. Further, learning an invalid world model can lead to the agent performing actions that may be invalid in an environment with state-dependent actions.

Model-based approaches have been successfully applied in various domains such as board games, video games, systems optimisation and robotics. Despite the apparent advantages of model-based RL with regards to reduced computation time, model-free reinforcement learning is by far the most popular approach and massive amounts of compute, typically by distributed training on clusters of GPUs/TPUs, is required overcome the sample inefficiency of model-free algorithms.

2.2.2 World Models

World models, first introduced by Ha and Schmidhuber [20], motivated and described an approach to model-based reinforcement learning in which we learn a model of the real environment using function approximators and train an agent using only predictions from the world model. Figure 2.2 shows the

design to utilise a world model as substitute for the real environment. In practise, a world model can be broken down into three main components. A visual model, V, that encodes the input into a latent vector z, a memory model, M the integrates the historical state to produce a representation that can be used as planning for future actions and rewards. Finally, a controller, C that uses both V and M to predict an action from the action set, $a \in \mathcal{A}$.

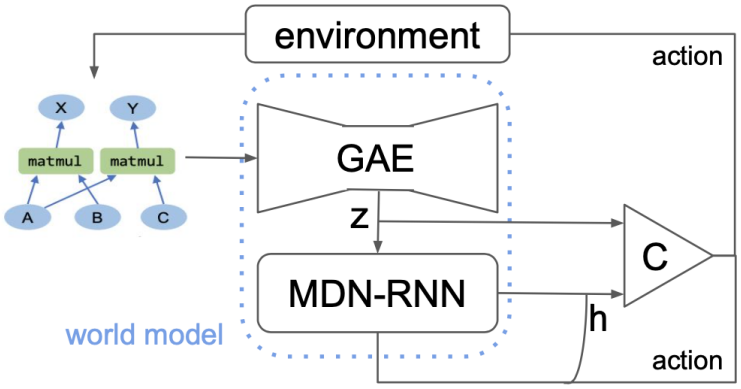


Figure 2.2: Diagrammatic representation of a world-model made up from an encoder (V) that transforms the input into latent space, a 'memory' module, M, that learns the behaviour of the environment from the latent vector z and a controller, C, that is trained using the latent vector of the encoder and the output features from the memory to choose an action which is either applied to the real or imagined environment. Figure adapted from [20].

Typically, a world model is trained using rollouts of the real environment that have been sampled using a random agent acting in the environment. The aim is to learn to accurately predict, given a state s_t, the next state s_{t+1} and the associated reward r_{t+1}. After training, the controller, C, can either learn using only observations from the world model, so called "training in a dream". Alternatively, the world model can be used to augment the observations from the real environment samples or used only for planning. To construct the world model, if the environment is simple and deterministic, it is possible to use a deep neural network to act as the world model, however, for environments that are only partially observable, a more complex model is required such as Recurrent Neural Networks (RNNs) [21, 22] or Long-short term memory (LSTMs) [23, 24]. The following two sub-sections describe the

fundamental concepts required to construct a world model.

Mixture Density Networks

Mixture Density Networks (MDNs) are a class of neural networks first described by Christopher Bishop [25] that were designed to deal with problems where there is an inherent uncertainty in the predictions. Given an input to the network, we wish to output a range of possible outputs conditioned on the input where we can assess the probability of each outcome. MDNs are commonly parameterised by a neural network that is trained using supervised learning and outputs the parameters for multiple mixture of Gaussians.

Recurrent Neural Networks

Recurrent Neural Networks are class of neural networks that allows for previous outputs to be re-used as inputs to sequential nodes while maintaining and updating their own hidden state. Primarily, RNNs are commonly used in the field of speech recognition and natural language processing as they can process inputs of an arbitrary length with a constant model size. In practise however, RNNs suffer from being unable to utilise long chains of information due to the vanishing/exploding gradient problem; the gradient can change exponentially changing in proportion to the number of layers in the network [22].

Motivated by the desire to overcome the limitations of RNNs, Hochreiter et al. [23] developed long-short term memory by describing Constant Error Carousel (CECs). The idea was further improved by Gers et al. [24] with the modern LSTM that is made up of four gates, each with a specific purpose that influences the behaviour of each cell and in combination, the properties of the network as a whole. Figure 2.3 shows the internal structure of an LSTM module.

An LSTM can be described used four "gates", where a gate influences a property of the behaviour of the LSTM cell. The *forget* gate dictates if

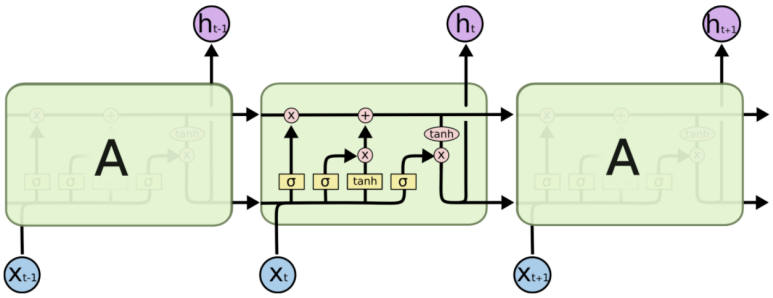


Figure 2.3: LSTM

the information stored in the cell should be erased by observing the inputs $[h_{t-1}, x_t]$ it outputs a value in the range $[0, 1]$, using the sigmoid function σ, where 1 means to completely forget the state. Secondly, the *input* gate calculates the new information to be stored in the cell state, generating a vector of candidates $\tilde{C}_t$. The *update* gate is to determine how much of the past state sequence should be considered using the outputs from the *forget* gate, the prior state C_{t-1} and the input gate $\tilde{C}_t$. Finally, the *output* gate determines the LSTM cell output based on the current, filtered state of the cell.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad \text{(1) Forget gate}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad \text{(2) Input gate}$$
$$\tilde{C}_t = \tanh(W_C) \cdot [h_{t-1}, x_t] + b_C) \quad \text{(3) Candidate value}$$
$$C_t = f_t C_{t-1} + i_t \tilde{C}_t \quad \text{(4) Update previous cell state}$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{(5) Output gate}$$
$$h_t = o_t \cdot \tanh(C_t) \quad \text{(6) Hidden state}$$

There are a number of popular varients of LSTM cells such as peephole LSTMs, GRUs, Grid LSTMs and ConvLSTM. There are many areas which have been revolutionised by the usage of LSTM cells in the network architecture. As we will describe in [TODO: ref chapter], LSTM cells are a key to allow world models to learn to simulate the behaviour of the environment

state-action transitions.

