FirePlace: Placing Firecracker Virtual Machines with Hindsight Imitation

Bharathan Balaji [1] **Christopher Kakovitch** [1] **Balakrishnan (Murali) Narayanaswamy** [1]

Abstract

Virtual machines (VM) form the foundation of modern cloud computing as they help logically abstract per-user compute from shared physical infrastructure. Users of these services require VMs of varying sizes and configurations, which the provider places on a set of physical machines (PMs). VMs on the same PM share memory and CPU resources so a bad placement directly impacts the quality of user experience. We consider the placement of Firecracker VMs (a form of Micro-VMs or μVMs) - lightweight VMs that are typically used for short lived tasks. Our objective is to place each VM as it arrives, so that the peak to average ratio of resource usage across PMs is minimized. Placement is challenging as we need to consider resource use in multiple dimensions, such as CPU and memory, and because resource use changes over time. Past approaches to similar problems have suggested that one could forecast VM resource use for placement. We see that in our production traffic, μVM resource use is spiky and short lived, and that forecasting algorithms are not useful. We evaluate Reinforcement Learning (RL) approaches for this problem, but find that off-the-shelf RL algorithms are not always performant. We present a forecasting free algorithm, called FirePlace, that learns the placement decision using a variant of hindsight optimization, which we call hindsight imitation. We evaluate our approach using a production traffic trace of μVM usage from AWS Lambda. FirePlace improves upon baseline algorithms by 10% on a production data trace of 100K μVMs.

1 Introduction

Virtual Machines (VMs) (Barham et al., 2003) are an essential technology in modern computing and form the core of many cloud services. A VM wraps the functionality of a physical compute system and presents it to the user, while sandboxing her from other users who may share the same Physical Machine (PM). VMs are useful because they allow providers to allocate resources efficiently by fitting many VMs on a single PM, while giving the functionality of a separate machine to the end user. VMs on the same physical PM share memory, compute and network resources. A bad packing of VMs would impact the stability, latency and throughput of computations and increase costs.

Algorithms for packing VMs efficiently into PMs have been studied for over a decade (Bobroff et al., 2007; Xiao et al., 2012; Meng et al., 2010b). Firecracker VMs are a recent innovation - a form of lightweight VMs that have fast startup times and can be packed with high density, while still providing strong security (Madhavapeddy et al., 2013; Agache et al., 2020). Firecracker VMs are an instantiation of MicroVMs, and we refer to them as μVMs for brevity. They are used to provide serverless services such as Function as a Service (Kratzke, 2018) with low overheads. We consider an online placement setting common in cloud computing, where μVMs are created and deleted based on exogenous demand. The objective is to place the VMs such that the total number of PMs used is minimized, while ensuring that resource use in any of the PMs does not exceed limits. A good packing directly translates to increased availability, reduced operational costs and energy savings (Fan et al., 2007).

Our VM placement problem is similar to the problem of online bin packing (Song et al., 2013; Gupta & Radovanovic, 2012). However, in our setting, not only do we need to consider multiple resource dimensions such as compute and memory, but also *how resource use changes over time*. Since vector bin packing itself is an NP complete combinatorial problem, and APX hard for 2 or more dimensions (Christensen et al., 2016), our problem in its general form is intractable. Related prior work has proposed forecasting the resources that will be used by each VM, followed by well-known heuristics such as Best-Fit or genetic algorithms to decide placement (Chen et al., 2018). In our production dataset, we find that μVM are short lived and their use is spiky, and hence, difficult to forecast. We can pack hundreds

[1]Amazon. Correspondence to: Bharathan Balaji <bhabalaj@amazon.com>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

of μVMs in a physical machine (PM) as each μVM resource use is small in comparison to the PM capacity. Therefore, forecasting solutions become untenable as the error in the prediction of each VM accumulates and leads to poor placement. In particular, the best p90 forecast for μVM CPU[1] use over its entire lifetime is 0.

Given the difficulty of forecasting, the best that any algorithm could realistically do is hedge placement decisions based on 'typical' compute and memory use of a new μVM when compared with the historical compute and memory use of the PM. This hedged decision rule could be state dependent. We formulate the problem as a Markov Decision Process (MDP) (Puterman, 1990), where the agent places a μVM at each time step. The MDP still suffers from the curse of dimensionality because of the large state and action spaces - in particular, the large number of PMs available at each time instant. Hence, we build on the *power of two choices* (Mitzenmacher, 2001) to reduce the action space. We find that off-the-shelf reinforcement learning (RL) algorithms do not perform well due to long horizon and noisy state space as μVM resource use change over time. We propose FirePlace, a hindsight imitation learning algorithm for μVM placement. We leverage historical data to identify placement decisions that could have been made using hindsight of future μVM compute and memory use, similar to hindsight optimization (Chong et al., 2000). We then cast placement as a supervised learning problem with the hindsight based decision as the label. We show that FirePlace does not require per μVM forecasting, outperforms off-the-shelf model-free deep RL algorithms, runs fast enough to deploy to a latency sensitive large scale production service, and generates a learned model that generalizes to unseen data.

2 μVM Placement in our fleet

Our problem formulation is informed by AWS Lambda[2], which provides function execution as a managed service. Users provide the function they want to execute, written in their preferred programming language, configure limits on memory use in 64 MB increments from 128 MB up to 3000 MB, and configure execution time maximums in 1 second increments up to 15 minutes. They can then execute the function as often as they like, using a variety of different event triggers. Users are alleviated from the burden of provisioning infrastructure and save on costs as they are billed only for the function execution time. The service provisions the compute resources, creates μVMs, installs the required dependencies and executes the function for the user. Users can use these functions for many different purposes, from real-time data processing to serving web requests. Our aim is to reduce the operational cost of the service.

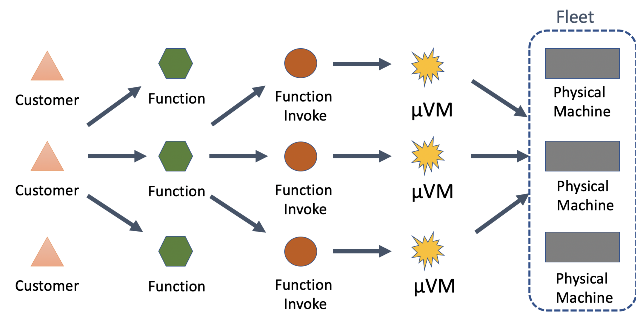

Figure 1. An overview of our model system that uses μVMs for invoke executions. Our objective is to pack the μVMs such that the demand for resources is satisfied and the size of the fleet is minimized.

