Ultron-AutoML: an open-source, distributed, scalable framework for efficient hyper-parameter optimization

Swarnim Narayan
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Swarnim.Narayan@walmartlabs.com

Chepuri Shri Krishna
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Chepurishri.Krishna@walmartlabs.com

Varun Mishra
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Varun.Mishra@walmartlabs.com

Abhinav Rai
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Abhinav.Rai@walmartlabs.com

Himanshu Rai
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Himanshu.Rai@walmartlabs.com

Chandrakant Bharti
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Chandrakant.Bharti@walmartlabs.com

Gursirat Singh Sodhi
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Gursirat.Singh@walmartlabs.com

Ashish Gupta
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Ashish.Gupta@walmartlabs.com

Nitinbalaji Singh
Catalog Data Science
Walmart Global Tech India
Bengaluru, India
Nitinraj.Balajisingh@walmartlabs.com

***Abstract*—We present Ultron-AutoML, an open-source, distributed framework for efficient hyper-parameter optimization (HPO) of ML models. Considering that hyper-parameter optimization is compute intensive and time-consuming, the framework has been designed for reliability – the ability to successfully complete an HPO Job in a multi-tenant, failure prone environment, as well as efficiency – completing the job with minimum compute cost and wall-clock time. From a user's perspective, the framework emphasizes ease of use and customizability. The user can declaratively specify and execute an HPO Job, while ancillary tasks – containerizing and running the user's scripts, model checkpointing, monitoring progress, parallelization – are handled by the framework. At the same time, the user has complete flexibility in composing the code-base for specifying the ML model training algorithm as well as, optionally, any custom HPO algorithm. The framework supports the creation of data-pipelines to stream batches of shuffled and augmented data from a distributed file system. This comes in handy for training Deep Learning models based on self-supervised, semi-supervised or representation learning algorithms over large training datasets. We demonstrate the framework's reliability and efficiency by running a BERT pre-training job over a large training corpus using pre-emptible GPU compute targets. Despite the inherent unreliability of the underlying compute nodes, the framework is able to complete such long running jobs at 30% of the cost with a marginal increase in wall-clock time. The framework also comes with a service to monitor jobs and ensures reproducibility of any result.**

***Index Terms*—Hyperparameter optimization, Machine Learning, Deep Unsupervised/Semi-supervised/Representation/ Self-supervised Learning**

978-1-7281-6251-5/20/$31.00 ©2020 IEEE

I. Introduction

Developing production grade AI systems for NLP or vision-based applications often requires training Deep Learning models in self-supervised/unsupervised mode followed by further finetuning in a supervised or semi-supervised mode for the task of interest [22], [25], [47], [52]. The training process itself is long-running since it needs to sweep over very large datasets, while the underlying models are memory and compute intensive, requiring large GPU units for speed-ups.

Additionally, the models need to be extensively fine-tuned for hyper-parameters such as the learning rate, regularization terms such as drop-out and architecture related choices related to the depth and width. This entails running many Deep Learning training jobs, each with a different hyper-parameter configuration, in parallel, or, if used in conjunction with a hyper-parameter optimization algorithm, sequentially with a lower degree of parallelism.

Training ML models at this scale and level of complexity requires access to an on-premise distributed computing environment or a cloud computing platform such as AWS, Azure or GCP. It also places additional burdens on the ML model developer: she needs to monitor and manage multiple training jobs, cater for various failure scenarios, and economize for compute. Additionally, most on-premise cluster environments run in multi-tenant mode, imposing additional headaches such as failure to provision the required compute for triggering the user's job.

Keeping these considerations in mind, we propose Ultron-AutoML, an open-source, distributed framework for easy specification and efficient execution of ML model hyper-parameter optimization (HPO) jobs. The Ultron-AutoML framework can be used by practitioners and researchers alike to develop production grade ML models as well as benchmark alternate ML algorithms, Deep Learning architectures and hyper-parameter tuning strategies. [1]

II. Background

A. Hyperparameter Optimization Review

The importance of tuning ML model hyper-parameters in the ML development cycle cannot be overstated: [40] shows that the LSTM architecture, proposed way back in 1997, outperformed two recent architectures, Recurrent Highway Networks [55] and NAS [56], on language modelling benchmarks after being tuned for regularization. In a similar vein, the original GAN's performance was found to be on par with contemporary versions after it was given a bigger computational budget for tuning hyper-parameters [37].

Hyperparameter Optimization (HPO), also referred to as AutoML in the literature, can be cast as the optimization of an unknown, possibly stochastic, objective function mapping the hyper-parameter search space to a real valued scalar, the ML model's accuracy or any other performance metric on the validation dataset. The search-space can extend beyond algorithm or architecture specific elements to encompass the space of data pre-processing and data-augmentation techniques, feature selections, as well as choice of algorithms. This is sometimes referred to as the CASH (Combined Algorithm Search and Hyper-parameter tuning) problem for which algorithms have been proposed [28], [48].

Neural Architecture Search (NAS) is a special type of HPO where the focus is on algorithm driven design of neural network architecture components or cells [26]. Models trained with architectures composed of these algorithmically designed neural network cells have been shown to outperform their hand-crafted counterparts in image recognition, object detection [57], and semantic segmentation [21], underscoring the practical importance of this field.

Random Search [18] and Grid Search are effective HPO strategies when the computational budget is limited or the hyper-parameter search space is high dimensional. Both are easy to implement and completely parallelizable. Random Search is also widely regarded as a good baseline for benchmarking new hyper-parameter optimization algorithms [33].

