Metareasoning in Modular Software Systems: On-the-Fly Configuration Using Reinforcement Learning with Rich Contextual Representations

Aditya Modi,[1] Debadeepta Dey,[2] Alekh Agarwal,[2]
Adith Swaminathan,[2] Besmira Nushi,[2] Sean Andrist,[2] Eric Horvitz[2]
[1]University of Michigan Ann Arbor, [2]Microsoft Research Redmond
[1]admodi@umich.edu, [2]{dedey, alekha, adswamin, besmira.nushi, sandrist, horvitz}@microsoft.com

Abstract

Assemblies of modular subsystems are being pressed into service to perform sensing, reasoning, and decision making in high-stakes, time-critical tasks in areas such as transportation, healthcare, and industrial automation. We address the opportunity to maximize the utility of an overall computing system by employing reinforcement learning to guide the configuration of the set of interacting modules that comprise the system. The challenge of doing system-wide optimization is a combinatorial problem. Local attempts to boost the performance of a specific module by modifying its configuration often leads to losses in overall utility of the system's performance as the distribution of inputs to downstream modules changes drastically. We present metareasoning techniques which consider a rich representation of the input, monitor the state of the entire pipeline, and adjust the configuration of modules on-the-fly so as to maximize the utility of a system's operation. We show significant improvement in both real-world and synthetic pipelines across a variety of reinforcement learning techniques.

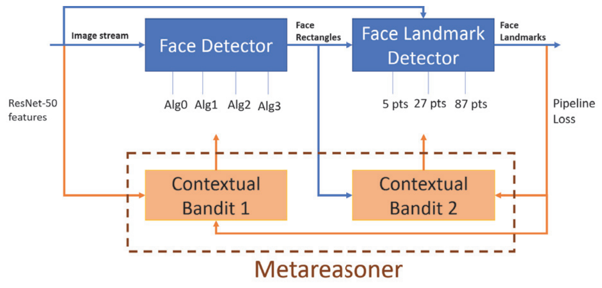

Figure 1: Face and landmark detection modular system. The input is an image stream to the face detection module which outputs locations of faces in the image which are then input to the face landmark detection module which outputs locations of eyes, nose, lips, brows etc on the detected faces. The metareasoning module receives the input stream of images along with intermediate outputs of the face detector to dynamically decide the configuration of the pipeline such that it optimizes the end system loss.

1 Introduction

The lives of a large segment of the world's population are greatly influenced by complex software systems, be it the software that returns search results, enables the purchase of an airplane ticket, or runs a self-driving car. Software systems are inherently modular, i.e. they are composed of numerous distinct modules working together. As an example, a self-driving car has modules for sensors such as cameras, lidars which poll the sensors and output sensor messages, and a mapping module that consumes sensor messages and creates a high-resolution map of the immediate environment. The output of the mapping module is then input to a planning module whose job is to create safe trajectories for the vehicle. These distinct modules often operate at different frequencies; the camera module may be producing images at 120Hz while the GPS module may be producing vehicle position readings at 1000Hz. Furthermore, they may each have their own set of free parameters which are set via access of a configuration file at startup. For example, the software serving as the driver of a camera in the self-driving pipeline may have a parameter setting for the rate at which images are pulled from the camera and another parameter for the resolution of the images. Similarly, the function of the mapping module may be controlled by a parameter that specifies the maximum amount of memory it is allowed to consume, leading to the continual removal of information about more distant and thus less relevant map content.

Large software systems typically are composed of a set of distinct modular components. The operating characteristics of all of the components are usually manually configured to achieve system performance targets or constraints like accuracy and/or latency of output. Configurations of parameters may result from the tedious and long-term tuning of one parameter at a time. Once such nominal configurations have been produced, they are then held constant during system execution. The reliance on such fixed policies in a dynamic world may often be sub-optimal. As an example, modules may take different amounts of time depending on the specific contents of the inputs they receive.

As a running example, we illustrate a pipeline for extracting faces with keypoint annotations from images in Figure 1. A natural performance metric for the pipeline might blend the prediction latency and accuracy, where the latency of a face-detection module may vary dramatically based on the number of people in the camera view. In this case, one might prefer switching to a parameter setting which allows the face detector to sacrifice some accuracy but which is much faster hence raising the overall utility of the entire pipeline. Also modules which are upstream from the face detector like the camera driver module might ideally throttle back the rate at which it is producing images since most of these images will not get processed anyways, due to a bottleneck at the face detector module. Attempts to separately optimize distinct modules can often lead to losses in utility (Bradley 2010) because of unaccounted shifts in the distribution of outputs produced by upstream modules.

Revisiting the self-driving car example, a basic utility function is to simply navigate passengers to their destination safely and in a reasonable amount of time. Highlighting the contextuality again, the emphasis on driving time might be higher when trying to get to an important meeting or a flight than going grocery shopping. Furthermore, the utility function will typically be specific to the user and has to be inferred over time. Importantly, this is a complex pipeline-level feedback which is hard to attribute to individual components.

In this work, we leverage advances in representation and reinforcement learning (RL) to develop metareasoning machinery that can optimize the configuration of modular software systems under changing inputs and compute environments. Specifically, we demonstrate that by having a metareasoner continuously monitor the entire system we can switch parameters of each module on-the-fly to *adapt* to changing inputs and optimize a desired objective. We also study the distinction between attainable performance by choosing the best configuration for the entire pipeline as a function of just the initial input, versus further choosing the configuration of each module based on all the preceding actions and outputs. We experiment with a synthetic pipeline meant to require adaptivity to the inputs, and we find that by doing so at each module, we improve by roughly 50% or more over the best constant assignment, and typically by a similar margin over the choice of a configuration just as a function of the initial input. For the face and landmark detection pipeline 1, we use the activations of a pretrained neural network model as a contextual signal and leverage this rich representation of context in decisions about the configuration of each module *before* the module operates on its inputs. We characterize the boosts in utility provided via use of this contextual information, improving 9% or more across different utility functions as opposed to the best static configuration of the system. Overall, our experiments demonstrate the importance of online, adaptive configuration of each module.

