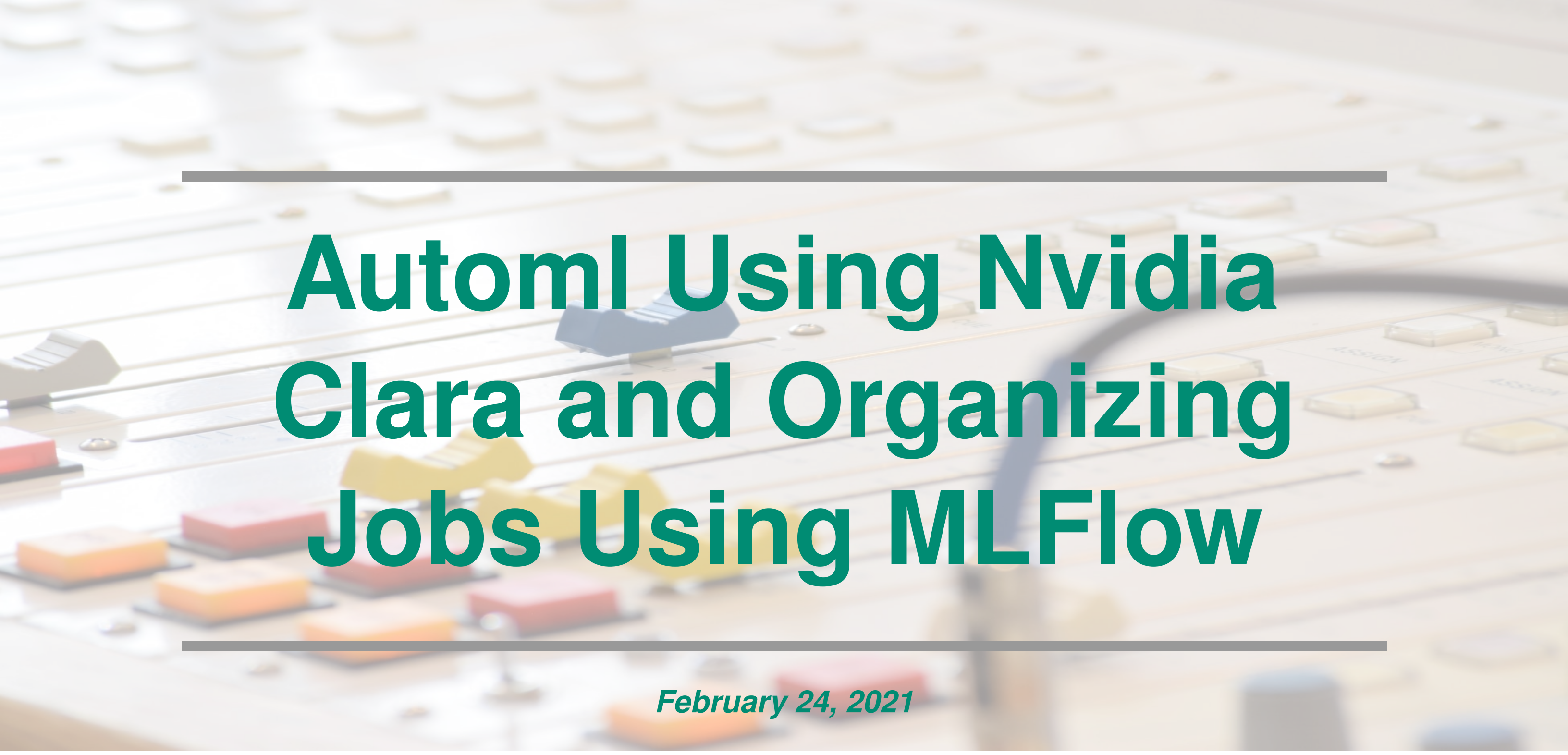

Automl Using Nvidia Clara and Organizing Jobs Using MLFlow

February 24, 2021

NVidia Clara train is a framework that attempts to provide an end-to-end workflow for all Deep Learning training needs involving Medical Imaging. From an engineering perspective, fundamentally, there are two loops in training a deep learning model, an outer loop through the hyper-parameters and an inner loop performing the weight training. The automation of the outer loop is achieved through the concept of AutoML. AutoML stands for Automated Machine Learning(AutoML). AutoML has a goal of automating every aspect of Machine Learning(ML), including data preparation, feature engineering, model selection, hyperparameter optimization, and selection of evaluation metrics. At the core, AutoML is a search problem. This search's algorithms range from heuristic manual search to brute-force grid search to highly sophisticated Bayesian optimization. The success of your research depends on how good you are at performing this search. Nvidia Clara Framework has support for both the above mentioned loops. Clara AutoML is a framework that offers help with the outerloop. Clara separates the autoML search into two parts. It calls the various areas mentioned above for AutoML as search spaces. Each search space offers capabilities to perform the search on the parameters that it calls parameter search.

While performing this search for the best model parameters, the training is run multiple times, and a run is called a job. A researcher needs to keep track of myriad things across the two loops mentioned above. MLFlow provides a platform to track all the jobs. It helps track the input data, hyperparameters for the run, metrics, and any other related files.