2.3 Graph Neural Networks

Graph neural network are a class of neural network that has seen considerable focus in recent years, with many successful applications being devised around the central idea of leveraging the structure of the graph input to aid in predicting attributes about the graph itself. The motivation factor for the use of graph networks is that, similar to the way in which convolutional neural networks revolutionised the application of neural networks to high dimensional inputs with images, video and audio—we desire an efficient way to generalise a similar idea onto graphs to take advantage of the inductive biases.

It is often difficult to model real-world problems in a way which we can train models to take advantage of the underlying structure. Much of the data generated by real-world systems and dynamics can be modelled easily in graph form; social networks, molecules, proteins, physical systems and text all exhibit a graph structure that can be leveraged. We point the reader to the survey performed by Zhou et al. [26] for an excellent overview of the methods and applications of graph neural networks.

Battaglia et al. [27] define a generalisable framework for entity/relation based reasoning with three main operators that act on edges, nodes, and on global features using user-defined functions. Within the framework described by Battaglia et al., a graph is defined as $G = (u, V, E)$ where u are the global attributes, $V = \mathbf{v}_{ii} = 1 : N^v$ is the set of vertices (with a cardinality of N^v) and finally, $E = (\mathbf{e}_k, r_k, s_k)_k = 1 : N^e$ is the set of edge attributes with their

sources and corresponding vertices.

Algorithm 1: Computation in a full GN block. Adapted from [27]

for $k \in \{1 \dots N^e\}$ **do**
  $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$
end
for $i \in \{1 \dots N^n\}$ **do**
  let $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$
  $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\to v}(E'_i)$
  $\bar{\mathbf{v}}'_i \leftarrow \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$
end
let $V' = \mathbf{v}'_{i=1:N^v}$
let $E' = (\mathbf{e}'_k, r_k, s_k)_{k=1:n^e}$
$\bar{\mathbf{e}}' \leftarrow \rho^{e\to u}(E')$
$\bar{\mathbf{v}}' \leftarrow \rho^{v\to u}(V')$
$\bar{\mathbf{u}}' \leftarrow \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$
return $(E', V', \mathbf{u}')$

We can define three update functions and three *aggregation* function. The update functions are ϕ^e, ϕ^v and ϕ^u for edges, vertices and globals respectively. The aggregation functions are $\rho^{e\to v}(E'_i)$, $\rho^{e\to u}(E')$, and $\rho^{v\to u}(V')$, for edges, vertices and globals respectively. To perform a single update given a set of input edges and vertices, we simply apply the three update and aggregation functions sequentially in the order of edges, vertices, then globals. Algorithm 1 describes, in general, the algorithm to perform an update of a graph block.

2.4 Related Work

Rule-based optimisation of computation graphs is the strategy by which we transform an input graph to alter its performance characteristics. Rule-based approaches such as TensorFlow [1] and TVM [7] used a pre-defined set of transformations that are applied greedily. We evaluated our approach against these traditional approaches in section [todo: ref + extend]. In addition, recent work, such as [10, 9] automatically search for transformations to apply to the input graph with the modification that we allow performance decreasing transformations. Their work is similar to our

approach, as we use the same automated method to discover and verify the operator transformations in an offline manner—prior to optimisation of the models. We also compare our work to TASO [9] as it is most similar in terms of substitution discovery and we present the results in section [todo].

Model-based Reinforcement Learning is a class of reinforcement learning algorithms in which we aim to learn—or use a given model—of the real environment in which an agent acts. The work in [20] proposed a novel approach to learn a "world model" using recurrent neural networks; we take inspiration from such work and use world models and a policy optimisation algorithm as the controller in the world model. Other work such as [15, 28] build discrete world models and train directly in latent space. Prior work on world models used a variation on a variational auto-encoders to generate a latent state of the pixel input, instead we use a graph neural network to generate a latent representation of the input computation graphs.

RL in Computer Systems is a relatively recent topic of research. In recent years there has been an increased focused on using model-free RL in variety of systems environments. For example, in [29, 30, 31, 32], reinforcement learning was used to optimise the placement of machine learning models to improve throughput. In [13], model-based RL was used successfully to optimise the selection of bitrate when streaming data across a network. This work takes inspiration from prior work and we use both model-free and model-based RL to optimise deep learning models by reducing estimated, on-device runtime.

Remarks. This work distinguishes itself from the aforementioned works as, to the best of our knowledge, there has not been an attempt to use reinforcement learning, neither model-free nor model-based, to the task of optimising a deep learning model by applying substitutions directly to the computation graph. Although there has been work using RL to the task of device placement [31, 32] which also aims to reduce runtime or memory usage, our work is in a different domain.

Chapter 3

Problem Specification

In this chapter we will introduce the graph optimisation problem and describe the approaches from prior work which we use as a baseline performance measure. Furthermore, we will frame the optimisation problem in the RL domain by describing the system environment, the reward calculation and the state-action space. Additionally, we describe the RL agents trained in the model-free and model-based domains and we also highlight limitations in the application of reinforcement learning to this problem.