Bayesian Optimization (BO) is a dominant paradigm for HPO [20], [27], [45]. Here, the objective function is modeled as a Gaussian Process [50], with the Kernel design reflecting assumptions about the objective function's smoothness properties. Under this assumption, the posterior distribution of the validation score for a candidate architecture is a Gaussian

[1]http://github.com/armlight/ux

```
1: procedure GENERIC-HPO-ALGORITHM
2:     Initialize State
3:     while $stopping\ criterion$ is False do
4:         Sample a batch of candidate architectures
5:         Fetch validation scores for these architectures
6:         Update state based on fetched validation scores
7:     end while
8: end procedure
```

Fig. 1. A generic HPO algorithm

whose mean and variance can be computed in closed form. The BO algorithm also needs to specify an acquisition strategy that balances exploit (choose an architecture with high mean) against explore (choose an architecture with high variance). Probability of Improvement and Expected Improvement (EI) [31], Entropy Search [30], GP Upper Confidence Bound [46] are some principled acquisition strategies to balance explore-exploit and find optimal architectures within an exploration budget. While BO based HPO is inherently sequential, the parallelism offered by multi-core or distributed computing platforms can be leveraged to sample batches in a principled manner such as Gaussian Process Bandit Optimization [24] or Monte Carlo acquisition for parallelizing BO [45]. Working with batches speeds up the search and is useful for HPO with large computational budgets.

Other HPO paradigms include evolutionary learning, model free Reinforcement learning (RL) and gradient based approaches. In evolutionary learning models, inspired by genetic evolution, the architectures are modeled as gene strings. The search proceeds by mutating and combining strings to discover promising architectures [43], [44], [51]. RL based techniques [17], [32], [56], [57] specify a policy network that learns to output desirable architectures. Search proceeds by training the policy network using Q-learning or policy gradient. These techniques are flexible in that they can search over variable size architectures and have shown very promising results for NAS. Gradient based methods specify the objective function as a parametric model and proceed to optimize it with respect to the hyper-parameters via gradient-descent [35], [38], [39].

Abstractly, a generic HPO algorithm distinct from Random Search or Grid Search maintains state and repeats steps based on a state-update procedure and a sampling procedure shown in Fig. 1. This abstraction can inform the design of a distributed framework that can support any HPO algorithm.

B. Related Work

AWS Sagemaker [1], Azure Machine Learning [2] and Google Cloud AI platform [6] are cloud-based services for hyper-parameter tuning. All three allow users to submit ML model training code-bases along with hyper-parameters to be tuned. However, the user can only pick one of their supported hyper-parameter optimization algorithms. There is limited scope for customizing these hyper-parameter tuning strategies or specifying a new strategy as part of the user's code-base.

Open-source frameworks include Hyperopt [19], Auptimizer [36], Katib [54] and Tune [34]. Auptimizer and Tune have been designed for ease of use and extendibility. They can integrate with multiple cloud platforms but they are not designed for reliability and economy of compute in a failure prone, multi-tenant environment. Hyperopt supports Random Search and Tree Parzen Estimator (TPE) but has limited support for scalability and distributed training.

Katib comes closest to Ultron-AutoML in that it is designed with Kubernetes [10] as the backbone. However, with Katib, the onus of containerizing ML model training jobs and hyper-parameter optimization jobs lies with the user. Further, Katib doesn't cater to failure scenarios for ML training jobs. This means the user needs to write code to resume training from a checkpoint.

III. Terminologies

We define the following terms that will be used in the rest of the paper for describing the working and design of the system:

A **HPO Job** refers to a user's request to execute an HPO algorithm and return the best trained ML model within a computation budget. The user's request will contain the code-base to train an individual ML model (referred to as **ML training job**) as a function of a **HP configuration** and the training datasets. The **HP configuration** is a dictionary object mapping hyper-parameters to their values. **Scores** are validation and test metrics of the model trained using a hyper-parameter configuration.

An **experiment** refers to running an independent set of *HPO Jobs*, each *HPO Job* taking the same payload but only differing in the starting seed. The underlying *HPO Jobs* are referred to as *trials*. The *experiment* then reports the statistics – mean and std-deviation – of the metric of interest from the *trials*. The *experiment* allows the user to measure the performance of the HPO algorithm or underlying ML algorithm. The Ultron-AutoML framework ensures that the results of an *experiment* are reproducible.

IV. Design

Ultron-AutoML can be installed to run as a service on any on-premise Kubernetes cluster or a cloud platform such as AWS, Google Cloud or Azure ML. From a performance perspective, the framework's design has been governed by the following aspects:

- Reliability: We define reliability as the likelihood that a user's *HPO Job* gets completed in a specified period of time. Given that the *HPO Job* is long-running and operates in a multi-tenant resource constrained environment, the framework should cater to various failure scenarios and be resilient to high load factors.
- Efficiency: The framework should complete a user's job in minimum time and compute cost. To do so, the framework must fully leverage the parallelism offered by the user's HPO algorithm, ensure high availability of various components, and durably maintain job state to ensure recoverability. In certain scenarios, the user should also be able to trade-off wall-clock time against compute cost.
- Scalability: The framework should scale with job load (determined by number of user requests and average size of a request), if running on a cloud platform.

```
{
    "model_classname" : "Model",
    "module_name" : "train",
    "package_name" : "modelpkg",

    "hp_ranges" : {
      "optimizer": ["adam","sgd","rmsprop"],
      "pca_components": [50, 51, 52],
      "num_layers": [1, 2, 3, 4],
      "lr": [1e-3, 5e-3, 1e-4]
      },
    "strategy" : "RANDOM",
    "num_iterations" : 150,
    "jobname" : "SampleModel",
    "parallelism_fact" : 8,
    "compute":"gpu",
    "file_name":"/sample_model.tar.gz",
    "cpu_per_pod":"5",
    "memory_per_pod":"100000Mi",
    "gpu_per_pod":"1"
}
```

Fig. 2. JSON encoding HPO Job specification

From a user's perspective, the framework's design emphasizes ease of use, flexibility and customizability. The user can declaratively specify and execute the salient aspects of a *HPO Job*, avoiding the need to write boilerplate code. The user can focus instead on code to implement the ML model training algorithm and, optionally, any custom HPO algorithm.