We now describe the system model we use to derive our problem formulation in the paper. The description is inspired by the real production system of AWS Lambda, but redacts sensitive proprietary information.

In our model system, the infrastructure consists of a fleet of data center servers, which we refer to as *physical machines (PMs)*. Once a user configures and calls the function, we create a μVM for that function in one of the PMs, and execute it. The μVM is not destroyed immediately, in case the user executes again immediately. The μVM is eventually deleted if it remains idle for a threshold amount of time. Each function call is called an *invoke*, and the user can execute the same function concurrently. Hence, a single function may have multiple μVMs. Each PM can host and execute multiple μVMs at a time. Figure 1 illustrates our model system.

Functions to μVMs have a one-to-many relationship, and μVMs are created and used for only one function. Two different functions cannot share execution on the same μVM. A μVM can only accommodate one invoke at a time. While the μVM is processing an invoke, it is said to be active. Once function execution ends for that invoke, the μVM enters an idle state and can be kept around to accommodate additional invokes if and when they arrive. Invoking on an idle μVM results in lower user latency compared to creating a new μVM.

Each decision in infrastructure management and function request routing impacts the operational cost and user experience. If an idle μVM for a particular function is available when an invoke arrives, then the latency of execution is reduced. μVMs use memory even when they are not executing a function. Hence, idle μVMs waste resources. If μVMs are packed in the PMs tightly, the number of PMs in the fleet

[1]CPU is used here to represent compute resources as measured by the number of cores on the physical machine.

[2]AWS Lambda - https://aws.amazon.com/lambda/

is reduced. However, we need to ensure the μVMs are not packed too tightly to ensure they can execute without hitting PM resource limits, or PM performance limits which would negatively impact the user observed latency or availability. We also need to ensure the PMs have enough resources to create new μVMs as they are needed. Proactive deletion of μVMs can reduce memory use. However, aggressive deletion may create churn with reactive creation of μVMs, and increase execution latency. Each of these decisions can be optimized to reduce cost. We focus on the placement of μVMs in PMs to ensure tight packing, and as a result, reduce the fleet cost.

3 Problem Setting

We formulate our problem by modeling the characteristics of a typical cloud system. μVMs are created by an external service based on user demand, and at each time step we receive a μVM to place. Our objective is to identify a PM in the fleet on which to place the μVM so that the Peak to Average Ratio (PAR) of resource use in the fleet is minimized while ensuring the resource use in any of the PMs does not exceed capacity *in the future* as the compute and memory use of the μVMs change. We can reduce our fleet size if the peak use of the fleet falls below a threshold. If the total number of active VMs on a PM require more compute or memory than the PM has available, then user experience is degraded. In practice, we observed that our PMs are not bottlenecked on resources such as network bandwidth. Therefore, we scoped the problem to only track the CPU and memory use of the PM. The placement decision needs to be made quickly (∼20ms) and with reasonable throughput (∼5 placements/s). Each fleet can consist of hundreds or thousands of PMs, and it is impractical to require the updated state of all PMs to make the placement decision. The algorithms we tried, including RL and baselines, did not scale well with large state and action spaces. Hence, we sample $\mathcal{K}$ PMs at a time and identify the PM to place. Our design is motivated by the power of two choices (Mitzenmacher, 2001), which shows that making an optimal choice out of two randomly sampled PMs is exponentially better than a random choice. This limits our action space to size $k = |\mathcal{K}|$, rather than the total number of active PMs.

μVMs start executing user invokes sometime after they are created, consuming memory and compute when they do so. An external service deletes the μVMs if it is idle for more than a specific period of time. We consider a fleet with a single type of PM with fixed compute and memory capacity.

A PM's resource use is simply the sum of resources consumed by the μVMs in the PM. We compared the sum of resource use by the μVMs to the total resource use of the PM measured in our system. We find the difference between the estimated and measured PM use can be explained with a constant bias term using regression analysis. The constant term does not impact our agent decisions, therefore we ignore overheads in our formulation.

μVMs arrive online, and the order of arrival cannot be changed. We cannot migrate μVMs once they have been placed. μVM resource use is unknown before placement and *changes over time* as invokes are executed. The memory use of a PM increases monotonically over time until deletion, since μVMs do not release memory. The μVM uses compute only when it is executed, thus the CPU use of the μVM is zero when it is idle (we ignore idle CPU overhead), and spikes up when it executes invokes.

Each μVM is represented by a timeseries of its resource use. Let c_t^v and m_t^v denote the CPU and memory use of μVM v at time t. Each PM consists of a collection of μVMs. Let $C_t^p = \sum_{v \in \boldsymbol{V}^p} c_t^v$ and $M_t^p = \sum_{v \in \boldsymbol{V}^p} m_t^v$ denote the CPU and memory use of PM $p \in \boldsymbol{P}$ at time t, where $\boldsymbol{V}^p$ is the set of μVMs in the PM and $\boldsymbol{P}$ is the set of PMs in the fleet. Let $\boldsymbol{c}^v = c_0^v, .., c_T^v$ and $\boldsymbol{m}^v = m_0^v, .., m_T^v$ denote the CPU and memory use timeseries of length T for μVM v. Let $\boldsymbol{C}^p = \sum_{v \in \boldsymbol{V}^i} \boldsymbol{c}^v$ and $\boldsymbol{M}^p = \sum_{v \in \boldsymbol{V}^p} \boldsymbol{m}^v$ denote the PM CPU and memory timeseries respectively. The placement algorithm places one μVM at a time in one of $\mathcal{K}$ PMs randomly sampled from the fleet.

We assume each PM in the fleet can be restarted any point with a low probability to ensure our formulation is robust to such changes in practice. In the simulation, each step corresponds to a VM placement. At each step, we decide to restart a PM with a low probability. When such a restart occurs, we randomly pick one of the PMs, and set its CPU and Memory use to zero. In practice, this corresponds to a PM that is no longer available for placement, and the VMs in that PM are deleted once they are idle for a period of time. A new PM is added to the fleet as replacement.

