Neural Network Pruning Through Constrained Reinforcement Learning

by

Shehryar Malik

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Science

at the

LAHORE UNIVERSITY OF MANAGEMENT SCIENCES

June 2021

© Lahore University of Management Sciences 2021. All rights reserved.

Author . . . .
Department of Computer Science
May 21, 2021

Certified by . . . .
Dr. Murtaza Taj
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Dr. Agha Ali Raza
Thesis Evaluator

Neural Network Pruning Through Constrained Reinforcement Learning

by

Shehryar Malik

Submitted to the Department of Computer Science
on May 21, 2021, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science

Abstract

Deep neural networks have proved to be extremely useful for several tasks such as those in computer vision and natural language processing. However, the problem with these networks is that they typically have large memory and compute requirements which makes it difficult to deploy them on small devices such as mobiles and tablets. Pruning reduces the size of neural networks by removing ('pruning') neurons such that the performance drop is minimal. Traditional pruning approaches focus on designing metrics to quantify the 'usefulness' of a neuron. However, designing such metrics is often quite tedious and suboptimal. Some recent approaches have instead focused on training auxiliary neural networks to automatically learn how useful each neuron is in the network we wish to prune. In many cases, we are constrained in the amount of computational power that we can use. However, these auxiliary networks often do not take this into account. In this work, we focus on using constrained reinforcement learning algorithms to train these auxiliary neural networks to respect pre-defined computational budgets. We also carry out experiments to demonstrate the effectiveness of our approach.

Thesis Supervisor: Dr. Murtaza Taj
Title: Assistant Professor

Acknowledgments

I would like to thank my supervisor Dr. Murtaza Taj for his continouous support and feedback throughout this project. I would also like to thank Dr. Agha Ali Raza for being on my thesis committee and for providing his feedback on this project. Finally, I would also like to acknowledge the support of people at CVGL lab especially Umair.

Contents

1 Introduction **13**

2 Background **17**

2.1 Neural Networks 17

2.1.1 Overview 17

2.1.2 Training 19

2.1.3 Convolutional Neural Networks 20

2.1.4 Residual Networks 20

2.2 Pruning 21

2.2.1 Definition 21

2.2.2 Budget Aware and Unaware Pruning Techniques 22

2.2.3 Magnitude-Based Pruning 22

2.2.4 Filter Pruning in CNNs 23

2.3 Reinforcement Learning 23

2.3.1 Markov Decision Processes 24

2.3.2 Deep Reinforcement Learning 25

2.3.3 Policy Gradients 26

2.3.4 Variance Reduction Methods 27

2.3.5 Constrained Markov Decision Processes 29

2.3.6 PPO-Lagrangian 29

3 Related Works **31**

3.1 Pruning 31

3.2 Constrained Reinforcement Learning 34

4 Proposed Methodology **35**

4.1 Formulation 36

4.1.1 Constrained Markov Decision Process 36

4.1.2 Action Representations 37

4.2 Envrionment 38

4.3 Algorithm 39

4.4 Implementation Details 41

4.5 Extending to Online Pruning Settings 42

5 Experiments **43**

5.1 Experimental Settings 43

5.2 Implementation Details 44

5.3 Baseline 46

5.4 Results and Discussion 46

6 Conclusions and Future Work **55**

List of Figures

2-1 A single neuron 17

2-2 A neural network with three layers. Each circle represents a neuron with an associated weight and bias. (Only the weights and biases of the middle layer are shown.) 18

2-3 Offline pruning: The prune-finetune stage is repeated until the desired level of sparsity is reached. 22

2-4 Filter pruning in convolutional neural networks: The above figures show a subsection of a convolutional neural network. We wish to prune the filters marked in red in the left figure. However, notice that the size of each filter in layer 4 is dependent on the number of filters in layer 3. Therefore, as shown on the right, we also remove the corresponding 'slices' from each filter in layer 4. 24

2-5 Reinforcement learning setup: The agent starts is some state and takes an action. The environment rewards the agent accordingly, and transitions it to a new state. 25

4-1 A graphical visualization of the environment we construct. The environment consists of a pretrained network that we are interested in pruning. At timestep t, the policy outputs an action that specifies how the tth convolutional layer should be pruned. The environment executes this action, i.e., it prunes the layer accordingly. It then returns the next state (i.e., layer $(t+1)^{th}$ specifications) and a reward and a cost associated with the action that the policy just took. 39

5-1 Training graphs for coarse-grained strategy on VGG11 for $\alpha = 20$. . . . 49
5-2 Training graphs for fine-grained strategy on VGG11 for $\alpha = 20$. . . . 49
5-3 Training graphs for coarse-grained strategy on VGG11 for $\alpha = 10$. . . . 50
5-4 Training graphs for fine-grained strategy on VGG11 for $\alpha = 10$. . . . 50
5-5 Training graphs for coarse-grained strategy on VGG16 for $\alpha = 20$. . . . 51
5-6 Training graphs for fine-grained strategy on VGG16 for $\alpha = 20$. . . . 51
5-7 Training graphs for coarse-grained strategy on VGG16 for $\alpha = 10$. . . . 52
5-8 Training graphs for fine-grained strategy on VGG16 for $\alpha = 10$. . . . 52
5-9 Training graphs for coarse-grained strategy on VGG19 for $\alpha = 20$. . . . 53
5-10 Training graphs for fine-grained strategy on VGG19 for $\alpha = 20$. . . . 53
5-11 Training graphs for coarse-grained strategy on VGG19 for $\alpha = 10$. . . . 54
5-12 Training graphs for fine-grained strategy on VGG19 for $\alpha = 10$. . . . 54

List of Tables

5.1 Hyperparameters for pretraining networks. 43

5.2 Accuracy of pretrained networks. 43

5.3 Hyperparameters for coarse and fine-grained strategies. For neural network architectures we report the number of hidden units in each layer. All hidden layers use the tanh activation function, whereas the output layer uses a sigmoid. 44

5.4 Results for VGG11 for $\alpha = 20$. Unpruned accuracy is 89.23%. 48

5.5 Results for VGG11 for $\alpha = 10$. Unpruned accuracy is 89.23%. 48

5.6 Results for VGG16 for $\alpha = 20$. Unpruned accuracy is 90.69%. 48

5.7 Results for VGG16 for $\alpha = 10$. Unpruned accuracy is 90.69%. 48

5.8 Results for VGG19 for $\alpha = 20$. Unpruned accuracy is 90.59%. 48

5.9 Results for VGG19 for $\alpha = 10$. Unpruned accuracy is 90.59%. 48

Chapter 1

Introduction

Neural networks are mathematical functions inspired by the structure and function of the human brain. The basic building block of a neural network is a neuron. Each network has several neurons connected together in different ways. Modern neural networks, for example, typically have millions (e.g., [42, 15]), and sometimes even billions (e.g., [12]), of neurons connected to one another. Each neruon-to-neuron connection has a weight associated with it. These weights are usually intialized randomly. Training a neural network refers to the process of adjusting these weights to achieve a certain desired output.

Neural networks, in the context of machines, were proposed as early as 1940s [47]. Followup works proved the universal function approximation property of these networks [20, 8], which essentially states that neural networks can (under certain mild assumptions) approximate any function. The 'backpropogation' algorithm [38] which can be used to train these networks was introduced in 1974. However, despite these theoretical advances, neural networks remained mostly irrelevant because of their prohitively huge compuational requirements at that time.

Substantial increases in the processing powers of machines over the past two decades have ushered in a new era of neural networks. Implementing and training neural networks composed of millions of neruons can now be easily and efficiently done using graphical processing units (GPUs), which are becoming increasingly affordable. As a result of these advances, neural networks have found their way into

almost all subjects including mathematics, natural and social sciences, economics, natural language processing and vision.

However, with these huge advances arise new challenges. One of these major challenges involves deploying neural networks on cheap hardware, such as mobile phones and tablets. These devices are usually limited in their processing power and storage capacities. While neural networks can be trained on GPUs in an offline manner, many applications require them be deployed on these 'small' devices in order to make decisions in real-time. Since modern neural networks typically have millions of neurons, they cannot simply be deployed on these devices directly.

One way to solve this problem is to find more efficient architectures for these networks. Another method, which is the focus of this thesis also, is pruning (see e.g., [21, 31, 22]). Pruning techniques generally start out with a big neural network and then iteratively reduce ('prune away') the number of its neurons and the connections between them. This then results in a smaller network that can be efficiently deployed on small devices.

Pruning has acheived surprisingly remarkable results. Even simple pruning techniques have been able to reduce the computational and space complexity of different neural networks by upto 80% without any significant drop in its performance [4]. These results have naturally raised the following question: can we directly train smaller networks to achieve the same level of performance of these bigger, deeper networks instead of first training a bigger network and then pruning it?

Several explanations have been proposed to explain the unreasonable effectiveness of pruning techniques. The lottery ticket hypothesis [10], for example, argues (with empitical support) that bigger, deeper neural networks contain subnetworks that are initialized in such a way that - when when trained in isolation - can almost reach the same accuracy as that of the original network. This has resulted in several recent efforts to find these subnetworks before the network's training begins. This has the addditional advantage of reducing the cost of training the network.

The goal of pruning is to reduce the computational and space complexity of a network in such a way that its performance drop is minimal. The extent to which

a network needs to be pruned can be captured via a budget [28] on some function of the remaining parameters in the network after pruning. For example, if we wish to restrict the space complexity of a network to one-half of its original complexity, then we can choose this function to be the ℓ_0 norm and the budget to be one-half of the original number of parameters. Most pruning techniques, however, ignore this concept of a budget. These techniques instead simply try to reduce the size of the network as much as possible such that the resulting drop in performance is below a certain threshold [28]. This means that the size or the inference time or the number of flops of the pruned network cannot be controlled directly. Budget-aware pruning tehcniques on the other hand allow for the budget to be directly controlled. However, one shortcoming of current budget-aware techniques is that they require the function on which the budget is being imposed to be either differentiable or fully specified. This, however, is not always possible when we, for example, want to impose budgets on metrics such as inference time. The main question, then, is the following: can we prune neural networks to respect budgets on arbitrary, possibly non-differentiable, functions? One way to solve this problem is to leverage some recent techniques in reinforcement learning which can optimize arbitrary non-differentiable functions while respecting budgets on arbitary non-differentiable functions.

Reinforcement learning is a branch of machine learning in which agents interact with the environment around them by taking different actions. Each action results in the agent receiving a reward depending upon how good or bad the action is. The agent's goal is to maximize its cumulative reward over a certain period of time. Constrained reinforcement learning is an extension of the reinfocement learning problem in which agents, in addition to the reward, also receive a cost. The agent's goal is to maximize its cumulative reward subject to its cumulative cost being less than some pre-defined threshold. One interesting property here is that neither the reward nor the cost function need to be differentiable or fully specified. The agent simply needs to be fed a scalar reward and a scalar cost value each time it takes an action.

In this thesis, we propose a general methodology for pruning neural networks. Our proposed methodology can prune neural networks to respect specified budgets

on arbitrary, possibly non-differentiable, functions. Furthermore, we only assume the ability to be able to evaluate these functions for different inputs, and hence they do not need to be fully specified beforehand. Specifically, we formulate the problem of pruning in the constrained reinforcement learning framework. We also carry out experiments to demonstrate the effectiveness of our approach.

This thesis is organized as follows: in Chapter 2 we provide an introduction to the problems of pruning and constrained reinforcement learning. Chapter 3 reviews different works in both of these fields. Chapters 4 and 5 present our proposed methodology and experiments respectively. Finally, Chapter 6 concludes this thesis.

Chapter 2

Background

This thesis primarily proposes an approach to use algorithms for constrained reinfocement learning to prune neural networks. This chapter primarily gives an introduction to both pruning and reinforcement learning.

2.1 Neural Networks

2.1.1 Overview

Artificial neural networks (or simply, neural networks) are a family of functions in artificial intelligence that are inspired by the strucutre and function of the human brain [36].

Figure 2-1 shows the building block of the neural network, called a neuron. Here $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{w} \in \mathbb{R}^m$ and $b \in \mathbb{R}$ are the input, weight and bias of the neuron respectively. The function $f : \mathbb{R} \mapsto \mathbb{R}$ is known as the activation function and is non-linear in nature. The output of the neuron is a scalar value denoted with y.

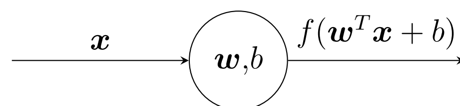

Figure 2-1: A single neuron

A neural network is typically composed of many *layers*. Each layer, in turn, is

composed of multiple independent neurons. Suppose that a single layer has m different neurons. Figure 2-2 shows an example. Each line represents a *connection* between two neurons. Notice that each element in $\boldsymbol{w}$ corresponds to a single connection..

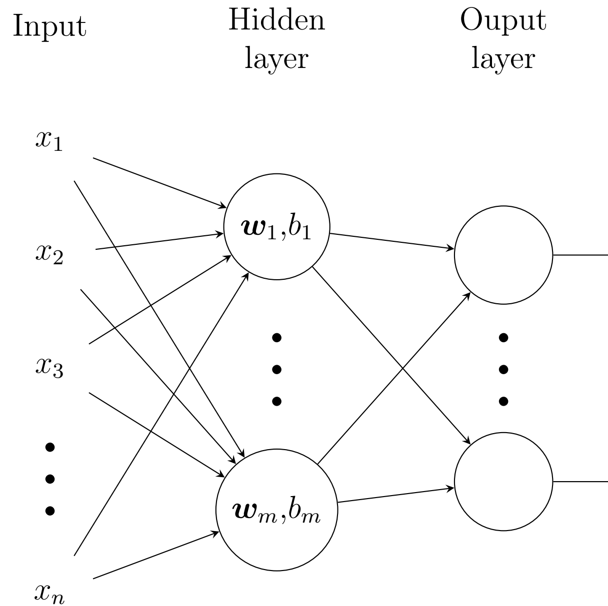

Figure 2-2: A neural network with three layers. Each circle represents a neuron with an associated weight and bias. (Only the weights and biases of the middle layer are shown.)

