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

June 1, 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 using RL to choose the graph transformations which are applied the networks graph representation. Reducing the hardware resource requirements of deep learning models is a open, active research question; current work has focused on the design optimal heuristic rules. In this work, we investigated the use of RL agents that can learn to perform optimal transformations, without the need of expert human heuristics to achieve a high level of performance. Recent work has aimed to apply reinforcement learning to computer systems with some success, especially using model-free RL techniques. However, model-based methods have seen an increased focus of research; it can be used to learn a model of the environment that can be leveraged to train an agent inside the learned world-model, thereby increasing sample efficiency compared to model-free RL. Furthermore, when using a world model as the environment, batch rollouts can occur safely in parallel and, especially in systems environments, it overcomes the possible latency impact of stepping a system environment that can take orders of magnitude longer to perform an action compared to a simple emulators for video games. This dissertation examines both the prior work for optimising deep learning models and the applicability of reinforcement learning to the problem. We show that by using model-based RL, we can reduce the runtime of deep learning models by up to 58% compared to current deep learning frameworks and up to 7% compared to the state of the art approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Services often taken for granted, such as search, social networks and language translators, are composed of complex systems. Modern services have components that are underpinned by machine learning (ML) models, specifically, deep neural networks (DNN). Over the past decade there has been a focus on developing frameworks that provide tools using which we can design, train and evaluate these deep learning models.

A common internal representation for neural networks inside deep learning frameworks is that of a computation graph—a directed acyclic graph where nodes represent a specific computation and edges the paths where data is transferred. Frameworks such as TensorFlow [1] and PyTorch [49] automatically apply optimisations in an effort to reduce computation resources during inference.

Currently, the optimisation performed in deep learning frameworks is performed using manually defined heuristics. For example, TensorFlow [1] uses 155 handwritten optimisations composed of 53,000 lines of C++. While such heuristics are applicable for current architectures, network design is consistently evolving. Therefore we require consistent innovation to discover and design rules that control the application of optimisations with guarantees that strictly improve efficiency. Eliminating the need for manual engineering

work that is required to design and implement the heuristics for applying optimisations is a primary focus of this work.

Recent work, namely TASO by Jia et al. [33, 34] has shown it is possible replace the heuristics with a cost-based search for the optimal graph. However, such approaches may not fully explore the potential search space due to the lack of planning in a cost-based optimisation. To address this issue in this work we explore the use of reinforcement learning (RL). RL is an area of machine learning in which an agent learns to act optimally, given a state and a suitable reward function, through interactions with an environment.

In this work, we focus on the use of RL for the task of optimising deep learning graphs. Specifically, we focus on a model-based reinforcement learning which aims to learn a model of the environment in which they act. In our work, the network learns to model the dynamics of the application of graph substitutions and its impact on the overall runtime of the model when executed on-device. Further, learning a model of the environment provides important benefits; for example, lookahead planning, low-cost state prediction and faster wall-clock training. We examine the use of world-models for learning the environment as well as training a controller inside a world model that removes the need of an expensive, time-consuming computer system to apply our chosen graph optimisations.

This dissertations key contributions are:

- Applies modern reinforcement learning approaches that eliminates the need for human engineered graph optimisations in machine learning frameworks. We show that our proposed method can improve runtime by up to 58% compared to current deep learning frameworks and up to 10% compared to the state-of-the-art.
- Provides a detailed discussion and analysis of our solution as well as comparison to the current state-of-the-art methods in published literature.
- Implemented a model-based RL agent (section 3.4), and environment

(section 3.2.1), for jointly choosing the optimal substitution and substitution location (section 3.2.3).

- This work, to the best of our knowledge, is the first that has applied model-based reinforcement learning in optimising computation graphs to reduce hardware resource requirements.

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 optimisation problem and formulates the problem in the context of reinforcement learning. Furthermore, we also describe our approach for applying reinforcement learning to optimise the computation graphs as well as learning an accurate model of the environment. Chapter 4 covers the evaluation setup, our experiments and results for different methodologies. Finally, in chapter 5 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 (for a variety of tasks such as object detection and classification), transformer networks, used for translation and generation of language, as well as recurrent networks that have shown to excel at exploiting long and short term trends in data.

Despite the improvements in the accuracy of machine learning (ML) models, the fundamental building blocks of deep learning models have remained largely unchanged. As the networks become more complex, it also becomes tedious to manually optimise the networks to reduce the execution time on hardware. Therefore, there is extensive work to 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.

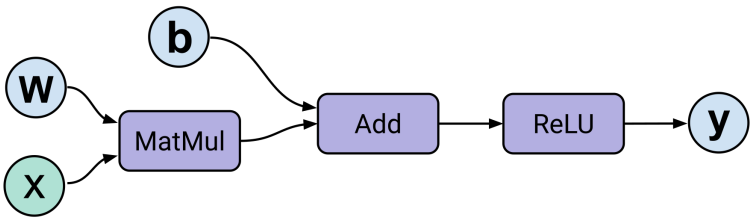


Figure 2.1: The operations shown in purple are the nodes of the computation graph which take an arbitrary number of inputs, performs a computation at the node and produces an output. The blue nodes represent the input nodes for tensors. The directed edges show the flow of tensors through the graph.

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. Second, 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 (Tensor Processing Unit) requires different data layouts and optimisations.

2.1.1 Current approaches to optimise deep learning models

In this section, we describe the two approaches for performing optimisation of computation graphs. Firstly, we describe rule-based optimisation which is the simplest method in which we use a set of hard-coded rules to greedily apply graph transformations. Secondly, we describe performant, cost-based

approaches which use heuristics to plan and apply optimisations automatically. We discuss these approaches in the following sections.

Rule-based Optimisation

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 [49], MXNet [11], and Caffe [32] 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 [13] or cuBLAS [45] that instead directly optimise individual tensor operations.

TensorFlow (TF) uses a system called *"Grappler"* that is the default graph optimisation system in the TF runtime [38]. 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 [1, 38]:

- Automatically improve performance through platform-dependent high-level graph optimisations
- Reduce peak memory usage on-device
- Optimising device placement

Although Grappler can automatically optimise the data-flow 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 [33]; 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.

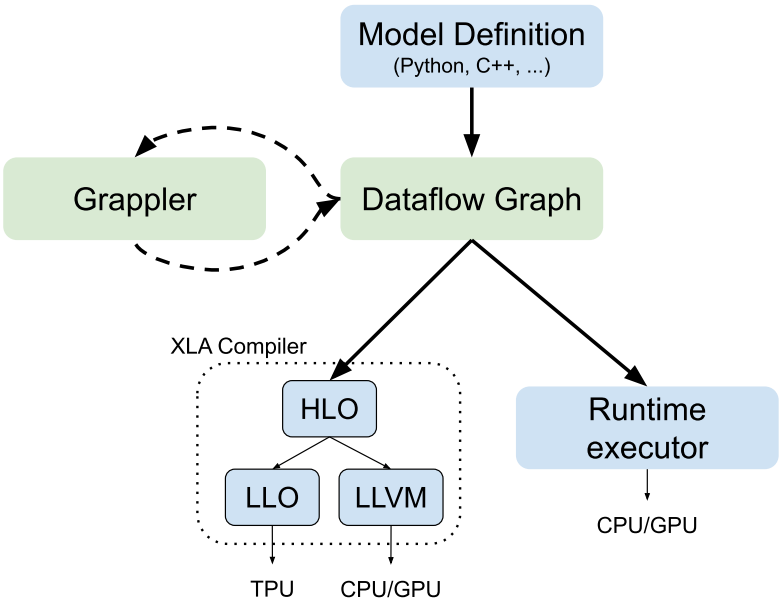


Figure 2.2: The machine learning model is processed prior to execution by either Grappler, the static graph optimiser in TensorFlow, or via JIT compilation of the model using XLA. Figure adapted from [38].

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 2.2 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 in both MetaFlow and TASO. The systems they designed can be used as a drop-in replacement of the Grappler and/or XLA compilation steps.

Finally, both TVM [12] and TensorRT [46] 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. These DNN

compilers also use greedy rule-based optimisation as part of the optimisation pipeline. In comparison to our work, we eliminated the need for manually designed heuristics and allowed a reinforcement learning agent to learn the optimal transformation sequence based on long-term expected rewards.

Cost-based Optimisation

As opposed to 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. [33] developed TASO that used a cost-based backtracking search to iteratively search through the state space of possible graphs that are formally equivalent. In contrast to using rule-based optimisation that applied hand-crafted optimisations heuristically, TASO generates the candidate subgraphs automatically and formally proves that the transformations are equivalent using an automated theorem prover.

Algorithm 1: Cost-based backtracking search. Adapted from [33].

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 $\texttt{cost}(\mathcal{G}') < \texttt{cost}(\mathcal{G}^*)$ **then**
      $\mathcal{G}^* = \mathcal{G}'$
    end
    if $\texttt{cost}(\mathcal{G}') < \alpha \times \texttt{cost}(\mathcal{G}^*)$ **then**
      $\mathcal{Q}.\texttt{enqueue}(\mathcal{G}')$
    end
  end
end
return $\mathcal{G}^*$

A key benefit of using a cost-based approach is that the search can take

into account 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. 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, TASO showed runtime of deep learning models can be increased up to 3x [33, 34], compared to baseline measurements using various deep learning compilers [13, 45, 46]. Principally, this approach is superior to the naive greedy optimisation as we can use the estimated runtime to guide the search and sacrifice immediate runtime improvement 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 [12] or Astra [56] 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 further reduced by up to 1.5x [33, 34] compared to the baseline optimisation.