2 Related Work

RL to control software pipelines: Decisions about computation under uncertainties in time and context have been described in (Horvitz and Lengyel 1997), which presented the use of metareasoning to guide graphics rendering under changing computational resources, considering probabilistic models of human attention so as to maximize the perceived quality of rendered content. The metareasoning guided tradeoffs in rendering quality under shifting content and time constraints in accordance with preferences encoded in a utility function. Principles for guiding proactive computation were formalized in (Horvitz 2001). (Raman et al. 2013) characterize a tradeoff between computation and performance in data processing and ML pipelines, and provide a message-passing algorithm (derived by viewing pipelines as graphical models) that allows a human operator to manually navigate this tradeoff. Our work focuses on the use of metareasoning to replace the operator by setting the best operating point for any pipeline automatically.

(Bradley 2010) proposed using subgradient descent coupled with loss functions developed in imitation learning in order to jointly optimize modular robotics software pipelines which often involve planning modules, when the modules are differentiable with respect to the overall utility function. This is not suited to most real-world pipelines with modules described not with parameters but lines of code. In this work we instead develop fully general methods, which only assume the ability to evaluate the pipeline. Another form of pipeline optimization is to accordingly pick or configure the machine where each module should be executed. Methods in this ambit (Mirhoseini et al. 2017) are complementary to this work in that optimizing the pipeline configuration per se remains a problem even with optimal device placement.

RL in distributed system optimization: The use of machine learning for optimizing resource allocation in distributed systems for data center and cluster management has been very well studied (Lorido-Botran, Miguel-Alonso, and Lozano 2014; Demirci 2015; Delimitrou and Kozyrakis 2013; 2014). Many of these techniques use supervised learning as well as collaborative filtering for resource assignment, which rely on the assumption of having a rich set of processes in the training data and might as a result suffer from eventual data bias for new workloads. Most recently, the use of reinforcement learning for learning policies which dynamically optimize resources such that service level agreements can be better satisfied has received a lot of attention especially with the rise of 'deep' reinforcement learning ((Li 2017)). Methods using model-free methods (Mao et al. 2016; Xu, Rao, and Bu 2012) have shown promise as modeling such large-scale distributed systems is a challenge in itself. Similarly, RL has found impressive success in energy optimization for data centers (Gao 2014; Memeti et al. 2018).

RL for scheduling in operating systems: Even at the single machine level, RL has found promise for thread scheduling and resource allocation in operating systems. For example (Fedorova, Vengerov, and Doucette 2007; Hanus 2013) use RL-based methods to learn adaptive policies which outperform the best statically optimal policy (found by solving a queuing model) as well as myopic reactive policies which greedily optimize for short term outcomes. The problem of scheduling in operating systems however differs from pipeline optimization in two fundamental ways. First, the operating system (as well as the scheduler) is oblivious to accuracy dependencies between different processes or threads. Second, due to either architectural or generality constraints, schedulers do not optimize process-level parameters but mainly focus on machine configuration.

3 Problem Definition

3.1 Formal Setting and Notation

A pipeline of M modules can be viewed as a directed graph where each node j is a module and an edge from j to k represents module k consuming the output of j as its input. We assume the graph does not have any cycles. Without loss of generality, let the modules be numbered according to their topological sort; i.e. j refers to the index of a module in a linear ordering of the DAG[1]. For each module j, we have a set of possible configurations—these are the actions that are available for the metareasoner to choose from. We denote this set by A_j. A module j can then be viewed as a mapping from its inputs $x \in {S_j}^{in}$ to outputs $z \in {S_j}^{out}$, and each configuration $a \in A_j$ implies a different mapping. As a running example, we will consider the face detection pipeline of Figure 1. The pipeline contains two modules with module 1 having 4 choices and module 2 having 3 choices. The input space to the first module ${S_1}^{in}$ is the space of images (possibly in a feature space). The output space ${S_1}^{out}$ is the same as ${S_2}^{in}$ and can encode the image, the locations of faces in the image, and the latency induced by the first module.

The quality of a pipeline's operation is measured using a loss function denoted by $L : {S_M}^{out} \mapsto \mathbb{R}$. In the example pipeline of Figure 1, the outputs from the landmark detector can be labeled by human evaluators to assess accuracy and L can be a complex trade-off between the latency incurred by the overall pipeline in processing an image vs. the accuracy of the detected landmarks. If labels are not available, accuracy might be inferred from proxies such as an incorrect denial of authentication for a user based on the landmark detector output, which can be observed when the user authenticates via alternative means such as a password. Crucially, we only observe the value of this loss-function for the specific outputs $z \in {S_M}^{out}$ that the pipeline generates based on a certain configuration of actions at each module in response to an input x. We highlight that the loss function L can be any function mapping the pipeline's final output and system state to a scalar value, such as a passenger's satisfaction with a ride in a self-driving car as discussed in Section 1.