Let us now introduce a cleaner notation to deal with neural networks with multiple layers. Let us stack the weights of the neurons in a layer into a matrix $\boldsymbol{W} = [\boldsymbol{w_1}, \ldots, \boldsymbol{w_m}]$, the biases into a vector $\boldsymbol{b} = [b_1, \ldots, b_m]$ and the outputs into a vector $\boldsymbol{y} = [y_1, \ldots, y_m]$ where w_i, b_i and y_i denote the weight, bias and output of the ith neuron. Note that $\boldsymbol{W} \in \mathbb{R}^{n \times m}$, $\boldsymbol{b} \in \mathbb{R}^m$ and $\boldsymbol{y} \in \mathbb{R}^m$. Using this notation we can concisely represent a layer of the neural network as $\boldsymbol{y} = f(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$. Note that here f acts individually on each of the elements in its input vector. These layers are also referred to as fully-connected layers

In this thesis, we will sometimes use $\boldsymbol{\theta}$ to refer to all of the neural network's parameters.

2.1.2 Training

Neural networks are function approximators that map inputs $\boldsymbol{x} \in \mathbb{R}^n$ to outputs $\boldsymbol{y} \in \mathbb{R}^d$. Let $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^N$ denote a set of inputs and desired outputs[1]. We assume that each input-output pair is sampled independently and identically from some (unknown) distribution p. Our goal is train a neural network, i.e., to find the appropriate weights and biases, such that the network maps each input $\boldsymbol{x} \sim p$ to the correct output $\boldsymbol{y}$. The main challenge here is that for the training process, we only have access to a limited number of samples from p, whereas our goal is to have the network map *any* $\boldsymbol{x} \sim p$ to the correct output.

The performance of a neural netwok is usually captured via a loss function, $\mathcal{L}$. Training a neural network, then, entails finding the optimal parameters θ^* that decrease the value of $\mathcal{L}$ maximally. This is typically done through gradient descent, which iteratively updates θ according to

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}), \tag{2.1}$$

where α is a constant known as the learning rate. Since the input to each layer in the network depends on the outputs, and hence the weights and biases, of the previous layers, calculating the gradients involves applying the chain rule. Specifically, the derivative of $\mathcal{L}$ with respect to the weight matrix $\boldsymbol{W}_l$ at layer l is given by

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_L} \left(\frac{\partial W_L}{\partial W_{L-1}} \frac{\partial W_{L-1}}{\partial W_{L-2}} \cdots \frac{\partial W_{l+1}}{\partial W_l} \right) \tag{2.2}$$

However, since neural networks are non-convex functions, gradient descent often converges to a local minima. Several variants have been proposed to this vanilla version of gradient descent to mitigate this problem (see [37] for an overview).

[1] We restirct ourselves to the so-called supervised learning setting where the outputs are known.

2.1.3 Convolutional Neural Networks

Convolutional neural networks [25] are neural networks that are composed of convolutional layers (and optionally, fully connected layers). The inputs and outputs of a convolutional layer are usually tensors of rank 3. Let $\boldsymbol{x} \in \mathbb{R}^{h \times w \times c}$ be the input. For an image, h and w would be its height and width respectively and c will be the number of color channels (usually three, corresponding to red, green and blue). Each convolutional layer l has L_l associated filters. Each filter $\boldsymbol{F}_l \in \mathbb{R}^{h'_l \times w'_l \times c}$ is separately convolved with the input $\boldsymbol{x}$ according to

$$\boldsymbol{y}_l[i,j] = \sum_{k=1}^{c} \sum_{n_1=1}^{h'_l} \sum_{n_2=1}^{w'_l} \boldsymbol{x}[i-n_1, j-n_2, k]\boldsymbol{F}[n_1, n_2, k], \tag{2.3}$$

where $\boldsymbol{x}[a,b,c]$ is the element at position (a,b,c) in $\boldsymbol{x}$. Note that each $\boldsymbol{y}_l$ is a rank 2 tensor. Stacking all of them together yields a rank 3 tensor. A more detailed discussion on convolutional layers can be found in [11].

One thing to note here, however, is that the size of the filter $\mathbb{F}$ depends on the number of input channels. We will make use of this later when we discuss pruning.

2.1.4 Residual Networks

Modern neural networks typically have tens, and even hundreds, of layers. Since the gradient of the parameters of a layer is equal to the product of the gradients at subsequent layers, the gradient for parameters of layers at the beginning of the network usually tends to either vanish (i.e., approach 0) or explore (i.e., approach ∞). One way to mitigate the problem of vanishing gradients is to add residual connections [16], which modify the output of the layer to be

$$\boldsymbol{y} = f(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) + \boldsymbol{x}. \tag{2.4}$$

2.2 Pruning

As discussed in the last chapter, moder neural networks typically have millions of parameters, which makes it challenging to deploy them on small devices like mobiles and tablets. Pruning aims to reduce the size of these networks by deleting neruon connections in such a way that the performance drop of the network is minimal.

2.2.1 Definition

Let $\boldsymbol{\theta}$ denote the parameters of a neural network. Each element of $\boldsymbol{\theta}$ represents the weight of single connection. Removing a connection is thus equivalent to mutliplying its weight by 0. Let $\mathcal{L}$ denote the loss function of the neural network. Pruning, in its most general form, tries to find a mask $\boldsymbol{M} \in \{0,1\}^{|\theta|}$ that solves the following optimization program:

$$\begin{aligned} &\underset{\boldsymbol{\theta},\boldsymbol{M}\in\{0,1\}^{|\theta|}}{\text{minimize}} \ \mathcal{L}(\boldsymbol{\theta} \odot \boldsymbol{M}) \\ &\text{subject to } f(\boldsymbol{M}) \leq \alpha, \end{aligned} \tag{2.5}$$

where $\odot$ is the element-wise Hadamard product, f be some arbitrary, possibly non-differentiable, function and α is a known constant. Here f represents the computational and space complexity of the neural network. For example, if we want to compress the network by at least 50% (in terms of the space it occupies) we can let f be the ℓ_1-norm and α to be equal to $0.5|\boldsymbol{\theta}|$. Similarly, if we wanted to optimize for speed, we could set f to be the number of flops the network consumes (or the time it takes) when it is pruned according to $\boldsymbol{M}$.

Traditionally, pruning techniques generally fall into two categories depending on how $\boldsymbol{\theta}$ and $\boldsymbol{M}$ are optimized.

1. Offline pruning: In this case, we first simply only optimize for $\boldsymbol{\theta}$. We then fix $\boldsymbol{\theta}$ and solve for $\boldsymbol{M}$. Next, we finetune our fixed network. This pruning-finetuning procedure can be repeated for a number of iterations. This is depicted in Figure 2-3. In this thesis, we will restrict ourselves to this offline pruning

setting.

2. Online pruning: In this case, both $\boldsymbol{\theta}$ and $\boldsymbol{M}$ are optimized jointly.

A new category of pruning techniques are, however, also now emerging which, inspired by the lottery ticket hypothesis, aim to optimize $\boldsymbol{M}$ before $\boldsymbol{\theta}$. This has the added advantage of reducing the cost of training the network also.

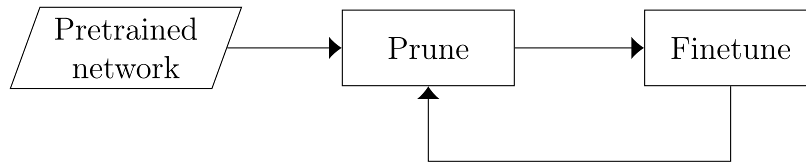

Figure 2-3: Offline pruning: The prune-finetune stage is repeated until the desired level of sparsity is reached.

2.2.2 Budget Aware and Unaware Pruning Techniques

While in its more general form, pruning tries to reduce the size of a network such that it respects a certain budget, many techniques instead follow the philosophy of simply pruning the network as much as possible without caring for the budget [28]. We will refer to these methods as budget unaware methods. On the other hand, budget aware methods instead prune a given network according to the budget specified. We review both of these methods in detail in Chapter 3.

2.2.3 Magnitude-Based Pruning

One simple technique used to prune networks is called magnitude-based pruning. Magnitude-based pruning simply removes connections within the network with the smallest weights. The intuition behind this is that connections with small weights have a very small effect on the final output of the network. While simple, magnitude-based pruning works well in practice.

2.2.4 Filter Pruning in CNNs

One drawback of the vanilla magnitude-based pruning technique discussed in the previous subsection is that it results in a sparse network. To fully exploit the sparsity in the network for optimizing performance, we often need to resort to specialized libraries, which is typically cumbersome. What we ideally need instead is to be able to prune the original bigger dense network in such a way that we are left with a smaller dense network. For convolutional neural networks, one approach used to achieve this is filter pruning which was introduced in [29].

Recall that each convolutional layer is composed of a set of L filters. Each of these filters is a tensor of size $h' \times w' \times c$. Instead of removing individual weight connections, filter pruning removes entire filters according to some metric. So, for example, a magnitude-based filter pruning scheme would remove filters with the smallest magnitude. We are, therefore, left with a subset of filters only. Note that this subset still constitutes a dense convolutional layer albeit smaller.

Convolutional neural networks are typically composed of multiple convolutional layers stacked together. The size of a filter in each layer depends on the number of input channels, which in turn is equal to the number of output channels of the previous layer. Hence, if we remove the ith filter from layer l, we would need to delete the ith slice (along the third dimension) in each of the filters in layer $l + 1$. This phenomenon is depicted in Figure 2-4.

2.3 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that focusses on training agents to maximize a numerical reward signal they receive when they interact with an environment. The agent usually starts off in some initial state, takes an action and transitions to a new state. Each state-action pair has a reward associated with it. The reward captures how good the action that the agent took was in that particular state. The agent's goal is to discover state-action pairs that yield high reward. This agent-environment interaction is depicted in Figure 2-5. One important point to note

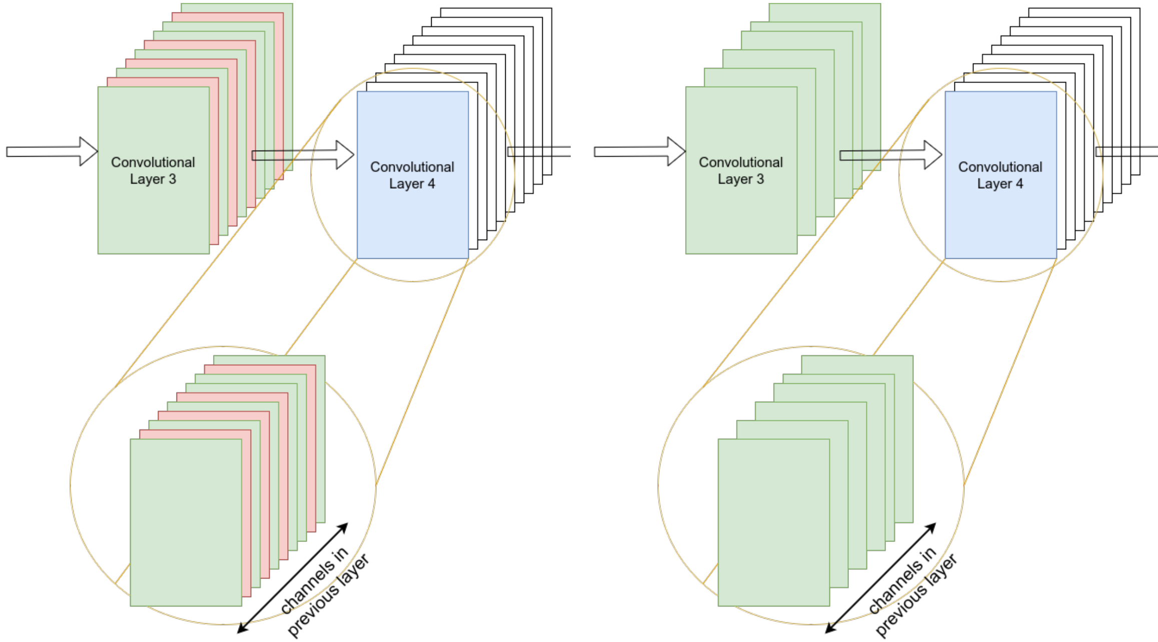

Figure 2-4: Filter pruning in convolutional neural networks: The above figures show a subsection of a convolutional neural network. We wish to prune the filters marked in red in the left figure. However, notice that the size of each filter in layer 4 is dependent on the number of filters in layer 3. Therefore, as shown on the right, we also remove the corresponding 'slices' from each filter in layer 4.

here is that actions may not just affect the immediate reward that the agent receives, but may also affect its subsequent state and hence all of its subsequent rewards. A comprehensive overview of reinforcement learning can be found in [43].

2.3.1 Markov Decision Processes

Mathematically, the notion of an environment is captured through a Markov Decision Process (MDP). A finite-horizon Markov Decision Process (MDP) $\mathcal{M}$ is a tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma, T)$, where $\mathcal{S} \in \mathbb{R}^{|\mathcal{S}|}$ is a set of states, $\mathcal{A} \in \mathbb{R}^{|\mathcal{A}|}$ is a set of actions, $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ is the transition probability function (where $p(s'|s, a)$ denotes the probability of transitioning to state s' from state s by taking action a), $r : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is the reward function, γ is the discount factor and T is the time-horizon. A trajectory $\tau = \{s_1, a_1, \ldots, s_T, a_T\}$ denotes a sequence of states-action pairs such that $s_{t+1} \sim p(\cdot|s_t, a_t)$.