In this blog, I plan to explain the integration of integrate Nvidia Clara's AutoML with an ML lifecycle management tool called MLFlow. I plan to provide the context for such an integration along with the code snippets.

What is the problem? Why AutoML?

At a very high level, deep learning does curve fitting. It maps a set of data points to a function. This curve fitting is done by minimizing an expected loss from the predictions made by training the model. The success of the model training involves optimizing the model parameters or weights. This training depends on the choices regarding the data preparation, feature engineering, model selection, parameters affecting the training loop, and model evaluation. These choices are called hyperparameters. Hyperparameter optimization is a challenging problem for large models used in deep learning. The following four equations explain this search problem.

“

$$\lambda^{(*)} = \underset{\lambda \in \Lambda}{\operatorname{argmax}} \, \mathbb{E}_{x \sim G_x} \left[\mathcal{L}\Big(x; \mathcal{A}_\lambda\big(\mathbb{X}^{(train)}\big)\Big) \right] \tag{1}$$

$$\lambda^{(*)} \approx \underset{\lambda \in \Lambda}{\operatorname{argmax}} \; \underset{\lambda \in \mathcal{X}^{(train)}}{\operatorname{mean}} \; \mathcal{L}\Big(x; \mathcal{A}_\lambda\big(\mathbb{X}^{(train)}\big)\Big) \tag{2}$$

$$\lambda^{(*)} \equiv \underset{\lambda \in \Lambda}{\operatorname{argmax}} \, \Psi(\lambda) \tag{3}$$

$$\lambda^{(*)} \equiv \underset{\lambda \in \{\lambda(1) \ldots \lambda(s)\}}{\operatorname{argmax}} \quad \Psi(\lambda) \equiv \hat{\lambda} \tag{4}$$

”

in “ Random Search for Hyper-Parameter Optimization.” by James Bergstra and Yoshua Bengio[1]

Figure 1: *Mathematical formulation of AutoML Problem*

Let $\mathcal{A}$ be the model we are trying to learn, and let us further assume that this training is trying to learn a set of weights θ. Equation 1 provides the formulation of the problem of hyperparameters. Say, $\lambda^{(\star)}$ is the set of hyperparameters we are trying to figure out. For a particular λ the model is $\mathcal{A}_\lambda$. Here we can observe the two loops involved in training deep learning. There is an inner loop to arrive at the model $\mathcal{A}$ to train θ. The outer loop tries to choose the best hyper-parameters λ.[1] The outer loop is the most challenging because this optimization does not have an analytical formulation. Equation 4 shows that without an analytical formulation or knowledge of the response function Ψ or the hyperparameter search space Λ. We try to find the best λ using a limited amount of training data. As the model complexity increases, the search space for these parameters increases. Brute forcing using grid search might become computationally expensive, and manual section using domain knowledge heuristics often does not lead to these parameters' optimal choice.

“the challenge of hyper-parameter optimization in large and multilayer models is a direct impediment to scientific progress”

in “Algorithms for Hyper-Parameter Optimization.” by James Bergstra, R Bardenet, Yoshua Bengio, and Bal azs Kegl.[2]

Figure 2: *Importance of Hyperparameter Optimization*

The success of deep learning training depends on the selection of hyperparameters. It is a challenge due to the sheer size of the search space and the computational expertise needed to perform such a search. AutoML helps with this task of hyperparameter search. AutoML is the process of automating various aspects of model training. AutoML brings the promise of making model training techniques accessible to non-experts. One of the essential tasks AutoML tries to address is hyperparameter optimization. Here, hyperparameter is referred to in a general sense relating to all the knobs during the model training. These choices are in feature engineering, model architecture selection, and training hyperparameter selection.

Many libraries and frameworks are offering AutoML[4]. There are two aspects of AutoML, the algorithms enabling the search and the tools implementing the algorithms. In this blog, our

> “
> The main advantages of AutoML are
>
> 1. Democratization of Data Science
> 2. Create a strong baseline for the training pipeline
> 3. Codify best practices
> 4. Reducing the tedious part of our work, freeing time to focus on problems humans do best (creativity, interpretation, . . .)
>
> ”
>
> in “Automatic Machine Learning (AutoML): A Tutorial” by Frank Hutter and Joaquin Vanschoren[3]

Figure 3: *Advantages of Hyperparameter Optimization*

focus is on Nvidia Clara Train Framework. Clara is a framework specializing in the needs of Healthcare and Life Sciences AI development. Nvidia Clara Train focuses on deep learning for medical imaging. Clara Train SDK, starting in version3.0, introduced the AutoML module. The main advantage of using the NVidia Clara framework is that it seamlessly integrates both the deep learning training loops. The outer loop performs hyperparameter optimization through the AutoML module and the inner loop, which trains a model given a set of hyperparameters.