A metareasoner can be represented as a collection of (possibly randomized) policies $\pi := \{\pi_1 \dots \pi_M\}$, where $\pi_j : {S_j}^{in} \mapsto \Delta(A_j)$ specifies a context-dependent configuration of the module and $\Delta(A_j)$ is the set of distributions over the action set A_j. We abuse the notation for ${S_j}^{in}$ here to denote any succinct representation of the preceding pipeline component's outputs, actions and system state variables which are needed to choose the appropriate action for module j. The pipeline receives a stream of inputs and we use t to index the inputs. At time t, the pipeline receives an initial input $x_t^1 \in S_1^{in}$, based on which an action $a_t^1 \sim \pi_1(x_t^1)$ is picked at the first module and it produces an intermediate output z_t^1. This induces the next input $x_t^2 \in {S_2}^{in}$ at the second module, at which point the policy π_2 is used to pick the next action and so on. At each intermediate module j, the input x_t^j depends on the outputs of all its parents in the DAG corresponding to the pipeline and we assume that the input spaces ${S_j}^{in}$ are chosen appropriately so that a good metareasoner policy for module j can solely depend on x_t^j instead of having to depend explicitly on the outputs of its predecessors. As a result, the interaction between the metareasoner and the environment can be summarized as follows:

1. $x_t^1 \in {S_1}^{in}$ is fed as input to the pipeline.
2. metareasoner chooses actions for each module based on the output of its predecessors and induces a trajectory: $(x_t^1, a_t^1, z_t^1, \dots, x_t^M, a_t^M, z_t^M)$; eventual output of the pipeline is z_t^M.
3. Observe loss $L(z_t^M)$.

Formulated this way, the task of the metareasoner can be viewed as an episodic fixed-horizon reinforcement learning problem, where the state transitions are deterministic (although the initial input can be highly stochastic, such as an image in the face detection example). Each input processed by the pipeline is an episode, the horizon is M, actions chosen by policies for the upstream modules affect the state distribution seen by downstream policies. The feedback is extremely sparse with the only loss being observed at the end of the pipeline. The goal of the metareasoner is to minimize its average loss: $\frac{1}{T}\sum_{t=1}^{T} L(z_t^M)$, and the ideal metareasoner can be described as:

$$\underset{\pi_1 \dots \pi_M}{\arg\min} \sum_{t=1}^{T} \mathbb{E}_{\pi_1,\dots,\pi_M}\left[L(z_t^M) \mid x_t^1\right] \doteq J(\pi). \qquad (1)$$

Our goal is to learn a metareasoner during the live operation of the pipeline. Since we only observe pipeline losses for the current choices of the metareasoner's policies, we must balance exploration to discover new pipeline configurations, and exploitation of previously found performant configurations. In such explore-exploit problems, we measure the average loss accumulated by our adaptive learning strategy as a benchmark; a lower loss is better. A better learning strategy will quickly identify good context-dependent configurations and hence have lower average loss as T increases.

3.2 Challenges

In this section we highlight the important challenges that a metareasoner needs to address.

Combinatorial action space: Viewing the entire pipeline as a monolithic entity, with an aim to find the best fixed assignment for each module with no input dependence, leaves the metareasoner with combinatorially many choices to consider. This can quickly become intractable even for modest pipelines (e.g. See Figure 2), despite the use of the simplest possible static policy class.

[1]For our solution methods, all we need is the raw input given to a module, current system parameters, and statistics of modules which have finished the computation. In general though, having an architectural graph helps because if the architecture is different from a pipeline, many modules may have finished the computation but not all them may be relevant to the current module. Thus, we use the topological sort for a DAG pipeline to establish a dependency order and easily fix an informative state representation for the policy.

Adaptivity to inputs: Having a static action assignment per module is overly simplistic in general and we typically need a policy for manipulating configurations that is context-sensitive. For example, in Figure 4, we observe that the number of faces in the input image implies a fundamentally different trade-off between latency and accuracy.

Credit Assignment: Since we only observe delayed episodic reward, we do not know which module was to blame for a bad pipeline loss. For non-additive losses, this is especially challenging (III et al. 2018).

Exploration: Pipeline optimization offers a fundamentally challenging domain for exploration. Though we employ ideas from contextual bandits here for exploration, we anticipate future directions that explore by using pipeline structure to derive better learning strategies.

4 Methods

The methods we outline now each address some of the challenges in Section 3.2. The simplest strategy is a non-adaptive approach that effectively handles combinatorial actions (Section 4.1) to return a locally optimal static assignment. A simple context-sensitive strategy that views the problem as a contextual bandit, and is vulnerable to a combinatorial scaling with pipeline size (Section 4.2). Finally, the most sophisticated strategy we develop produces a context-adaptive policy, exploits pipeline structure to learn per-module policies and uses policy-gradient algorithms to quickly reach a locally optimal configuration policy (Section 4.3).

4.1 Greedy Hill Climbing

The simplest (infeasible) strategy for pipeline optimization with input examples $x_1^1, \ldots, x_T^1$ is to brute-force try every possible configuration for each of the T inputs and pick the configuration that accumulates the lowest loss. This strategy will identify the best non-adaptive (i.e. context-insensitive) configuration, but needs $T \cdot \prod_{j=1}^{M} |A_j|$ executions of the pipeline to find this configuration. Since this is intractable even for modest values of T and A_j (especially in real-time), we now describe a tractable alternative to find an approximately good configuration via random co-ordinate descent.

Rather than identifying the best configuration, suppose we aim to find a "locally optimal" configuration – that is, for every module, if we held all other module configurations fixed then deviating from the current configuration can only worsen the pipeline loss. To achieve this, we begin by randomly picking an initial configuration. In each epoch, we first sample K out of T examples uniformly with replacement from the dataset, where K is a hyperparameter that can be set based on the available computational budget. We then choose one of the modules $j \in \{1, 2, \ldots, M\}$ uniformly at random and keep the configurations of all other modules fixed. We cycle through every possible action for that module (using, for instance, $K/|A_j|$ examples for each choice of action at this module) and pick the action that achieves the lowest accumulated loss. We then repeat this process until our training budget is exhausted, or we cycled through every module without changing the configuration (which indicates a local optimum). This is akin to a greedy hill-climbing strategy, and has been used in many diverse applications of combinatorial optimization as an approximate heuristic, for instance in page layout optimization (Hill et al. 2017). More sophisticated variants of this approach can use best-arm identification techniques during each epoch, but fundamentally, this strategy finds an approximately optimal context-insensitive policy.