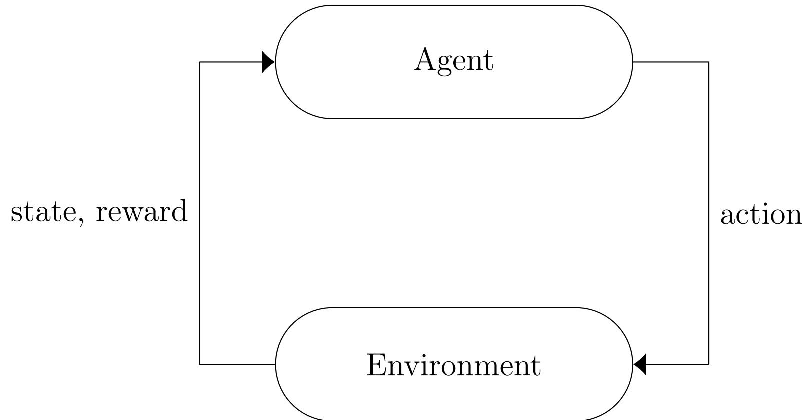

Figure 2-5: Reinforcement learning setup: The agent starts is some state and takes an action. The environment rewards the agent accordingly, and transitions it to a new state.

The goal of an agent is to learn a *policy* about which action to take in each state. Formally, a policy $\pi : \mathcal{S} \mapsto \mathcal{P}(\mathcal{A})$ is a map from states to probability distributions over actions, with $\pi(a|s)$ denoting the probability of taking action a in state s. We will sometimes abuse notation to write $\pi(s, a)$ to mean the joint probability of visiting state s and taking action a under the policy π and similarly $\pi(\tau)$ to mean the probability of the trajectory τ under the policy π.

Let $r(\tau) = \sum_{t=1}^{T} \gamma^t r(s_t, a_t)$ denote the total discounted reward of a trajectory. The problem of RL is to find a policy π^* that maximizes the expected total discounted reward, i.e.,

$$\pi^* = \arg\max_{\pi} J(\pi) = \mathbb{E}_{\tau\sim\pi}[r(\tau)]. \tag{2.6}$$

2.3.2 Deep Reinforcement Learning

When dealing with large (possibly infinite) state and action spaces, it is often useful to parameterize π as a neural network. When the action space is discrete (i.e., there are only a fixed number of actions that the agent can choose from), the neural network can simply output a probability score for each action. When the action space is continuous, then one common approach is to model π as a Guassian distribution $\mathcal{N}(\mu_\theta(\boldsymbol{s}), \boldsymbol{\sigma} I)$, where $\mu_\theta : |\mathcal{S}| \mapsto |\mathcal{A}|$ is a neural network. $\boldsymbol{\sigma} \in \mathcal{R}^{|\mathcal{A}|}$ is a trainable vector and I is the identity matrix. We will denote all of the parameters of π with $\boldsymbol{\theta}$.

2.3.3 Policy Gradients

The policy gradients algorithm [48] finds π^* through stochastic gradient descent. That is, we update θ according to

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}). \tag{2.7}$$

The gradient of J with respect to θ can be shown to be

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau)] = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\theta}(\tau) r(\tau)]. \tag{2.8}$$

The expectation in the above expression can be approximated using samples from $\pi_{\boldsymbol{\theta}}$. The policy gradients algorithm (also known as the REINFORCE algorithm) is given as follows:

1. Intialize $\boldsymbol{\theta}$ randomly.
2. Repeat until convergence:
   (a) Sample a set of trajectories $\mathcal{D} = \{\tau^{(i)}\}_{i=1}^m$ from $\pi_{\boldsymbol{\theta}}$.
   (b) Estimate $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) \approx 1/m \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \log \pi_{\theta}(\tau^{(i)}) r(\tau)$.
   (c) Update $\boldsymbol{\theta} := \boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$.

One problem with policy gradients is that we need to sample from $\pi_{\boldsymbol{\theta}}$ each time we update $\boldsymbol{\theta}$. This is usually computationally expensive. What we would like to do instead is to just sample from $\pi_{\boldsymbol{\theta}}$ after, say, K updates of θ. This would, however, require that we approximate the expectation in (2.8) from a distibution different than $\pi_{\boldsymbol{\theta}}$. Let $\pi_{\bar{\boldsymbol{\theta}}}$ denote the distribution from which we draw samples. Then we can rewrite the expectation as

$$\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\theta}(\tau) r(\tau)] = \mathbb{E}_{\tau \sim \pi_{\bar{\boldsymbol{\theta}}}} \left[\frac{\pi_{\boldsymbol{\theta}}(\tau)}{\pi_{\bar{\boldsymbol{\theta}}}(\tau)} \nabla_{\boldsymbol{\theta}} \log \pi_{\theta}(\tau) r(\tau) \right]. \tag{2.9}$$

We can now approximate this using samples from $\pi_{\bar{\boldsymbol{\theta}}}$. This estimate, however, improves when $\pi_{\bar{\boldsymbol{\theta}}}$ is closer $\pi_{\boldsymbol{\theta}}$. To make use of this fact, trust region methods modify the original problem by adding a constraint on the KL-divergence, D_{KL}, betweent

the two policies. The policy is updated each time according to

$$\begin{aligned}\pi^* &= \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi_{\bar{\boldsymbol{\theta}}}}\left[\frac{\pi_{\boldsymbol{\theta}}(\tau)}{\pi_{\bar{\boldsymbol{\theta}}}(\tau)}r(\tau)\right] \\ &\text{subject to } D_{KL}(\pi_{\bar{\boldsymbol{\theta}}}||\pi_{\boldsymbol{\theta}}) \leq \delta,\end{aligned} \tag{2.10}$$

for some known constant δ. The proximal policy algorithm (PPO) [41] makes a first order approximation to this optimization problem. Specifically, it proposes to update $\pi_{\boldsymbol{\theta}}$ by solving

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi_{\bar{\boldsymbol{\theta}}}}\left[\min\left(\frac{\pi_{\boldsymbol{\theta}}(\tau)}{\pi_{\bar{\boldsymbol{\theta}}}(\tau)}r(\tau), \text{clip}\left(\frac{\pi_{\boldsymbol{\theta}}(\tau)}{\pi_{\bar{\boldsymbol{\theta}}}(\tau)}, 1-\epsilon, 1+\epsilon\right)r(\tau)\right)\right], \tag{2.11}$$

for some known constant ϵ. Here clip bounds its first argument between the other two. The objective function above essentially removes any incentive to move the ratio between the two policies outside of the interval $[1-\epsilon, 1+\epsilon]$. While a crude approximation, PPO usually performs comaprable to other more theoretically grounded approaches such as trust region policy optimization (TRPO) [40].

2.3.4 Variance Reduction Methods

One problem with policy gradients method is that we we cannot compute the expectation $\mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) r(\tau))]$ (we only focus on the vanilla case, but the same discussion can easily be extended to the PPO setting), exactly for large state and action spaces. We, therefore, need to resort to approximating it by drawing a limited number of samples from $\pi_{\boldsymbol{\theta}}$. The problem with this sampling-based method, however, is that it usually has very high variance.

One method to reduce variance is to decompose this expectation over trajectories into expectations over state-action pairs, as follows

$$\mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) r(\tau))] = \sum_{t=1}^{T} \mathbb{E}_{s_t, a_t\sim\pi_{\boldsymbol{\theta}}}\left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\left(\sum_{t'=1}^{T} r(s_t, a_t)\right)\right], \tag{2.12}$$

and to note that since because of causality future actions and states cannot affect

previous ones, we can rewrite this expectation as:

$$\sum_{t=1}^{T} \mathbb{E}_{s_t, a_t \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \left(\sum_{t'=1}^{T} r(s_t, a_t) \right) \right) \\ = \sum_{t=1}^{T} \mathbb{E}_{s_t, a_t \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \left(\sum_{t'=t}^{T} r(s_t, a_t) \right) \right]. \tag{2.13}$$

An approximation of the term on the right hand side using samples has a lower variance than the one for the term on the left. The sum $\sum_{t'=t}^{T} r(s_t, a_t)$ is often referred to as the reward-to-go.

Another method to reduce variance is to subtract a state-dependent baseline b from the rewards.

$$\sum_{t=1}^{T} \mathbb{E}_{s_t, a_t \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \left(\sum_{t'=t}^{T} r(s_t, a_t) \right) \right] \\ = \sum_{t=1}^{T} \mathbb{E}_{s_t, a_t \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \left(\sum_{t'=t}^{T} r(s_t, a_t) - b(s_t) \right) \right]. \tag{2.14}$$

The two expectations above are equal because we can show that

$$\sum_{t=1}^{T} \mathbb{E}_{s_t, a_t \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \sum_{t'=t}^{T} b(s_t) \right] = 0. \tag{2.15}$$

A common baseline which works well in practice is the so-called value function given as

$$V^{\pi_{\boldsymbol{\theta}}}(s_t) = \sum_{t'=t}^{T} \mathbb{E}_{s_{t'}, a_{t'} \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)}[r(s_{t'}, a_{t'})]. \tag{2.16}$$

The value functions at two consecutive timesteps are related as follows

$$V^{\pi_{\boldsymbol{\theta}}}(s_t) \approx r(s_t, a_t) + V^{\pi_{\boldsymbol{\theta}}}(s_{t+1}) \tag{2.17}$$

where a_t is sampled from $\pi_{\boldsymbol{\theta}}$. In practice, $V^{\pi_{\boldsymbol{\theta}}}$ is usually parameterized as a separate neural network with parameters ϕ. This network is trained by gradient descent to

minimize the following loss function

$$\mathcal{L}(\boldsymbol{\phi}) = ||r(s_t, a_t) + V_{\boldsymbol{\phi}}^{\pi_{\boldsymbol{\theta}}}(s_{t+1}) - V_{\boldsymbol{\phi}}^{\pi_{\boldsymbol{\theta}}}(s_t)||_2^2. \tag{2.18}$$

Both of these variance reduction methods usually result in significant improvements in performance.

2.3.5 Constrained Markov Decision Processes

A Constrained MDP (CMDP) [2] $\mathcal{M}^c$ is a simple MDP augmented with a cost function $c : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ and a budget $\alpha \geq 0$. Let $c(\tau) = \sum_{t=1}^{T} \gamma^t c(s_t, a_t)$ be the total discounted cost of the trajectory τ. The constrained RL problem is to find a policy π^* such that

$$\begin{aligned} \pi^* = \arg\max_{\pi} \mathbb{E}_{\tau \sim \pi}[r(\tau)] \\ \text{subject to } \mathbb{E}_{\tau \sim \pi}[c(\tau)] \leq \alpha. \end{aligned} \tag{2.19}$$

As before, π will be parameterized as a neural network with parameters θ.

2.3.6 PPO-Lagrangian

One method to solve (2.19) was introduced in [44] which rewrites the original problem as an unconstrained min-max problem on the Lagrangian function as follows.

$$\min_{\lambda \geq 0} \max_{\pi_{\boldsymbol{\theta}}} J_{\text{LAG}}(\lambda, \pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[r(\tau)] - \lambda(\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[c(\tau)] - \alpha)]. \tag{2.20}$$

Here $\lambda \in \mathbb{R}$ is the Lagrangian multiplier. Note that as λ increases the solution to (2.20) converges to that of (2.19). We can use the primal-dual optimization (PDO) algorithm to solve this min-max problem. The PDO algorithm is as follows:

1. Initililize $\boldsymbol{\theta}$ and λ.
2. Repeat until convergece:

(a) Update θ by gradient ascent using the current value of λ:

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \alpha_1 \nabla_\theta J_{\text{LAG}}(\lambda, \pi_{\boldsymbol{\theta}}), \tag{2.21}$$

where α_1 is the learning rate for $\boldsymbol{\theta}$.

(b) Update λ by gradient descent using the current value of θ:

$$\lambda := \lambda + \alpha_2 \nabla_\lambda J_{\text{LAG}}(\lambda, \pi_{\boldsymbol{\theta}}), \tag{2.22}$$

where α_2 is the learning rate for λ.

Note that we need to take gradients with respect to $\boldsymbol{\theta}$ (2.21). As discussed earlier, to do so would require drawing samples from $\pi_{\boldsymbol{\theta}}$, which is compuationally expensive. Therefore, we can instead replace J_{LAG} with the PPO objective function, i.e.,

$$J_{\text{LAG}}^{\text{PPO}}(\lambda, \pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\tau \sim \pi_{\bar{\boldsymbol{\theta}}}} \left[\frac{\pi_{\boldsymbol{\theta}}(\tau)}{\pi_{\bar{\boldsymbol{\theta}}}(\tau)} r(\tau) - \lambda(\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[c(\tau)] - \alpha) \right], \tag{2.23}$$

where $\pi_{\bar{\boldsymbol{\theta}}}$ is the distribution of which we have samples from. We will refer to the modified PDO algorithm that optimizes this new objective as PPO-Lagrangian.

In this thesis, we will use the PPO-Lagrangian algorithm for pruning neural networks. We, however, also review additional techniques to solve the constrained RL problem in (2.19) in Chapter 3.

Chapter 3

Related Works

In this chapter, we review different pruning and constrained reinforcement learning methods.