3.1 Compute Based Packing

Figure 2 shows the memory and compute characteristics of an example μVM. CPU use predictability is much lower than memory predictability, and for most algorithms compute violations are much more common than memory violations. As a result, we also report results for a version where we only consider compute use during the placement. We formulate the problem as an MDP, where the agent places the μVMs across an episode. We introduce $\tau \in [0, T]$ as the wall clock time at which μVM v at time step t is placed in the PM. The input to the agent is the state: $s(C_t^p, c_t^v)$ for $p \in \mathcal{K}_t, t \in [0, \tau]$, and the agent takes action by picking one of $\mathcal{K}_t$ PMs. If the agent picks a PM that is too full to fit the μVM - which we can tell in simulation by considering future memory and compute use - we repeatedly pick another PM in the fleet at random, and try to place the μVM there, until we find a PM that can hold the μVM. The agent receives

a penalty, since bad placements degrade performance. The episode terminates when all μVMs are placed or if all the PMs in the fleet are full. We define the reward function at each time step t as:

$$R_t = \frac{\frac{1}{|\boldsymbol{P}|}\sum_{p\in\boldsymbol{P}} \max_{t\in\tau} C_t^p}{\max_{p\in\boldsymbol{P},t\in\tau} \boldsymbol{C}^p} - W_t \tag{1}$$

where $|\boldsymbol{P}|$ is the number of PMs in the fleet. $W_t = 1$ if the PM picked is too full to place this μVM, and $W_t = 0$ otherwise. PM resource use is represented by its maximum CPU use, as the maximum use should not exceed capacity of the PM. We consider CPU and memory use smoothed over a minute in our experiments. Therefore, momentary spikes due to OS scheduling bottlenecks are not considered as violations.

The first term in Equation 1, encourages the agent to increase the mean CPU use of the fleet compared to the max use (PAR) which encourages the agent to pack μVMs tightly yet maintain the same CPU use across PMs. This choice of metric represents our desire to be efficient, yet robust to e.g. single instance failures, at any time. The fleet is sized so that there are adequate resources for peak traffic. An algorithm that is efficient in μVM packing will ensure that the peak resource use is minimized, allowing for reduction in fleet size. The PAR metric normalizes the peak use at each decision time. The W_t parameter accounts for egregious violations in resource use.

3.2 Compute and Memory Based Packing

The above formulation can be extended to include memory use. The state includes both CPU and memory use: $s(C_t^p, M_t^p, c_t^v, m_t^v)|p \in \mathcal{K}_t, t \in [0, \tau]$. The reward function is modified as:

$$R_t = \frac{\frac{1}{|\boldsymbol{P}|}\sum_{p\in\boldsymbol{P}} \max_{t\in\tau} C_t^p}{\max_{p\in\boldsymbol{P},t\in\tau} \boldsymbol{C}^p} + \frac{\frac{1}{|\boldsymbol{P}|}\sum_{p\in\boldsymbol{P}} \max_{t} M_t^p}{\max_{p\in\boldsymbol{P},t\in\tau} \boldsymbol{M}^p} - W_t \tag{2}$$

Our formulation can be extended to heterogeneous fleets, where the PMs have different amount of compute and memory resources, as we use normalized measures of CPU and memory in our metrics. However, we focus on homogeneous fleet setting in this paper.

A simple baseline algorithm is to pick a PM uniformly at random, and try to place the μVM. An ideal algorithm will pick the PM in the fleet that minimizes the overall PAR *in the long term*. While we only have $|\mathcal{K}| \ll |\boldsymbol{I}|$ PMs from the fleet to choose from, prior work has shown that if we pick the best of $\mathcal{K}$ PMs, we can perform much better than random (Richa et al., 2001).

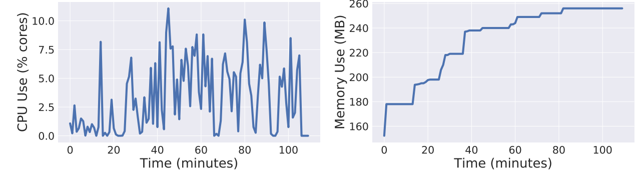

Figure 2. CPU and memory use of a long lived μVM in our dataset. Compute is used only when the μVM is executing invokes and memory use is the maximum of memory used in μVM history.

4 DATA CHARACTERISTICS

We collect a dataset with ~200K μVMs across 24 hours of use from AWS Lambda, and analyze its characteristics. Figure 2 shows the CPU and memory use of a particularly extreme μVM in our dataset, across the 2 hours it was active. Data is recorded every minute and consists of μVM CPU and memory use. The CPU use spikes up whenever invokes are executed. Most μVMs are relatively small (median values – memory: ~100MB, CPU: ~15-20% of a cpu core, execution time: 0.2-0.6s, inter arrival time: 50-100s) but a small percentage of μVMs have extreme values (99th percentile (p99) values – memory $>$ 1GB, CPU $>$ 150%, execution time $>$ 90s, inter arrival time: $>$10 min). The median life of a μVM is ~15 minutes and the p99 value is $>$ 2 hours. The memory use grows gradually as invokes take up more memory and assigned memory is not reclaimed until μVM is deleted. At the time of the placement decision, these detailed μVM characteristics are unknown, since each μVM is unique from the perspective of the placement algorithm.

To make good placement decisions, we may need to forecast how μVMs use resources in aggregate. In particular, we need to place μVMs such that the aggregate use does not exceed the capacity of the PM. In case of memory, an individual μVM use has low variance. Hence, aggregate memory use can be approximated as sum of individual μVM memory use. However, aggregate CPU use varies depending on when invokes are executed and how the invokes are interleaved across μVMs. Hence, estimating aggregate CPU use is challenging. Aggregate resource use also depends on when the μVMs are deleted.

5 METHODS

5.1 Hindsight Placement

One approach to the problem could be to split it into two parts: forecasting and optimization (Meng et al., 2010a; Chen et al., 2016; 2018). We could forecast individual μVM use and estimate the aggregate PM use given the μVMs in it. We then need to optimize the placement of μVMs to minimize PAR. In our approach, we instead start with the optimization problem, assuming we have perfect forecasts

available using real μVM use data from our historical data. Given this future knowledge, we can pack μVMs to greedily maximize the reward in Equations 1 or 2 at each step. We call this algorithm Hindsight based packing.