A. Specifying and Running an HPO Job

The user submits a *HPO Job* via HTTP POST method. The HTTP POST payload, shown in Fig. 2, is a JSON object that fully defines the *HPO Job*. The JSON object values include the packaged code-base for executing a model training job, the hyper-parameter search space, the choice of hyper-parameter tuning strategy from supported strategies – Random Search, TPE [20], and REINFORCE based methods [32]. The overall computational budget, characterized in terms of number of training jobs, is set via the *num_iterations* key.

The user can also specify compute targets for training jobs via the *cpu_per_pod*, *memory_per_pod* and *gpu_per_pod* keys. Finally, the user can speed up training by setting the level of parallelism via *parallelism_fact*. Ultron-AutoML will make a best effort attempt to provide the specified level of parallelism subject to availability of computational resources.

The user has complete control and flexibility over composing the model training code-base, including choice of dependencies. The user can specify these dependencies, including any version of popular ML frameworks such as Pytorch [13], Tensorflow or scikit-learn along with other libraries, in the *setup.py* file. The Ultron-AutoML framework manages the installation of these dependencies and execution of the training job as a containerized application (refer section V-B).

```python
from setuptools import setup
setup(install_requires=['torch>=0.4.1',
                        'numpy',
                        'boto3',
                        'requests',
                        'tqdm',
                        'regex'], other_entries)
```

Fig. 3. Custom libraries in setup.py

```python
class ModelInterface:
    def score(self, hyperparameters,
              checkpoint_path, resume):
        """score the hyperparameter"""
        pass
```

Fig. 4. Interface for the ML model score function

Within the user supplied code-base, the user only needs to write a class implementation for the interface in Fig. 4. The abstract method *score* takes as argument, *hyperparameters* which is an *HP configuration* object and returns a *score*.

B. Bring Your Own Hyperparameter Optimization Algorithm

The user can specify a custom HPO algorithm by writing a class for the interface based on the abstraction in Fig. 5. The class needs to implement the following methods:

- *sample_hyperparameter_candidates* takes as argument an integer N and returns a set of *HP configurations* of size N.
- *update_state* takes as argument a set of tuples, whose components comprise a *HP configuration* and associated ML model validation score, and updates some internal state.

C. Architecture Overview

To achieve reliability, efficiency and scalability, Ultron-AutoML uses Kubernetes. All components processing a *HPO Job* run as containerized applications on Kubernetes *pods* with *Controllers*. Components are loosely coupled using a messaging system.

For scalability, we rely on the Kubernetes *cluster auto-scaler*. The *cluster auto-scaler* minimizes node under-utilization and therefore compute cost by dynamically commissioning and decommissioning nodes. Kubernetes can also shift in-progress jobs out of unavailable nodes to available ones. This allows us to effectively leverage intrinsically unreliable computes like pre-emptible GPU nodes which are up-to 70% more cost effective than regular nodes for training models.

```python
class HPOAlgorithmInterface:
    def sample_hyperparameter_candidates(self, n):
        """Sample n candidate HP configurations"""
        pass

    def update_state(self, hyperparameter_candidates
, validation_scores):
        """Sample N candidate HP configurations"""
        pass
```

Fig. 5. Interface for the HPO algorithm

The messaging system is implemented with *RabbitMQ* [11] which is the queue orchestrator. A downstream component (a consumer) uses positive and negative acknowledgments for respectively queuing and dequeuing messages. When a consumer detects a problem within itself, or within a downstream component, it uses a negative acknowledgment with a timer to perform a delayed retry. The queue orchestrator also maintains heartbeats with consumers to detect dead consumers and to re-queue any messages which were being processed by them.

Queues are mirrored for high availability. Certain components require a key value store for which we use *Redis* [14]. It is made highly available using Redis sentinels. For durability of *Redis* and *RabbitMQ*, we rely on persistent volumes mounted on their *pods*.

A high-level view of the process flow of the *HPO Job* is as follows:

1) The user fires an HTTP POST request containing the *HPO Job* payload shown in Fig. 2.
2) The *Ingress Controller* forwards the incoming request to the *Entry Point Service*, which validates the request and forwards it to the *On Demand Containerizer*.
3) The *On Demand Containerizer* creates *master* and *worker* images based on the user-supplied code-base. The *master* and *worker* images implement the interfaces in Fig. 5, Fig. 4 respectively. Finally, it forwards the request to the *Job Agent*.
4) The *Job Agent* acquires resources from the *Resource Limiter*, launches the *master* and creates two queues: *Work* queue, *Results* queue.
5) The *master*, on initialization, starts a pool of parallel *worker pods*. The *master* and *workers* co-ordinate via the *Work* queue, *Results* queue to run the HPO algorithm to completion.
6) Finally, the *Completion Manager* deletes the *master*, associated *workers*, *Results* queue and *Work* queue, marking the end of the *HPO Job*.

V. Request Lifecycle

A. Ingress and Entry Point Service

The *Ingress* serves all requests originating outside the cluster. It redirects an incoming HTTP POST based new *HPO Job* request to a Kubernetes *Service* called *Entry Point Service*. Upon receiving the request, the *Entry Point Service* validates it, uploads the user's code-base to a persistent store, and forwards the request to the *Processing* queue.