3.1 Pruning

Magnitude-based pruning is one of the oldest techniques for pruning neural networks. This approach is based on the 'magitude equals saliency' principle and consequently prunes neuron connections with the smallest magnitude. The intuition behind this is that connections with small weights affect the output of the network the least. One way to make this approach more effective is by adding a sparsity loss (e.g., ℓ_1 regularization) while training the network. This would encourage the network to keep most of its connections small. A recent extension of this approach called lookahead pruning [26] argues that it is not enough to only look at the magnitude of each connection separately. A connection with a large weight may also have a small affect on the output if the neurons that it connects have smaller connections in the previous and subsequent layers and vice versa. Consequently, it proposes a method to take this additional information into account.

A more theorectically-grounded approach called optimal brain damage instead analyzes the derivatives of the loss function of the network with respect to the weight of each connection [24]. The idea behind this is to find the set of connections which

result in the least change in the value of the loss function. In the most general case, this problem is insoluble, so it instead introduces an approximation (specifically, that the change in the loss resulting from the deletion of several different connnections is equal to the sum of the changes in the loss when each of these connections is deleted individually) that result in near-optimal solutions. An extension of this approach is presented in [14]. However, one problem with this approach is that it requires the second-order derivatives of the loss function with respect to the connections to be computed which is often computationally expensive, especially for very large networks. An extension of this approach [9], instead proposes a layer-wise loss function. For each layer, we prune connections that result in the least change in the value of the loss function for that layer. This means that each layer can be pruned independently and in parallel, which significantly speeds up the computational complexity of the pruning process.

A more recent class of techniques poses the problem of pruning as a learning problem. Usually, these approaches multiply the network weights with a mask vector, whose entries are restricted to be within 0 and 1. Mutliplying a weight with a small value is essentially equivalent to removing that weight from the network. The goal is then to find the mask vector that has most of its elements close to 0 and, simultaneously, does not degrade the performance of the network much. Once the best mask vector is found, connections for which the corresponding element in the mask vector is small are simply removed from the network. There are two main approaches for learning this mask vector which we discuss below. One point to note here, however, is that both of these approaches can be used in either online or offline pruning setting.

The first approach treats the mask vector as a trainable parameter and directly optimizes it through gradient descent. The loss function for the mask vector is usuaully a combination of two terms: the sparsity loss of the vector itself and an estimate of the performance of the network (usually its loss function). An example of this appproach is [13].

The second approach instead treats the mask vector as the output of a second

neural network with its own set of parameters. The loss function for this network is defined in a similar way as the first approach (i.e., a combinartion of the sparsity of the mask vector and the original network's performance). An example of this approach is [30]. One work in this paradigm, which is also relevant to this thesis, is [17]. Instead of having the second network predict the entire mask vector for each layer, we ask it to predict a sparsity ratio for the layer. Each layer is then pruned according to magnitude-based pruning upto predicted sparsity ratio. The main motivation behind this work is that different layers have different impacts on the network's performance and computational and space complexity, and hence should be pruned differently.

Inspired by the observations made in [10], which introduced the lottery tickey hypothesis, a new class of pruning methods have now emerged which aim to prune deep neural networks even before they are trained on their relevant task. For example, [27] prunes networks so that the initial loss does not change much. Another method [46] instead tries to prune connections such that the gradient flow is preserved (or increased). This is because a larger gradient flow essentially corresponds to faster training.

One problem with the approaches discussed so far is that they do not offer a natural way to control the computational and space complexity of the pruned network. Budget-aware techniques instead focus on pruning networks to respect specified *budgets* on compuational and space complexities. The first work to introduce this concept of a budget was [28], which proposed a variant of the log-barrier function to penalize the network heavily if it violated the specified budget.

Pruning is just one of many techniques for network compression. Another technique for compression is known as knowledge distillation [19]. Instead of reducing the orignal, deep network (called the "teacher") by pruning connections, knowdledge distillation trains another, smaller network (called the "student") to mimic the output of the teacher network.

3.2 Constrained Reinforcement Learning

We now review different techniques for solving the constrained RL problem. The framework for constraned MDPs, which was introduced in [2] is the de-facto framework for constrained RL.

The primal-dual optimization algorithm presented in the previous chapter was introduced in [44]. An extension of this work instead argues that the Lagrange mutliplier should be state dependent [6], and shows simple cases where a state-independent formulation leads to suboptimal solutions. [50] formulates the problem of recommendation in the constrained RL framework and adapts the primal-dual method to solve it. However, since the problem of constrained RL is not convex, these primal-dual algorithms need not converge. [34] shows that the constrained RL problem actually has a zero-duality gap (under mild assumptions) and hence primal-dual algorithms will, in fact, converge to the optimal solution.

Several other methods of constrained RL have also been proposed in literature. [1] proposes one scheme, wherein for each update of the network, it constructs a trust-region containing all feasible solutions (i.e., solutions that respect the constraints specified), and then performs the update within that trust region only. [7] derives another algorithm for this problem by using the Lyapunov method. An interesting work is the one in [39] which translates the constraints on cumulative cost over the entire trajectory into state-based constraints. It then proposes an algorithm to optimize the policy so that it respects the constraints at each timestep. Other approaches to constrianed RL include [49, 5, 33]. Constrained RL actually falls under the domain of safe RL, where we are interested in training agents to respect *safety* constraints. A more detailed review can be found in [35].

Chapter 4

Proposed Methodology

In this chapter we present our methodolgy. We will follow the notation from Chapter 2.

As discussed previously, one of the problem with most pruning approaches is that they offer no natural way to control the computational and space budget of the pruned network. In most cases, the only way one can prune networks to respect arbitrary budget requirements on, say, sparisty or the number of flops is through trial-and-error (as discussed in the previous chapters). The few techniques that do allow the budget to be controlled require f to either be differentiable or fully specified [28, 17]. While this can be done in the case of sparsity or even the number of FLOPs, it is obvioulsly not possible to this for, say, inference time. Consider building an application that involves mutliple deep neural networks and several other components interlinked with one another. The application takes some sort of an input and produces an output. The developer, primarily, is interested in bounding the inference time of these different networks. This, however, is not possible with current pruning methods.

Recall that in constrained RL, the cost function can be non-differentiable. Furthermore, we do not need pre-define the cost function. Instead, we only require the ability to *evaluate* the cost function for different actions that the agent takes (in this case, corresponding to, say, the inference time a pruned network requires, which clearly is possible to compute). We, therefore, formulate pruning in the constrained RL framework.

4.1 Formulation

4.1.1 Constrained Markov Decision Process

In order to formulate our problem as a constrained RL problem, we need to first define our constrained MDP (CMDP). Let $g = g_F \circ g_T \circ g_{T-1} \circ \ldots g_1$ define a neural network with T convolutional layers, $g_1, \ldots, g_T$ followed by a few fully connected layers collectively denoted as g_F. We wish to prune g. Furthermore, let θ_t denote the parameters corresponding to layer t. We define the key components of the CMDP as follows:

1. State, s: Each convolutional layer corresponds to a single state. Similar to the scheme in [17] each of these layers is represented by the following tuple:

   (t, input channels, number of filters, kernel size, stride, padding)

   where t is the index of that layer and the remaining entries are the attributes of a convolutional layer (as discussed in Chapter 2).

2. Action, a: We use two different representations for the action, which we discuss in the next subsections. However, for both representations we end up with a mask vector M_t at each layer g_t. The lenght of this mask vector will be equal to the number of filters for g_t, i.e., we recieve a scalar value between 0 and 1 inclusively for each filter. The filter weights are then simply multiplied by this value. We will use M to collectively denote the mask vectors for all layers.

3. Transition function: The agent always transitions from state g_t to state g_{t+1}, and so the transitions are fixed and independent of the agent's actions.

4. Reward function, r: Let B be a batch of input examples (uniformly) sampled from the training dataset. We define the reward as follows:

$$r(s_t, a_t) = \begin{cases} -\mathcal{L}(\theta \odot M) & \text{if } t = T \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where $\mathcal{L}$ is the loss function of the network evaluated on the batch B.

5. Cost function, c: Similarly the cost function is defined as

$$c(s_t, a_t) = \begin{cases} f(M) & \text{if } t = T \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

   where f is our constraint function evaluated on the batch B.

6. Budget, α: This the budget on f we wish our pruned network to respect. For example, if we want the network to be at least 30% sparse, then f can be defined to be the average of M (assuming M is binary) and α to be 70%.

4.1.2 Action Representations

We now discuss two different representations for actions.

Coarse-grained strategy

In this case, the policy predicts a sparsity ratio for each layer. Filters in that layer are then pruned using magnitude-based pruning upto the desired sparsity.

Fine-grained strategy

One problem with the coarse-grained strategy is that it relies on magnitude-based pruning which is a greedy strategy. This leads to suboptimal solutions. We would ideally like to allow the policy more control over which connections to turn and turn off. However, since each layer may be very high dimensional (i.e., have too many connections), predicting a separate mask value for each connection may be too computationally expensive. Since we restrict ourselves to convolutional layers, we instead predict one scalar value per filter. One obvious issue with this is at that the number of filters may vary for each layer. To overcome this, at each timestep we have the network predict a mask vector of length equal to the maximum number of filters in any layer. We then simply ignore the extraneuous mask elements.

One point to note here is that the policy outputs continuous values in the range 0 to 1. During training, when computing the reward, we simply multiply these continuous values with their corresponding filter weights. To compute the cost, however, we threshold these values so that we have a binary mask vector. Similarly, we also threshold the values for evaluation once the policy has been trained.

4.2 Envrionment

As discussed previously, an agent primarily interacts with an environment. Environments generally provide the following interace to the agent [45] (in case of simple reinforcement learning, the 'cost' is omitted):

```
next_state, reward, cost, done = environment(action)
```

Here the agent feeds an action to the environment that it wants to take. The environment executes that action and returns the agent with its next state alogn with a scalar reward and a scalar cost for the action that was passed to it. In addition to this, the environment also returns a 'done' flag, indicating if the maximum number of timesteps is reached or if the simulation has ended (e.g., when the agent has been killed).

We define our environment in a similar way. At the beginning, the environment loads a pretrained network (e.g., VGG11). The agent is then fed the layer representations of the first convolutional layer in the network (as defined in the previous section). The agent then specifies an appropriate action (either a scalar or a vector depending upon whether we are using the coarse-grained or fine-grained approach). This action is then recorded and agent is then fed the next state. Once the agent has finished predicting its actions for all convolutional layers, the network is pruned accroding to the agent's specified actions and finetuned on the training set (this is the same training set that was used to pretrain the network). The accuracy of the finetuned network is returned as the agent's reward (the reward at all other timesteps is 0). Similarly, the cost is also calculated and returned.

Figure 4-1 shows a visualization of the environment.

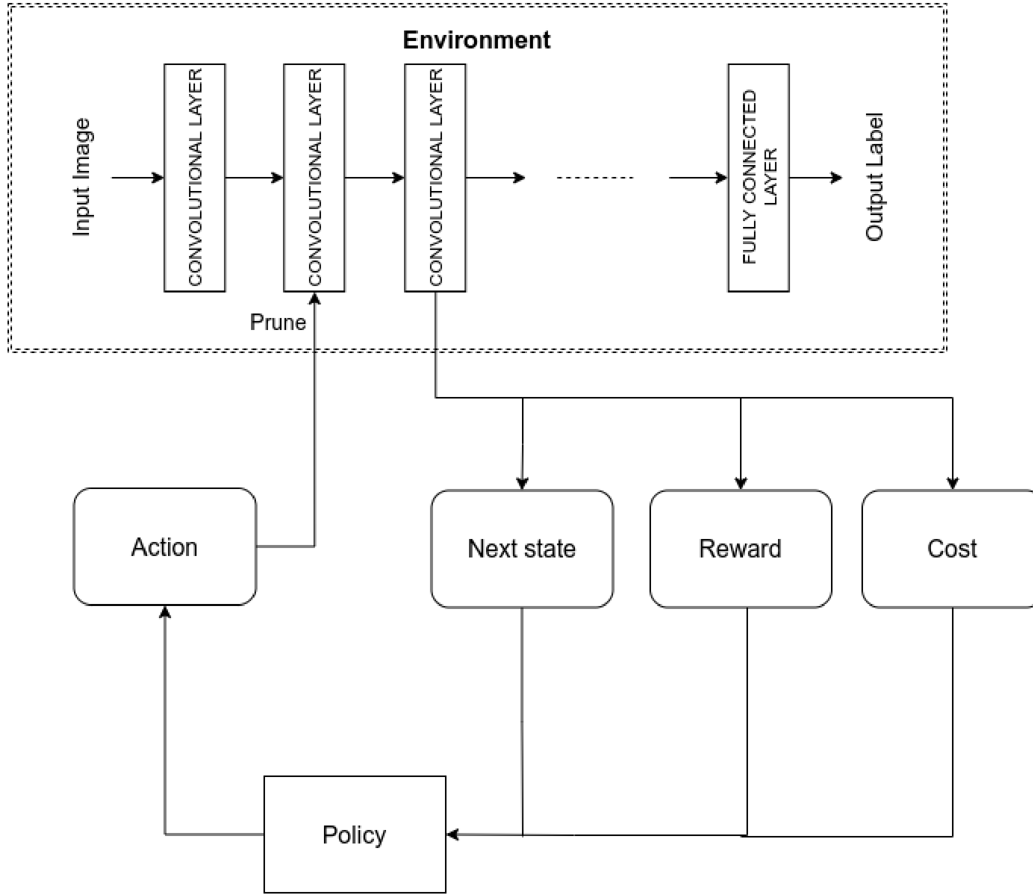

Figure 4-1: A graphical visualization of the environment we construct. The environment consists of a pretrained network that we are interested in pruning. At timestep t, the policy outputs an action that specifies how the tth convolutional layer should be pruned. The environment executes this action, i.e., it prunes the layer accordingly. It then returns the next state (i.e., layer $(t+1)^{th}$ specifications) and a reward and a cost associated with the action that the policy just took.