Hindsight based placement is still greedy, even though it uses perfect information, since we do not attempt to account for future μVM arrivals. Still, a well trained placement agent could learn to hedge against the 'typical' distribution of future arrivals, and predict future resource requirements of already placed VMs.

For the hindsight CPU packing problem, we assume we have access to the entire future timeseries of μVM use and PM use, together with the μVMs already placed in it: $(C_t^p, c_t^v) | p \in \mathcal{K}, t \in [0, T]$. We pick the PM that minimizes the maximum CPU use if the μVM was placed in it:

$$B_k^{cpu} = \max_{t \in T}(c_t^v + C_t^k) \tag{3}$$

$$A = \arg\min_{k \in \boldsymbol{K}} B_k^{cpu} \tag{4}$$

where A is the PM picked by the algorithm. We call this a *Hindsight* algorithm as it utilizes future information unknown in production. For 2-dimensional CPU/memory packing (or, 2D packing for brevity), we pick the PM that minimizes the L^2-norm of CPU and memory use if the μVM were placed in that PM:

$$B_k = \left\| \left(\max_{t \in T}(c_t^v + C_t^k), \max_{t \in T}(m_t^v + M_t^k) \right) \right\|^2 \tag{5}$$

This choice of metric is somewhat arbitrary, and comes from domain experience, not first principles.

In full generality, hindsight optimal placement can be treated as a combinatorial optimization based bin packing problem and we can use classic techniques like dynamic programming. However, the high dimensional state space and the large number of μVMs to be placed make these approaches infeasible, even offline. Online bin packing is a much easier problem when all the items are of the same size (Shen et al., 2017). Since our μVM resource use is small when compared to PM capacity, we can justify use of greedy hindsight techniques. We thus focus on the time varying nature of resource use, rather than the combinatorial optimization aspects.

5.2 Simple Baselines

We present a *Baseline* algorithm that simply uses the mean values to represent the μVM and PM CPU/memory use. This algorithm picks the the PM that minimizes the following for CPU packing:

$$B_k^{cpu} = \frac{1}{T} \sum_{t=0}^{T} c_t^v + \frac{1}{T} \sum_{t=0}^{T} C_t^k \tag{6}$$

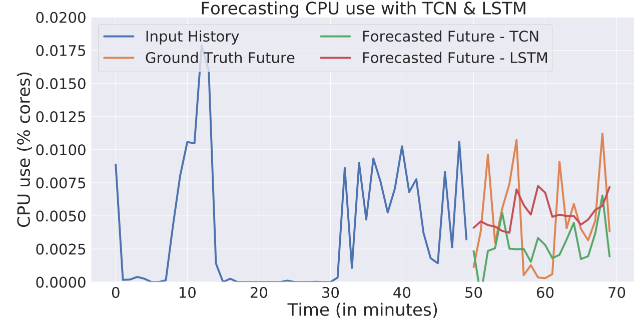

Figure 3. An example of forecasting CPU use with TCN and LSTM for 20 time steps in future given 50 time steps of history. We trained both neural networks for 1000 epochs with a batch size of 10 in Tensorflow Keras. Both LSTM and TCN consisted of 1 layer with 64 units each.

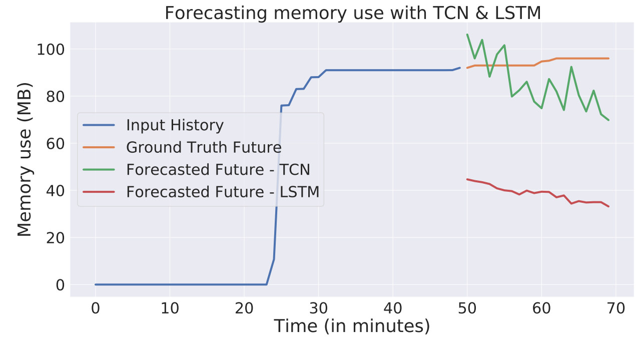

Figure 4. An example of forecasting memory use with TCN and LSTM for 20 time steps in future given 50 time steps of history. We trained both neural networks for 1000 epochs with a batch size of 10 in Tensorflow Keras. Both LSTM and TCN consisted of 1 layer with 64 units each.

For 2D packing, we use the L^2-norm to represent both cpu and memory use:

$$B_k^{mem} = \frac{1}{T} \sum_{t=0}^{T} m_t^v + \frac{1}{T} \sum_{t=0}^{T} M_t^k \tag{7}$$

$$B_k = \| (B_k^{cpu}, B_k^{mem}) \|^2 \tag{8}$$

In addition to the Baseline algorithm, we also present results of randomly picking the PMs as a comparison.

5.3 Forecasting

One approach to this problem is to featurize the inputs available at decision time, forecast the VM use, and use the same optimizations in Equations 3-5. For the optimization, we could forecast the μVM use based on historical μVM use, e.g. using other μVMs that belong to the same user, and forecast the PM use by forecasting the resource use by individual μVMs in that PM (Chen et al., 2016; 2018).

However, we found that there are multiple challenges to this forecasting based approach. μVM use, especially CPU use, is inherently difficult to predict as it depends on user demand at the invoke level and how invokes are routed to a μVM when multiple μVMs per user exist at a time. We need to forecast the μVM use for its entire lifetime of up to $\sim$2 hours. We evaluated LSTMs (Hochreiter & Schmidhuber, 1997) and TCNs (Bai et al., 2018) for forecasting of μVM resource use. We split the μVM timeseries to feed 50 timesteps of historical data and predict 20 timesteps of future use, where each timestep is a minute. Both methods failed to learn predict CPU and memory use on test data for even such a short horizon. As we add individual forecasts for hundreds of μVMs in a PM, the uncertainty in forecasts increases further. While it is possible to mitigate some of these uncertainties using techniques like hierarchical forecasting (Hyndman et al., 2011) and multi-horizon recurrent forecasting (Wen et al., 2017), we have a tight budget of 20ms to make placement decisions, and using sophisticated forecasting approaches easily exceeded the budget. As a result, we focus on forecasting free algorithms for placement.

