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

April 30, 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

Write a summary of the whole thing. Make sure it fits in one page.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Background and Related Work

2.1 Introduction to Deep Learning Models

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

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

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

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

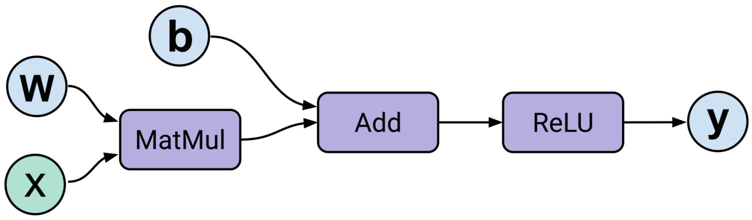


Figure 2.1: The operations shown in purple are the nodes of the computation graph which take an arbitraty number of inputs, performs a computation at the node and produces an output. The blue nodes represent the input nodes for tensors. The directed edges show the flow of tensors through the graph.

Similarly, the whole model can be converted into a stateful dataflow graph in this manner. By using a stateful dataflow (or computation) graph, we can use any optimisation technique for backpropagation of the model loss though the graph [TODO rewrite last sentence]. We consider two key benefits of this representation. First, we can execute the model on any hardware device as the models have a single, uniform representation. Secondly, it allows for pre-execution optimisations based on the host device, for example, we may perform different optimisations for executing on a GPU compared to a TPU.

2.1.1 Current approaches to optimising deep learning models

Due to the prevalence and importance of machine learning, especially deep networks, there is a focus on finding ways decrease the inference runtime and by extension, increasing the model throughput. All major frameworks such as TensorFlow [1], PyTorch [2], MXNet [3], and Caffe [4] have some level of support for performing pre-execution optimisations. However, the process of performing such optimisations is often time-consuming and cannot be completed in real-time. Rather, it is common to use a deep learning optimisation library such as cuDNN [5] or cuBLAS [6] that instead directly optimise individual tensor operations.

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

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

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

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.

- TODO Should also mention POMDPs?

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

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

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

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

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

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

2.2.1 Model-Free and Model-Based RL

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

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

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

Unfortunately, learning a model of the environment is not trivial. The most challenging problem that must be overcome is that if the model is imperfect, the agent may learn to exploit the models deficiencies, thus making it effectively useless in the real environment.

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

2.2.2 World Models

World models, first introduced by Ha and Schmidhuber, motivated and described an approach to model-based reinforcement learning in which we learn a model of the real environment using function approximators and train an agent using only predictions from the world model. Figure 2.2 shows the design to utilise a world model as substitute for the real environment. In practise, a world model can be broken down into three main components.

A visual model, V, that encodes the input into a latent vector z, a memory model, M the integrates the historical state to produce a representation that can be used as planning for future actions and rewards. Finally, a controller, C that uses both V and M to predict an action from the action set, $a \in \mathcal{A}$.

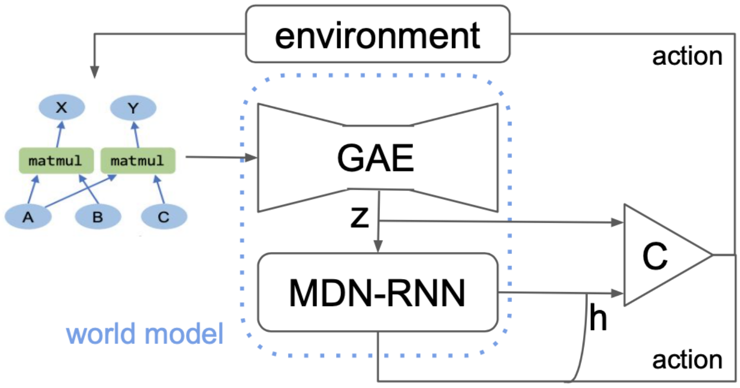


Figure 2.2: Diagrammatic representation of a world-model made up from an encoder (V) that transforms the input into latent space, a 'memory' module, M, that learns the behaviour of the environment from the latent vector z and a controller, C, that is trained using the latent vector of the encoder and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

Typically, a world model is trained using rollouts of the real environment that have been sampled using a random agent acting in the environment. The aim is to learn to accurately predict, given a state s_t, the next state s_{t+1} and the associated reward r_{t+1}. After training, the controller, C, can either learn using only observations from the world model, so called "training in a dream". Alternatively, the world model can be used to augment the observations from the real environment samples or used only for planning. To construct the world model, if the environment is simple and deterministic, it is possible to use a deep neural network to act as the world model, however, for environments that are only partially observable, a more complex model is required such as Recurrent Neural Networks (RNNs) or Long-short term memory (LSTMs). 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 [21] that were designed to deal with problems where there is an inherent uncertainty in the predictions. Given an input to the network, we wish to output a range of possible outputs conditioned on the input where we can assess the probability of each outcome. MDNs are commonly parameterised by a neural network that is trained using supervised learning and outputs the parameters for multiple mixture of Gaussians.