4.2 Global Bandit from Initial Input

For many real-world pipelines, the modules' characteristics are sensitive to the initial input, meaning that a context-insensitive policy can be very sub-optimal w.r.t. the pipeline loss. This motivates our approach to find a context-adaptive policy using contextual bandit (henceforth CB) algorithms.

A CB algorithm receives a context x_t in each round t, takes an action $a \in A$ and receives a reward r_t. The algorithm learns a policy $\pi : x \mapsto \Delta(A)$ that is context-sensitive and adaptively explores/exploits to maximize $\sum_t r_t$. In our setting x_t is the input example to the pipeline, A is the Cartesian product of all module-specific configurations and the reward is simply the negative of the observed pipeline loss.

In our experiments, we use a simple CB algorithm that uses Boltzmann exploration (see e.g. (Kaelbling, Littman, and Moore 1996)). Concretely, the policy is represented by a parametrized scoring function $s_\theta : {S_1}^{in} \times A \mapsto \mathbb{R}$. The score for each global configuration is computed $s_\theta(x, a)$ and the policy is a softmax distribution of these scores:

$$\pi_\theta(a \mid x) = \frac{\exp(\lambda s_\theta(x, a))}{\sum_{a'} \exp(\lambda s_\theta(x, a'))}, \tag{2}$$

where $\lambda > 0$ is a hyperparameter that governs the trade-off between exploration and exploitation. The score function is typically updated using importance-weighted regression (Bietti, Agarwal, and Langford 2018) (henceforth `IWR`); that is, if we observe a reward r_t after configuring the pipeline with action $a_t \sim \pi_\theta(a \mid x_t)$, then the score function is optimized to minimize $\frac{1}{\pi_\theta(a_t|x_t)}(r_t - s_\theta(x_t, a_t))^2$.

These contextual bandit algorithms can very effectively find context-sensitive policies π and adaptively explore promising configurations. However, by viewing the entire pipeline as one monolithic object with combinatorially many actions, they cannot scale to even moderate-sized pipelines.

4.3 Per-module Bandit: Using Immediate Observations

The contextual bandit approach of Section 4.2 does not scale well with the size of the pipeline, but it does guarantee (under mild assumptions, like an appropriate schedule for λ, see e.g. (Singh et al. 2000)) that we will eventually find the best context-adaptive policy expressible by our scoring function s_θ. It also does not capture the outputs of prior modules in choosing the configuration at a successor, which can be vital such as when a previous module incurs a large latency. Suppose we again relax the goal to instead find an approximately good "locally optimal" policy. Our key insight is to now employ a CB algorithm for each module, so that the algorithm for module j only needs to reason about A_j actions. Moreover, for each module, the metareasoner can use up-to-date information (e.g. latencies introduced by upstream modules) as part of the context for the downstream bandit algorithm.

One can again perform a variant of randomized co-ordinate ascent as in Section 4.1, holding all but one module fixed and running a CB algorithm for that module. This ensures that each bandit algorithm faces a stationary environment and can reliably identify a good context-sensitive policy quickly. However, this can be very data-inefficient; we will next sketch an actor-critic based reinforcement learning algorithm that can apply simultaneous updates to all modules.

Suppose we consider stochastic policies of the form (2) for a module j, but where $x_t^j \in S_j^{in}$ and $a_t^j \in A_j$. A common approach to optimize the policy parameters θ is to directly perform stochastic gradient descent on the average loss, which results in the policy gradient algorithm. Specialized to our setting, an unbiased estimate of the gradient for the parameters θ_j of π_j, that is $\nabla_{\theta_j} J(\theta)$ (recall (1) is given by $L(z_t^M)\nabla_{\theta_j} \log \pi_\theta(a_t^j \mid x_t^j)$ since the loss is only incurred at the end. Typically, policy gradient techniques use an additional trained critic $C(x_t^j)$ as a baseline to reduce the variance of the gradients (Konda and Tsitsiklis 2000). We train the critic to minimize the mean squared error between the observed reward and the predicted reward, $(L(z_t^M) - C(x_t^j))^2$.

5 Experiments

The algorithms discussed in the previous section are tested on two sets of pipelines: a synthetic pipeline with strong context dependence and a real-world perception pipeline. For all our experiments, we use a PyTorch based implementation (Paszke et al. 2017) with RMSProp (Hinton, Srivastava, and Swersky 2012) as the optimizer. For hyperparameter tuning, we perform a grid search over the possible choices. The common hyperparameters for both methods are: (i) Learning rate $\in \{0.0001, 0.0004, 0.001, 0.005\}$, (ii) Minibatch size $\in \{5, 10, 20, 50, 100\}$ and (iii) ℓ_2-weight decay factor $\in \{0.01, 0.05, 0.1, 1\}$. All our plots include 5 different runs with 5 randomly chosen random seeds with standard error regions. The specific details for each algorithm are as follows:

Greedy hill-climbing For finding the greedy step in each iteration, we use a minibatch of 1000 samples per action ($K = A_j * 1000$). The procedure is run until it converges to a fixed assignment. In the plots, we outline this as the non-adaptive baseline with which each method is compared. The final assignment obtained by the procedure is evaluated using Monte Carlo runs with sufficiently large number of samples from the input distribution (synthetic pipeline) or using samples present in a holdout set (face detection pipeline).