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 [7]; 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 [7].

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

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 for applications such as Atari [41, 35] and Go [55]. 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 [47], datacenter power management, device placement [2, 40], 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 [10, 35, 50], 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 [55] or ExIt [3]. Alternatively, we can use use data augmentation methods to learn a representation of the underlying environment behaviour, and either use an imagined model or augment real experiences to train an agent in the domain [35, 15, 16].

Understandably, a relevant question is why one would prefer a model-free over model-based approach and the benefits of the respective methods. The primary benefit of model-based RL is that it has 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 which allows the agent to plan ahead, the agent can choose from a range of possible trajectories by taking actions to maximise its reward. The agent that acts in this "*imagined*" or "*hallucinogenic*" environment can be a simple MLP [21] to a model-free agent trained using modern algorithms such as PPO [52], A2C [42] or Q-learning [58, 41]. Further, training an agent in the world model is comparatively cheap, especially in the case of complex systems environments where a single episode can 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. Consequently, 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, the models are trained on distributed clusters of GPUs/TPUs; large amounts of compute is required overcome the sample inefficiency of model-free algorithms.

2.2.2 World Models

World models, first introduced by Ha and Schmidhuber [21], 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 2.3 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 that 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, $a \in \mathcal{A}$.

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

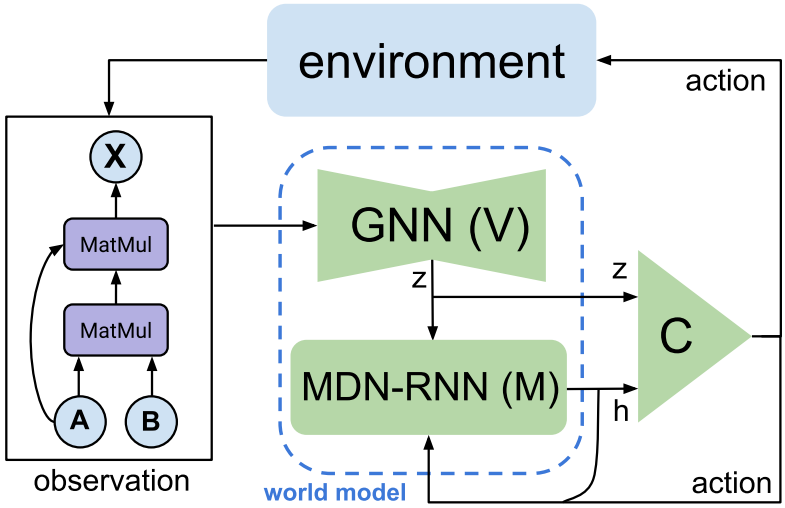


Figure 2.3: Diagrammatic representation of a world-model made up from an encoder (V) that transforms the input into latent space, a 'memory' module, M, that learns the behaviour of the environment from the latent vector z and a controller, C, that is trained using the latent vector of the encoder and the output features from the memory to choose an action which is either applied to the real or imagined environment. Figure adapted from [21].

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) [53, 29] or Long-short term memory (LSTMs) [30, 18]. 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 [8] 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.

MDNs can be used to learn to output parameters to a probabilistic Gaussian mixture model (GMMs) [8]. 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 are a 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 [29].

Motivated by the desire to overcome the limitations of RNNs, Hochreiter et al. [30] developed long-short term memory by describing Constant Error Carousel (CECs). The idea was further improved by Gers et al. [18] 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 2.4 shows the internal structure of an LSTM module.

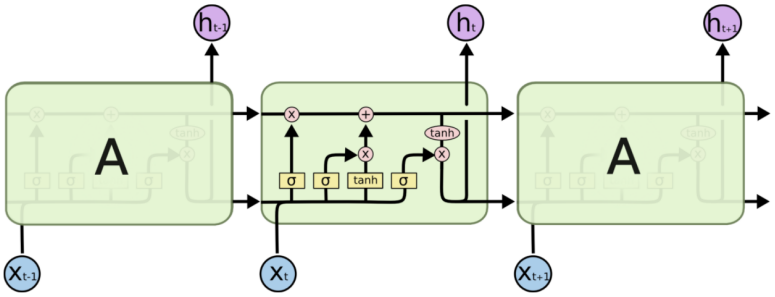


Figure 2.4: LSTM

An LSTM can be described using four "gates", where a gate influences a specific property of the LSTM cell; the four gates are as follows. The *forget* gate dictates if the information stored in the cell should be erased after observing the inputs $[h_{t-1}, x_t]$; the forget gate outputs a value in the range $[0, 1]$ using the sigmoid function σ. When the forget gate output is 1, it completely forgets the current state. Secondly, the *input* gate calculates the new information to be stored in the cell; it generates a vector of candidate values defined by $\tilde{C}_t$. The *update* gate is used to determine how much of the prior 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 which is based on the current, filtered state of the cell as a combination of the prior gates' output.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad \text{(1) Forget gate}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad \text{(2) Input gate}$$

$$\tilde{C}_t = \tanh(W_C) \cdot [h_{t-1}, x_t] + b_C) \quad \text{(3) Candidate value}$$

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t \quad \text{(4) Update previous cell state}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{(5) Output gate}$$

$$h_t = o_t \cdot \tanh(C_t) \quad \text{(6) Hidden state}$$

There are a number of popular variants of LSTM cells such as peephole LSTMs, GRUs, Grid LSTMs and ConvLSTM [60]. Many areas have been revolutionised by the usage of LSTM cells in network architectures to learn to predict sequences of data with high accuracy. As we will describe in chapter 3.4.2, LSTM cells are the key which allows world models to learn to simulate the behaviour of the environment via 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. [61] for an excellent overview of the methods and applications of graph neural networks.

Battaglia et al. [5] 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. [5], a graph is defined as $G = (u, V, E)$ where u are the global attributes, $V = \{\mathbf{v}_i\}_{i=1:N^v}$ 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 edges with their sources and corresponding vertices.

Algorithm 2: Computation in a full GN block. Adapted from [5]

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 2 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 [12] used a pre-defined set of transformations that are applied greedily. We evaluated our approach against these traditional approaches in section 4.3.1. In addition, recent work, such as [34, 33] 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 [33] as it is most similar in terms of substitution discovery and we present the results in section 4.3.1.

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 where an agent acts. The work in [21] 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. In contrast, alternative approaches have been proposed such as imagination-augmented agents [59] and model-based value estimation for model-free agents [15]. Furthermore, Nagabandi et al. [43] proposed an method to combine the sample efficiency of model-based RL and the high-task performance of model-free agents; our work differs as we use a world model to RNN-based network to simulate the environment dynamics. Other work such as [50, 24] build discrete world models and train directly in latent space. Prior work on world models used a variation on a variational auto-encoders [21, 23] to generate a latent state of the pixel input, instead we use a graph neural network [5] 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 [39, 40, 2, 48], reinforcement learning was used to optimise the placement of machine learning models to improve throughput. In [10], 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 [2, 48] which also aims to reduce runtime or memory usage, our work is in a different domain. Moreover, applying model-based RL to graph optimisation poses significant challenges; tuning the world-model to reduce the likelihood of inaccuracies in the *imagined* environment and each roll-out environments are of variable length, generating a homogenous, uniform embedding of the state required careful design of the system architecture.

Chapter 3

XFlowRL: Reinforcement Learning Optimisation

In this chapter we will introduce the graph optimisation problem and describe the technical details of the design of the two reinforcement learning agents and their components in relation to prior work. 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 as well as highlighting limitations in the application of reinforcement learning to this problem. Finally, we discuss the relative benefits of each approach and 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.

3.1 Graph-level optimisation

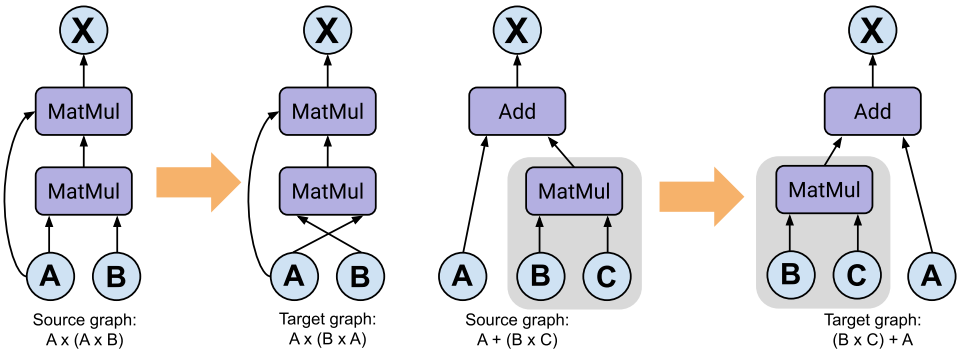
Performing optimisations at a higher, graph-level means that the resulting graph is—in terms of execution methodology—no different than the original graph prior to optimisation. Therefore, by performing graph-level optimisa-

tion 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 in Figure 3.1a and 3.1b respectively.