B. On Demand Containerizer

The *On Demand Containerizer* is a Kubernetes *Deployment* that creates images for the *master* and associated *workers*. It functions as follows:

1) Consumes a message from *Processing* queue

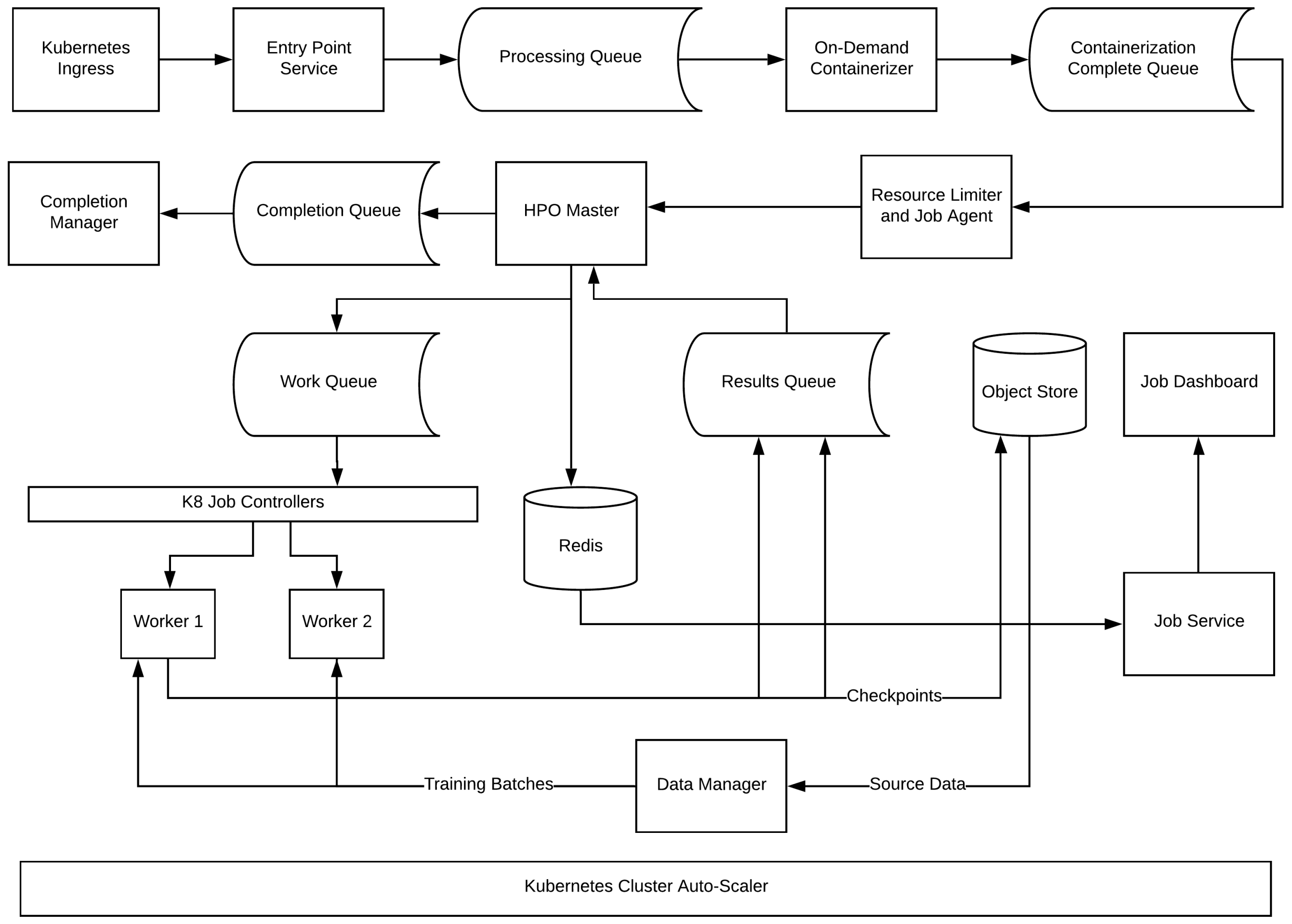

Fig. 6. Architecture of the platform

2) Creates *worker* image: Selects a base *worker* image as per the user's request. Every model code-base requires a different set of toolkits and the platform comes with a choice of *worker* images, each containing a standard ML toolkit combination.
3) Downloads the user code distributable from the persistent store, installs it and all its dependencies into the base *worker* image. It also installs *worker* boilerplate logic which manages communication with the *Work* queue, *Results* queue, heartbeats and acknowledgments into the base image.
4) If the user has supplied a custom HPO algorithm, it creates a custom *master* image.
5) Pushes the images into a pre-configured registry and forwards the request to the next queue in the pipeline, the *Containerization Complete* queue.

C. Resource Limiter

To make the platform multi-tenant, there is a need to safeguard against over-consumption of resources by any tenant. For this, a *Resource Limiter* imposes an upper limit ($max\_resources$) on resource consumption. One unit of a resource is defined as one GPU or one CPU. The quantum of resource which the limiter can allocate or de-allocate is the *parallelism_fact* (P) times the *cpu_per_pod* or *gpu_per_pod* (N_R) for an *HPO Job* as shown in Fig. 2. The *Resource Limiter* is a semaphore $resource\_counter$, in Redis. It relies on race-condition safe operations: $INCRBY(W, N)$ and $DECRBY(W, N)$ ("increment/decrement integer W by N") [8] [9] [3]. We optimistically increase this semaphore before the *master pod* is started. The procedure is shown in Fig. 7.

D. Job Agent

The *Job Agent* is responsible for acquiring resources via the *Resource Limiter* and on successful acquisition, setting up components for downstream steps. By design choice, the *Job Agent* is a singleton because it acquires resources. The Kubernetes *Deployment Controller* ensures that despite being singleton, it is never a single point of failure. It functions as follows:

1) Consumes a message from the *Containerization Complete* queue
2) Creates the job's *Work* queue, *Results* queue and inserts a sentinel into the *Work* queue.

```
procedure LOCK-RESOURCES($P, N_R$)
    $R_{req} \Leftarrow P \cdot N_R$
    $E \Leftarrow R_{req} + resource\_counter$
    if $E > max\_resources$ then
        Re-queue request for delayed retry
    else
        $INCRBY(resource\_counter, R_{req})$
        Trigger Master Start
        if Master Start Failed then
            $DECRBY(resource\_counter, R_{req})$
            Re-queue request for delayed retry
        end if
    end if
end procedure
```

Fig. 7. Resource Locking Procedure

3) Spawns a *Data Manager* for the job
4) Acquires resources via the *Resource Limiter* and spawn the job's *master pod*.