4.3 Algorithm

Let d_a be the dimension of the action (1 in case of coarse-grained strategy and equal to the largest number of filters in any given convolutional layer in the network for the fine-grained strategy). Also, let d_s denote the dimension of the state vector (recall that each state vector corresponds to a single layer).

We model our policy as a (diagonal) multivariate Gaussian distribution $\mathcal{N}(\mu_{\boldsymbol{\theta}}, \boldsymbol{\sigma I})$. Here $\mu_{\boldsymbol{\theta}} : \mathbb{R}^{d_s} \mapsto \mathbb{R}^{d_a}$ is a neural network with parameters $\boldsymbol{\theta}$, $\sigma \in \mathbb{R}^{d_a}$ is a trainable vector and $\boldsymbol{I} \in \mathbb{R}^{d_a \times d_a}$ is the identiy matrix (hence $\boldsymbol{\sigma I}$ is the covariance matrix of

the distribution). The network $\mu_{\boldsymbol{\theta}}$ takes in as input a state vector s provided by environment and outputs a vector of dimension d_a. We then simply sample an action from $\mathcal{N}(\mu_{\boldsymbol{\theta}}(\boldsymbol{s}), \boldsymbol{\sigma I})$ and feed it to the environment.

In parallel, we also train two other neural networks, $V^r_{\boldsymbol{\phi_r}} : \mathbb{R}^{d_s} \mapsto \mathbb{R}$ and $V^c_{\boldsymbol{\phi_c}} : \mathbb{R}^{d_s} \mapsto \mathbb{R}$ with parameters $\boldsymbol{\phi_r}$ and $\boldsymbol{\phi_c}$. The reward value function has already been discussed in Chapter 2. The cost value function is also defined similarly for the cost function. Recall that having these networks helps us reduce variance.

We intialize our policy and value networks randomly and collect data from the environment using the method described in the above. We will denote this data using $\mathcal{D}$. Each data point is essentially a tuple $(\boldsymbol{s}_t, \boldsymbol{a_t}, \boldsymbol{s}_{t+1}, r, c)$ where $\boldsymbol{s}_t$ and $\boldsymbol{s}_{t+1}$ are the states at time t and $t+1$ respectively, $\boldsymbol{a}_t$ is the action taken at time t and r and c are the reward and cost recieved as a consequence of taking action $\boldsymbol{a}_t$. We use this dataset to update our parameters (discussed below). Furthermore, we initialize our Lagrange multiplier λ with a fixed constant value and also update it using this dataset. This entire process is repeated until convergence.

Recall that at each iteration, we are interested in optimizing (here we decompose the expectation over trajectories into expecation over states and actions):

$$J^{\text{PPO}}_{\text{LAG}}(\lambda, \pi_{\boldsymbol{\theta}}) = \sum_{t=1}^{T} \sum_{\boldsymbol{s}_t, \boldsymbol{a}_t \sim \mathcal{D}} \left[\frac{\pi_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{a}_t)}{\pi_{\bar{\boldsymbol{\theta}}}(\boldsymbol{s}_t, \boldsymbol{a}_t)} J^r(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) - \lambda(\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[J^c(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})] - \alpha) \right], \tag{4.3}$$

where

$$J^r(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) = \sum_{t'=t}^{T} r(\boldsymbol{s}_t, \boldsymbol{a}_t) - V^r_{\boldsymbol{\phi_r}}(\boldsymbol{s}_{t+1}), \tag{4.4}$$

and

$$J^c(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) = \sum_{t'=t}^{T} c(\boldsymbol{s}_t, \boldsymbol{a}_t) - V^c_{\boldsymbol{\phi_c}}(\boldsymbol{s}_{t+1}). \tag{4.5}$$

All parameters are updated via to gradient descent. Specifically, the policy network is updated according to:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha_1 \nabla_{\boldsymbol{\theta}} J^{\text{PPO}}_{\text{LAG}}(\lambda, \pi_{\boldsymbol{\theta}}), \tag{4.6}$$

and the Lagrange multiplier according to

$$\lambda := \lambda - \alpha_2 \nabla_\lambda J^{\text{PPO}}_{\text{LAG}}(\lambda, \pi_{\boldsymbol{\theta}}), \tag{4.7}$$

where α_1 and α_2 are learning rates.

Furthermore, we define the loss for the reward value function network as:

$$\mathcal{L}^r = \sum_{t=1}^{T} \sum_{\boldsymbol{s}_t, \boldsymbol{a}_t \sim \mathcal{D}} ||r(s_t, a_t) + \gamma V_{\boldsymbol{\phi_r}}(s_{t+1}) - V_{\boldsymbol{\phi}_r}(s_t)||_2^2, \tag{4.8}$$

and update its parameters according to:

$$\boldsymbol{\phi_r} := \boldsymbol{\phi_r} - \alpha_3 \nabla_{\boldsymbol{\phi_r}} \mathcal{L}^r(\boldsymbol{\phi_r}), \tag{4.9}$$

where α_3 is the learning rate.

Similarly, we define the loss for the cost value function network as:

$$\mathcal{L}^c = \sum_{t=1}^{T} \sum_{\boldsymbol{s}_t, \boldsymbol{a}_t \sim \mathcal{D}} ||c(s_t, a_t) + \gamma V_{\boldsymbol{\phi_c}}(s_{t+1}) - V_{\boldsymbol{\phi}_c}(s_t)||_2^2, \tag{4.10}$$

and update its parameters according to:

$$\boldsymbol{\phi_c} := \boldsymbol{\phi_c} - \alpha_4 \nabla_{\boldsymbol{\phi_c}} \mathcal{L}^c(\boldsymbol{\phi_c}), \tag{4.11}$$

where α_4 is the learning rate.

4.4 Implementation Details

In this section, we summarize some of the different techniuqes we used to speed up and stabilize the learning process.

Firstly, as discussed in Chapter 2, in offline pruning, once the network has been pruned, it is usually finetuned for a certain number of iterations. Therefore, ideally we would like to finetune the network before computing the policy's reward. However,

finetuning repeatedly is computationally expensive. So instead we adopt a finetuning schedule. In the beginning, when the policy is random and not good, we finetune the network less. Once the policy starts improving, we start finetuning more. Specifically, we employ a staircase-like schedule, where we divide our total number of iterations into a fixed number of equal intervals. In each interval, we finetune the network for a constant number of iterations.

Secondly, in practice, we normalize our rewards and cost values with a running mean and standard deviation which is continually updated as more data is collected. Furthermore, we also normalize the state vector in the similar way. Normalization has known to improve stability in practice..

Thirdly, for the coarse-grained strategy we generally clip the action to a value less than 1. This is because, initially, when the policy is not so good, it tends to take large random actions which result in entire layers being pruned. This then results in the policy receiving very low rewards, and hence it is not able to learn anything useful. So to prevent this we clip the action's value, similar to as in [17].

4.5 Extending to Online Pruning Settings

In this thesis, we restrict ourselves to the offline setting only. However, note that our method can easily be used in online settings, with any modifications. In the online setting, we would simply start out with a randomly intialized network we wish to prune, instead of a pretrained one. Then we would simply train this network in conjunction with all of the other networks described above.

Chapter 5

Experiments

5.1 Experimental Settings

We experiment with three different neural networks: VGG11, VGG16 and VGG19 [42]. We pretrain each of these networks on the CIFAR-10 dataset [23]. The hyper-paramters for pretraining are summarized in Table 5.1. This dataset contains $60,000$ colored images (therefore, three input channels) of size 32×32 belonging to 10 different classes. The accuracy of these networks on this dataset are summarized in Table 5.2.

Table 5.1: Hyperparameters for pretraining networks.

	VGG11	VGG16	VGG19
Optimizer	Adam	Adam	Adam
Learning Rate	3.0×10^{-4}	3.0×10^{-4}	3.0×10^{-4}
Iterations	35,000	50,000	60,000
Batch Size	60	60	60

Table 5.2: Accuracy of pretrained networks.

	VGG11	VGG16	VGG19
CIFAR-10	89.23	90.69	90.59

5.2 Implementation Details

Table 5.3: Hyperparameters for coarse and fine-grained strategies. For neural network architectures we report the number of hidden units in each layer. All hidden layers use the tanh activation function, whereas the output layer uses a sigmoid.

	Coarse-Grained	Fine-Grained
Policy, $\pi_{\boldsymbol{\theta}}$		
Architecture		
Policy Network	64, 64	64, 64
Value Network	64, 64	64, 64
Cost Value Network	64, 64	64, 64
Batch Size	64	64
PPO Target KL	0.01	0.01
Optimizer	Adam	Adam
Learning Rate	3×10^{-4}	3×10^{-4}
Reward-GAE-γ	0.99	0.99
Reward-GAE-λ	0.95	0.95
Cost-GAE-γ	1.00	1.00
Cost-GAE-λ	1.00	1.00
Lagrangian, λ		
Initial Value	1.0	1.0
Learning Rate	0.1	0.1
Pruning Environment		
Batch Size	20,000	20,000
Finetuning Schedule	0,32,128	0,32,128
Initial Action Clip Value[1]	0.8/0.9	-
Action Clip Gradient	0.05	-
Timesteps	40,000	40,000

[1]Values are for $\alpha = 20$ and 10 respectively

We model our policy as a (multivariate) Gaussian distribution $\mathcal{N}(\mu_\theta, \sigma)$ where μ_θ is a neural network parameterized by θ and σ_ϕ is a trainable parameter. We feed each state into μ_θ and then sample an action from the resulting distribution.

For the coarse-grained approach, the policy outputs a single scalar value while for the fine-grained approach it outputs a mask of length equal to the maximum number of filters in any layer of the network we wish to prune (in all of the three networks that we prune, this is equal to 512.)

The exact hyperparameter settings are summarized in Table 5.3. We elaborate on

some of the rows in more detail below.

As discussed in the previous chapter, offline pruning techniques typically finetune a network after pruning it. However, finetuning is expensive, so instead we use a schedule in which we finetune less in the beginning and more towards the end. This schedule is defined as follows: we divide the total number of timesteps (i.e., the steps that the agent takes in the environment) into equal number of intervals. During each interval, we finetune for a fixed number of iterations. So, for example, as reported in Table 5.3, for the coarse-grained strategy we divide the total number of training iterations into three intervals. For the first interval, we do not finetune the network to compute the accuracy. For the second and third intervals, we finetune for 32 and 128 iterations before computing the network's accuracy.

One problem that we noticed in the case of the coarse-grained strategy was that in the beginning when the policy is not so good, it tends to prune away all of the filters in each layer (in a bid to reduce its cost maximally). This prevents any meaningful learning from happening. To remedy this, we clip the value of the action that the policy takes (recall that the action is equal to the sparsity ratio). This value at which we clipped is then increased linearly throughout training (the gradient reported in the tables controls the increase in this value per timestep). Specifically, at timestep t, the value at which the action is clipped is computed as

$$\text{action clip value} \; = \; \text{initial action clip value} \; + \frac{t}{T} \times \text{action clip gradient} \tag{5.1}$$

where T is the total number of timesteps.

Finally, our implementation is based on the stable baselines library [18] which implements several deep reinforcement learning algorithms using PyTorch [32]. We use W&B [3] to manage and visualize our experiments.

5.3 Baseline

We use magnitude-based pruning with the same pruning ratio for each layer as our baseline. Specifically, given a sparsity budget α we simply prune away $\alpha\%$ of the connections in each layer. This ensures that the final network respects the specified budget. We then finetune the pruned network and report its accuracy on the test set.

5.4 Results and Discussion

We report our results in Tables 5.5 to 5.8. To compute the final accuracy, we finetune the pruned VGG11, VGG16 and VGG19 networks for $25,000$, $35,000$ and $40,000$ iterations respectively. Recall that for the fine-grained approach the policy predicts a scalar value in the range of 0 to 1. For evaluation, we threshold this value at 0.5.

One interesting phenomenon to notice is here is that the coarse-grained approach, in general, works better than the fine-grained one. We hypothesize that this is because the solution space of the fine-grained approach is much larger than that of the coarse-grained, which makes the optimization problem much harder to solve.

We also include our training graphs in Figures 5-1 to 5-12 that track the reward, cost, Lagrange multiplier (λ) and the mean of the actions that the agent took. The x-axis corresponds to the number of timesteps that the agent has taken in the environment.

In the figures, note that the reward often follows a staircase-like pattern. This is a direct conseuqence of employing a finetuning schedule. As the number of finetuning interations increase, the network's accuracy too starts to increase. Note that the reward is equal to the network's accuracy on a batch of data sampled from the training set. Furthermore, note that the Lagrange multiplier initially increases as long as the agent violates the specified budget. Once the agent's cost decreases to below the specified budget, the Lagrange multplier begins to decrease. Finally, note the oscillation in mean of the agent's action at different timesteps. This oscillation is desirable, since it indicates that the agent is taking different actions at each timestep,

and hence at each layer. In other words, this indicates that the agent is focussing on finding the optimal pruning strategy for each layer individually.

Table 5.4: Results for VGG11 for $\alpha = 20$. Unpruned accuracy is 89.23%.

	Final sparsity (%)	Final accuracy (%)
MP (baseline)	80.00	85.50
Coarse-grained	83.48	89.11
Fine-grained	83.29	86.47

Table 5.5: Results for VGG11 for $\alpha = 10$. Unpruned accuracy is 89.23%.

	Final sparsity (%)	Final accuracy (%)
MP (baseline)	90.00	83.80
Coarse-grained	90.75	88.09
Fine-grained	96.89	76.61

Table 5.6: Results for VGG16 for $\alpha = 20$. Unpruned accuracy is 90.69%.