(a) Tensor renaming substitution

(b) Common subgraph substitution

Figure 3.1: 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.

3.1.1 Graph Embedding

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) [19] which reduce data along edges and between nodes in the graph. Alternatively, we considered using graph convolutional networks (GCNs) [36], however, we found that using messages passing networks produced an adequate generalisable embedding that leverages the relational biases in the graph structure without added complexity of graph auto-encoders or graph convolution.

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. [5], we implemented a graph neural network to learn a latent space embedding of the graph using message passing networks to gather the

global learned features from the graph.

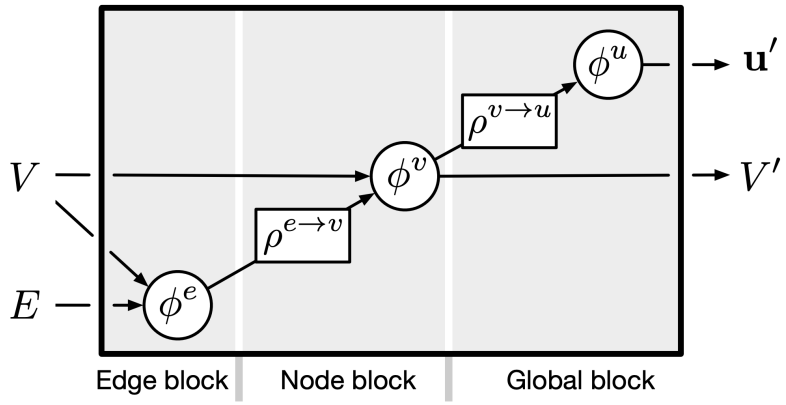


Figure 3.2: Message-passing Neural Network design using the three-step architecture described by Battaglia et al. [5].

As part of the design of the graph embedding network, we used a graph message passing neural network [19] that uses the input node, edge and global attributes to predict the node, edge and global attributes. We use five message passing rounds and a unsorted segment sum reducing function in each round of the node block for the function $\rho^{e\to v}$.

Furthermore, we 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 a model-free reinforcement learning agent [17]. We used their work as a foundation upon which we continued development and research into model-free and model-based RL.

3.2 Reinforcement Learning formulation

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

3.2.1 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 [9] or OpenSpiel [37] 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 3.3.

To simplify the implementation of the environment, we used made extensive use of the work by Jia et al. [33] 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 functions—we provide more detail in section 3.2.4.

The scope of our work meant that there was no existing prior work that

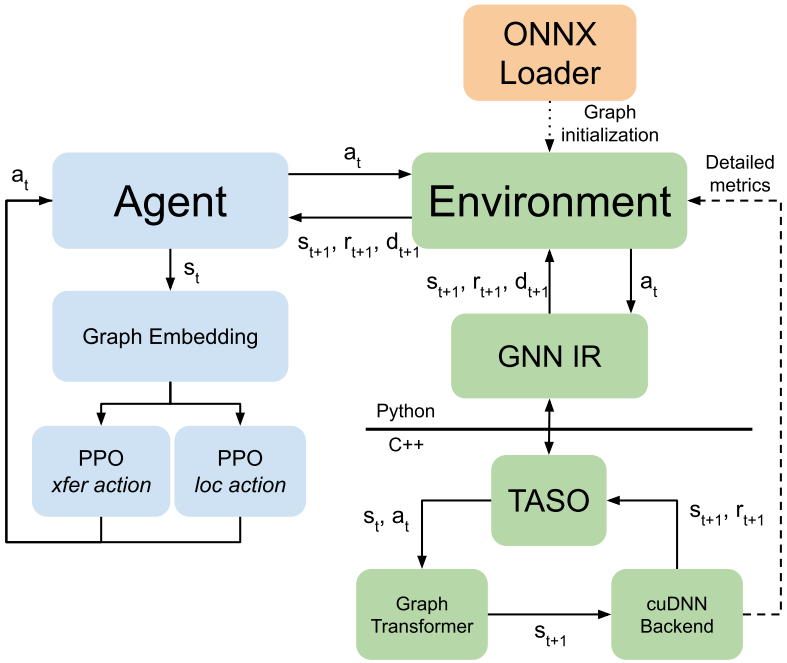


Figure 3.3: Data flow between components of the RL system. Although this diagram shows the setup for both the training of the model-free and model-based world model, we can use the environment in figure 2.3 as a drop-in replacement of the environment to train the model-based controller.

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 [6]. 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 sections 3.4 and 4.3.3 we will further explore world models and evaluate our implementation respectively.

3.2.2 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 [4] format which is a open-source standard for defining the structure of deep learning models. By extension, we can support any deep learning framework that supports serialisation of models into the ONNX format such as TensorFlow [1], PyTorch [49] and MXNet [11].

Next, we parse the ONNX graph 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 3.2.1. 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.

3.2.3 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. [40] used hierarchical RL with multiple actions to find the optimal device placement and Addanki et al. [2] 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

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

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 transformations 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.

3.2.4 Reward function

The design of a reinforcement learning agent consists of three key elements, the agent, environment and reward function. Most importantly, we require a reward function that captures dynamics of the environment in such a way that we can directly indicate to the agent if we consider the action to be "good" or "bad". For example, we wish to prevent the agent from performing actions that would be invalid in the environment, therefore, using the reward signal we provide a large negative reward to disincentivize the agent from replicating the behaviour. Conversely, we need to provide a positive reward, dependent on a chosen action and its impact on the agent performance.

Selecting optimal actions can be challenging in any deep reinforcement learning system, especially those with either long-term action dependencies or a large number of possible actions in any given state. Importantly, in our environment, the selection of a poor action be impactful on both subsequent action space and the resulting reward generated by the environment. Therefore, we used multiple reward functions to investigate the resulting performance of the agent. First, we used a simple reward function that is commonly used in sequential RL applications:

$$r_t = \begin{cases} RT_{t-1} - RT_t, & \text{if valid action} \\ \text{-100}, & \text{otherwise} \end{cases}$$

Using the reward function defined above, we use the previous estimated runtime, RT_{t-1} of the computation graph and the estimated runtime of the current graph, RT_t, to determine the step-wise, incremental change in graph runtime as the reward. This simple, yet powerful function has the benefit of a very low overhead as we only need to store the last runtime. Furthermore, as our primary goal is to reduce the execution time of the graphs, rather than for example the system memory, it captures our desired metric which we wish to optimise.

Secondly, we instrumented TASO to extract detailed metrics in an attempt

to engineer a more complex reward function; we used the runtime, FLOPS, memory accesses and kernel launches to perform experiments to determine if using a combination of the metrics could yield a higher performance RL agent. We defined modified reward function as shown below; where RT is the graph runtime, M is the memory accesses, α and β are two hyperparameters for weighting the runtime and memory accesses respectively.

$$r_t = \begin{cases} r_t = \alpha(RT_{t-1} - RT_t) + \beta(M_{t-1} - M_t), & \text{if valid action} \\ \text{-100}, & \text{otherwise} \end{cases}$$

We provide further discussion and motivation for our chosen reward functions in section 4.3.2 as well as an analysis of the detailed runtime metrics and the impact on improving graph runtime.

Finally, we note that TASO used a simple method to estimate the runtime of tensor operators that is 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 and can lead to a poor estimation of the hardware impact. As such, we investigated the use of using real runtime measurements during training rather than a estimation of operator runtime. After performing experiments with a modified version of TASO which averages the real runtime over N rounds, we found that it increases duration of each training step to such a degree that any possible performance improvements achieved using real hardware costs are not worth the trade-off.

3.3 Model-free Agent

In section 3.1.1 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. [52] 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 [51], 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. 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 describe the reward calculation in section 3.2.4.

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 3.2.3, the alternative approach we considered would lead to lower agent performance during training due to the larger action space.

3.4 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.

3.4.1 Benefits of model-based RL

When using a model-based approach over a model-free approach, we must weigh the benefits but also drawbacks. Importantly for system environment problems, a model-based approach decreases the wall-clock training time

by leveraging a reduction of the environment step time; in section 4.3.3 we show a 85x reduction in the wall-clock training time of model-based agents compared to the model-free agents.

Furthermore, using a world model can aid in stabilising agent reward variance we observe while training a controller inside the world model. By tuning the hyperparameters associated with the world model (i.e. the temperature τ), we can provide softer targets for the agent to learn. As we show in section 4.3.3, the world model temperature has a significant influence on the stability of the model-based agent during training. Additionally, in section section 4.3.3 show that the model-based agent is generally more stable with optimised hyperparameters during training.

3.4.2 World Models

World models, introduced by Ha et al. [21], 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 aim to learn the world model using the latent space from the environment.

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 [53]. Importantly, as the output of an RNN is deterministic, we use the outputs from the RNN as the parameters for a probabilistic model to insert

a controllable level of stochasticity in the output predictions; a method first proposed by Graves [20].