E. Master and Workers

The *master* and *worker pods* execute the *HPO Job* as follows:

1) The *master pod* samples *HP configurations* and fetches their scores by calling a stub of the model's *score* function as in step 5, Fig. 1. When the stub is called, the framework pushes the *HP configurations* into the *Work* queue and the *master pod* waits for their scores on the *Results* queue.
2) A worker-crew model [29] is used to score the *HP configurations*. The framework spawns a pool of *worker pods*. Each *worker pod* consumes a *HP configuration* from the *Work* queue, scores it, and writes the score to the *Results* queue.
3) When all scores are in, the *master pod* stops waiting and resumes from step 6, Fig. 1. When the *stopping criteria* is fulfilled, it sends a message to the *Completion* queue.

The *master pod* caches all scores in Redis with the *HP configuration* as the keys. This allows scores of duplicate *HP configurations* to be retrieved from the cache instead of a fresh scoring.

The *master* and *worker pods* are tracked by separate *Job Controllers*. The platform uses the *Jobs Controller* because it supports a *restartPolicy* of *onFailure*. This policy setting releases all resources consumed by the *worker* and *master pods* when they successfully terminate.

We use one *worker pod* mapped to one *Job Controller* as opposed to multiple *worker pods* mapped to one *Job Controller* because we observed that in the presence of the *cluster auto-scaler*, the latter option results in a faulty infinite loop.

F. Handling master pod failures

The *master pod* checkpoints its state after each iteration in Fig. 1. If it fails, the *Job Controller* restarts it and it makes a recovery using the most recent checkpoint. Since the scores are also cached by the *master pod* in *Redis*, they too remain intact during *master pod* failure.

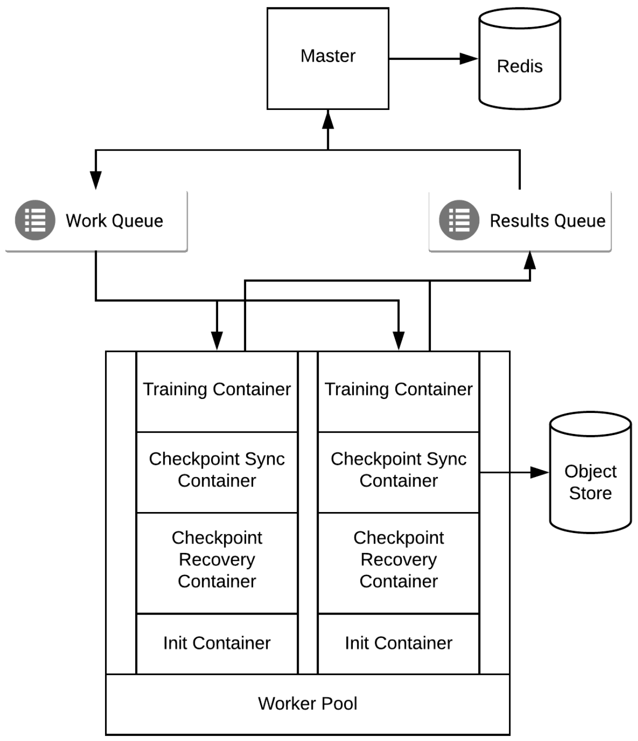

Fig. 8. Master and Worker

G. Handling worker pod failures

Negative acknowledgments, heartbeats and delayed retries ensure no *HP configuration* is lost when *worker pods* fail. Since *worker pods* are tracked by *Job Controllers*, a minimum number of them is always maintained in the pool. Model checkpointing ensures that the ML training job does not start from scratch when the *worker pods* themselves restart.

H. Checkpointing

Model checkpointing is critical because training can be preempted due to node failure. Model checkpoints can be several Gigabytes in size and are created every few minutes during training. With multiple *HPO Jobs* running concurrently, the checkpoint creation rate can be very high.

Using a *NFS* like *Gluster* [5] entails high storage costs and complexity. Further, *NFSes* are typically shared across an organization and disproportional load of the checkpoints may also result in its outage impacting other applications. We therefore use object stores, which are more cost effective, for checkpointing.

Object stores can have lower write speeds which can become a bottleneck for the training thread. To mitigate this, the thread only writes the checkpoints to a *checkpoint path* on the locally attached ephemeral storage of the node which is as fast as writing to a local disk.

These checkpoints are synchronized to the *object store* by a separate container running on the *worker pod*. The implementation of this container can be extended to incorporate the efficient *upload* implementation for a cloud provider. For example, a concrete implementation of this container which comes with the platform uses *gsutil* with parallel file-chunk upload for syncing to Google Cloud Storage Buckets. The *checkpoint recovery container* runs immediately before the *score* function is called. It fetches the latest checkpoint from the *object store* and places it in the *checkpoint path*. This path is then passed as an argument to the *score* function from where the model can recover it's state.

This container can also be extended. Models can save checkpoint files within separate folders named as per the respective epochs. As soon as all checkpoint files for an epoch have been written to its corresponding folder, the model can create an indicator file inside it which commands the checkpoint syncer to upload the epoch's checkpoint folder to the object store. During training, the *checkpoint recovery container* retrieves all folders from the *object store*, sorts them in descending order to get the checkpoint files for the latest epoch.

I. Data Manager

For training models over large data sets with deep unsupervised or semi-supervised algorithms, the *Data Manager* component implements a data pipeline which streams shuffled and augmented batches of training data to the *worker pods*. The *Data Manager* component shown in Fig. 9, is a set of *pods*, tracked by a *Deployment Controller* and it's REST API exposed via a Kubernetes *Service*. Multiple *pods* ensure high availability and the *Deployment Controller* ensures reliability. All *worker pods* in an *HPO Job* receive their training batches from a common *Data Manager* component.

Since the *Data Manager pods* run parallelly with the *worker pods*, it ensures that training batches are always available for consumption by the training GPU, thereby minimizing data starvation. The data pipeline is implemented using *Tensorflow tf.data* [16] and provides interfaces for the user to specify data augmentation operations.