3.1 Introduction

The major deep learning frameworks such as TensorFlow [1] and PyTorch [2] used greedy rule-based graph transformation prior to execution. Furthermore, in Chapter 2.1.1 we described the prior work upon which this work builds. Namely, we introduced the work by Jia et al. [9, 10] that proposed an approach for performing an offline optimisation of deep learning computation graphs using a recursive backtracking search in the action space. Specifically, the authors developed a framework that uses a pre-generated set of formally verified, semantically equivalent graph substitutions that can be used to modify the graph to search for a reduced runtime.

3.2 Optimisation of deep learning graphs

TensorFlow (TF) uses a system called *"Grappler"* that is the default graph optimisation system in the TF runtime [33]. By natively performing the graph optimisation at runtime, it allows for a interoperable, transparent optimisation strategy via protocol buffers. To improve the performance of the underlying model, Grappler supports a range of features such as the pruning of dead nodes, removal of redundant computation and improved memory layouts. Concretely, Grappler was designed with three primary goals:

- Automatically improve performance through graph simplifications and high-level optimisations to benefit the most target architectures
- Reduce device peak memory usage
- Improve hardware utilisation by optimising device placement

On the other hand, although Grappler can automatically optimise the dataflow graphs of deep learning models, such a complex optimisation system presents challenges. Firstly, significant engineering effort is required to implement, verify and test the optimiser to ensure the correctness of the graph rewrites rules; TF contains a set of 155 substitutions that are implemented in 53,000 lines of code; to further complicate matters, new operators are continuously proposed, such as grouped or transposed convolutions, all of which leads to a large amount effort expended to maintain the library. Secondly, and perhaps more importantly, as TF uses Grappler at runtime by default, it adds overhead to execution as extra graph conversions are performed at runtime rather than offline.

Alternatively, both TensorFlow, and more recently PyTorch, support automatic graph optimisation by JIT (just-in-time) compilation through XLA and the `torch.jit` package respectively. In Figure 3.1 we can see a high-level view of the components of the optimisation system. In order to motivate the reasoning to perform offline optimisation rather than JIT optimisation we consider the work proposed by Jia et al. in both MetaFlow and TASO, the systems they design can be used as a drop-in replacement of the Grappler

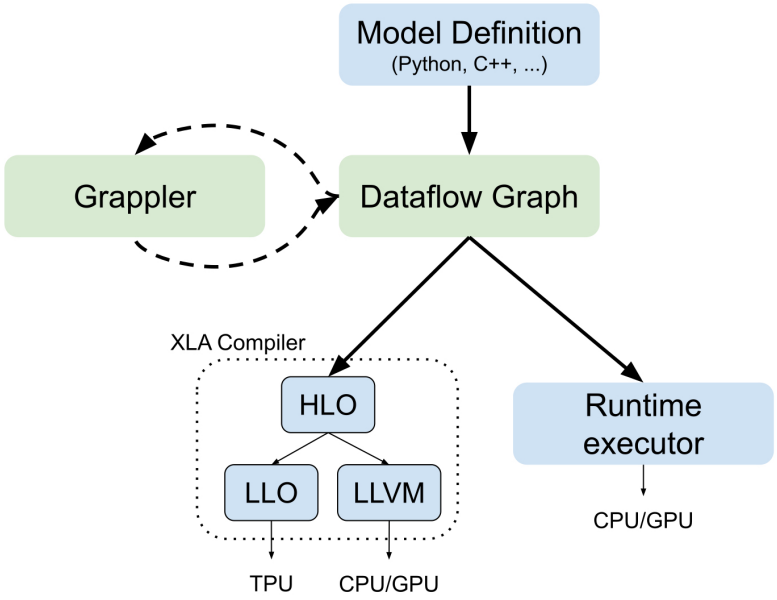


Figure 3.1: The machine learning model is processed prior to execution by either Grappler, the static graph optimiser in TensorFlow, or via JIT compilation of the model using XLA. Figure adapted from [33].

and/or XLA compilation steps.

TASO applies all possible candidate transformations at each step and estimates the runtime (or cost) of the final graph. Next, TASO chooses the highest performing candidates for the proceeding iteration of candidate evaluations. Principally, this approach is superior to the naive greedy optimisation approach as we can use the estimated runtime to guide the search and forego immediate improved runtime to increase the potential search space of candidate graphs.

In addition, as TASO operates at the graph-level, its optimisations are completely orthogonal to operator-level optimisations; thus it can be combined with code generation techniques such as TVM [7] or Astra [34] to further improve overall performance. We also note that TASO performs tensor data layout and graph transformation simultaneously rather than sequentially. It has been shown that by considering it as a joint optimisation problem end-to-end inference runtime can be reduced by up to 1.5x [9, 10].

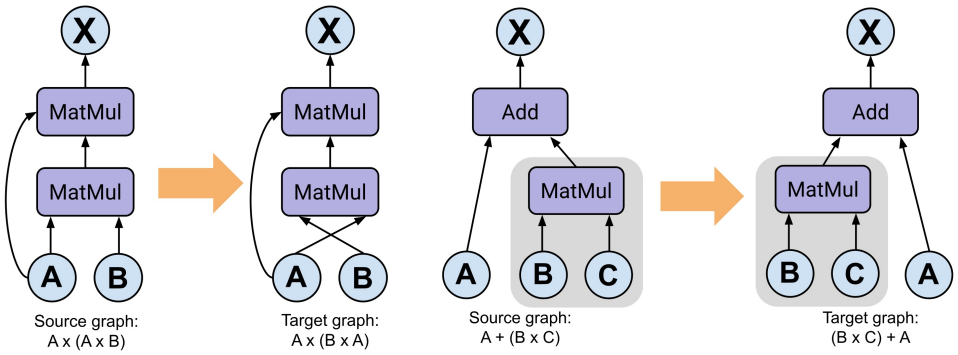
3.2.1 Graph-level optimisation

Performing optimisations at a higher, graph-level means that the resulting graph is - in terms of execution methodology - no different than the original graph prior to optimisation. Therefore, by performing graph-level optimisation we generate a platform and backend independent graph representation which can be further optimised by specialised software for custom hardware accelerators such as GPUs and TPUs.