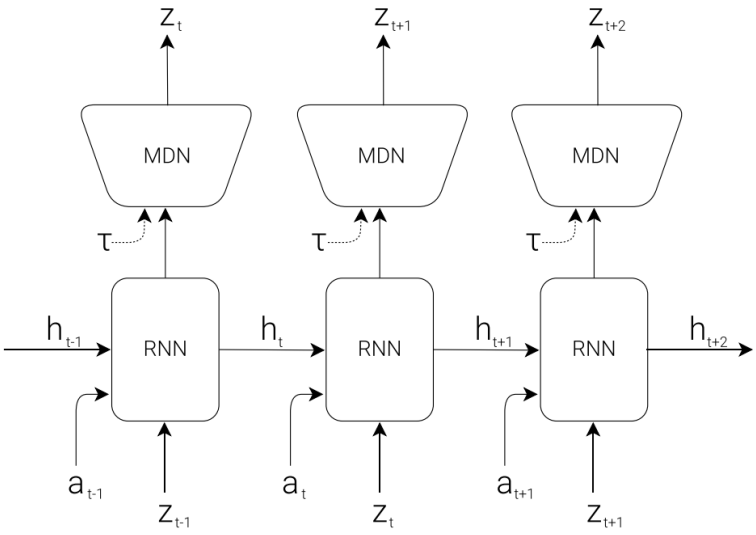


Figure 3.4: Structure of an unrolled MDN-RNN. The MDN outputs the parameters of a Gaussian mixture distribution used to sample a prediction of the next latent vector z_{t+1}, the MDN is controlled by the temperature parameter τ.

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 (World Model)

By combining the mixture density and recurrent networks, we can use rollouts of the environment sampled using a random agent to train the combined net-

work, 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 3.4 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 3.4 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 $\tau \in [0, 1)$, 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. [21] 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 per-

forming 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 [28]. 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 [21] who performed 10,000 rollouts of the environment offline using a random policy to collect the data, we took a different approach.

Rather than generating large rollouts offline, 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.

3.4.3 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 [21] 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 [21] the authors used an evolutionally algorithm, covariance matrix adoption evolution strategy (CMA-ES) [26, 25], which optimises the weights of the network based on the reward produced by the world model. Alternatively, recent work by Hafner et al. [23, 24] 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 [10].

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. For our work, we used the same algorithm to train the agent inside the real environment and in the world model, thus, acting as a good point of comparison for performance of the two methods. We show the results for training the model-free and model-based controllers in sections 4.3.2 and 4.3.3 respectively.

Chapter 4

Evaluation

4.1 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 the world models accurately model the reward estimation from the graphs latent state?
4. Are the agents trained in an imagined world model applicable to the real-world environment?

Throughout this chapter, we aim to answer these questions by a series of experiments which provide evidence to support our claims. Finally, we conclude with an overall discussion of our findings and its impact.

4.2 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 [1]. 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.

To interface with the internal representation of the computation graphs, as previously discussed, we used the open-sourced version of TASO [33] 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. [5] to process our input graphs which we described in chapter 3.2.1. The PPO agent was implemented based upon the implementation provided by Schulman et al. [52].

4.2.1 Graphs Used

	InceptionV3	ResNet-18	ResNet-50	SqueezeNet1.1	BERT
Type	Convolutional	Convolutional	Convolutional	Convolutional	Transformer
Layers	43	18	50	21	12
Unique Layers	12	6	6	3	3
Substitutions	56	40	228	288	80

Table 4.1: Properties of the five evaluation graphs used in the experiments contained in this chapter. We differentiate the total number of layers in a network from the number of unique layers used in composing the network to provide a more accurate representation of its complexity.

We chose to use five real-world deep learning models to evaluate our project. InceptionV3 [57] is a common, high-accuracy model for image classification trained on the ImageNet dataset [2]. ResNet-18 & ResNet-50 [27] are also deep convolutional networks that are 18 and 50 layers deep respectively.

[1] `https://www.gitlab.com/CamRL/xflowrl`

[2] `https://image-net.org/index.php`

SqueezeNet [31] is a shallower yet accurate model on the same ImageNet dataset. BERT [14] is a recent large transformer network that has been to improve Google search results [44]. As these graph were also used in the evaluation of TASO [33], we can show a direct comparison of the performance between the different approaches.

4.3 Experiments

4.3.1 Baselines

In this section, we will establish the baseline performance results from prior work and modern machine learning frameworks such that we can compare against our proposed approach and quantitatively analyse the results. We show the runtime metrics of the five graphs described in section 4.2.1 that are optimised using TensorFlow [1], TensorRT [46] and TASO [33].

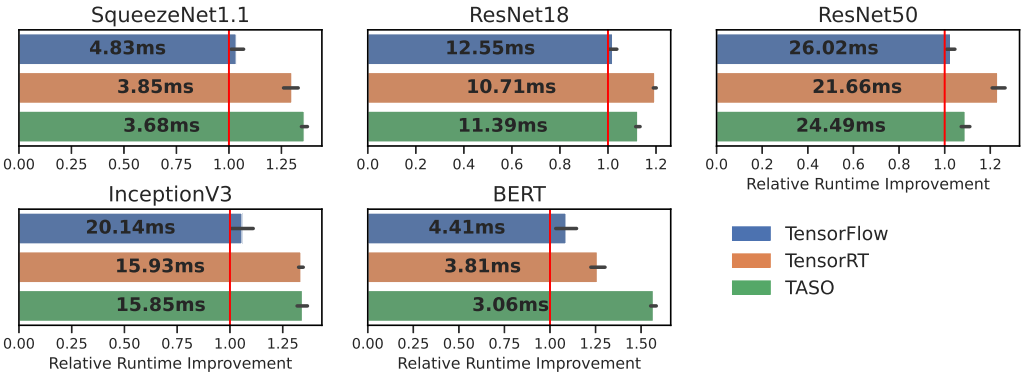


Figure 4.1: Runtime of optimised graphs using the three baseline optimisation methods. The x-axis shows the relative runtime improvement, a higher relative runtime is better.

Figure 4.1 shows the runtime of each optimised graph described in section 4.2.1 using the three baseline methods, TensorFlow [1], TensorRT [46] and TASO [33]. We observe that TASO outperforms TensorFlow Grappler and TensorRT on BERT by 50.5% and 43.6% respectively. On the other hand,

with convolutional networks, the optimised graph discovered by TASO has a runtime within ±6% compared to TensorRT. Furthermore, we note that during our reproduction of the results found by Jia et al. [33], we used the same value of $\alpha = 1.05$ and a search budget of 50,000 steps. TASO often found the optimal graph within ∼5000 steps and the remaining computation steps failed to further improve the estimated runtime.

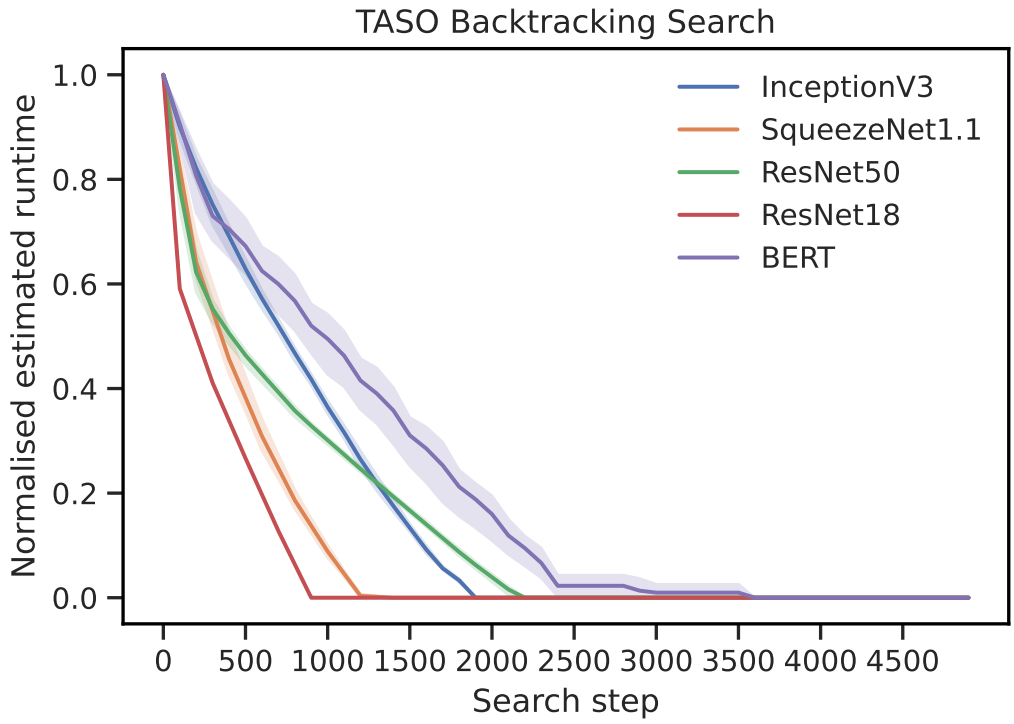


Figure 4.2: We show the estimated runtime improvement, normalised between the minimum and maximum obtained runtimes, for each tested graph. We performed each experiment five times and show the 95% confidence interval for each graph. We also note that we clipped the plot as after 5000 steps, there was no increase in performance for the remaining 45000 search steps.