Since the model training may be implemented using a different framework like *PyTorch* [13], the framework enables inter-operability between *Tensorflow* and frameworks such as *PyTorch*. It works as follows:

1) During training, inside the *Data Manager*, the data from the *object store* is incrementally streamed, passed through the *shuffling-augmenting-batching* operations and the output batches are pushed into the *tf.data Queue*. When the *Data Manager API Controller* receives a request for a set of training batches, it dequeues them from the *tf.data Queue* and serves them. The pipeline works to ensure that the *tf.data Queue* is always filled to capacity. The *Data Manager* is multi-threaded, with a pipeline running in each, all injecting data into the *tf.data Queue*.

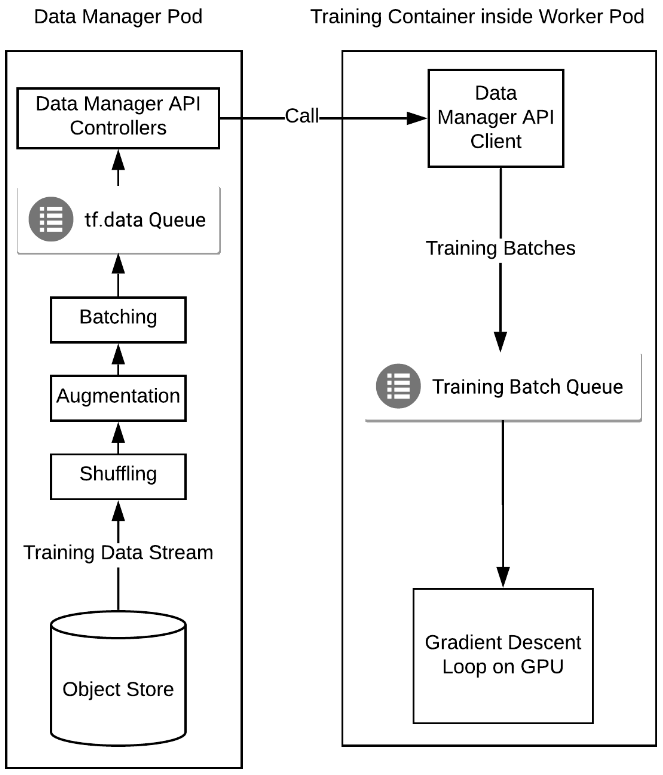

Fig. 9. The Data Manager and Worker Interaction

2) A client of the *Data Manager API Controller*, named the *Data Manager API Client*, which comes pre-packaged inside the training container of a *worker pod*, makes calls to the *Data Manager API Controller* for receiving training batches. These are pushed into the *training batch queue* from where they are consumed by the GPUs for executing the gradient descent steps. The client and the gradient descent run on different threads with the former on a CPU and the latter on a GPU.

J. Completion Manager

The *Completion Manager* frees up resources, executes finalizers and marks the completion of the *HPO Job*. It functions as follows:

1) Consumes a message from *Completion* queue
2) Frees-up resources by calling the *Resource Limiter* which in turn will decrease the $resource\_counter$.
3) Executes all finalizers: For example, certain flows may want to ensure that a particular checkpoint exists on the *object store* before deleting the *worker pods*. This is the case because the synchronization of checkpoints to the *object store* may lag the training itself due to low write speed of the *object store*.
4) Deletes all queues created for the job i.e. the *Work* queue, *Results* queue to free-up memory on the RabbitMQ cluster. Deletes the *master* and *worker Job Controllers*. Even though the *pods* terminate on their own,

their *Controllers* need deletion to eliminate potential resource leaks.

5) The *HPO Job* is marked complete

VI. Reproducibility of Results

Reproducibility is important in cases where researchers would want to use the platform for benchmarking. The platform can run in a mode which guarantees reproducible results for GPU based models. This is achieved by having a support for setting seeds in the entire flow and in case of *Tensorflow* models, by imposition of version standards not plagued by the problem of in-determinism. Problems of in-determinism in *Tensorflow* have been widely studied and solutions proposed [4]. Base *worker* images with *Tensorflow* setups supporting reproducibility have specifically been pre-packaged for this purpose with the platform.

VII. Monitoring a user's job

An *HPO Job* is monitored by an *HPO Job portal* , a web application written in *Angular 8*. The *HPO Job portal* displays the degree of completion, best result obtained thus far etc.

The portal communicates with a REST API which fetches the data from *Redis* for metrics that need to be displayed. Since the *master pod* reads and writes to the *Redis* cache, the portal reads the same and displays near real-time accurate information to the user. The REST API and the portal are deployed as a set of *pods*, tracked by a *Deployment Controller*, load balanced by a Kubernetes *Service*.

VIII. Results and Case-studies

We present results from three case-studies to demonstrate different performance aspects of the Ultron-AutoML framework. In the case study under section VIII-A, we investigate the ability of the framework to reliably and efficiently complete a BERT pre-training job. In the case study under section VIII-B, we assess the ability of Ultron-AutoML to effectively parallelize an *HPO Job* and reduce wall-clock time. In the case study under section VIII-C, we showcase how Ultron-AutoML can be used to benchmark HPO algorithms. All case studies were run by installing Ultron-AutoML on *Google Cloud Platform.*

A. Running a BERT pre-training job

Pre-training a BERT model [25] is both time and compute intensive. Moreover, the pretraining task requires a huge amount of data to learn meaningful features. Depending on the volume of training data, the pre-training job could take several days even when using multiple GPUs. So, it becomes essential for the platform executing such jobs to be reliable and efficient. We show that Ultron-AutoML is able to reliably execute such jobs while at the same time be cost efficient.