Recurrent Neural Networks

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

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

An LSTM can be described used four "gates", where a gate influences a property of the behaviour of the LSTM cell. The *forget* gate dictates if the information stored in the cell should be erased by observing the inputs $[h_{t-1}, x_t]$ it outputs a value in the range $[0, 1]$, using the sigmoid function

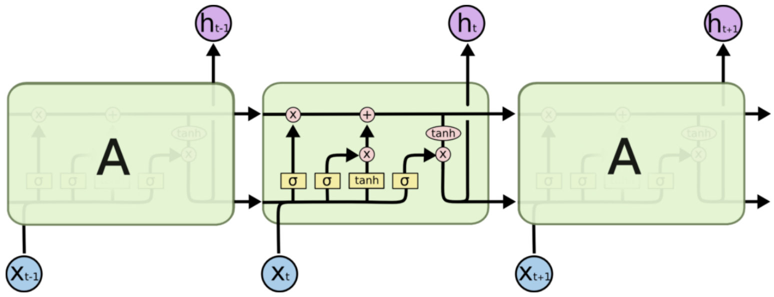


Figure 2.3: LSTM

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

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

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

2.3 Graph Neural Networks

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

Battaglia et al. [25] defines a generalisable framework for entity/relation based reasoning with three main operators that act on edges, nodes, and on global features using user-defined functions. Within the framework described by Battaglia et al., a graph is defined as $G = (u, V, E)$ where u are the global attributes, $V = \mathbf{v}_{ii} = 1 : N^v$ is the set of vertices (with a cardinality of N^v) and finally, $E = (\mathbf{e}_k, r_k, s_k)_k = 1 : N^e$ is the set of edge attributes with their sources and corresponding vertices.

Algorithm 1: Computation in a full GN block. Adapted from [25]

for $k \in \{1 \ldots N^e\}$ **do**
  $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$
end
for $i \in \{1 \ldots N^n\}$ **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 $\to$ vertices $\to$ globals. Algorithm 1 describes, in general, the algorithm to perform an update of a graph block.

- Graph Auto-encoders

Chapter 3

Problem Specification

In this chapter we will introduce the graph optimisation problem and establish the baseline performance using cost-based backtracking optimisation. Next, we will frame the optimisation problem in the RL domain by describing the system environment, the reward calculation and the state-action space. Additionally, we describe the RL agents trained in the model-free and model-based domains and we also highlight limitations in the application of reinforcement learning to this problem.

3.1 Introduction

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

3.2 Optimisation of deep learning graphs

TensorFlow (TF) uses a system called *"Grappler"* that is the default graph optimisation system in the TF runtime [26]. By natively performing the graph optimisation at runtime, it allows for a interoperable, transparent optimisation strategy via protocol buffers. To improve the performance of the underlying model, Grappler supports a range of features such as the pruning of dead nodes, removal of redundant computation and improved memory layouts. Concretely, Grappler was designed with three primary goals:

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

On the other hand, although Grappler can automatically optimise the 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; to further complicate matters, new operators are continuously proposed, such as grouped or transposed convolutions, all of which leads to a large amount effort expended to maintain the library. Secondly, and perhaps more importantly, as TF uses Grappler at runtime by default, it adds overhead to execution as extra graph conversions are performed at runtime rather than offline.

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

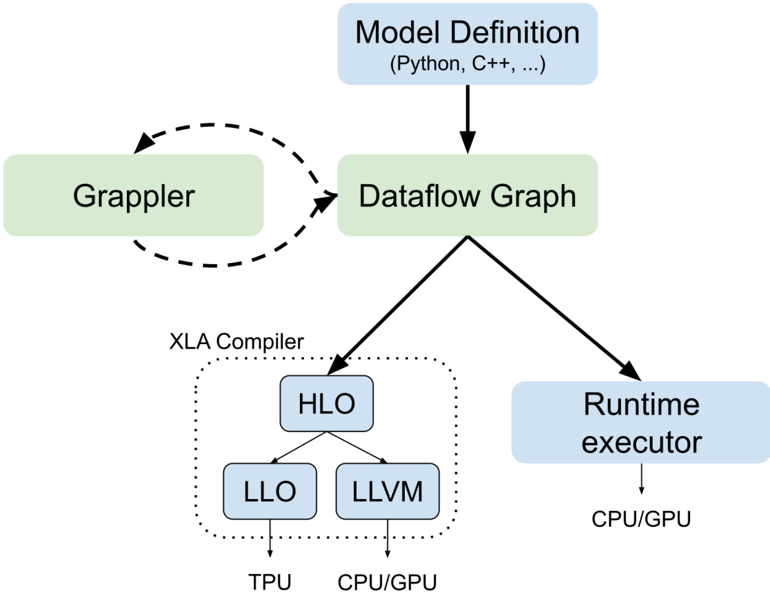


Figure 3.1: The machine learning model is processed prior to execution by either Grappler, the static graph optimiser in TensorFlow, or via JIT compilation of the model using XLA. Figure adapted from [26].

and/or XLA compilation steps.