	Final sparsity (%)	Final accuracy (%)
MP (baseline)	80.00	88.40
Coarse-grained	83.81	90.96
Fine-grained	78.24	89.55

Table 5.7: Results for VGG16 for $\alpha = 10$. Unpruned accuracy is 90.69%.

	Final sparsity (%)	Final accuracy (%)
MP (baseline)	90.00	87.10
Coarse-grained	92.90	89.89
Fine-grained	90.65	87.89

Table 5.8: Results for VGG19 for $\alpha = 20$. Unpruned accuracy is 90.59%.

	Final sparsity (%)	Final accuracy (%)
MP (baseline)	80.00	88.40
Coarse-grained	83.48	91.06
Fine-grained	87.59	90.04

Table 5.9: Results for VGG19 for $\alpha = 10$. Unpruned accuracy is 90.59%.

	Final sparsity (%)	Final accuracy (%)
MP (baseline)	90.00	86.90
Coarse-grained	92.31	91.31
Fine-grained	77.95	90.62

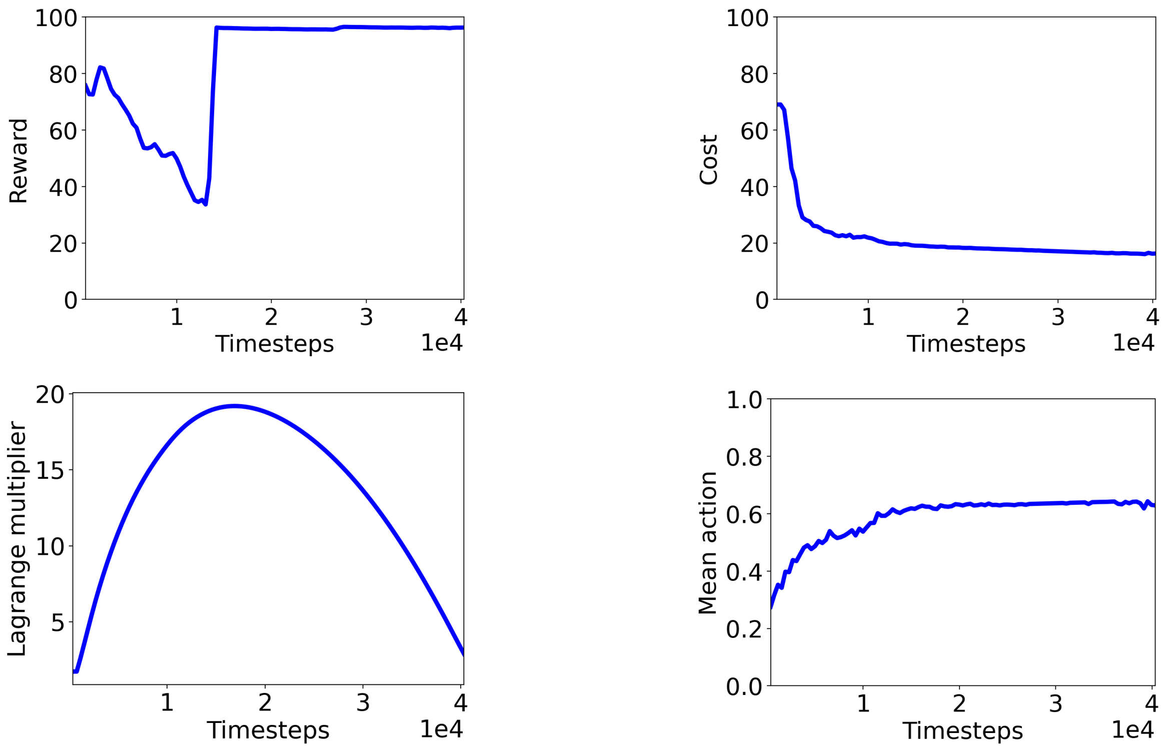

Figure 5-1: Training graphs for coarse-grained strategy on VGG11 for $\alpha = 20$.

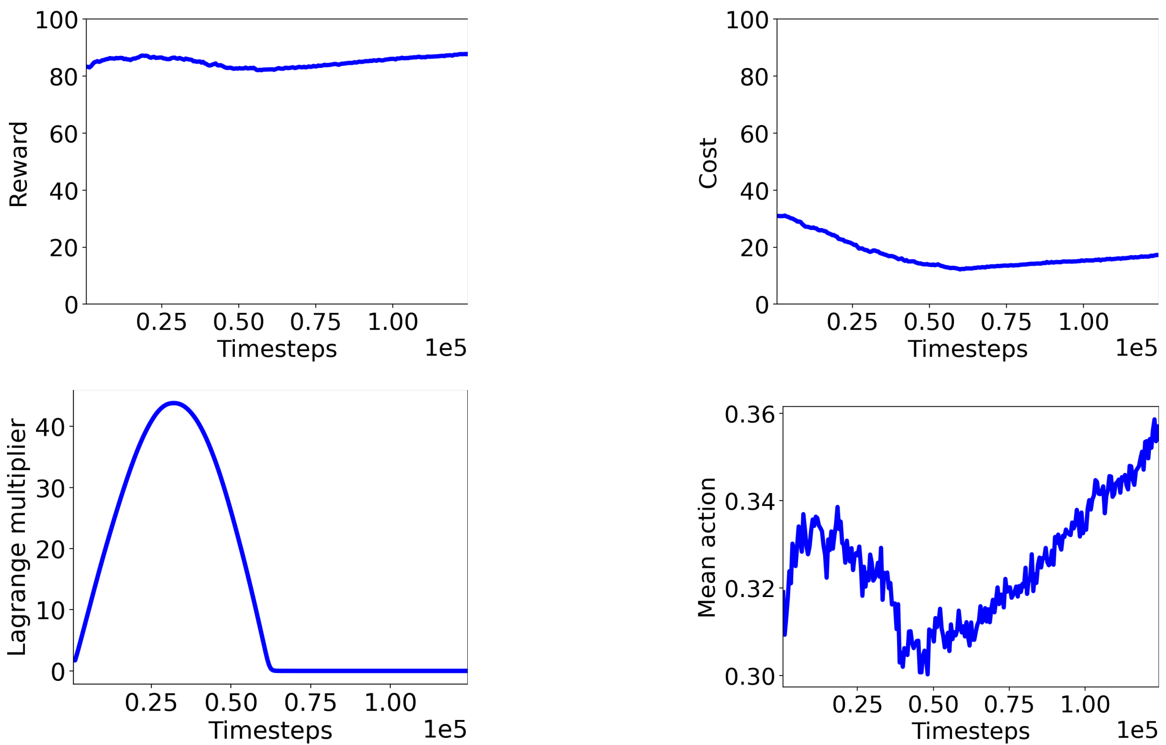

Figure 5-2: Training graphs for fine-grained strategy on VGG11 for $\alpha = 20$.

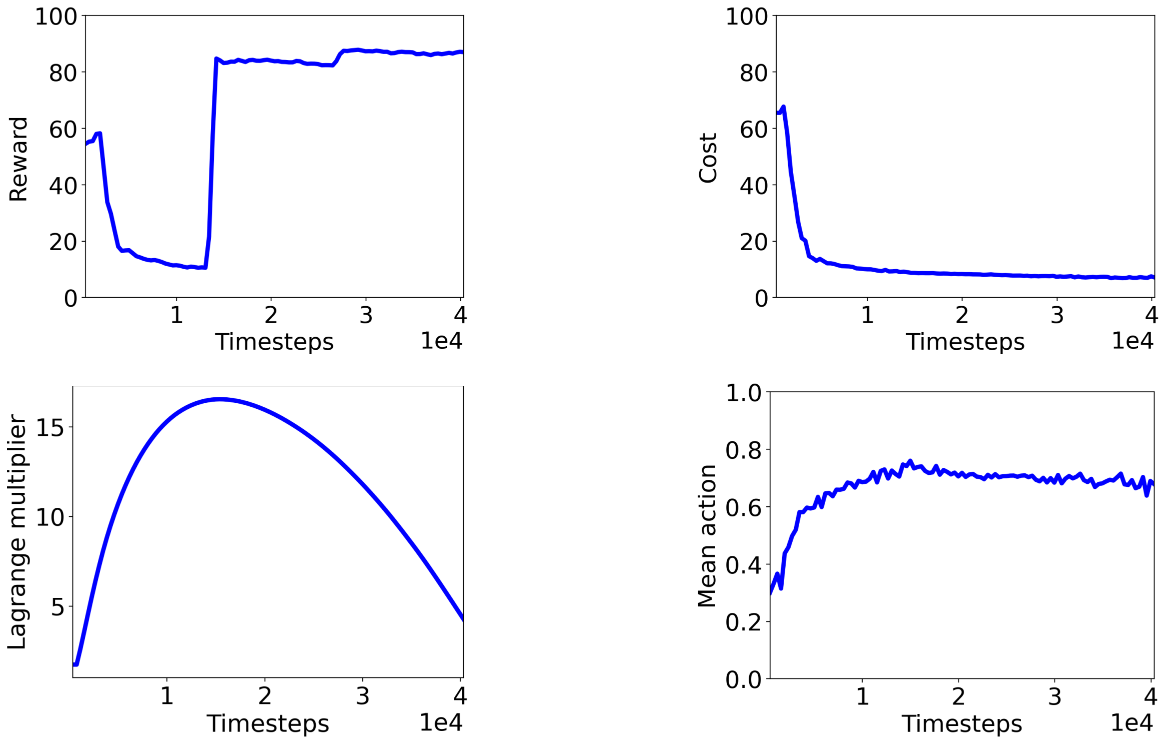

Figure 5-3: Training graphs for coarse-grained strategy on VGG11 for $\alpha = 10$.

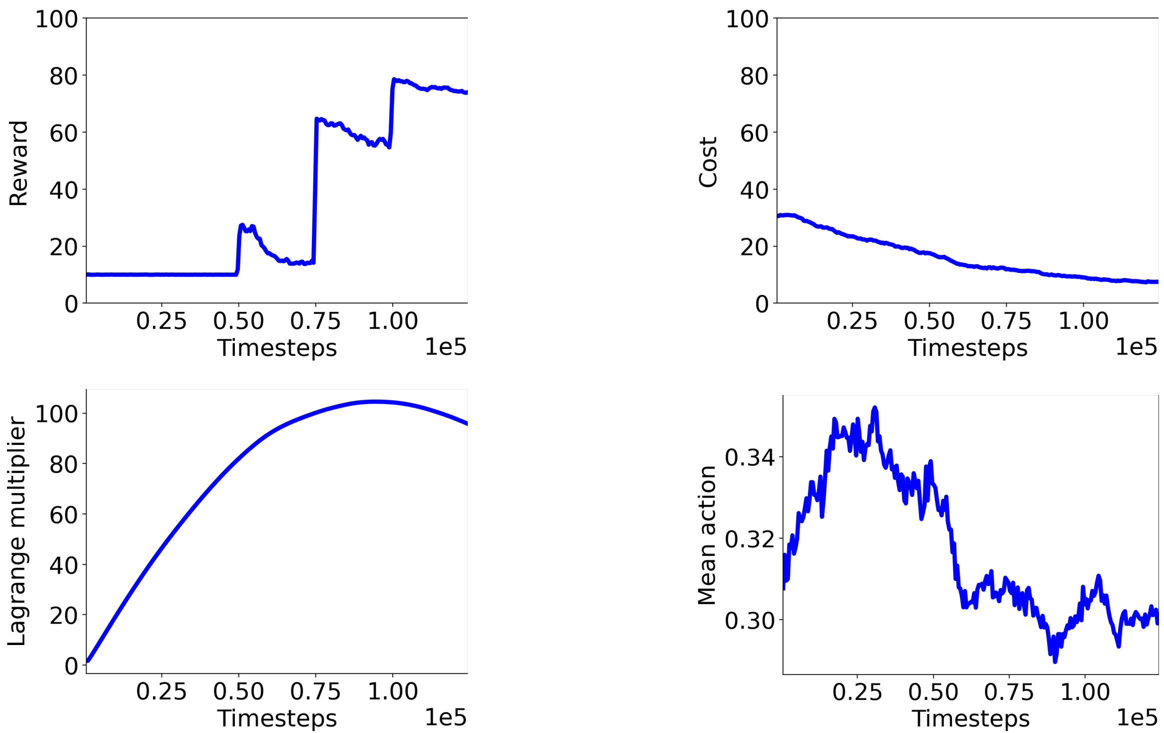

Figure 5-4: Training graphs for fine-grained strategy on VGG11 for $\alpha = 10$.