5.4 Reinforcement Learning

Having already defined the problem as an MDP, reinforcement learning (RL) algorithms are a natural fit for the problem. In our case, the states are quantiles of CPU and memory over different windows, for each of the $|\mathcal{K}|$ PMs and the specific μVM we are placing in that time step. We have $|\mathcal{K}|$ actions corresponding to each of the PMs where we can place the μVM. We evaluate off-the-shelf deep RL algorithms implemented in the Ray RLlib library (Liang et al., 2017). We use the PPO (Schulman et al., 2017) algorithm, and experimented with a number of hyper-parameters and discuss results of the best settings we could find. As we will see in the results (Figure 5), RL is sometimes able to match or even slightly outperform the Hindsight algorithm we described in the previous section. But for many settings it is much worse.

5.5 FirePlace: Hindsight Imitation Learning

We propose a form of imitation learning - a strategy that directly learns to mimic the actions of the Hindsight algorithm, using it as a teacher (Hussein et al., 2017; Pomerleau, 1989). As we have historical data from our production workloads, we can compute hindsight based decisions at each time step and create a dataset for training. Use of hindsight imitation learning circumvents the issues associated with forecasting, as we directly learn a relationship between input features and output decision that hedges placements. The idea is similar to hindsight optimization proposed by Chong et al. for tabular problems (Chong et al., 2000).

The input to the imitation model is features derived from the state and the output is the action. We found that the choice of features, model and dataset is important to avoid overfitting, even with large amounts of data, given the variance in state and action spaces. We use a random forest classifier (Liaw et al., 2002) and support vector machines (Suykens & Vandewalle, 1999) as our model, whichever gives us the the best classification accuracy on our validation data. Simple 2 layer neural networks (Haykin, 1994) did not perform well in our datasets. We use quantiles of PM resource use over different windows as our features. Specifically, we use the p10, p25, p50, p75, p100 and mean values for CPU use, and just the p100 and mean values for memory use because the memory use is monotonic till deletion. We split the μVMs from historical production data to two partitions, use one partition to train our agent and the other to test. We simulate the online μVM placement in a fleet with our train partition, and use the features computed from state as well as the Hindsight algorithm actions to create our training dataset for imitation learning. We refer to the learned model as *FirePlace*.

We list the hyper-parameters we use for each method in Appendix A.

6 RESULTS

We use multiple datasets from a single region: one with 20,000 μVMs and another with 100,000 μVMs respectively. We refer to them as the 20K and 100K datasets. We set $|\mathcal{K}| = 2$, i.e. the algorithm picks from 2 randomly sampled PMs in the fleet and places the μVM on one of these. The performance of the algorithm increases slightly with increase in $|\mathcal{K}|$ (Mitzenmacher, 2001), but $|\mathcal{K}| = 2$ gives us a large improvement over random. We use PMs with 4 CPU cores and 16GB memory in the 20K dataset and use 8 CPU cores and 64GB memory in the 100K dataset. As the number of PMs in the fleet decreases, the placement becomes harder until all PMs very quickly become too full to accommodate μVMs. We vary the number of PMs in our experiments to show the impact of fleet size on the performance of placement algorithms. We run each experiment 10 times, and report mean rewards with error bars.

We assign alternate μVMs to two partitions - this makes the training data representative, but also well separated from test data. We use the train partition to create our dataset with the Hindsight algorithm. We use 75% of the dataset for training and 25% for validation.

The results in Figure 5 show that the best RL algorithm sometimes performed as well as or even slightly better than the Hindsight algorithm, but usually performed worse. Figure 6 shows the training progress of the RL algorithms for the 2D packing case in 20K and 100K datasets. While RL

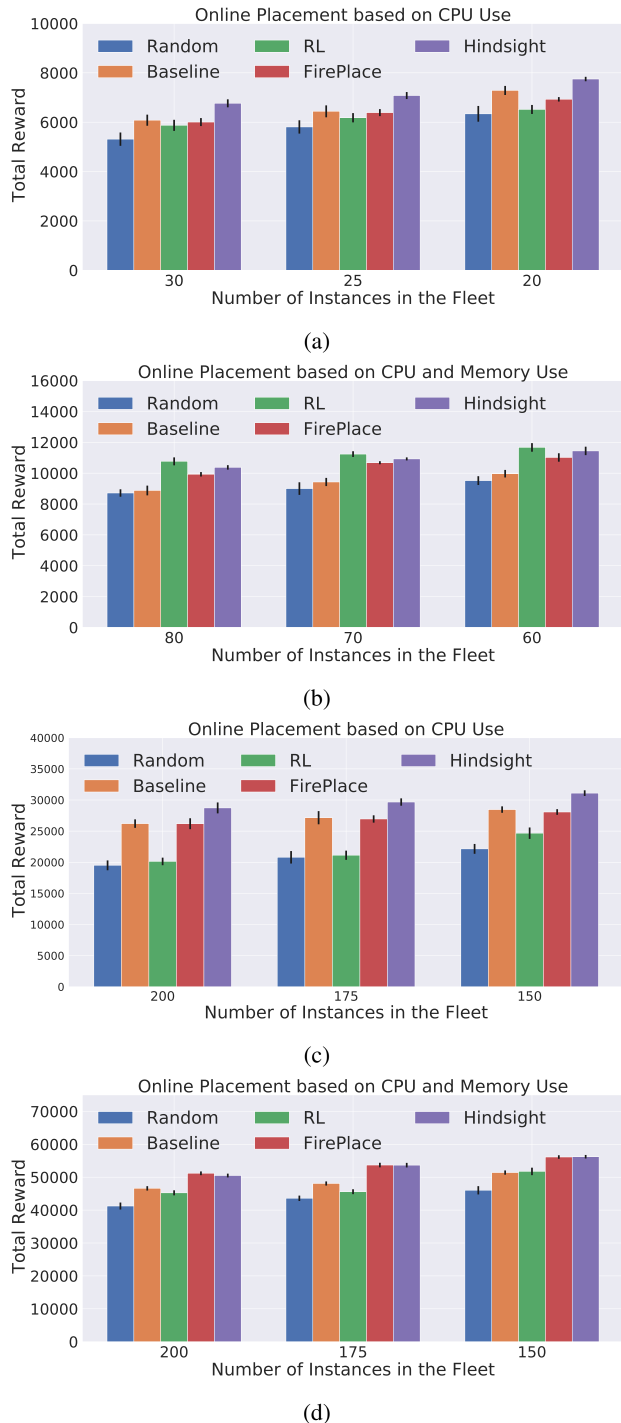

Figure 5. Online μVM packing based on CPU use in (a) and (c) for 20K and 100K dataset respectively. μVM packing based on CPU and memory use in (b) and (d) for 20K and 100K datasets respectively.