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

In addition, as TASO operates at the graph-level, its optimisations are completely orthogonal to operator-level optimisations; thus it can be combined with code generation techniques such as TVM [7] or Astra [27] to further improve overall performance. We also note that TASO performs tensor data layout and graph transformation simultaneously rather than sequentially. It has been shown that by considering it as a joint optimisation problem end-to-end inference runtime can be reduced by up to 1.5x [9, 10].

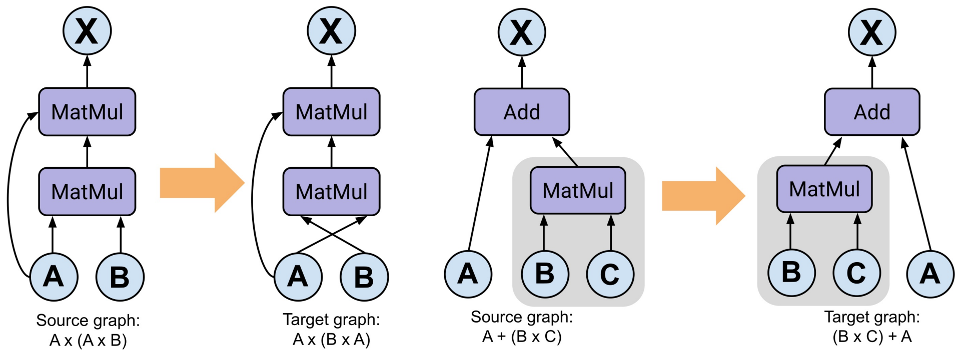
3.2.1 Graph-level optimisation

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

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

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

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



(a) Tensor renaming substitution

(b) Common subgraph substitution

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

in Figure 3.2a and 3.2b respectively.

- TODO: TASO algo for searching for optimal graph

3.2.2 Baselines

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

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

3.3 Reinforcement Learning formulation

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

3.3.1 System environment

In order to train a reinforcement learning agent, it necessary that we have access to an environment that, given the current environment state, the agent can take an action. After taking the chosen action, the environment is updated into a new state and the agent receives a reward signal. Typically, one uses a mature environment such as OpenAI Gym [28] or OpenSpiel [29] as the quality of the environment often has a significant effect on the stability of training. Moreover, using an environment that uses a common interface allows researchers to implement algorithms with ease and, importantly, reproduce results from published conference papers.

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

To simplify the implementation of the environment, we used made extensive use of the work by Jia et al. [9] with the open source version of TASO. We provide a computation graph and the chosen transformation and location; TASO then applies the requested transformation and returns the newly transformed graph. Further, we use internal TASO functions that calculates estimates of the runtime on the hardware device which we use as our reward signal for training the agent. During our experiments we modified TASO to extract detailed runtime measurements to analyse the rewards using a range of different reward formulae - we describe our approach further in section 3.3.3.

The scope of our work meant that there was no existing prior work that applied reinforcement learning to the task of optimising deep learning computation graphs. Thus, we required an environment in which an agent can act efficiently. Due to the nature of systems environments, the interactions with the real-world environment can be often slow, especially compared to those such as Arcade Learning Environment [30]. An aim of this work was to train a simulated environment, a "world model", that if accurate in relation to the real environment, we can train an agent far more efficiently than would be possible with the real-environment. In section [TODO] we will further explore world models and evaluate our implementation.

3.3.2 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. [31] used hierarchical RL with multiple actions to find the optimal device placement and Addanki et al. [32] that also aided in the design choice of input/output graph sizes.

The environment expects two actions:

- Transformation (which we refer to as an `xfer`) to apply, `xfer_id`
- Location at which to apply the transformation

Therefore, 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 100 possible for any given transformation, the remaining locations are considered invalid locations. 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 of invalid locations.

3.3.3 Reward determination

- Runtime difference

- Inclusion of detailed measurements

- Real-time measurements instead of estimated?

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

Bibliography

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

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

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

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

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

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

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

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

[9] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 4762, New York, NY, USA, 2019. Association for Computing Machinery.

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

[11] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[12] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand, 2019.

[13] Harrison Brown, Kai Fricke, and Eiko Yoneki. World-models for bitrate streaming. *Applied Sciences*, 10(19), 2020.

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

[15] Jan Robine, Tobias Uelwer, and Stefan Harmeling. Smaller world models for reinforcement learning, 2021.

[16] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[17] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search, 2017.

[18] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning, 2018.

[19] C. Daniel Freeman, Luke Metz, and David Ha. Learning to predict without looking ahead: World models without forward prediction, 2019.

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

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

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

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

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

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

[26] Rasmus Munk Larsen and Tatiana Shpeisman. Tensorflow graph optimizations, 2019. `http://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf`.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.