Global contextual bandit The policy parameters consist of a single policy that maps the input x_t^1 to a configuration for the entire pipeline, and policy class is a neural network with a single hidden layer. The inverse temperature coefficient for Boltzmann exploration, λ, is considered to be a hyperparameter. We use the `IWR` loss with minibatches to perform updates to the policy.

Per-module contextual bandit The policy function at each module is a single hidden layer neural network with a softmax layer at the end. We use the policy gradient update rule as discussed in Section 4.3. The context for each module is the concatenation of the sequence of actions chosen for previous modules, current latency and the initial input to the system. Additionally, for each module, we implement a critic which predicts the final loss of the pipeline for the given context as described in Section 4.3. The critic is again a single hidden layer neural network with a single output node and is trained using squared loss over the observed and predicted loss. We use the same learning rate for both networks. We use minibatches for training the networks for each module and these are concurrently updated for each minibatch. In addition, we also use entropy regularization weighted by `ent_wt` with the policy gradient loss function (Haarnoja et al. 2018).

For hyperparameter tuning, we choose the setting with the minimum cumulative loss across the input stream.

Method	Hyperparameter choices
Global CB	$\lambda \in \{0.1, 0.3, 1, 5, 10\}$
Per-module CB	`ent_wt` $\in \{0.01, 0.03, 0.1, 0.3, 1\}$

Table 1: Algorithm specific hyperparameter choices

At a high-level, our experiments seek to uncover the importance of adaptivity to the inputs in configuring the pipeline. To capture practical trade-offs, we consider loss functions which combine the latency incurred for an input, along with the accuracy of the final prediction.

5.1 Synthetic Pipelines

We begin with an illustrative synthetic pipeline designed to highlight: (1) benefits of adaptivity to the input over a static assignment, and (2) infeasibility of the global CB approach for even modestly long and representationally simpler pipelines. The structure of the synthetic pipeline with n modules is a linear chain of length n. Each module has two possible actions: 0 and 1 (cheap/expensive action) which incur a latency cost of 0 and 1 respectively. Inputs to the pipeline consist of uniformly sampled binary strings from $\{0, 1\}^n$, with the i_{th} bit encoding the preferred action for module i. If the i_{th} bit is set to 0, both actions give an accurate output and if it is 1, only the expensive action gives an accurate output. If we make an incorrect prediction at module i, then the final prediction at the end of the pipeline is always incorrect. At each episode t, we provide an input to the pipeline by first sampling a random binary string as mentioned above, but then add uniform noise in the interval $[-0.3, 0.3]$ to each entry and this perturbed input constitutes the initial context x_t^1 for the pipeline. The loss function for the final output of the pipeline is $\ell(a) := \frac{4}{n^2}(\text{latency} - n/2)^2 + \text{error}$.

We center the latency term at $n/2$, which is the latency of the optimal policy that routes each input perfectly to the cheapest action that makes the correct prediction for it and the normalization keeps this term in $[0, 1]$. The second term measures the error in the eventual prediction, which requires each module to make an accurate prediction. The value is set to 1 for an incorrect output and 0 otherwise. While the initial input encodes the optimal configuration,suited to global CB, there is further room to adapt. When module i makes an error, then all modules $j > i$ should pick the cheap action.

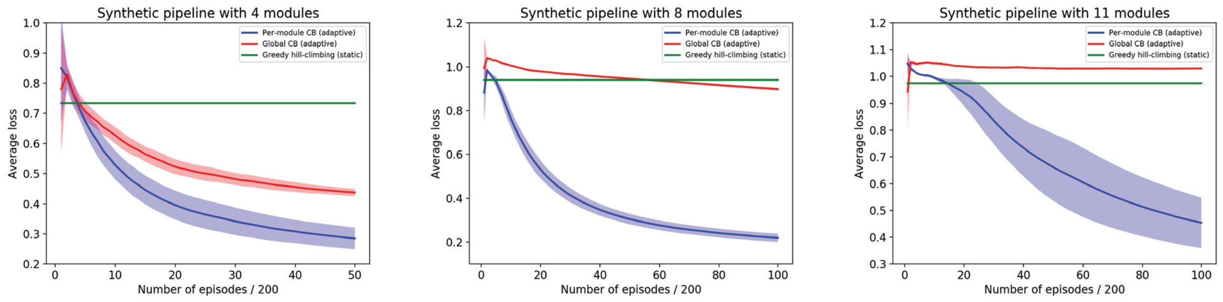

Figure 2: Average loss as a function of the number of examples for the synthetic pipeline. The flat line corresponds to the expected loss of the best constant assignment. The shaded region represents one standard error over 5 runs.

We show results of our algorithms for $n = 4, 8$ and 11. For static assignments, we compute both the solution of the greedy hill climbing strategy and a brute force search over all assignments, which results in similar average losses under the input distribution. The context for each module for per-module CB contains the pipeline's input, a binary string to denote upstream actions and the current latency. We use ReLU activations with the number of hidden layers for each network in our experiments as the average of input dimension and the output dimension. For instance, for global CB, the number of hidden units for $n = 4$ is $h = 10$.

We show the evolution of the average loss as a function of the number of examples for different values of n in Figure 2. Our results show significant gains for being adaptive over the constant assignment baseline in all the plots. For $n = 4$, the total number of assignments is 16 and it can be clearly seen that global CB is effective when compared to the per-module counterpart. However, global CB is slower in convergence than per-module CB. For $n = 8$, the difference between the two is more pronounced as the per-module CB method converges rapidly. For $n = 11$, the total number of assignments for the pipeline is 2048 and global CB completely fails to learn a better adaptive policy. The per-module CB has a slower convergence in this harder case, but still improves upon the best constant assignment extremely quickly.