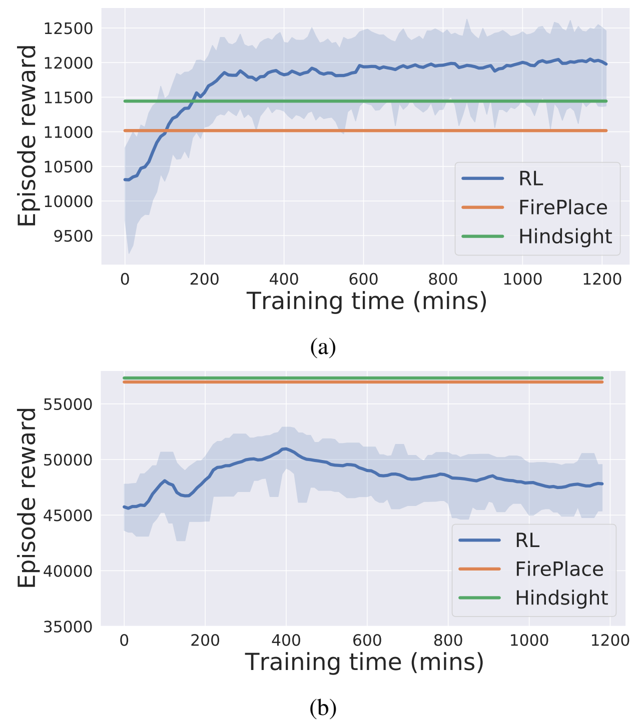

Figure 6. RL training curves for 2D packing based on memory and CPU use for the 20K (a) and 100K (b) datasets. We see that when RL succeeds, it does so quickly. However, often the performance plateaus much before performance of even the Baseline algorithm is reached.

performs well in the 20K dataset, it fails to even beat the baseline algorithm in 100K dataset. RL also performs only slightly better than Random in the CPU packing case, where the stochasticity of the environment is higher.

To evaluate FirePlace, we ran the Hindsight algorithm, logged its state and actions. We then used various ML models to learn to predict the Hindsight decision from the state. We got the best performance using both Random Forests (Liaw et al., 2002), and Support Vector Machines (Suykens & Vandewalle, 1999), we report whichever yielded the higher accuracy on the validation data here. For the CPU packing case, the models yield a validation accuracy of 91% and 92% for 20K and 100K datasets respectively. For the 2D packing case, the models have a validation accuracy of 79% and 85% for 20K and 100K datasets respectively. We used the learned model to make placement decisions for the test μVM partition. We use the same learned model across different fleet sizes to test generalization. Figures 5 shows the results for all datasets across 10 runs on the test μVM partition.

Figure 5a shows the results of the CPU based packing. The Hindsight algorithm performs 9% better than Baseline on average and 33% better than random. Due to the sporadic nature of CPU use (Figure 2), the temporal characteristics of μVM and PM CPU use are useful in determining a good placement. The Hindsight algorithm has access to the entire

future of PM and μVM use and can pick the PM whose low usage period (troughs in the timeseries plot) align with the peak μVM use. The Hindsight algorithm uses Equations 3 and 4 to determine the PM whose maximum CPU use is minimized after the μVM is placed. This temporal information is lost in the Baseline algorithm because of the coarse featurization, and hence the performance is worse. The Baseline algorithm performs better than random, by 22% on average, as it places the μVM in PMs which have more capacity using the best of $\mathcal{K}$ choices. Figure 5b shows the results of the 2-dimensional CPU and memory based packing. The trends in this case are similar to CPU packing, with the Hindsight outperforming the Baseline and random algorithms by 12% and 21% on average respectively.

The FirePlace performance is somewhere between Baseline and Hindsight, demonstrating that the model has indeed learned to make good decisions from the Hindsight dataset, while relying only on features available at decision time. FirePlace improves upon Baseline by 11% on average in the 2D packing problems, and performs almost as well as the Hindsight algorithm. However, in the case of CPU packing, the FirePlace performance is only as good as the Baseline algorithm. We hypothesize that the stochasticity in the CPU packing is too high to bridge the gap between Hindsight and Baseline algorithm.

FirePlace generalizes to the 2D case and is able to find a good trade-off between CPU and Memory features. In one 2D packing case, Figure 5b, we see that RL outperforms the Hindsight expert. This indicates that better trade-offs between CPU and Memory than the L^2 norm we used in Equation (8) are possible in the short term to minimize the long term PAR. If we introduced μVM features or raw timeseries as features and used LSTMs to represent the policy, the model overfitted to the training data and gave poor validation results. This indicates that, while the data we have seems large, it is still insufficient for data hungry RL models. FirePlace is much more sample efficient than RL, and we plan to explore the use of FirePlace as a pre-trained model for RL training in future work. Finally, we observe that within an episode, there were occasionally large jumps in the reward, where a few placement decisions lead to large shift in rewards, while most decisions have no impact. In ongoing work, we are working to characterize these 'important' placement decisions, and adjust the loss function used in training FirePlace to further improve performance.

7 DISCUSSION

7.1 Train Test Split

The evaluation assigns alternate micro VMs from the production trace to the training and test data partitions, which ensures the train and test data have similar distributions. However, in practice, the model needs to generalize to unseen data in another day or a different region. When we evaluated our trained models on data from a different day, the performance of the model indeed dropped. Upon analysis, we identified that the shift in distribution is primarily due to shift in demand from train to test data. We were able to mitigate any performance impact by normalizing the features on a per decision basis. We normalized the CPU and memory use features such that the model is given a comparative state of the PMs rather than their absolute values. More representative features are likely to yield higher performance gains.

7.2 A/B Testing in Production

We briefly describe how FirePlace can be evaluated in a production environment. The FirePlace model can be trained offline using historical data, and used in production for inference. Historical μVM use data is fed into our simulator to collect Hindsight decisions, and the resulting dataset can be used to train the FirePlace imitation model. We ensure the inference time of the model is sufficient for production settings, and validate the model performance in held out test set. Methods to ensure the simulator is representative to the production infrastructure is paramount to ensure model performs well in practice.