Next, we define that two computation graphs, $\mathcal{G}$ and $\mathcal{G}'$ are semantically equivalent when $\forall \mathcal{I} : \mathcal{G}(\mathcal{I}) = \mathcal{G}'(\mathcal{I})$ where $\mathcal{I}$ is an arbitrary input tensor. We aim to find the optimal graph $\mathcal{G}^*$ that minimises the cost function, $\texttt{cost}(\mathcal{G})$, by performing a series of transformations to the computation graph - at each step, the specific transformation applied does not need to be strictly optimal. In fact, by applying optimisations that reduce graph runtime we further increase the state space for the search; a large state space is preferable in the reinforcement learning domain.

An important problem in graph-level optimisation is that of defining a set of varied, applicable transformations that can be used to optimise the graphs. As previously noted, prior work such as TensorFlow use a manually defined set of transformations and optimise greedily. On the other hand, TASO uses a fully automatic method to generate candidate transformations by performing a hash-based enumeration over all possible DNN operators that result in a semantically equivalent computation graph.

In this work, we take the same approach as that of TASO and automatically generate the candidate graphs. We perform this as an offline step as it requires a large amount of computation to both generate and verify the candidate substitution; to place an upper bound on the computation, we limit the input tensor size to a maximum of 4x4x4x4 during the verification process. Following the generation and verification steps, we prune the collection to remove substitutions that are considered trivial and as such would not impact runtime. For example, trivial substitutions include input tensor renaming and common subgraphs, we show both techniques diagrammatically



(a) Tensor renaming substitution

(b) Common subgraph substitution

Figure 3.2: Two examples of trivial graph substitutions that does not impact the overall runtime of the computation graph. The left sub-figure shows a simple renaming of the tensor inputs. The figure on the right shows that we have a common sub-graph between the source and the target graphs. In both cases we eliminate the duplicates as the hash of the two graphs will be identical.

in Figure 3.2a and 3.2b respectively.

Algorithm 2: Cost-based backtracking search. Adapted from [9].

Input: Initial computation graph $\mathcal{G}_0$, a cost function $\texttt{cost}(\mathcal{G})$, a list of valid graph substitutions $\{S_1, \ldots, S_m$, and the hyperparameter α
Output: An optimised computation graph $\mathcal{G}^*$
// $\mathcal{Q}$ is a priority queue of graphs sorted by `cost`.
$\mathcal{Q} = \{\mathcal{G}_0\}$
while $\mathcal{Q} \neq \{\}$ **do**
  $\mathcal{G} = \mathcal{Q}.\texttt{dequeue()}$
  for $i = 1 \ldots m$ **do**
    $\mathcal{G}' = S_i(\mathcal{G})$
    if $\mathit{cost}(\mathcal{G}') < \mathit{cost}(\mathcal{G}^*)$ **then**
      $\mathcal{G}^* = \mathcal{G}'$
    end
    if $\mathit{cost}(\mathcal{G}') < \alpha \times \mathit{cost}(\mathcal{G}^*)$ **then**
      $\mathcal{Q}.\texttt{enqueue}(\mathcal{G}')$
    end
  end
end
return $\mathcal{G}^*$

3.2.2 Baselines

In order to establish a baseline performance measure for performing graph-level optimisation of deep learning models we have two different sources. Firstly, we can measure the performance of a select number of deep learning models in the standard DL frameworks, TensorFlow and PyTorch. In this project, there are common, standardised mechanisms for evaluating the performance of models using these frameworks - we show the results of the baseline measurements in the following section.

Secondly, in this work, we replicate the experiments as performed by Jia et al. [9] and use the results as our benchmark to compare our work against. However, for the majority of evaluated graphs we used a lower budget than that of the authors in the original paper. We found that using a lower search budget, without alteration of the hyperparameter α, it did not result in a lower performance compared to the original experiments. Figure [TODO] shows the results of the heuristic search for the graph $\mathcal{G}^*$ and Figure [TODO] shows the relative performance of the methods on each chosen deep learning model.

We further note that Jia et al. [9] used a simple method to estimate the runtime of tensor operators that are executed using low-level CUDA APIs and the runtime is averaged over N forward passes.However, this approach to runtime estimation is imperfect as there is a non-negligible variance of the runtime on real hardware. We investigated the use of using real runtime measurements during training rather than a estimation of operator runtime. We found that it increases duration of each training step such that performance improvements are not worth the trade-off.

3.3 Reinforcement Learning formulation

In the following section we will describe how to represent the computation graph optimisation problem in the reinforcement learning domain by de-

scribing the key components of the system. We describe the system environment in which the agents act, the state-action space, and finally the reward functions for both the model-free and model-based agents which we used to determine the optimal reward signal to train the agents.

3.3.1 System environment

In order to train a reinforcement learning agent, it necessary that we have access to an environment that, given the current environment state, the agent can take an action. After taking the chosen action, the environment is updated into a new state and the agent receives a reward signal. Typically, one uses a mature environment such as OpenAI Gym [35] or OpenSpiel [36] as the quality of the environment often has a significant effect on the stability of training. Moreover, using an environment that uses a common interface allows researchers to implement algorithms with ease and, importantly, reproduce results from published conference papers.