This case study presents a modification of the BERT pre-training task [25]. We perform finetuning over the pretrained weights instead of training them from scratch. The problem comprises of two tasks:

TABLE I
Experiment Specs For Masked LM and Sentence Prediction

Number of documents	9 Million
GPU Family	Tesla V100
GPUs per Worker Pod	4
Total Checkpoints Created	270 Gigabytes
Epochs	2
Node Type	Pre-emptible, Non Pre-emptible Nodes

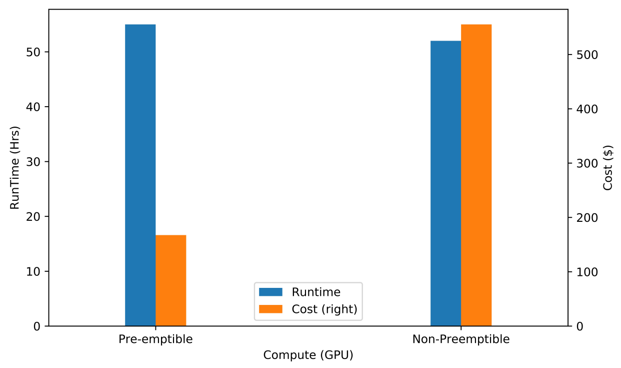

Fig. 10. Trade-off between cost and time efficiency

1) Masked Language Model: For a given sequence of corrupted tokens, $w_1, w_{\text{mask}}, w_3, \ldots, w_n$, predict the masked token using contexts from both left and right of it.
2) Next Sentence Prediction: Given a pair of sentences, predict whether the pair consists of consecutive sentences or not.

The experiment specifications are listed in Table I. The *Data Manager* ensures robust and reliant data loading. Also, the checkpoint recovery feature as provided by the platform ensures that if there is a failure at any point, the checkpoint can be restored, and training can continue from that point onward which is critical for such time intensive jobs. Results are presented in Fig. 10.

As it is evident from the figure, the run-time on the pre-emptible compute is slightly higher but the reliability of Ultron-AutoML platform and the efficiency of the *Data Manager* leads to nearly 70% cost savings.

B. Finetuning a BERT model with limited labeled data

A pretrained BERT model can be fine-tuned with very little labeled data for downstream applications such as text classification. Performance of the text classification model can be further improved by tuning for a key hyper-parameter: the number of training steps. Limited labeled data implies a small sized validation data-set, posing a problem for tuning hyper-parameters [41].

We get around this via a modified K-fold cross-validation algorithm. We train K BERT models independently for a predetermined number of training steps. We accumulate the validation metrics at predefined training intervals for all K

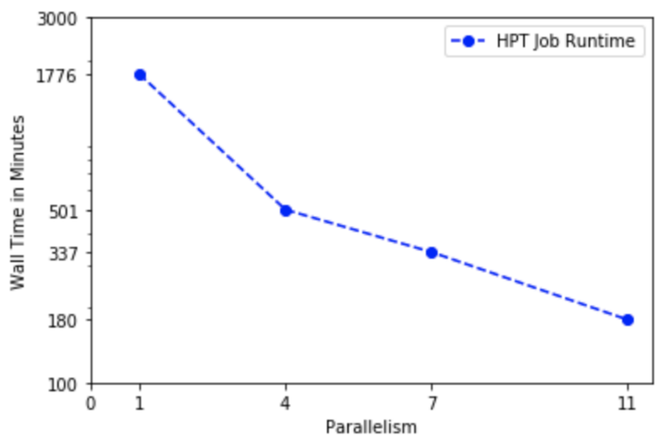

Fig. 11. Effect of increasing workers

TABLE II
EXPERIMENT SPECS FOR MODIFIED K FOLD

Training Points	5000
GPU Family	Tesla V100
GPUs per Worker Pod	2
Hyperparameter Space	$\{number\ of\ steps\}$
Method	Modified K-Fold Cross Validation
Pretrained Model	BERT Base
Training Type	All BERT layers opened for fine-tuning
Max Steps	10,000
Node Type	Pre-emptible Nodes

models. By averaging these metrics across the K models, we can obtain more reliable estimates of the model's validation error. We then choose the training step with the best validation error, and retrain the model with all the training data up to that step.

This algorithm can be parallelized up to a factor of K, which was 11 in our case. In our study, we plot the wall-clock time to complete a job against the parallelism factor which varies from 1 to 11 (Fig. 11). Other specifications for running the job via Ultron-AutoML are in Table II. We note that the wall-clock time decreases almost linearly with increase in parallelism.

C. Reproducible benchmarking of custom HPO algorithms

In this case study, we highlight the ability of the platform to benchmark any custom *HPO algorithm* in a reproducible manner. The platform was used to benchmark ReMAADE [32], a novel *HPO algorithm* against TPE [20] and Random Search. The best ML model for a regression task on the Naval Propulsion data-set [23] was discovered via the *HPO algorithm*. Each *HPO algorithm* experiment comprised 100 trials. The specifications for running the experiments are listed in Table III. The results of the benchmarking are shown in Fig. 12.

The Ultron-AutoML platform ensures reproducibility by setting seeds across the *master pod*, *worker pods* and by choosing the base *worker* image which guarantees reproducible results with *Tensorflow* and GPU computes.

TABLE III
EXPERIMENT SPECS FOR BENCHMARKING REMAADE

HPO Algorithms compared	Random, TPE, ReMAADE
Datasets	Naval Propulsion
Trials performed per Experiment	100
HP Configurations explored per Trial	100
Parallelism per Trial	8
Total Models Built	30000
GPU Family	Tesla K80
GPUs per Worker Pod	1
Node Type	Non Pre-emptible Nodes