We create two fleets of equal size distinct from the production fleet, and direct a small but equal percentage of traffic to both fleets. For the test to be effective, we recommend sizing the fleet to be just large enough to accommodate the traffic for the period of the test. We maintain historical memory and CPU use for each instance in the fleet required for the features to be fed to the FirePlace model. Our formulation uses all the historical data available, but we found that even 10 minutes of historical data is sufficient to train a performant model in practice. While we do not use μVM features in our model, one can create representative features given historical use by the same function that creates a particular μVM. When μVM arrives for placement, the system randomly picks $\mathcal{K}$ PMs from the fleet and uses FirePlace for placement. The performance of the model is aggregated for a fixed period of time. The performance of FirePlace can be compared with baseline algorithms using aggregate CPU and memory use of the A and B fleet.

8 RELATED WORK

VM packing has been studied extensively in literature (Bobroff et al., 2007; Xiao et al., 2012; Meng et al., 2010b). A popular class of problems is VM migration (Bobroff et al., 2007; Xiao et al., 2012; Feller et al., 2012; Lebre et al., 2015). Here VMs are migrated periodically to reduce hot spots in the data center and exploit freed resources. The VMs and the invokes associated have longer lifetime

compared to the μVMs we studied, and forecasting algorithms such as exponentially weighted mean average, sufficed to predict the workload well. In contrast, we consider placement of μVMs when they are created, and do not consider migration in our formulation. Our invokes are short and bursty, making its resource use prediction hard. Some works consider the dependencies that exist between invokes (Grandl et al., 2016; Agarwal et al., 2012; Ferguson et al., 2012; Meng et al., 2010b) using heuristics (Grandl et al., 2016; 2014) and even RL recently (Mao et al., 2019). We work in an orthogonal setting, where the invokes and μVMs are independent from each other.

Our problem setting can be considered as a dynamic bin packing problem (Coffman et al., 1983), where items are packed into bins one at a time, their arrival order is unknown and they depart at an unknown time. The theoretical properties of the problem has been studied extensively, and it has been shown that even myopic algorithms like First Fit perform well compared to optimal packing (Gupta & Radovanovic, 2015; Chan et al., 2009; Stolyar & Zhong, 2013). Xin et al. (Li et al., 2013) show that multi-dimensional online packing is an NP-complete problem. They present a practical algorithm that avoids resource fragmentation and outperforms First Fit. However, in these works, the VM resource use is considered fixed and the algorithms have full access to the state of all PMs. In our use case, VM resource use changes over time. We need to pack VMs such that their peak usages are not aligned, so that the resource capacity of the PM is never exceeded.

Several works have considered packing of VMs such that their usage is *anti-correlated* with each other and leads to tight packing (Meng et al., 2010a; Chen et al., 2016; Kim et al., 2013; Shaw et al., 2018; Chen et al., 2018). Kim et al. (Kim et al., 2013) consider the covariance of the VM use from historical data and use a Pearson correlation based heuristic for placement. Meng et al. (Meng et al., 2010a) use ARMA and kernel density estimation based forecasting model to predict VM use. Chen et al. (Chen et al., 2016) use a neural network model for forecasting. In a follow up work (Chen et al., 2018), they improve their forecasting using PCA with ARIMA models. The forecasting models are used for placement using solutions like First Fit, Best Fit or genetic algorithms. All of these consider time scales of hours to days, making resource use pattern prediction feasible. They also consider the VM migration setting, where the horizon of forecasting is only one placement time step. In contrast, our work considers online placement with no migration, and hence we need to forecast the VM resource for its lifetime. In our μVM dataset, the time scales of invoke execution are of the order of seconds, and the bursty nature of invokes makes it a difficult forecasting problem. Therefore, we take a hindsight imitation approach, which circumvents the forecasting model and directly learns a hedged placement policy.

The hindsight optimization algorithm was proposed by Chong et al. (Chong et al., 2000). They used hindsight information to learn a Q function in a tabular setting for a network traffic control problem. Tamar et al. (Tamar et al., 2017) proposed a similar algorithm where they learn a invoke execution plan with model predictive control (MPC) (Camacho & Alba, 2013) using hindsight data, available only after decisions have been made. The learned plan is used as to shape the cost function of the online MPC algorithm that plans on a shorter horizon. Our work is related to, and draws inspiration from these approaches, but we directly use classification based imitation learning instead of MPC or Q learning to learn from hindsight. Our work is orthogonal to Hindsight Experience Replay (Andrychowicz et al., 2017) and Hindsight Policy Gradients (Rauber et al., 2017), which are designed for goal oriented problems.

9 Conclusions

We have demonstrated that we can exploit historical data to learn to optimize μVM placement. The results presented here are an initial proof of concept, and the algorithms need to evaluated on a variety of datasets and tested for robustness over longer periods of time in production settings. We continue to evaluate how to seed RL models with the FirePlace model, since RL demonstrated the ability to perform better than the Hindsight algorithm. It may also be possible to improve on the results with better feature engineering and machine learning models. Much of our current fleet cost can be attributed to how many μVMs are present at a time, and we can use similar machine learning approaches to identify when to create and destroy μVMs to save on infrastructure costs. The approaches presented here can be extended to reduce the operational cost of other compute services such as allocating PMs for autoscaling and many other managed serverless services.

References

Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pp. 419–434, 2020.

Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., and Zhou, J. Reoptimizing data parallel computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 281–294, 2012.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong,

R., Welinder, P., McGrew, B., Tobin, J., Abbeel, O. P., and Zaremba, W. Hindsight experience replay. In *Advances in neural information processing systems*, pp. 5048–5058, 2017.

Bai, S., Kolter, J. Z., and Koltun, V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

Bobroff, N., Kochut, A., and Beaty, K. Dynamic placement of virtual machines for managing sla violations. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 119–128. IEEE, 2007.

Camacho, E. F. and Alba, C. B. *Model predictive control*. Springer Science & Business Media, 2013.

Chan, J. W.-T., Wong, P. W., and Yung, F. C. On dynamic bin packing: An improved lower bound and resource augmentation analysis. *Algorithmica*, 53(2):172–206, 2009.

Chen, T., Zhu, Y., Gao, X., Kong, L., Chen, G., and Wang, Y. Correlation-aware virtual machine placement in data center networks. In *Cloud Computing, Security, Privacy in New Computing Environments*, pp. 22–32. Springer, 2016.