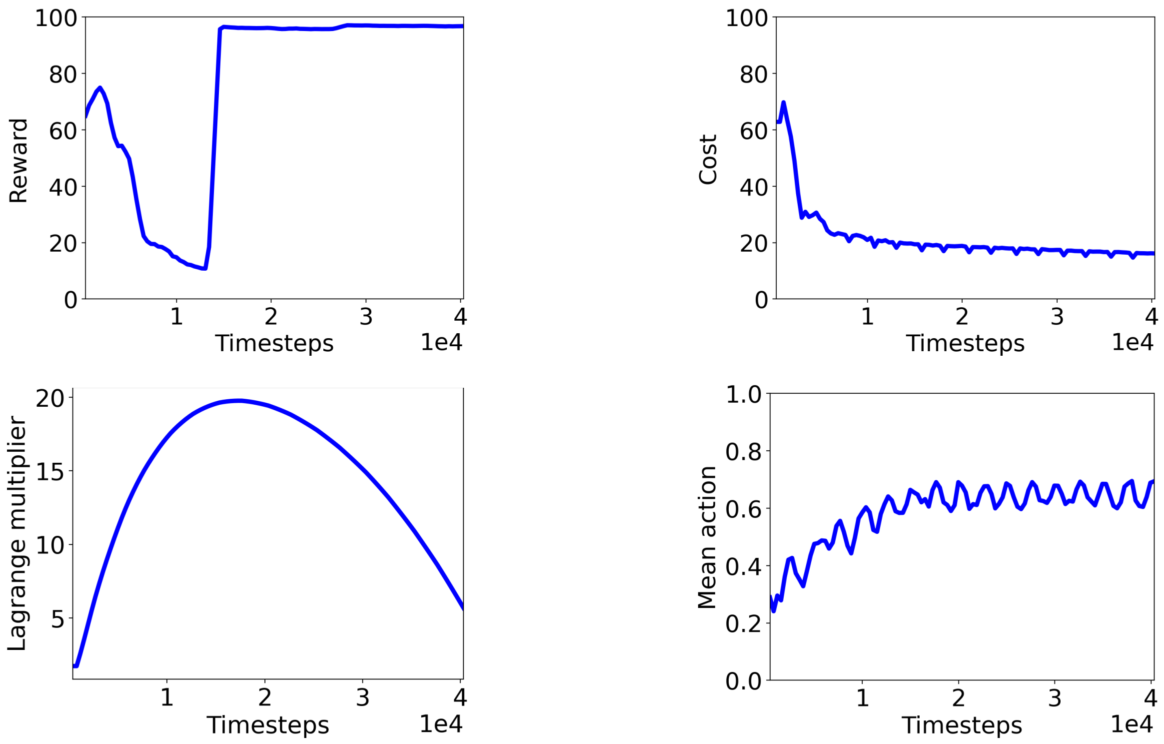

Figure 5-5: Training graphs for coarse-grained strategy on VGG16 for $\alpha = 20$.

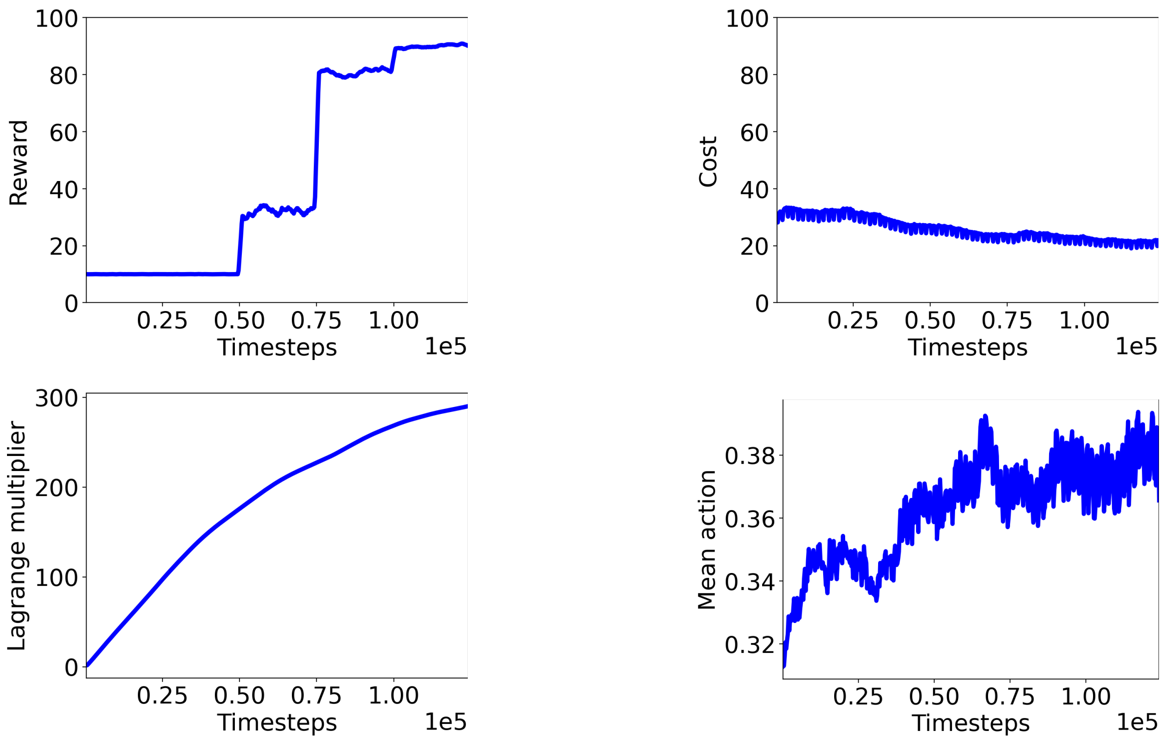

Figure 5-6: Training graphs for fine-grained strategy on VGG16 for $\alpha = 20$.

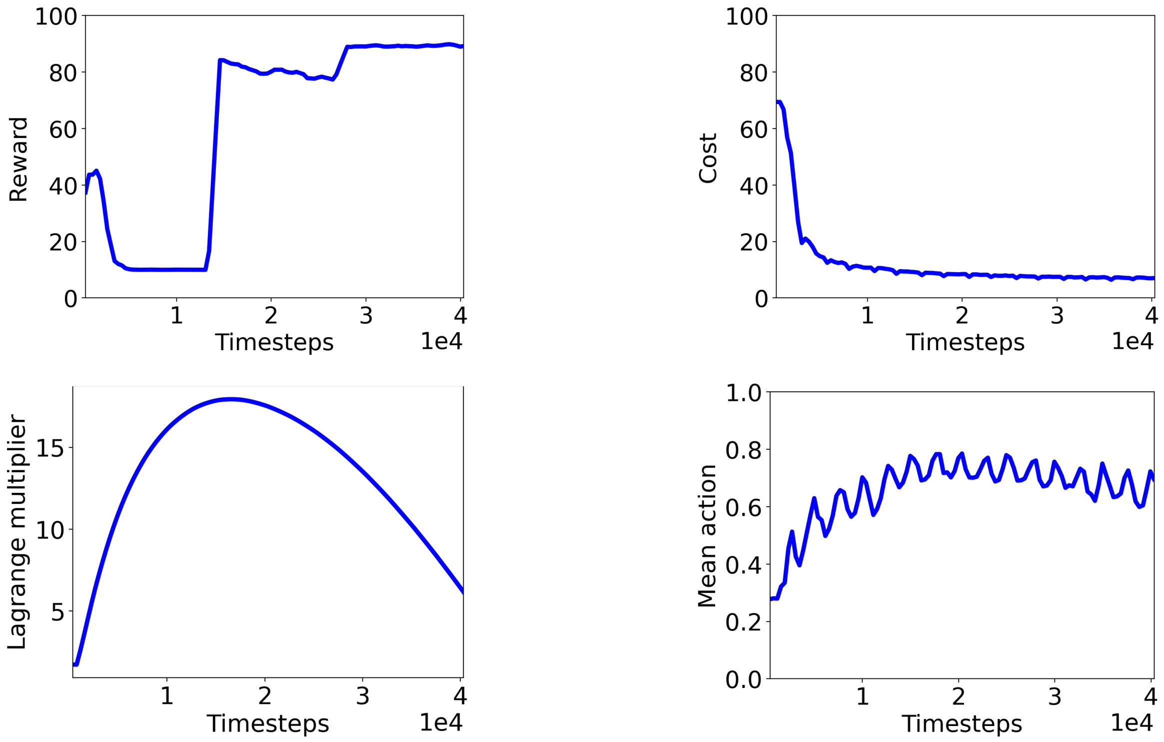

Figure 5-7: Training graphs for coarse-grained strategy on VGG16 for $\alpha = 10$.

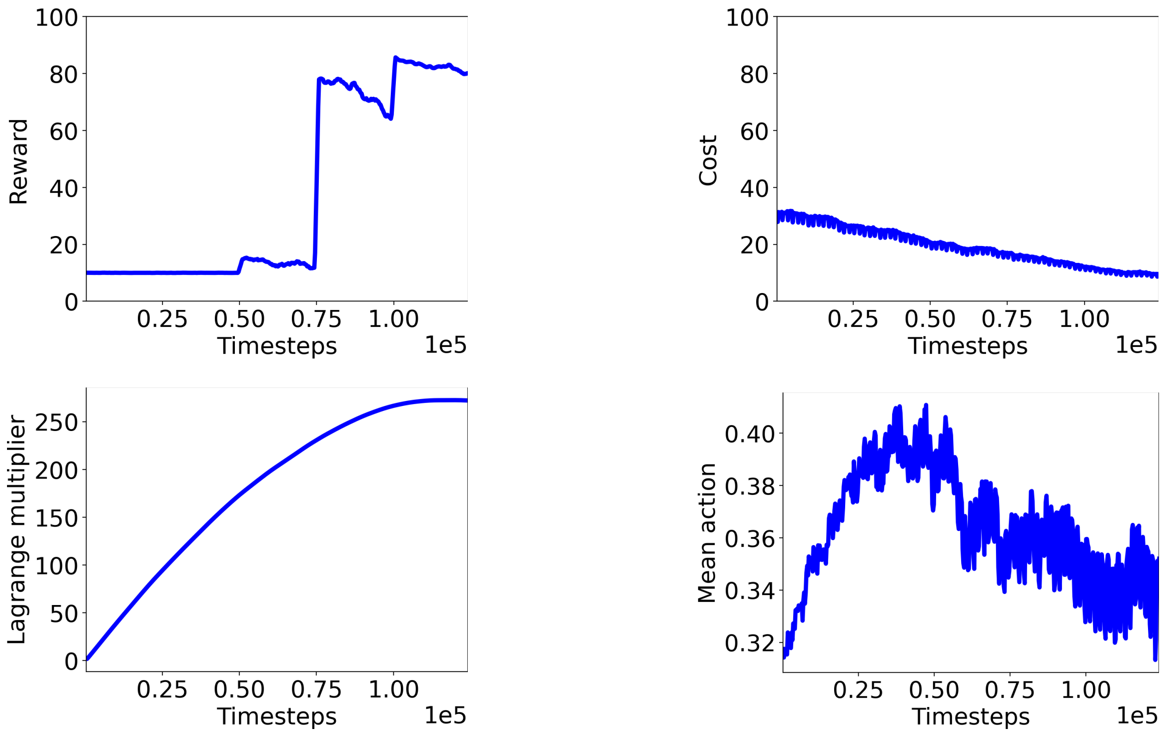

Figure 5-8: Training graphs for fine-grained strategy on VGG16 for $\alpha = 10$.

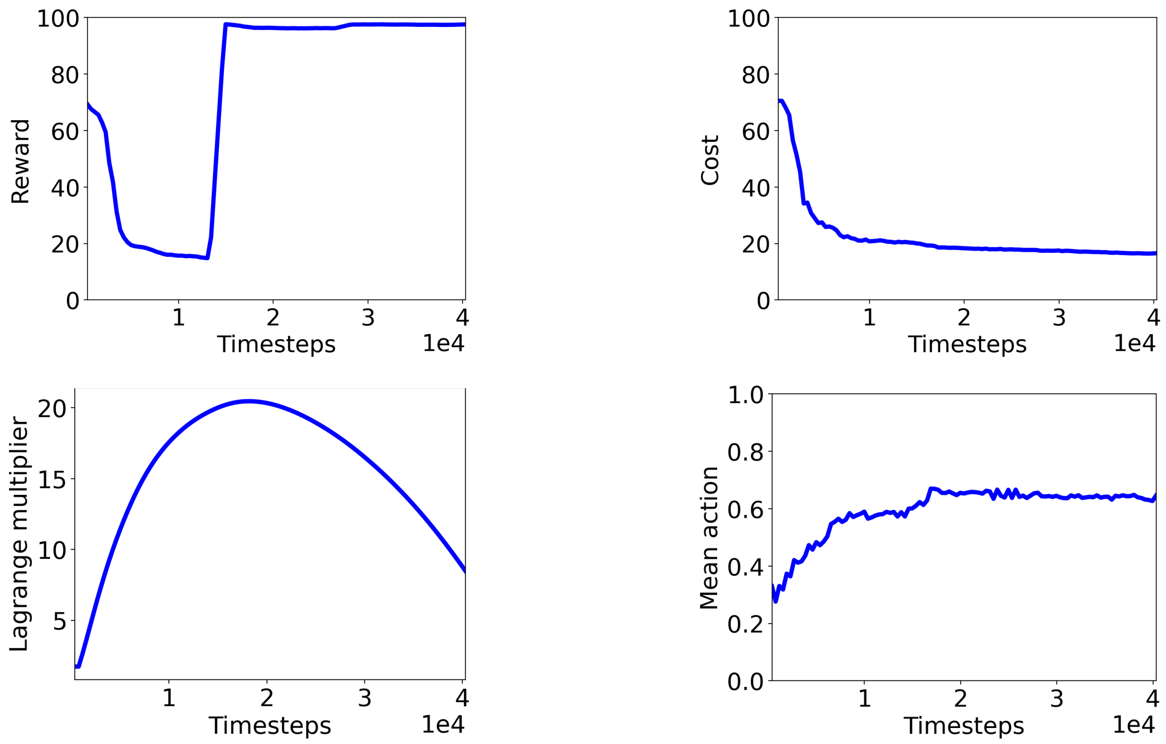

Figure 5-9: Training graphs for coarse-grained strategy on VGG19 for $\alpha = 20$.