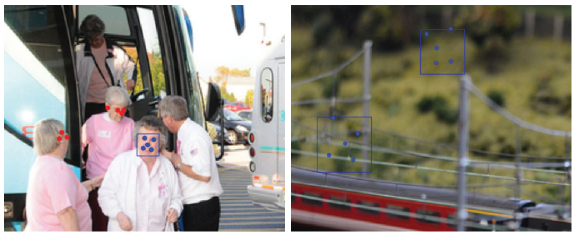

Figure 3: Example face and landmark detections from COCO validation set. (Left) Face detected (blue rectangle) and landmarks detected within the face (blue dots). The red dots represent undetected groundtruth face landmarks by the pipeline. (Right) False face detections and wrong landmarks within the rectangles.

5.2 Face and Landmark Detection

Pipeline and dataset: We use a two-module production-grade real-world perception pipeline service to empirically study the efficacy of our proposed methods (Figure 1). The first module is a face detection module which takes as input an image stream and outputs the location of faces present in the image as a list of bounding box rectangles. This module has four different algorithms for detecting faces. The exact details of the algorithms are proprietary and hence we only have black-box access to them. We benchmarked the latency and accuracy of the algorithms on 2689 images from the validation set of the 2017 keypoint detection task of the open source COCO dataset (Lin et al. 2014). COCO has ground truth annotations of up to 17 visible keypoints per person in an image. We notice that not only do each of the algorithm choices have large variation in latency and accuracy on average when compared to each other, more crucially their latencies and accuracies vary drastically with the number of true faces present in the incoming images, i.e. they are *context dependent*. Specifically, we observe that latency drastically increases with the number of faces present in the image. Figure 4 shows the latencies of all four detection algorithms vs. number of true faces present in the image. Note that different algorithms have different latencies *on average* with Algorithm 0 being the fastest (~ 0.2 seconds) and Algorithm 3 the slowest (~ 2.5 seconds).

The second module is a face landmark detection module which takes as input the original image and the predicted face rectangles output by the face detection module and computes the location of landmarks on the face like nose, eyes, ears, mouth etc. There are three different landmark detector algorithm choices: 5 points, 27 points or 87 points landmark detector. Again we observe in our benchmarking that the landmark detector which outputs 87 points takes the most time at 0.25 ms per image on average vs. 0.17 ms and 0.08 ms per image for the 27 and 8 points algorithms respectively. Since the landmark detectors are applied on each face rectangle detected by the face detector, the computational time required goes up proportional to the number of faces. Figure 3 shows example face detections and landmarks detected on images from the validation sets of the COCO dataset.

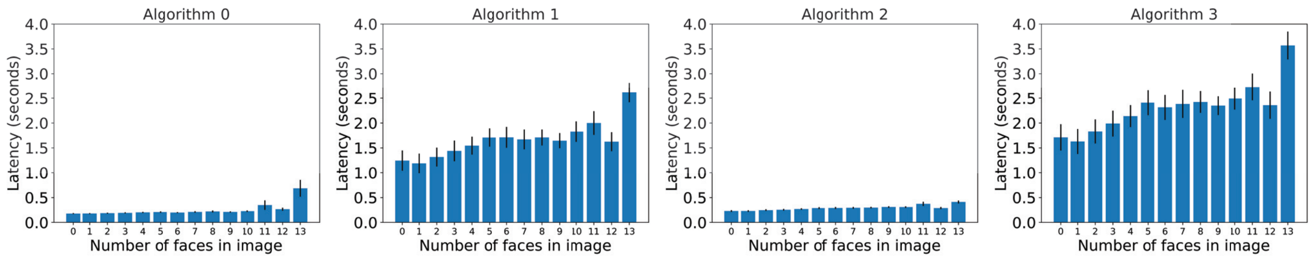

Figure 4: Face detection algorithm choices vs. latency in seconds as a function of true number of faces present in the image. Algorithm 0 and 2 are much faster than Algorithm 1 and 3. All algorithms exhibit increasing latencies as the number of faces goes up in the image.

Accuracy calculation: For evaluating when a prediction by the face detection module is a true/false positive/negative, we closely follow the scheme laid out in the COCO Keypoints evaluation page (Lin et al. 2014). Specifically a rectangle location on the image is considered a true positive if it is within 30 pixels of a ground truth face annotation which is quite conservative as the images we use are all resized to constant size of 1280(W) $\times$ 960(H) pixels. Otherwise, it is marked as a false positive. Ground truth faces which are not "covered" by any of the predicted faces cause an entry in the false negative count. If an image contains no faces and the face detection module also predicts no faces then we count such scenarios as true negatives.

For the face landmark module, we mark a prediction as a true positive if it is within 5 pixels of the ground truth landmark, else a false positive. All landmarks not "covered" by any of the predicted faces are counted as false negatives. Since the COCO keypoint annotations include only 17 keypoint annotations on the entire human body including only 5 face landmarks, we don't penalize predictions of the 27 or 87 landmark detection algorithms which are not within threshold distance of any ground truth landmark as that unfairly counts as false positives (due to lack of ground truth annotations).

Results: The dataset of 2689 images is divided into train and test sets of size 2139 and 550 respectively. For training, we use minibatches randomly sampled from the training set and test curves are plotted using the average loss over the complete test set.[2] We use the embedding from the penultimate layer of ResNet-50 (He et al. 2016) as the contextual representation for each image for both adaptive methods. Thus, the context is a 1000 dimensional real valued vector. For per-module CB, the first module's policy network gets the embedding as input whereas the second one gets additional concatenated values of number of faces detected by module 1 and its latency. All networks here have a hidden layer with 256 units with ReLU activations. For evaluating the final loss function of the pipeline, we consider three metrics:

[2]As the number of episodes is much larger than the size of our data set, algorithms can overfit to the data set. So we evaluate the average test performance on held out examples as a proxy.