Figure 4.2 shows a plot of the runtime estimated by TASO at each step in the search. We repeated each experiment five times and show the confidence interval for each graph. Based upon the results, we observe that TASO did not just take the longest to discover the optimised graph, the variance between runs was the highest compared to other graphs. One reason for this

disparity is that it has a vastly different architecture; BERT is a transformer network which, compared to convolution networks, has a greater breadth than depth. As such, when TASO performs the backtracking search, there are far more initial locations in the graph where a substitution can be applied.

4.3.2 Model-Free Agent

In this section we describe our experiments performed using the model-free agent which acts inside the real environment. Firstly, we trained the agent on each graph under consideration and evaluated its optimised runtime, the results of which we present in figure 4.3. In the worst case, the model-free agent performs a series of optimisations that increases runtime by 9% compared to those discovered by TASO.

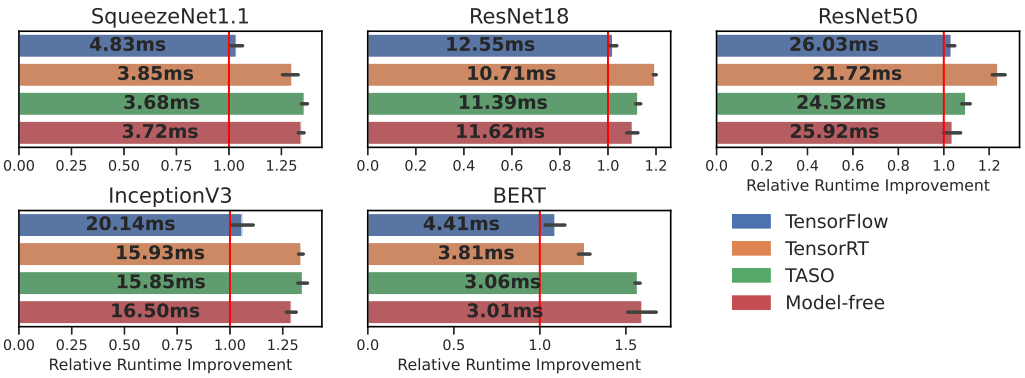


Figure 4.3: Runtime of optimised graphs using an agent trained using the model-free PPO algorithm. We also show the baseline results as comparison. The x-axis shows the relative runtime improvement, a higher relative runtime is better.

Figure 4.4 shows the reward produced by the model-free agent acting inside the real environment for each graph. Due to the difference in estimated runtime between graphs, we used min-max normalisation to scale the rewards into the same range. First and foremost, it is evident that the graph optimisation using the model-free agent reward converge quickly after only $\sim$1000 epochs with low variation in the average epoch rewards after convergence.

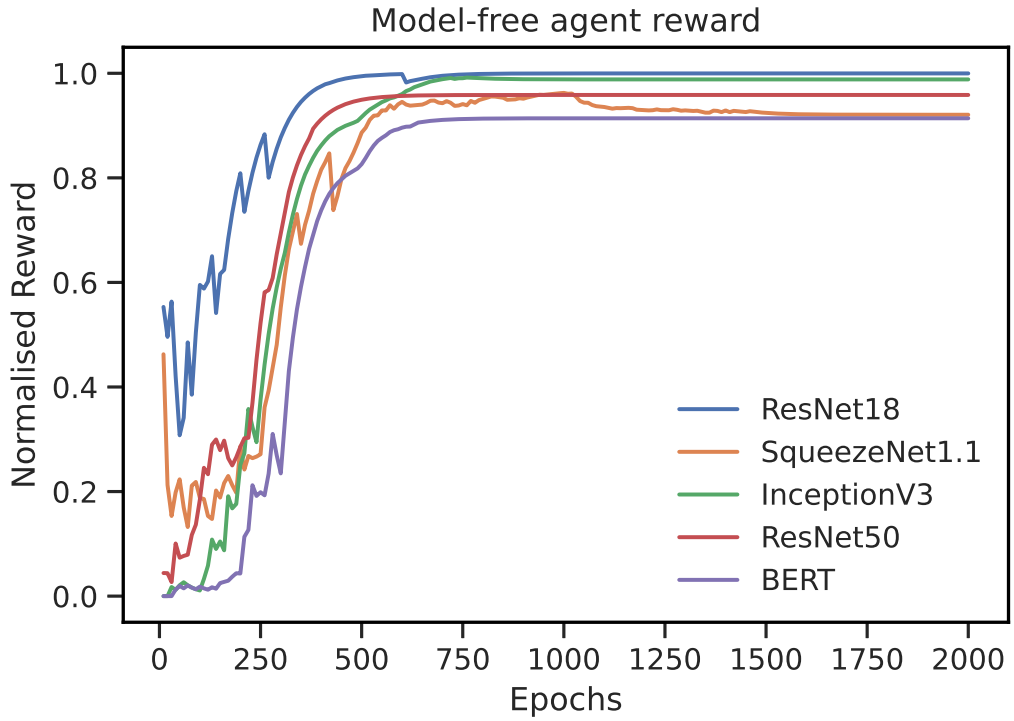


Figure 4.4: Normalised reward of model-free agent produced by the real environment in response to selected actions by the agent

Reward functions

As we described in section 3.2.4, the design of the reward function used in the training of RL agents is a pivotal part of the agents architecture. In this section, we analyse our proposed reward functions and the effect on the convergence as well as final performance of the trained agents.

Figure 4.5 shows the pairwise relationship between the four detailed runtime measurements which we record at each step of the training process for the BERT graph. We collected the runtime, FLOPS, memory access and number of kernel launches and show the relationship between the values. We note that the estimated runtime and memory accesses have a strong correlation; when the memory access decreases, we see a notable decrease in estimated runtime.

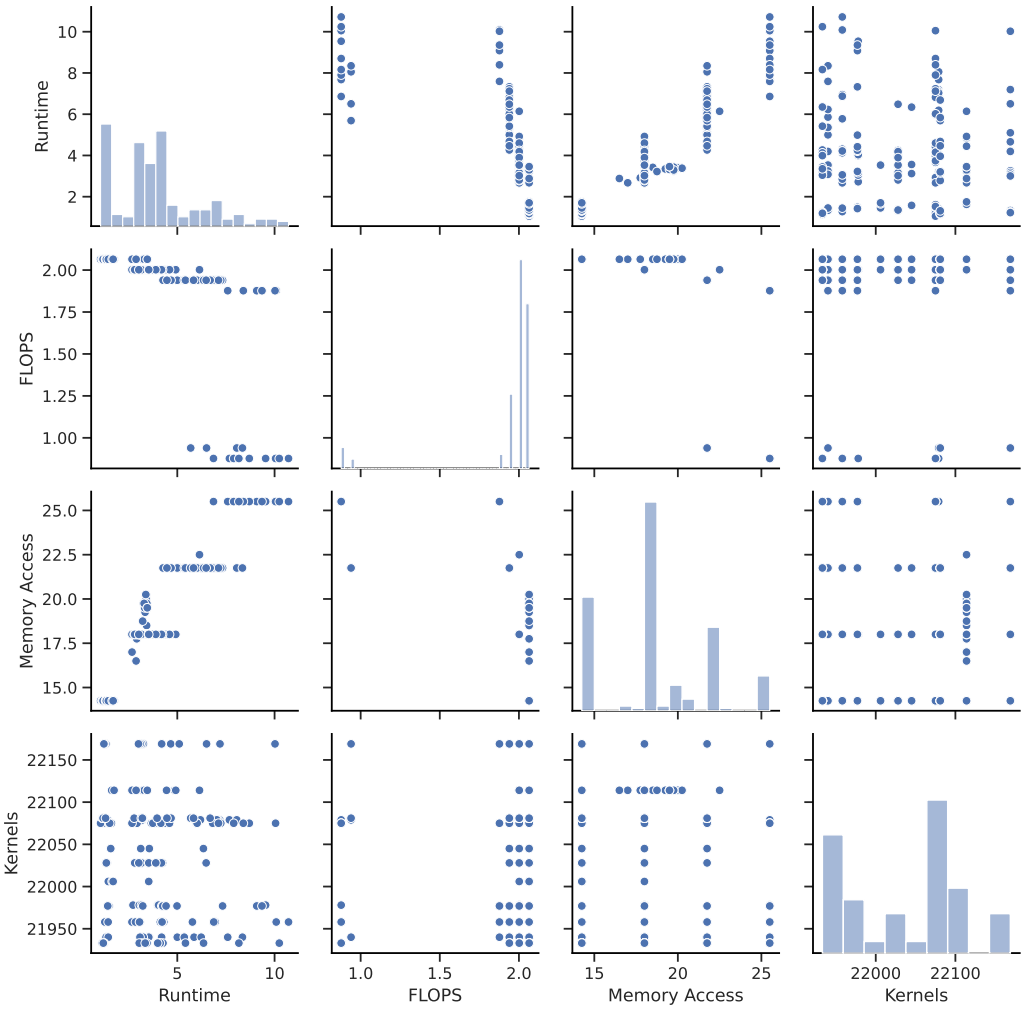


Figure 4.5: We show the pairwise correlation between the detailed runtime metrics recorded directly from the TASO environment while training on the BERT graph. We note the strong correlation between the number of memory accesses and estimated runtime improvement.