In our work, we implemented an environment that follows the OpenAI Gym API standard stepping an environment, that is, we have a function `step(action)` that accepts a single parameter, the action requested by the agent to be performed in the environment. The `step` function returns a 4-tuple `(next_state, reward, terminal, extra_info)`. `extra_info` is a dictionary which can store arbitrary data. The environment in our project has a structure that is shown diagrammatically in Figure [TODO].

To simplify the implementation of the environment, we used made extensive use of the work by Jia et al. [9] with the open source version of TASO. We provide a computation graph and the chosen transformation and location; TASO then applies the requested transformation and returns the newly transformed graph. Further, we use internal TASO functions that calculates estimates of the runtime on the hardware device which we use as our reward signal for training the agent. During our experiments we modified TASO to extract detailed runtime measurements to analyse the rewards using a range of different reward formulae - we describe our approach further in section

3.3.4.

The scope of our work meant that there was no existing prior work that applied reinforcement learning to the task of optimising deep learning computation graphs. Thus, we required an environment in which an agent can act efficiently. Due to the nature of systems environments, the interactions with the real-world environment can be often slow, especially compared to those such as Arcade Learning Environment [37]. An aim of this work was to train a simulated environment, a "world model", that if accurate in relation to the real environment, we can train an agent far more efficiently than would be possible with the real-environment. In section [TODO] we will further explore world models and evaluate our implementation.

3.3.2 Computation Graphs

The first step prior to optimising a deep learning graph is that we must load, or create on-demand, the model in a supported deep learning framework. In our project, we can support any model that is serialised into the ONNX [38] format which is a open-source standard for defining the structure of deep learning models. Thereby, by extension, we can support any deep learning framework that supports ONNX serialisation such as TensorFlow [1], PyTorch [2] and MXNet [3].

Next, we parse the ONNX graph internal representation by converting all operators into the equivalent TASO tensor representations such that we can modify the graph using the environment API as we described in section 3.3.1. Although our environment does not support conversion of all operators defined in the ONNX specification [1], the majority of the most common operators for our use case are supported; therefore we still maintain the semantic meaning and structure of the graph. Additionally, after performing optimisations of the graph, we can export the optimised graph directly to an ONNX format.

[1] ONNX operator specification: `https://github.com/onnx/onnx/blob/master/docs/Operators.md`

3.3.3 State-Action space

In this project we modelled the state and action space in accordance with prior research, specifically we referenced work in a similar domain of system optimisation using reinforcement learning; Mirhoseini et al. [30] used hierarchical RL with multiple actions to find the optimal device placement and Addanki et al. [31] that also aided in the design choice of input/output graph sizes.

Next, we require two values in order to update the environment. First, we need a select a transformation (which we refer to as an `xfer`) to apply to the graph. Secondly, the location at which to apply the transformation. As we need to select two actions that are dependent on each other to achieve a higher performance, it requires selecting the actions simultaneously.

However, this would require a model output of $N \times L$ values, where N is the number of transformations, L is the number of locations. Such an action space is too large to train a model to efficiently predict the correct action. Additionally, after choosing a transformation, we ideally mask the available locations as not all locations can be used to apply a transformation. Therefore, using the same trunk network, we first predict the transformation, apply the location mask for the selected transformation, then predict the location.

We define the action as 2-value tuple of (xfer_id, location). There is a special case for the xfer_id. When it equals N (the number of available transformations), we consider it the NO-OP action. Therefore, in this special case we do not modify the graph, rather we terminate the current episode and reset the environment to its initial state.

As explained in the previous section, we used an step-wise approach where at each iteration, we provide a 2-tuple of the transformation and location, to apply in the current state. The updated state from the environment is a 4-tuple consisting of `(graph_tuple, xfer_tuples, location_masks, xfer_mask)`.

`xfer_mask` refers to a binary mask that indicates the valid and invalid trans-

formations that can be applied to the current computation graph as not every transformation can be applied to every graph. If the current graph has only four possible transformations that can be applied, all other transformations considered to be invalid. Thus, we return a boolean location mask where only valid transformations are set to 1, or `true`. This can be used to zero-out the model logits of invalid transformations (and thereby actions also) to make ensure the agent always selects a valid transformation from the set.

Similarly, for each transformation selected by the agent, there are a number of valid locations where this transformation can be applied. We set a hardcoded, albeit configurable, limit the number of locations to 200 in this work. If the current graph has fewer than 200 possible locations for any given transformation, the remaining are considered invalid. Therefore, we again return a boolean location mask, which is named `location_masks` in the 4-tuple defined above, which can be used to zero out the model logits that which the locations are invalid.

3.3.4 Reward function

- Runtime difference

- Inclusion of detailed measurements

- Real-time measurements instead of estimated?

- Look up research on RL rewards (what makes a good reward signal)

Chapter 4

Reinforcement Learning Agent Design

In this chapter we describe the technical details of the design of the two reinforcement learning agents and their components in relation to prior work. Further, we also discuss the relative benefits of each approach as well as the significant challenges that we must overcome to apply RL to this problem and establish the baselines to compare the model-free and model-based agents.

4.1 Graph Embedding

As we described in the previous section, the reinforcement learning agent must learn to choose two actions at each step in an epoch; the transformation and location from the set available dependent on the current computation graph. To learn the optimal action selection, we must create an embedding from the computation graph representation in the machine learning framework to our internal, manipulable graph representation inside our environment, our modified TASO backend.

When developing the project, a pivotal part of the project is the decision as the representation of the GNN as there are a wide variety of forms which

it can take. For example, a common implementation are message-passing networks (MPNNs) [39] which reduce data along edges and between nodes in the graph. Alternatively, we considered using graph convolutional networks (GCNs) [40], however, we found that using messages passing networks produced a more generalisable embedding as we leverage the relational biases in the graph structure and avoid imposing restrictions on the learned embedding accidentally.