Chen, T., Zhu, Y., Gao, X., Kong, L., Chen, G., and Wang, Y. Improving resource utilization via virtual machine placement in data center networks. *Mobile Networks and Applications*, 23(2):227–238, 2018.

Chong, E. K., Givan, R. L., and Chang, H. S. A framework for simulation-based network control via hindsight optimization. In *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No. 00CH37187)*, volume 2, pp. 1433–1438. IEEE, 2000.

Christensen, H. I., Khan, A., Pokutta, S., and Tetali, P. Multidimensional bin packing and other related problems: A survey, 2016.

Coffman, Jr, E. G., Garey, M. R., and Johnson, D. S. Dynamic bin packing. *SIAM Journal on Computing*, 12(2): 227–258, 1983.

Fan, X., Weber, W.-D., and Barroso, L. A. Power provisioning for a warehouse-sized computer. *ACM SIGARCH computer architecture news*, 35(2):13–23, 2007.

Feller, E., Rilling, L., and Morin, C. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 482–489. IEEE, 2012.

Ferguson, A. D., Bodik, P., Kandula, S., Boutin, E., and Fonseca, R. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 99–112, 2012.

Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44 (4):455–466, 2014.

Grandl, R., Kandula, S., Rao, S., Akella, A., and Kulkarni, J. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 81–97, 2016.

Gupta, V. and Radovanovic, A. Online stochastic bin packing. *arXiv preprint arXiv:1211.2687*, 2012.

Gupta, V. and Radovanovic, A. Lagrangian-based online stochastic bin packing. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):467–468, 2015.

Haykin, S. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

Hyndman, R. J., Ahmed, R. A., Athanasopoulos, G., and Shang, H. L. Optimal combination forecasts for hierarchical time series. *Computational statistics & data analysis*, 55(9):2579–2589, 2011.

Kim, J., Ruggiero, M., Atienza, D., and Lederberger, M. Correlation-aware virtual machine allocation for energy-efficient datacenters. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1345–1350. IEEE, 2013.

Kratzke, N. A brief history of cloud application architectures. *Applied Sciences*, 8(8):1368, 2018.

Lebre, A., Pastor, J., and Südholt, M. Vmplaces: A generic tool to investigate and compare vm placement algorithms. In *European Conference on Parallel Processing*, pp. 317–329. Springer, 2015.

Li, X., Qian, Z., Lu, S., and Wu, J. Energy efficient virtual machine placement algorithm with balanced and improved resource utilization in a data center. *Mathematical and Computer Modelling*, 58(5-6):1222–1235, 2013.

Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381*, 2017.

Liaw, A., Wiener, M., et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 270–288. 2019.

Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E., and Pendarakis, D. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing*, pp. 11–20, 2010a.

Meng, X., Pappas, V., and Zhang, L. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *2010 Proceedings IEEE INFOCOM*, pp. 1–9. IEEE, 2010b.

Mitzenmacher, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pp. 305–313, 1989.

Puterman, M. L. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

Rauber, P., Ummadisingu, A., Mutz, F., and Schmidhuber, J. Hindsight policy gradients. *arXiv preprint arXiv:1711.06006*, 2017.

Richa, A. W., Mitzenmacher, M., and Sitaraman, R. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Shaw, R., Howley, E., and Barrett, E. A predictive anti-correlated virtual machine placement algorithm for green cloud computing. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pp. 267–276. IEEE, 2018.

Shen, B., Sundaram, R., Russell, A., Aiyar, S., Gupta, K., Nagpal, A., Ramesh, A., and Shukla, H. High availability for vm placement and a stochastic model for multiple knapsack. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–9. IEEE, 2017.

Song, W., Xiao, Z., Chen, Q., and Luo, H. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Transactions on Computers*, 63(11):2647–2660, 2013.

Stolyar, A. L. and Zhong, Y. A large-scale service system with packing constraints: Minimizing the number of occupied servers. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):41–52, 2013.

Suykens, J. A. and Vandewalle, J. Least squares support vector machine classifiers. *Neural processing letters*, 9 (3):293–300, 1999.

Tamar, A., Thomas, G., Zhang, T., Levine, S., and Abbeel, P. Learning from the hindsight plan—episodic mpc improvement. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 336–343. IEEE, 2017.

Wen, R., Torkkola, K., Narayanaswamy, B., and Madeka, D. A multi-horizon quantile recurrent forecaster. *arXiv preprint arXiv:1711.11053*, 2017.

Xiao, Z., Song, W., and Chen, Q. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE transactions on parallel and distributed systems*, 24(6):1107–1117, 2012.

APPENDIX

A HYPERPARAMETERS

Below we provide the hyperparameters used for each experiment. Note that no formal hyperparameter search was conducted and the hyperaparameters were generally set to default values found in the Ray RLlib (Liang et al., 2017) and Scikit-learn (Pedregosa et al., 2011) libraries.

Table 1. Hyperparameters used for RL experiments. We used the Proximal Policy Optimization algorithm (Schulman et al., 2017), as implemented in Ray RLlib repository (Liang et al., 2017).

Hyperparameter	Value
Gamma	0.995
KL coefficient	1.0
SGD iterations	5
Minibatch size	512
Train batch size	8192
Learning rate	0.00001
Hidden layers	[256, 256]
Use GAE	False

Table 2. Hyperparameters used for Hindsight Imitation Learning experiments with Support Vector Machines. We used the implementation in the Scikit-Learn library (Pedregosa et al., 2011).

Hyperparameter	Value
C	10
Cache size	200
Class weight	None
Coef 0	0.0
Decision function shape	ovr
Degree	3
Gamma	Auto Deprecated
Kernel	RBF
Max Iter	-1
Probability	False
Random State	None
Shrinking	True
tol	0.001

Table 3. Hyperparameters used for Hindsight Imitation Learning experiments with Random Forests. We used the implementation in the Scikit-Learn library (Pedregosa et al., 2011).

Hyperparameter	Value
Bootstrap	True
Class Weight	None
Criterion	gini
Max Depth	None
Max Features	auto
Max Leaf Nodes	0.None
Min Impurity Decrease	0.0005
Min Impurity Split	None
Min Samples Leaf	1
Min Samples Split	2
Min Weight Fraction Leaf	0.0
Number of Estimators	50
Number of Jobs	None
OOB Score	False
Random State	1
Warm Start	False