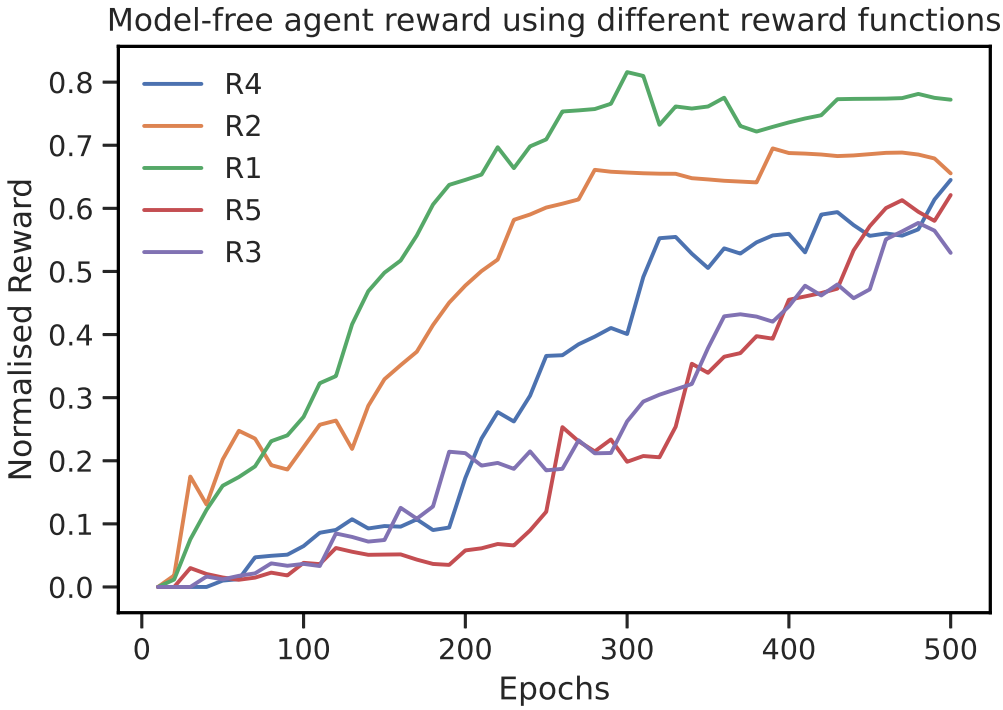


Figure 4.6: We show the normalised reward of each agent using various reward functions while being trained for 500 epochs. R1 uses the second reward function with tuned parameters, R2 uses new runtime reward, R3 uses $\alpha = 0.1, \beta = 0.9$, R4 uses $\alpha = 0.5, \beta = 0.5$ and finally, R5 uses incremental runtime improvement.

Figure 4.6 shows the effect on convergence of model-free RL agents while training on the BERT graph using various reward functions. Significantly, using a standard, yet naive approach of the first reward function described in section 3.2.4 shows that the model-free agent improves at a linear rate as shown by R4 in figure 4.6. Whereas, using the second reward function and tuned hyperparameters of α and β, shown by R1, converges the fastest.

However, the rate of convergence does not show the whole picture of the agents performance. We note that the highest performing agent, after the restricted 500 epoch training time, was R1 with an average runtime improvement of $48.7 \pm 3.2\%$. Surprisingly, the second highest performing agent was the agent using R4, the simplest reward function from the set tested, with a

performance of $43.2 \pm 2.3\%$.

In order to find the values of hyperparameters α and β which results in the agent with the maximum performance, we performed a grid search for α and $\beta = 1 - \alpha$ between the values of $[0, 1]$; we searched between the bounds in increments of 0.1. After training each agent for 500 epochs and evaluate the performance of each agent three times, we found that the reward function resulting in the highest performance was using the values 0.8 and 0.2 for α and β respectively.

4.3.3 Model-based Agent

In this section we first present the results for training an agent inside a world model for each graph individually. Secondly, we compare the model-based agent performance to baseline measurements as well as showing the change in memory usage which is a by product from the applying graph transformations in order to reduce runtime. Furthermore, we also discuss the impact of hyperparameter selection on the agent performance as well as showing the accuracy of the world model in regards to graph reward prediction.

Runtime Performance

Figure 4.7 shows the runtime of the optimised graphs for the model-based agents trained inside the fully hallucinogenic world model. Each agent was trained inside a world model using rollouts from its respective graph as described in section 3.4.3. We trained the agents for a maximum of 1000 epochs, in mini-batches of 10 epochs. Additionally, we used a fixed learning rate for both the policy and value networks during training of the controller agent network.

Firstly, we note that training the agents on convolutional networks, especially SqueezeNet1.1 and InceptionV3, the model-based agent failed to outperform TASO, we still decreased the runtime compared to the baseline graph pro-

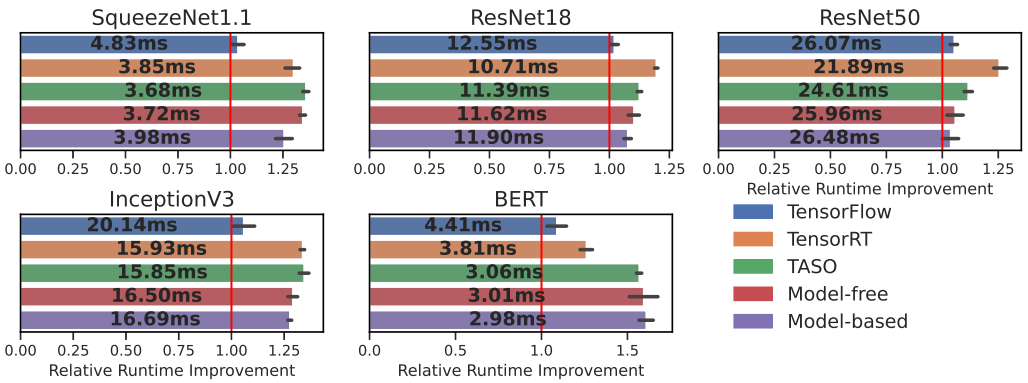


Figure 4.7: Runtime of optimised graphs using an agent trained using the model-based world model. We also show the baseline results as comparison. The x-axis shows the relative runtime improvement, a higher relative runtime is better.

duced by TensorFlow Grapper. Importantly, we observe that the model-based agent outperformed all baseline approaches on the BERT transformer network; we improved the runtime by 54.1% and 7.3% compared to TensorFlow and TASO respectively. Figure 4.11 shows the transformations applied by the model-based agent on the test graphs and compared to TASO, we only apply a single transformation over 20 times—compared to TASO which uses four distinct transformations produce the optimised graph.

Compared to the strictly model-free agent that was trained in the real environment, our model-based agent achieved a similar level of performance in the majority of the tested graphs. The model-free agent was trained for 2000 epochs and, by extension, over 4,000,000 interactions with the real environment. Comparatively, the model-based agent performed approximately 1,000,000 interactions with the real environment as the agent did not interact with the real environment while training inside the world model. Therefore, it is evident that by training inside the world model we improved the sample efficiency of the agent. On the other hand, the performance of the agent decreased compared to the model-free agent in four of the five tested graphs.

Furthermore, an important consideration when training inside a systems environment is the wall-clock time for stepping the environment to a new state

based upon the agent action. We analysed the time required to perform a single step while training the ResNet50 graph. We found that stepping the world model (performing inference of the world-model) takes, on average, 10ms whereas stepping the real environment takes on average 850ms. Thus, although the performance of the model-based agent was comparatively lower, our wall-clock time for required for training was reduced by a factor of 85x.

Memory Usage

	Baseline (TF)		Optimised (MB-RL)	
	Inf. time (ms)	Mem. usage (GiB)	% Improvement	
ResNet18	12.2	1.18	3.0%	1.1%
ResNet50	26.7	2.34	1.0%	0.6%
InceptionV3	17.6	2.11	12.5%	2.3%
SqueezeNet1.1	4.6	1.14	18.9%	1.8%
BERT	4.1	0.26	54.1%	4.5%

Table 4.2: Relative performance improvement of the graphs optimised by the model-based agent. We show the inference time, and memory used for performing inference on the model.

Table 4.2 shows the percentage improvement of both the inference time and the memory used for performing inference on the optimised models. Importantly, although we tasked the agent to optimise for reducing the runtime of the graphs, we observe an unintended secondary effect of a reduction in memory usage of up to 4.5% over the baseline TensorFlow model.