[TODO] cite related work that used GNN for systems work?

During training of the reinforcement learning agents, we convert the internal graph representation to a graph neural network. In order to train the model-free and model-based agents, a latent space embedding of the computation graph is required. Therefore, using the `graph_nets` package developed by Battaglia et al. [27], we use the graph neural network to learn a latent space embedding of the graph using message passing networks to gather the global learned features of the graph.

- Auto-encoders

4.2 Model-free Agent

In section 4.1 we described the process for translating the computation graph, built in a machine learning framework, into an internal message passing graph neural network that can produce a latent space embedding, z_t, of the graph state s_t at a time t. In our work, we used the PPO algorithm described by Schulman et al. [41] as it brings three advantages, it was deliberately designed to be sample efficient, easy to implement, and stable to a wide range of values in hyperparameter selection. Its predecessors, such as TRPO [42], required off-policy learning using replay memory, which is often challenging to implement efficiently - especially with systems environments where rollouts are expensive to collect and store. Moreover, PPO is an on-policy algorithm that is compatible with stochastic gradient descent (SDG), meaning we can use it in combination with our graph network to train using hierarchical

reinforcement learning. Algorithm 3 shows a variant of the PPO algorithm using a clipped objective, resulting in a simpler implementation compared to KL-penalty objective.

Algorithm 3: PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ

for *k = 0, 1, 2, ...* **do**

Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using GAE with the value function V_{ϕ_k}

Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{\text{CLIP}}(\theta)$$

by taking K steps of minibatch SDG (using Adam), where

$$\mathcal{L}_{\theta_k}^{\text{CLIP}}(\theta) = \mathbb{E}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi_k})\right]\right]$$

Fit value function using using MSE loss using minibatch SDG

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T}\sum_{t=0}^{T}\left(V_\phi(s_t) - \hat{R}_t\right)^2$$

end

We use short online rollouts to collect a mini-batch of observations where a single trajectory begins with the unmodified graph and we iteratively apply transformations until we reach a terminal action or no further transformations can be applied. After each rollout we estimate the runtime which is used to calculate the reward for the rollout - we described the reward calculation in section 3.3.4.

After collection of n rollouts, we train the agent using the data produced during each action step which is used to update the weights of the policy and value neural networks according to the PPO algorithm. One should note that as we require two actions to be selected (`xfer_id` and `location_id`), it requires two sets of results to be collected during the rollout, one for each action performed. Additionally, as we perform two actions, it doubles our

overhead during training as we both store and perform backpropagation for four neural networks, the policy and value networks for each action. However, as we discussed in 3.3.3, the alternative approach we considered would lead to lower agent performance during training due to the larger action space.

4.3 Model-based Agent

Unlike model-free reinforcement learning, in the domain of model-based reinforcement learning we aim to learn a model of the environment such that we no longer need the real simulator, providing numerous benefits such as improved sample efficiency, ability to plan trajectories of actions forward in time and decreased training time for systems environments. The primary task is model-based RL is to learn a model of the environment. Concretely, we aim to learn a function $f(z_t, a_t)$ that predicts the latent next state z_{t+1} based on the action a_t being performed in the state z_t, the reward r_t and the terminal flag d_t which indicates the end of the trajectory. Many environments, especially systems tasks, state transitions are stochastic and we must accurately represent such transitions in order to have a useful world model for planning. This section will further discuss how we designed the world model for learning the environment behaviour.

4.3.1 World Models

World models, introduced by Ha et al. [20], create an imagined model of the true environment by observing sequences of states, actions and rewards from the environment and learning to estimate the transitions between states based upon the actions taken. Ha et al. showed that the world models can learn the environment transitions and achieve state-of-the-art results on visual learning tasks such as CarRacing and VizDoom. One should note that Ha & Schmidhuber used a latent space embedding from the convolutional neural network based on the RGB pixel image; in this work we instead use

the latent space produced by the graph neural network - in either case, we are learning the world model using the latent space of the environment. World models are constructed from three components. The "visual" module, taking the raw state from the environment and transforming into latent space, as well as the "memory" and "controller" modules which are discussed below.

Mixture Density Networks

Mixture Density Networks (MDNs) are a class of network that can learn to output parameters to a probabilistic Gaussian mixture model (GMMs) [25]. A GMM is a function that is composed of several gaussians, each given a label $k \in \{1, \ldots, K\}$, where K is the number of components. Each gaussian is formed from three parameters μ_i, the mean of component i, σ_i the variance of component i and π the mixing probability/weight of each component. Unlike the networks used in supervised learning tasks that are trained using regression, training a GMM instead attempts to maximise the likelihood that the gaussians fit the data points in each minibatch. Inside a world model we use the predictions of an MDN at time t to choose the parameters of the gaussian distribution for the next latent vector at time $t + 1$. Notably, one can either use expectation maximisation to find the parameters of the model, or alternatively, can use a parameterised GMM which is trained in conjunction with the RNN using stochastic gradient descent.

Recurrent Neural Networks

Recurrent Neural networks (RNNs) are a class of architectures in which the connections between the nodes form a directed graph in a temporal sequence [21]. There are many forms which an RNN can take, each providing features and levels of stability which one many find useful for the task at hand. Importantly, the output of an RNN is deterministic, however, we can use the outputs from the RNN as the parameters for a probabilistic model to insert a controllable level of stochasticity in the output predictions [43].