- **Pure latency:** Squared loss between the pipeline's latency and a threshold t_0: $\ell(a) := (\text{latency} - t_0)^2$. For fair treatment across all inputs, just minimizing latency might not always be the actual goal and will result in a trivial policy which learns to pick the least expensive option for all inputs. Our metric incentivizes the system to stay around the *threshold* and penalizes under-utilization.
- **Latency and accuracy:** In addition to the squared distance, we now consider the false negative rate (FNR) of the pipeline for the landmarks detected in each image. Since false negative rate is always in $[0, 1]$, it is robust to different number of landmarks in different images as well as different number of predicted landmarks(5, 27 and 87), unlike a direct classification error. In this case, $\ell(a) := (\text{latency} - t_0)^2 + \text{FNR}$.
- **Latency, accuracy and false detection penalty:** For the face detection module, in many cases there are non-zero false positives. This further increases the number of false positive landmarks for those cases and therefore we add another penalty of the false discovery rate for face detection.

In our experiments, we choose a value of $t_0 = 1.3$ and $t_0 = 1.7$ for all three loss functions for the pipeline. Note that, if one tries to optimize total latency of the pipeline, then the non-adaptive solution of choosing the cheapest action for both modules works well. Therefore, we choose the bell shaped squared loss for latency which reflects the specification of aiming for a target latency. We show the test set performance in Figure 5. For $t_0 = 1.3$, we see a more interesting contrast across the two methods, whereas, for $t_0 = 1.7$, both Global CB and Per-module CB exhibit similar performance. Per-module CB and global CB show improvement for all loss functions against the constant assignment baseline found by greedy hill climbing. The numbers in Table 2 show the context-dependency of the pipeline. The benefits of algorithms which are able to effectively utilize context (Global CB and Per-module CB) is really highlighted in the parts of the dataset with more faces. As the number of faces in an image increases, the percentage gain increases as well. The observed gains of approximately 15, 22 and 24 percent in respecting the utility function are arguably significant for sensitive mission-critical applications.

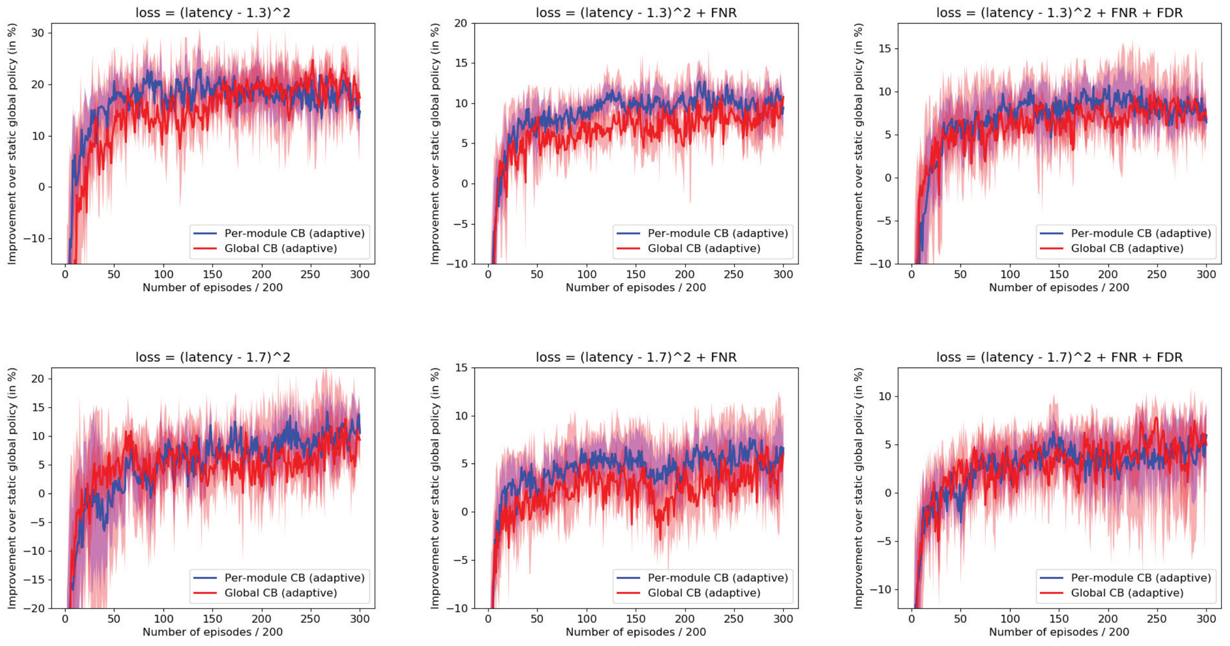

Figure 5: Test performance curves for the Face Detection and Landmark pipeline with $t_0 = 1.3$ (top) and $t_0 = 1.7$ (bottom). The Y-axis is the performance percentage improvement over static global policy after every 200 episodes of learning on held-out examples. The plots use a latency-based loss (left), latency and FNR (middle) and latency, FNR and FDR (right). The adaptive approaches significantly improve over the best fixed configuration in all cases. Shading represents std. error across 5 runs.

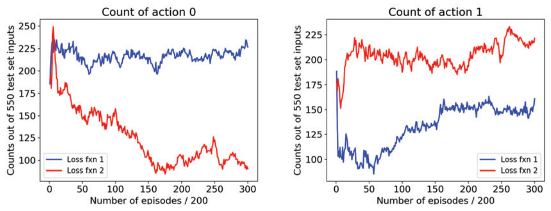

Figure 6: Action counts in module 2 for per-module CB

#Faces	#Images	Global CB	Per-module CB
$\geq$3	124	11.82%	15.23%
$\geq$4	83	18.58%	22.51%
$\geq$5	63	23.04%	24.12%

Table 2: Performance percentage improvement over static global policy broken down by the number of true faces in the image for latency and accuracy loss with $t_0 = 1.3$.