World-model accuracy

The training of the model-based agent is split into two parts. First, we train the world-model, the network that learn to simulate the environment dynamics, and secondly, we train the controller network inside the world-model. In this section, we show the convergence of the world model during training in figure 4.8. The figure is a plot of the log-likelihood loss per training

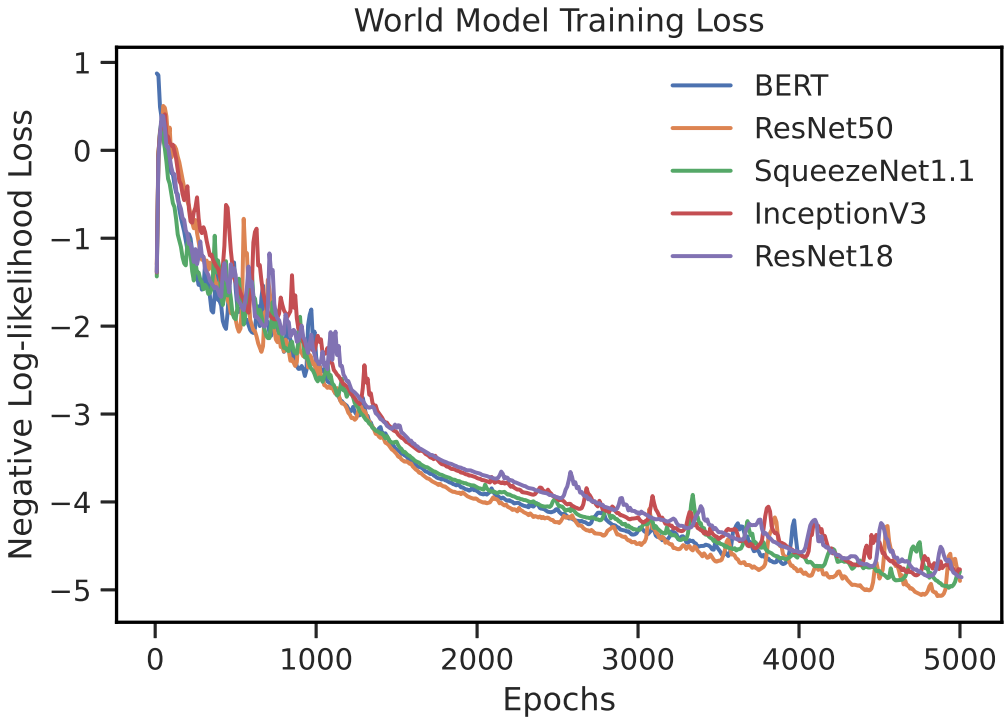


Figure 4.8: Training plot of the log-likelihood loss for our world model on the five test graphs.

epoch for each graph. We used the same hyperparameters for training each world model as well as decaying the learning rate over the course of 2000 epochs with a 2nd-degree polynomial decay policy. The MDN-RNN is trained with 8 Gaussians and 256 hidden units, all other hyperparameters used in training the MDN-RNN world model are the same as those used by Ha and Schmidhuber [21], unless otherwise stated.

Figure 4.9 shows the reward (decrease in estimated runtime) for each graph as predicted by the world model during training. As the tested graphs have a wide range of epoch rewards, we perform min-max normalisation to scale the plots into the same range. We observe the same results as figure 4.7 in which the optimisations applied to BERT during training results in the optimal graph found after 700 epochs. On the other hand, graphs such as ResNet 18/50 are less stable during training with a high epoch to epoch variation

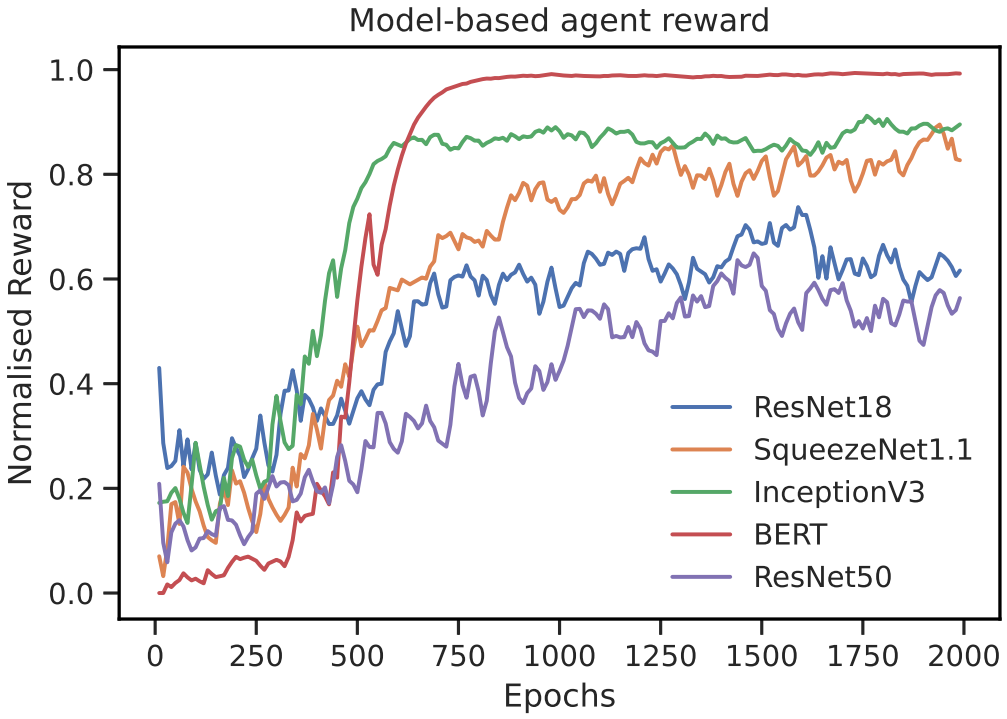


Figure 4.9: Predicted reward produced by the world model while training the agent inside a the imagined environment. All rewards are normalised into the same range.

in rewards. In comparison to the rewards received by the model-free agent during training, we note that the strictly model-free agent was more stable during training, and additionally, consistently found the optimised graph after approximately 1000 epochs.

Discussion

If we assume that both the model-free and model-based agents should achieve a similar level in performance once trained, the results in figure 4.9 and 4.4 show that the agents trained in the world model are less stable with a higher reward variance. We hypothesise that there are three factors for such a discrepancy to occur:

- Imperfect world-model reward predictions leading to incorrect (or in-

valid) actions being performed

- Next state prediction by the world-model generating states that are invalid due to poor generalisation of the model
- Incorrect action mask predictions that would lead to a divergence in between the world-model state and real environment state

In an attempt to resolve the issues highlighted above, we performed further experiments which we believe would aid in both reducing the variance in reward prediction as well as stabilise the world-model during training to prevent state divergence over time. We performed a temperature sweep of the hyperparameter τ which is used in training agent inside the world model, shown in section 4.3.3.

Temperature Sweep

Temperature	World-model Score	Real Score
0.1	-6.67% ± 0.6%	-43.92% ± 5.1%
0.5	-7.75% ± 0.3%	-55.33% ± 6.7%
0.75	-9.10% ± 0.4%	-55.80% ± 5.2%
1.0	-8.85% ± 1.2%	-55.78 ± 4.0%
1.2	-9.91% ± 0.8%	-57.01% ± 3.9%
1.5	**-8.37% ± 0.6%**	**-58.23% ± 3.6%**
1.75	-9.92% ± 1.0%	-52.07% ± 5.8%
2.0	-9.65% ± 0.8%	-46.12% ± 5.4%
2.5	-10.04% ± 2.0%	-41.14% ± 10.2%
3.0	-10.38% ± 1.9%	-51.32% ± 7.2%

Table 4.3: Temperature sweep of trained model-based agent optimising the BERT network. We ran each experiment five times and show both the average performance improvement as well as the variance between runs.

Table 4.3 shows the results from performing a temperature sweep in which we used different values of τ while training the agent in a world model. After training, we evaluated the agent which produced an optimised graph that we

evaluated to determine average runtime. The table shows the average reduction in runtime and standard deviation, averaged over five runs, compared to the unoptimised graph. The motivation for using a range of temperatures is that a higher value of τ leads to softer targets for the agent to predict, thereby improving generalisation. Conversely, a lower value of τ presents hard targets and thus when $\tau = 1.0$, it is equivalent to using the unmodified mixing weight, π, of the MDN.

Based upon the results in table 4.3 from the conducted experiments, we note that the world model agents are stable to temperatures within the range of $\tau = 0.5$ to $\tau = 1.75$. Although the runtime improvement world-model from the environment is consistently above 6%, we observe a large difference between the predicted runtime improvement and the real environment reward.

Agent Convergence

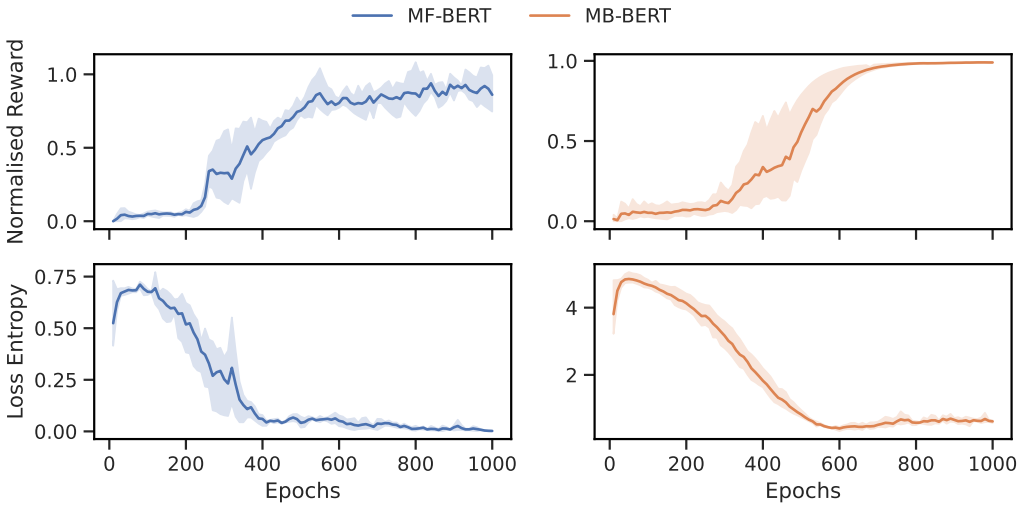


Figure 4.10: Agent reward and loss for model-free and model-based agents shown on the left (blue) and right (orange) respectively. We trained the agents on the BERT graph for 1000 epochs and repeated the experiments five times. We show the 95% confidence interval in the shaded area of each plot.