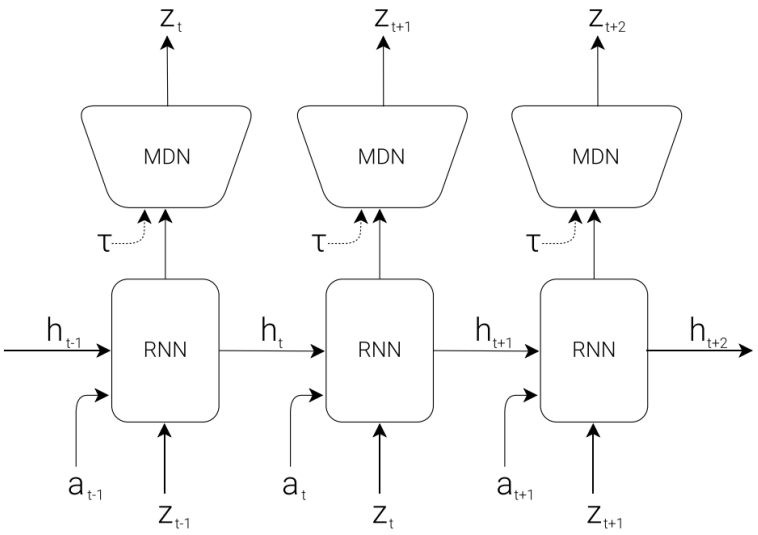


Figure 4.1: Structure of an unrolled MDN-RNN. The MDN outputs the parameters of a Gaussian mixture distribution used to sample a prediction of the next latent vector z_{t+1}, the MDN is controlled by the temperature parameter τ.

A constraint of using RNNs is that they expect a fixed sized input sequence. However, in our work, both the shape of the latent state tensor, and the number of actions performed by the agent in a rollout is variable. As such, we employ a common approach to mitigate this problem is by prepending zero values to the input sequence until the desired length is reached, commonly referred to as padding. After performing inference on the model and retrieving the predicted state, we mask the results based on the input padding to ensure we only use valid predictions to select the next action using the controller.

MDN-RNN

By combining the mixture density and recurrent networks, we can use rollouts of the environment sampled using a random agent to train the combined network, called an MDN-RNN. We use the network to model $P(z_{t+1} \mid a_t, z_t, h_t)$, where z_t, z_{t+1} is the latent state at the times t and $t+1$ respectively, a_t is

the action taken at time t, and h_t is the hidden state from the RNN network at time t. Figure 4.1 shows the combination of the RNN and MDN networks and how we calculate the predictions of the next latent state in sequence.

Furthermore, after training the world model, we must train an agent (or controller) to perform actions in the world model and learn to take optimal actions that maximise reward. During inference of the world model, we use a softmax layer which outputs π in the form of a categorial probability distribution which we sample under the Gaussian model parameterised by (μ_i, σ_i).

In figure 4.1 we show that one of the inputs to the MDN is τ, the temperature. By altering the temperature it allows us to control the stochasticity of the agent during training of the controller. The logits of the RNN that represent the predictions for the values of π are divided by the temperature prior to being passed into the softmax function which converts the logits into pseudo-probabilities. We incorporate the temperature, τ, into the function using the following equation.

$$\text{softmax}(\mathbf{x}_i) = \frac{\exp\left({}^{x_i}/_{\tau}\right)}{\Sigma_j \exp\left({}^{x_j}/_{\tau}\right)}$$

Typically, temperature is a real number in the range $\tau \in [0, \ldots, 1)$, where a value of zero leads to completely deterministic predictions generated by the RNN, whereas larger values introduces a greater amount of stochasticity in the predictions. As larger values of τ increases the probability of samples with a lower sampling likelihood being selected it leads to a greater diversity of actions taken by the agent in the environment. Importantly, Ha et al. [20] found that having a large temperature can aid in preventing the agent from discovering strategies to exploit in the world model which are not possible in the real environment due to imperfections in the model.

Modifying the softmax activation function in this way is equivalent to performing knowledge distillation between two models; learnt information is transferred from a large teacher model, or ensemble model, to a smaller model

which acts as a student model. In both the context of knowledge distillation and training a controller actor inside the world model, a high temperature will generate a softer targets. Specifically, in this work a higher temperature produces a softer pseudo-probability distribution for π in the GMM. Additionally, using soft targets will provide a greater amount of information for the model to be learn by forcing the model to learn more aggressive policies, thus outputting stochastic predictions which is beneficial to encourage exploring the environments state-action space.

Furthermore, we consider how the world model is trained. For any supervised learning task we require target data to which we can compare our predictions, calculate a loss and perform backpropagation to update the weights in the network. To train the world model, we use a random agent, one that has an equal probability of choosing any action from the valid set of actions in a given state. Unlike Ha and Schmidhuber [20] who performed 10,000 rollouts of the environment offline using a random policy to collect the data, we took a slightly different approach. Instead, we generated minibatch rollouts using the random agent online, and directly used the observations to train the world model. Although this approach reduces the data efficiency as we only use each state observation once, we benefit from removing the need to generate the data prior to training. In systems environments, it is often expensive—in terms of computation time—to step the environment collect a diverse dataset. Therefore, we found generating short rollouts and training on the minibatch was beneficial without any perceivable impact on performance.

4.3.2 Action Controller

Finally, we discuss the design of the "controller", the network/agent that learns to output actions based upon the output from the MDN-RNN world model. Ha and Schmidhuber [20] used an evolution based controller defined as a simple multi-layer-perceptron, $a_t = W_c[z_t, h_t] + b_c$, that accepts the hidden and current states from the recurrent network to predict the next action to be taken. A challenge when training the controller inside the fully

imagined world environment is that we no longer have access to the ground truth state nor the reward produced by the real environment, therefore, we cannot use supervised learning to train the controller.

In [20] the authors used an evolutionally algorithm, covariance matrix adoption evolution strategy (CMA-ES) [44, 45], which optimises the weights of the network based on the reward produced by the world model. Alternatively, recent work by Hafner et al. [46, 28] has shown to achieve state-of-art results in the Atari environment using an actor-critic method as the controller in the world model. Furthermore, prior work on the application of world models to systems environments has shown one can train a model-free controller inside the world environment [13].