Although these two methods are hard to distinguish on average, we think this is due to the small length of the pipeline and the intermediate context for the second module's policy not being very informative. In order to show that the adaptivity to the final loss function influences the chosen actions, a comparison between the action counts for two different loss functions can be seen in Figure 6.

6 Discussion and Conclusion

We observe that contextual optimization of software pipelines can provide drastic improvements in the average performance of the pipeline for any loss function. Our experiments show that for small pipelines, both global CB and per-module CB can give potential improvement over a static assignment. However, these experiments should only be considered as a controlled study of the power of contextual optimization and there are additional caveats which we defer for future work:

Computational overhead: In addition to the pipeline's latency, any metareasoning module will add to the cost. In our experiments, the total time for inference and updates is less than 5-7 `ms` per input which is orders of magnitude less the the pipeline's latency. Moreover, making the pipeline configurable in real-time might induce further communication/data re-configuration costs. We focus on the potential improvements from adaptivity in this paper and leave the engineering constraints for future work.

Non-stationarity during learning: For the per-module CB algorithm, the input given to each network is ideally the input for the corresponding module. Changes in the configuration can vary the distribution of the inputs to these modules drastically. The pipelines in our experiments do not showcase this issue. We ignore this aspect in our current exposition and leave a more involved study to future work.

In summary, we presented the use of reinforcement learning to perform real-time control of the configuration of a modular system for maximizing a system's overall utility. We employed contextual bandits with a holistic representation and showed significant improvement with use of metareasoning in our experiments. Future directions include studies of scaling up the mechanisms we have presented to more general systems of interacting modules and the use of different forms of contextual signals and their analyses.

Acknowledgements

Part of this work was done while AM was at Microsoft Research. AM acknowledges the concurrent support by a grant from the Open Philanthropy Project to the Center for Human-Compatible AI and NSF grant CAREER IIS-1452099.

References

Bietti, A.; Agarwal, A.; and Langford, J. 2018. A contextual bandit bake-off. *arXiv preprint arXiv:1802.04064*.

Bradley, D. M. 2010. Learning in modular systems. Technical report, Carnegie-Mellon University, Pittsburgh PA.

Delimitrou, C., and Kozyrakis, C. 2013. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, 77–88. ACM.

Delimitrou, C., and Kozyrakis, C. 2014. Quasar: resource-efficient and qos-aware cluster management. In *ACM SIGARCH Computer Architecture News*, volume 42, 127–144. ACM.

Demirci, M. 2015. A survey of machine learning applications for energy-efficient resource management in cloud computing environments. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 1185–1190. IEEE.

Fedorova, A.; Vengerov, D.; and Doucette, D. 2007. Operating system scheduling on heterogeneous core systems. In *Proceedings of the Workshop on Operating System Support for Heterogeneous Multicore Architectures*.

Gao, J. 2014. Machine learning applications for data center optimization.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 1856–1865.

Hanus, D. 2013. *Smart scheduling: optimizing Tilera's process scheduling via reinforcement learning*. Ph.D. Dissertation, Massachusetts Institute of Technology.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Hill, D. N.; Nassif, H.; Liu, Y.; Iyer, A.; and Vishwanathan, S. 2017. An efficient bandit algorithm for realtime multivariate optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, 1813–1821.

Hinton, G.; Srivastava, N.; and Swersky, K. 2012. Neural networks for machine learning, lecture 6a: Overview of mini-batch gradient descent. URL:https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Horvitz, E., and Lengyel, J. 1997. Perception, attention, and resources: A decision-theoretic approach to graphics rendering. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, 238–249. Morgan Kaufmann Publishers Inc.

Horvitz, E. 2001. Principles and applications of continual computation. *Artificial Intelligence* 126(1-2):159–196.

III, H. D.; Langford, J.; Mineiro, P.; and Sharaf, A. 2018. Residual loss prediction: Reinforcement learning with no incremental feedback. In *International Conference on Learning Representations*.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4:237–285.

Konda, V. R., and Tsitsiklis, J. N. 2000. Actor-critic algorithms. In *Advances in neural information processing systems*, 1008–1014.

Li, Y. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

Lin, T.-Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; and Zitnick, C. L. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*, 740–755. Springer.

Lorido-Botran, T.; Miguel-Alonso, J.; and Lozano, J. A. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing* 12(4):559–592.

Mao, H.; Alizadeh, M.; Menache, I.; and Kandula, S. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 50–56. ACM.

Memeti, S.; Pllana, S.; Binotto, A.; Kołodziej, J.; and Brandic, I. 2018. Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review. *Computing* 1–44.

Mirhoseini, A.; Pham, H.; Le, Q. V.; Steiner, B.; Larsen, R.; Zhou, Y.; Kumar, N.; Norouzi, M.; Bengio, S.; and Dean, J. 2017. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 2430–2439. JMLR. org.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in pytorch. In *NIPS Autodiff Workshop*.

Raman, K.; Swaminathan, A.; Gehrke, J.; and Joachims, T. 2013. Beyond myopic inference in big data pipelines. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 86–94.

Singh, S.; Jaakkola, T.; Littman, M. L.; and Szepesvári, C. 2000. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning* 38(3):287–308.

Xu, C.-Z.; Rao, J.; and Bu, X. 2012. Url: A unified reinforcement learning approach for autonomic cloud management. *Journal of Parallel and Distributed Computing* 72(2):95–105.