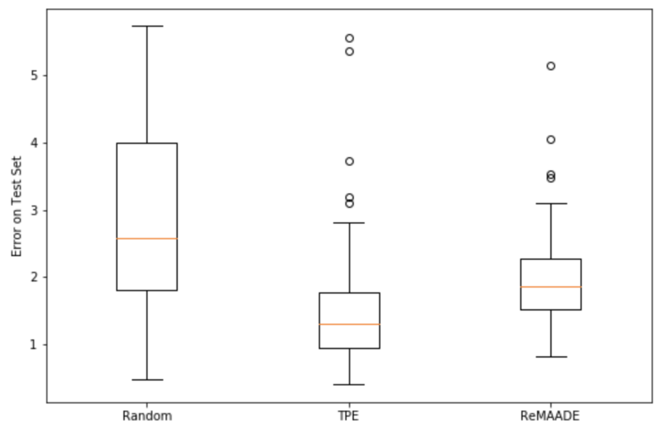

Fig. 12. Benchmarking on the Naval Propulsion Dataset

IX. FUTURE WORK

The Data Manager can be enhanced by leveraging *RAPIDS* [12] to enable efficient GPU based augmentation operations. There is scope to use *RSocket* [15], *GRPC* [7] or raw UDP sockets [42] to reduce latency of calls between the *Data Manager Client* and the *API Controller* as shown in Fig. 9. *RAPIDS* and *Apache Spark* [53] can also be leveraged to compute aggregate statistics and perform data pre-processing when required such as in UDA [52] or to normalize data of any modality. Furthermore, there is scope for building a feature where custom Extract-Transform-Load pipelines [49], pre-processing and augmentation can be injected as hyperparameters themselves.

REFERENCES

[1] Aws sagemaker. https://aws.amazon.com/sagemaker. Accessed: 2020-08-04.
[2] Azure machine learning. https://docs.microsoft.com/en-us/azure/machine-learning/. Accessed: 2020-08-04.
[3] Decrby - redis. https://redis.io/commands/decrby. Accessed: 2020-08-04.
[4] Github - nvidia/framework-determinism: Providing determinism in deep learning frameworks. https://github.com/NVIDIA/framework-determinism. Accessed: 2020-08-04.
[5] Gluster docs. https://docs.gluster.org/en/latest/. Accessed: 2020-08-04.
[6] Google cloud ai platform. https://cloud.google.com/ai-platform. Accessed: 2020-08-04.
[7] grpc. A high-performance, open source universal RPC framework. Accessed: 2020-08-04.
[8] How fast is redis. https://redis.io/topics/benchmarks. Accessed: 2020-08-04.

[9] Incrby - redis. https://redis.io/commands/incrby. Accessed: 2020-08-04.

[10] Kubernetes. https://kubernetes.io/. Accessed: 2020-08-04.

[11] Messaging that just works: Rabbitmq. https://www.rabbitmq.com/. Accessed: 2020-08-04.

[12] Open gpu data science — rapids. https://rapids.ai/. Accessed: 2020-08-04.

[13] Pytorch. https://pytorch.org/. Accessed: 2020-08-04.

[14] Redis. https://redis.io/. Accessed: 2020-08-04.

[15] Rsocket. https://rsocket.io/. Accessed: 2020-08-04.

[16] tf.data: Build tensorflow input pipelines. https://www.tensorflow.org/guide/data. Accessed: 2020-08-04.

[17] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[18] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[19] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123, 2013.

[20] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[21] Liang-Chieh Chen, Maxwell Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jon Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in neural information processing systems*, pages 8699–8710, 2018.

[22] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. *arXiv preprint arXiv:2002.05709*, 2020.

[23] Andrea Coraddu, Luca Oneto, Aessandro Ghio, Stefano Savio, Davide Anguita, and Massimo Figari. Machine learning approaches for improving condition-based maintenance of naval propulsion plants. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment*, 230(1):136–153, 2016.

[24] Thomas Desautels, Andreas Krause, and Joel W Burdick. Parallelizing exploration-exploitation tradeoffs in gaussian process bandit optimization. *Journal of Machine Learning Research*, 15:3873–3923, 2014.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[26] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[27] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

[28] Nicolo Fusi, Rishit Sheth, and Melih Elibol. Probabilistic matrix factorization for automated machine learning. In *Advances in neural information processing systems*, pages 3348–3357, 2018.

[29] Rajat P Garg, Ilya A Sharapov, and Illya Sharapov. *Techniques for optimizing applications: high performance computing*. Sun Microsystems Press Palo Alto, 2002.

[30] Philipp Hennig and Christian J Schuler. Entropy search for information-efficient global optimization. *The Journal of Machine Learning Research*, 13(1):1809–1837, 2012.

[31] Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.

[32] Chepuri Shri Krishna, Ashish Gupta, Himanshu Rai, and Swarnim Narayan. Neural architecture search with reinforce and masked attention autoregressive density estimators. *arXiv preprint arXiv:2006.00939*, 2020.

[33] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

[34] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[35] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[36] Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. Auptimizer-an extensible, open-source framework for hyperparameter tuning. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 339–348. IEEE, 2019.

[37] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are gans created equal? a large-scale study. In *Advances in neural information processing systems*, pages 700–709, 2018.

[38] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in neural information processing systems*, pages 7816–7827, 2018.

[39] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[40] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

[41] Avital Oliver, Augustus Odena, Colin A Raffel, Ekin Dogus Cubuk, and Ian Goodfellow. Realistic evaluation of deep semi-supervised learning algorithms. In *Advances in neural information processing systems*, pages 3235–3246, 2018.

[42] Jon Postel et al. User datagram protocol. 1980.

[43] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[44] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[45] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[46] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[47] Antti Tarvainen and Harri Valpola. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *Advances in neural information processing systems*, pages 1195–1204, 2017.

[48] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[49] Panos Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.

[50] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[51] Lingxi Xie and Alan Yuille. Genetic cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1379–1388, 2017.

[52] Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V Le. Unsupervised data augmentation for consistency training. *arXiv preprint arXiv:1904.12848*, 2019.

[53] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[54] Jinan Zhou, Andrey Velichkevich, Kirill Prosvirov, Anubhav Garg, Yuji Oshima, and Debo Dutta. Katib: A distributed general automl platform on kubernetes. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, pages 55–57, 2019.

[55] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnık, and Jürgen Schmidhuber. Recurrent highway networks. In *International Conference on Machine Learning*, pages 4189–4198, 2017.

[56] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[57] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.