In our work, we use PPO, an on-policy algorithm which uses the world model state, rewards and terminal flags to optimise the control policy. Although any controller, from an shallow MLP to model-free RL algorithms can be used, the PPO algorithm is shown to be extremely robust to a range of parameters and for your work, a good comparison as we used the same algorithm to train the agent inside the real environment. We show the results for training the model-free and model-based controllers in sections 5.3.2 and 5.3.3 respectively.

Chapter 5

Evaluation

5.1 Aims

In this chapter, we look to assess aims we presented at the beginning of this work where we claimed to use reinforcement learning to perform automated optimisation of deep learning computation graphs. Thus, this evaluation seeks to answer the following questions:

1. Are model-based reinforcement learning methods able to model the transition dynamics of the environment?
2. Is the agent policies able to generalise to unseen states of the same graph to act in accordance to our performance objectives?
3. Do they accurately model the reward estimation from the graph latent state?
4. Are the agents trained in an imagined world model applicable to the real-world environment?

5.2 Experimental Setup

All the experiments presented in this chapter, both training various agent models and testing, is performed using the codebase available in the GitLab repository for this project [todo: link]. The project was developed, and the experiments were performed using a single machine running Ubuntu Linux 18.04 with a 6-core Intel i7-10750H@2.6GHz, 16GB RAM and an NVIDIA GeForce RTX 2070.

We chose to use five real-world deep learning models to evaluate our project. InceptionV3 [47] is a common, high-accuracy model for image classification trained on the ImageNet dataset. ResNet-18 & ResNet-50 [48] are also deep convolutional networks, 18 and 50 layers deep respectively as well as SqueezeNet [49], a shallower yet accurate model. BERT [50] is a recent large transformer network that is used to improve Google search results [51]. As these graph were also used in the evaluation of TASO [9], we can show a direct comparison of the performance between the different approaches.

To interface with the internal representation of the computation graphs, as previously discussed, we used the open-sourced version of TASO [9] which we modified to extract detailed runtime information. Further, we implemented the reinforcement learning algorithms in TensorFlow 2 [1] and utilised the `graph_nets` package developed by Battaglia et al. [27] to process our input graphs which we described in chapter 3.3.1. The PPO agent was implemented based upon the implementation provided by Schulman et al. [41].

Finally, we would like to acknowledge the work by Kai Fricke and Michael Schaarschmidt who developed the initial Python interface with TASO, the algorithm for converting the C++ TASO graph representation into a Cython object and performed experiments with model-free reinforcement learning agents [52].

5.3 Experiments

5.3.1 Baselines

TensorFlow Grappler

TASO

5.3.2 Model-Free Agent

5.3.3 Model-based Agent

5.4 Discussion

Chapter 6

Conclusion and Future Work

6.1 Conclusion

6.2 Future Work

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems, 2015.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, page 675678, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning, 2014.

[6] NVIDIA. cuBLAS Library. `https://developer.nvidia.com/cublas`, 2008.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, 2018.

[8] NVIDIA. TensorRT: Programmable Inference Accelerator. `https://developer.nvidia.com/tensorrt`, 2017.

[9] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 4762, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Zhihao Jia, James Thomas, Tod Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. *SysML 2019*, 2019.

[11] Richard Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[12] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert,

Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving Rubik's Cube with a Robot Hand, 2019.

[13] Harrison Brown, Kai Fricke, and Eiko Yoneki. World-Models for Bitrate Streaming. *Applied Sciences*, 10(19), 2020.

[14] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-Based Reinforcement Learning for Atari, 2020.

[15] Jan Robine, Tobias Uelwer, and Stefan Harmeling. Smaller World Models for Reinforcement Learning, 2021.

[16] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017.

[17] Thomas Anthony, Zheng Tian, and David Barber. Thinking Fast and Slow with Deep Learning and Tree Search, 2017.

[18] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning, 2018.

[19] C. Daniel Freeman, Luke Metz, and David Ha. Learning to Predict Without Looking Ahead: World Models Without Forward Prediction, 2019.

[20] David Ha and Jürgen Schmidhuber. Recurrent World Models Facilitate Policy Evolution, 2018. `https://worldmodels.github.io`.

[21] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[22] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[24] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM, 1999.

[25] Christopher M Bishop. Mixture density networks, 1994.

[26] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[27] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[28] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering Atari with Discrete World Models, 2021.

[29] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org, 2017.

[30] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[31] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879*, 2019.

[32] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs, 2020.

[33] Rasmus Munk Larsen and Tatiana Shpeisman. TensorFlow Graph Optimizations, 2019. `http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf`.

[34] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.

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

[36] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR*, abs/1908.09453, 2019.

[37] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253279, Jun 2013.

[38] Junjie Bai, Fang Lu, Ke Zhang, et al. ONNX: Open Neural Network Exchange. `https://github.com/onnx/onnx`, 2019.

[39] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry, 2017.

[40] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[41] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[42] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization, 2017.

[43] Alex Graves. Generating Sequences With Recurrent Neural Networks, 2014.

[44] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[45] Nikolaus Hansen. The CMA Evolution Strategy: A Tutorial, 2016.

[46] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to Control: Learning Behaviors by Latent Imagination, 2020.

[47] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision, 2015.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, 2015.

[49] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, 2016.

[50] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[51] Pandu Nayak. Understanding searches better than ever before. `https://blog.google/products/search/search-language-understanding-bert`, 2019.

[52] Kai Fricke and Michael Schaarschmidt. XflowRL. `https://gitlab.com/CamRL/xflowrl`, 2019.