As we have previously claimed, there are many benefits to using a world

model as the environment where we train the controller agent. In Figure 4.10 we show the agent reward and loss during training for 1000 epochs on the BERT graph using model-free (MF) and model-based (MB) agents. Also, we used the results from our previous temperature sweep experiment by choosing the temperature $\tau = 1.5$ for use when training the controller. We found that the model-based agent, when trained using tuned hyperparameters, was more stable than the MF agent trained inside the real environment; the variance between runs was lower for the model-based agent compared to the MF agent. However, we found that the variance in the optimised runtime of the models produced by the MF agent was larger than the MB agent, with a variance of 6.1% and 3.6% respectively.

Graph transformations

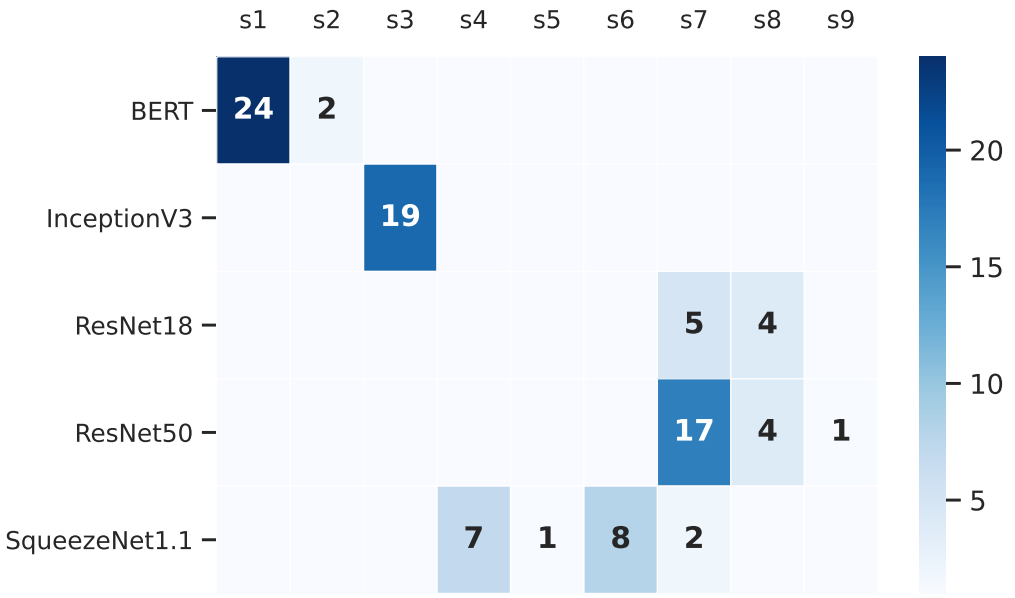


Figure 4.11: Heatmap showing the transformations applied by the trained controller acting inside the world model. Although there are over 100 possible transformations, we only show the transformations applied onto at least one graph. The counts for each transformation show the number of times it has been applied.

Figure 4.11 shows a heatmap of the various graph transformations which have been applied by a trained model-free agent during evaluation. Notably, the optimisations applied to the ResNet18/50 graphs apply similar transformations, those targeting the convolutions in the network; as the networks are composed of alike convolutional operators, with different depths, we apply analogous transformations. On the otherhand, for recurrent networks such as BERT, we only apply relatively few transformations. This is in stark contrast to the series of transformations found by TASO which applies four distinct substitutions in comparison to the two applied by our approach. Despite the large disparity in both the specific transformations as well as number of times we applied transformations, the performance difference between TASO and our proposed method is small.

4.4 Discussion

Throughout this chapter we have evaluated our claims which we formed in the introduction. We have provided the results to various experiments covering the reinforcement of our baseline measurements and those which show the performance of the proposed agents. Overall, we have found that our proposed approach outperforms the TensorFlow optimisation strategy on all graphs. However, in some cases, namely the deeper convolutional networks such as ResNet50 and InceptionV3, our agents failed to apply optimisations that outperform those performed by TensorRT and TASO.

In examining the model-based agent performance, we found that a world-model is able to accurately learn and simulate the transition dynamics of an environment, showing that despite the complex nature of the state-action transitions the world-model is flexible to learn its behaviour and adapt to previously unseen state-action pairs.

Notably, although the state-action prediction and the terminal state prediction was accurate, we found the reward prediction had a large error when compared to the real reward produced by the environment. However, and

surprisingly, this disparity between the two rewards did not have a significant impact on the agent performance as we showed in section 4.3.3, the agent trained inside the world model is stable to a wide range of operating conditions.

Finally, the primary takeaway from these experiments is that reinforcement learning, especially model-based methods, are extremely powerful and can learn to model complex dynamic systems and provide important benefits. Nonetheless, we note that such methods still suffer instability during training, and imperfections in the world-model leads to agent exploitation of the faulty environment and therefore poor performance in the real environment—tuning of the world-model hyperparameters remains vital to producing a stable, performant agent.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this work, we have shown the result of applying deep reinforcement learning techniques to the task of optimising deep neural networks. Our approach uses RL agents to select optimal actions that optimise the computation graphs with the goal of improving on-device runtime. We performed experiments that show RL agents decreased the runtime on all of the five test graphs, each of which had unique properties and architectures. Notably, we provide evidence to support our claim that it is possible to learn a world-model of the environment which is sufficiently accurate to enable the end-to-end training of an agent inside a fully imagined world-model. Furthermore, we built upon prior work by Jia et al. [33], and developed a deeply instrumented environment in which we can train model-free agents as well as world-models for model-based agents.

In addition, this work has highlighted that the performance of model-based agents trained inside a hallucinogenic is highly dependent on the accuracy of the world-model. Inaccuracies in the model, can lead to compounding errors and thus the agent choosing sub-optimal, or invalid, actions that diverges the imagined state from the true environment state. Hence, there are still

significant fundamental difficulties in training stable, accurate world-models that can simulate the true environment; if one can train such a model by carefully tuning hyperparameters, we can gain substantial benefits through increased sample efficiency and decreased training time.

5.2 Future Work

In this work, we have presented an approach to using message-passing neural networks to exploit the relational biases in the graph structure of the input and produce a embedding of the graph in latent space. One possible direction for future work is to investigate the use of graph auto-encoders [5] to produce a reconstructed graph which can be used for planning. Recent work [22, 54] has shown that using an accurate world-model, we can exploit the model to plan our actions into the future state space to gain higher performance than would be possible using single-step state predictions.

Model-based reinforcement learning is an active area of research and recent work has shown significant improvement in the performance of model-based methods on tasks in which model-free methods traditionally excel. Hafner et al. [24] show that using discrete world models that learn directly in latent space using planning, actor-critic agents can surpass model-free in the Atari environment. We propose that a future direction of work can be to investigate the use of discrete world models with actor-critic agents that offer greatly stability during training and learn more accurate world models which is especially critical to the performance of agents in system environments.

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. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] 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.

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

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

[5] 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, Fran-

cis 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.

[6] 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. ISSN 1076-9757. doi: 10.1613/jair.3912. URL `http://dx.doi.org/10.1613/jair.3912`.

[7] Richard Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. ISSN 00959057, 19435274. URL `http://www.jstor.org/stable/24900506`.

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

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

[10] Harrison Brown, Kai Fricke, and Eiko Yoneki. World-Models for Bitrate Streaming. *Applied Sciences*, 10(19), 2020. ISSN 2076-3417. doi: 10.3390/app10196685. URL `https://www.mdpi.com/2076-3417/10/19/6685`.

[11] 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.

[12] 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.

[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen,

John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning, 2014.

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

[15] 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.

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

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

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

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

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

[21] David Ha and Jürgen Schmidhuber. Recurrent World Models Facilitate Policy Evolution, 2018. URL `https://papers.nips.cc/paper/7512-recurrent-world-models-facilitate-policy-evolution`. `https://worldmodels.github.io`.

[22] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning Latent Dynamics for Planning from Pixels. *arXiv preprint arXiv:1811.04551*, 2018.

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

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

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

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

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

[28] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[29] 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.

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

[31] 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.

[32] 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. ISBN 9781450330633. doi: 10.1145/2647868.2654889. URL `https://doi.org/10.1145/2647868.2654889`.

[33] 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. ISBN 9781450368735. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630.

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

[35] 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.

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

[37] 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. URL http://arxiv.org/abs/1908.09453.

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

[39] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Ben-

gio, 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.

[40] 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.

[41] 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.

[42] 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*, pages 1928–1937, 2016.

[43] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning, 2017.

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

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

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

[47] 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.

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

[49] 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. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

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

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

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

[53] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. doi: 10.1109/78.650093.

[54] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models, 2020.

[55] 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.

[56] 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.

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

[58] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[59] Thophane Weber, Sbastien Racanire, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomnech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning, 2018.

[60] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

[61] 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.