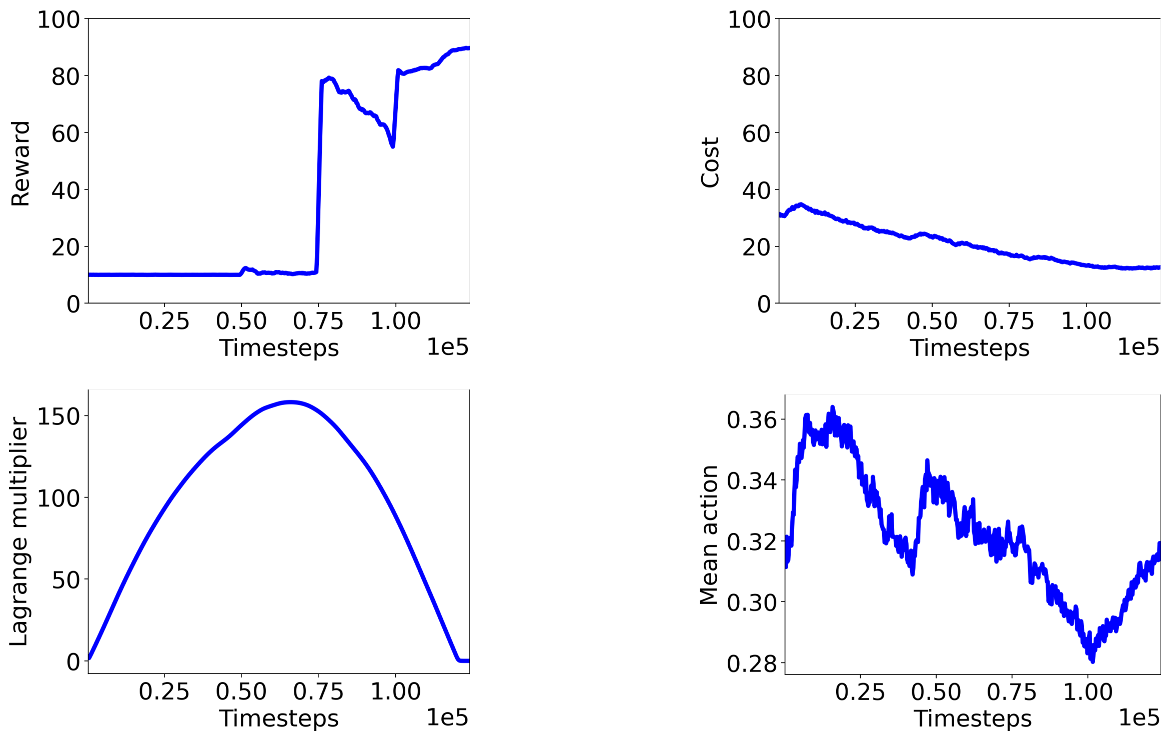

Figure 5-10: Training graphs for fine-grained strategy on VGG19 for $\alpha = 20$.

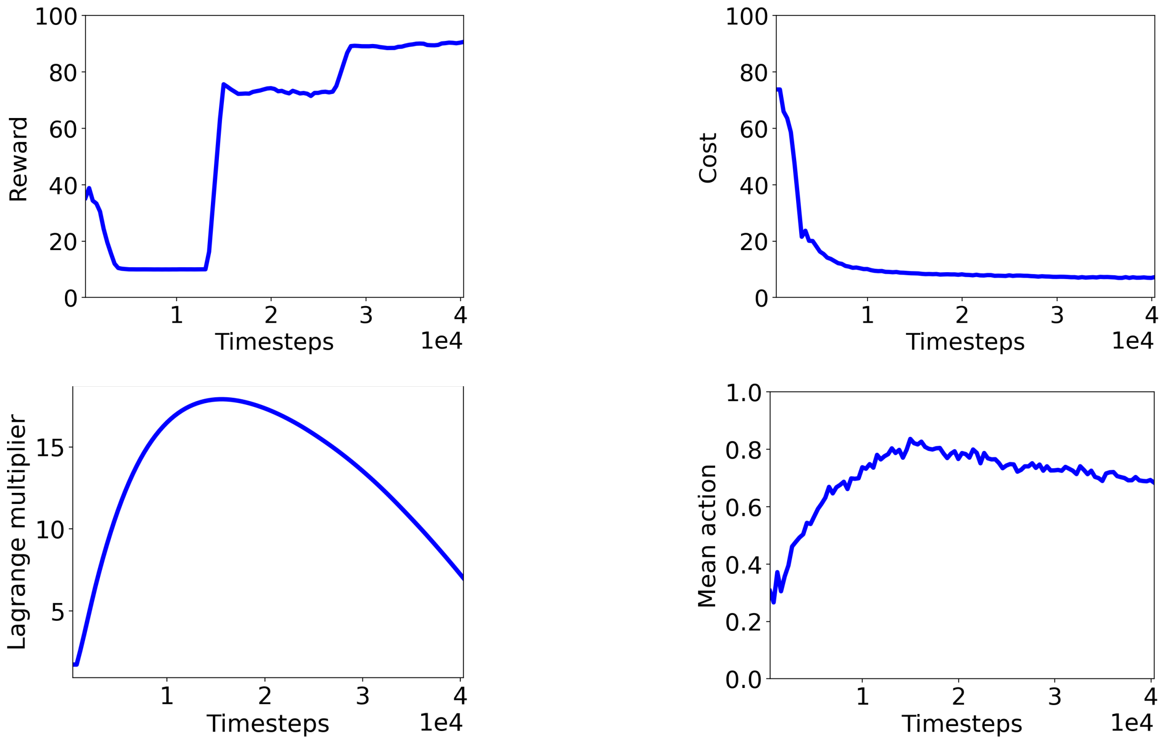

Figure 5-11: Training graphs for coarse-grained strategy on VGG19 for $\alpha = 10$.

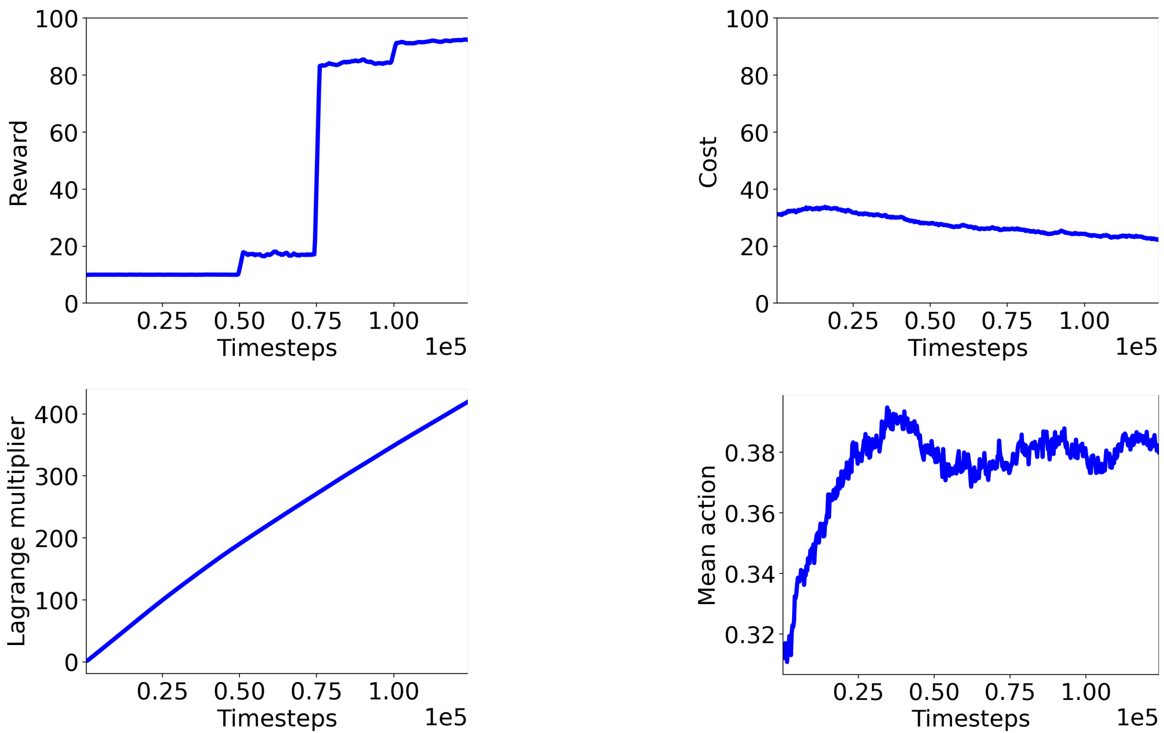

Figure 5-12: Training graphs for fine-grained strategy on VGG19 for $\alpha = 10$.

Chapter 6

Conclusions and Future Work

In this thesis, we first formulated pruning in the framework of constrained Markov decision processes and then proceeded to use constrained reinforcement learning algorithms to solve it. Unlike previous works, our approach can prune deep neural networks upto arbitrary pre-defined budgets on functions that need not be differentiable or fully specified. Furthermore, we carried out experiments to demonstrate the effectiveness of our approach.

Several extensions of this work are possible. First, we only tested our method with constraints on sparsity. Future works can focus on testing this method on other constraints (such as those on inference time or number of flops). Second, we only used one constrained reinforcement learning algorithm. Future works can explore more recent and state-of-the-art algorithms. Finally, as noted in Chapter 5, the coarse-grained approach tends to work better than the fine-grained approach, which we hypothesize is because the solution space of the latter is much larger than that of the former. Future works can also focus on making the optimization in the larger solution space of the fine-grained approach more efficient.

Bibliography

[1] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *International Conference on Machine Learning*. PMLR, 2017.

[2] Eitan Altman. *Constrained Markov Decision Processes*. Chapman and Hall, 1999.

[3] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[4] Davis Blalock, Gonzalez Ortiz, Jose Javier, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 129–146, 2020.

[5] Dan A. Calian, Daniel J. Mankowitz, Tom Zahavy, Zhongwen Xu, Junhyuk Oh, Nir Levine, and Timothy A. Mann. Balancing constraints and rewards with meta-gradient D4PG. In *International Conference on Learning Representations*, 2021.

[6] Miguel Calvo-Fullana, Santiago Paternain, Luiz F. O. Chamon, and Alejandro Ribeiro. State augmented constrained reinforcement learning: Overcoming the limitations of learning with rewards, 2021. arXiv:2102.11941.

[7] Yinlam Chow, Ofir Nachum, Edgar Duenez-Guzman, and Mohammad Ghavamzadeh. A lyapunov-based approach to safe reinforcement learning. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2018.

[8] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4), 1989.

[9] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017.

[10] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[12] Priya Goyal, Mathilde Caron, Benjamin Lefaudeux, Min Xu, Pengchao Wang, Vivek Pai, Mannat Singh, Vitaliy Liptchinsky, Ishan Misra, Armand Joulin, and Piotr Bojanowski. Self-supervised pretraining of visual features in the wild, 2021. arXiv:2103.01988.

[13] Muhammad Umair Haider and Murtaza Taj. Comprehensive online network pruning via learnable scaling factors, 2020. arXiv:2010.02623.

[14] Babak Hassibi, David G. Stork, Gregory Wolff, and Takahiro Watanabe. Optimal brain surgeon: Extensions and performance comparisons. In *Advances in Neural Information Processing Systems*. Morgan-Kaufmann, 1993.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[17] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *European Conference on Computer Vision*. Springer International Publishing, 2018.

[18] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018. Last accessed on May 20, 2021.

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. arXiv:1503.02531.

[20] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), 1989.

[21] Steven A. Janowsky. Pruning versus clipping in neural networks. *Phys. Rev. A*, 39:6600–6603, Jun 1989.

[22] E. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, 1990.

[23] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[24] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*. Morgan-Kaufmann, 1990.

[25] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*, page 319, Berlin, Heidelberg, 1999. Springer-Verlag.

[26] Jaeho Lee, Sejun Park, and Jinwoo Shin. Learning bounds for risk-sensitive learning. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2020.

[27] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. SNIP: single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*, 2019.

[28] Carl Lemaire, Andrew Achkar, and Pierre-Marc Jodoin. Structured pruning of neural networks with budget-aware regularization. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[29] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representation*, 2017.

[30] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017.

[31] Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann, 1989.

[32] 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*. Curran Associates, Inc., 2019.

[33] Santiago Paternain, Miguel Calvo-Fullana, Luiz F. O. Chamon, and Alejandro Ribeiro. Safe policies for reinforcement learning via primal-dual methods, 2019. arXiv:1911.09101.

[34] Santiago Paternain, Luiz Chamon, Miguel Calvo-Fullana, and Alejandro Ribeiro. Constrained reinforcement learning has zero duality gap. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2019.

[35] Alex Ray, Joshua Achiam, and Dario Amodei. Benchmarking Safe Exploration in Deep Reinforcement Learning. 2019.

[36] Frank Rosenblatt. The perceptron - a perceiving and recognizing automation. Technical report, 1957.

[37] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017. arXiv:1609.04747.

[38] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Neurocomputing: Foundations of Research*, 1988.

[39] Harsh Satija, Philip Amortila, and Joelle Pineau. Constrained Markov decision processes via backward value functions. In *International Conference on Machine Learning*. PMLR, 2020.

[40] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In *International Conference on International Conference on Machine Learning*. PMLR, 2015.

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

[42] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations*, 2015.

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

[44] Chen Tessler, Daniel J. Mankowitz, and Shie Mannor. Reward constrained policy optimization. In *International Conference on Learning Representations*, 2019.

[45] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[46] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020.

[47] Haohan Wang and Bhiksha Raj. On the origin of deep learning, 2017. arXiv:1702.07800.

[48] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 1992.

[49] Tengyu Xu, Yingbin Liang, and Guanghui Lan. A primal approach to constrained policy optimization: Global optimality and finite-time analysis, 2020. arXiv:2011.05869.

[50] Ruiyi Zhang, Tong Yu, Yilin Shen, Hongxia Jin, and Changyou Chen. Text-based interactive recommendation via constraint-augmented reinforcement learning. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2019.