Nvdia Clara Train Design Philosophy

The pipeline for deep learning training provided by the Nvidia Clara train framework is based on the “Inversion of Control(IOC)” design pattern using “dependency injection(DI)” and “Event Driven Programming(EDP)”. As the name says, it inverts the control of code execution for the loops of deep learning training. IOC is about the separation of the concerns. Clara provides the skeleton developed using its engineering expertise, and it gives back control of the pipeline using two patterns called “dependency injection” and “Event-driven programming”. The framework transfers the task of creating the python objects to the researchers and using the researcher-developed object in the code flow is called dependency injection. It provides a scalable, robust, and well-tested skeleton code. It takes away the burden of doing the necessary house-keeping code to establish the hyper-parameter and the training loop, which allows the researcher to focus on the algorithm and model development. Further, the pipeline's main code knows about the researcher-developed dependencies using a configuration file called *config_train.conf*. Here comes another significant advantage of using the Clara train. In general, these dependencies are the pipeline's critical components like data transformers, models, loss functions, metrics, etc. Nvidia provides a rich library of these components out-of-the-box, which could be configured directly without writing any code. If these dependencies are developed by the user Nvidia calls it "**B**ringing **Y**our **O**wn **C**omponents" (BYOC). For researchers, the flexibility brought by BYOC is critical for using Clara for their deep learning training needs.

" Inversion of control is a design pattern where the "main" code does not control the program's execution flow. Instead the framework (caller) receives the business code as parameter and decides when and where it is executed. This allows common and reusable code being developed independently from problem-specific code, producing valuable abstraction layers. "

in "Tackling Algorithmic Skeleton's Inversion of Control." by Gustavo Pabon and Mario Leyton.[5]

Figure 4: *Inversion of Control design pattern*

Apart from the dependency injection, Clara provides access to the run-time information during the pipeline execution using event handlers. When events occur, the framework invoking hook methods on the handler objects registered in the configuration file. Another object which helps pass information from one step to another step in the pipeline is the context object. These context objects are accessible in the objects for dependency injection and in the event handler providing access to the information held during the prior steps and also pass information to the next steps in the data processing pipeline.

Inversion of Control Concerns

The very advantage of using the IOC pattern of helping the researcher from establishing the loops and the associated house-keeping might become a disadvantage. With the IOC, the control is inverted, and it is abstracted away from the user. The configuration file replaces the intuitive and straightforward logic of the loops. These configuration files are not intuitive enough, and this might add to the learning curve. Clara is not open source, so it would be hard to understand the errors and debug them. It might not be possible to implement user specific optimization to the main loop.

Nvidia tries to provides a great deal of flexibility by using DI and EDP. It also implements the most common components to follow the design principle of "Convention over configuration", allowing users to perform the deep learning training with zero code for some scenarios. This approach makes configuration choices for the users using industry best practices. There are limitations to Clara's flexibility, and care should be exercised to use the framework only for the supported use cases. For these supported cases, a lot of engineering advantages like "Automatic Mixed Precision", "Data Parallelism", "Determinism", "Smart Cache", "AutoML" and "Federated Learning" are available without much coding.

Finally, the anxiety of using a framework can be summed up by highlighting all the "Zen of Python" rules it breaks as shown in figure 5. except for one which reads **"Although practicality beats purity"**. This rule alone has immense engineering advantage and benefits explained in figure 3.

Medical Model ARchive (MMAR)

Configuration files are fundamental for implementing the IOC, DI, EDP design patterns, and the principle of "Convention over configuration". Clara uses JSON for creating these configuration files. Along with the configuration, a framework needs a project structure to organize all the artifacts. Clara Train calls this Medical Model Archive. It is a self-containing directory structure to

“

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren’t special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one– and preferably only one –obvious way to do it.
14. Although that way may not be obvious at first unless you’re Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it’s a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea – let’s do more of those!

”

in “Zen of Python” by Tim Peters[6]

Figure 5: *Zen of Python*

hold all the config files, shell scripts, documentation, other artifacts needed for housekeeping, and the generated models. Following are the advantages of this strategy.

1. It is easy to establish a project root directory and user a relative path in all the scripts and config files. This relative root path is a common strategy employed by major frameworks to achieve the portability of the project artifacts. It is easy to make a zip file of the directory structure and move it to a different location without modifying the scripts or config files.
2. It is easy to version all the artifacts together. This versioning capability is an important feature for researchers to create an experiment to track changes. Also, a unified project structure makes it easy to share this code as well as results.
3. Nvidia also uses MMAR structure to share various Clara train projects through its registry called NGC[7].
4. A standard directory structure also establishes a standard way of managing the life-cycle of a feature. It reduces the learning curve. If a researcher is familiar with Clara Train, intutivly he/she can learn how to use AutoML, Federated Learning, etc.,.
5. The project structure also makes it easy to establish governance and manage resources.

In figure 6 highlighted parts are the autoML artifacts. “automl.sh” is the main script. It launches all the jobs. “automl_train_round.sh” is the script used for launching the individual jobs. The configurations are stored in “config_automl.json”. The output is stored in a directory called automl.

Types of Hyperparameters

A framework supporting AutoML needs to provide a way to create the search space for two types of hyperparameter. The first type is individual hyperparameters, and the second type is conditional hyperparameters.

```
ROOT
    config
        config_automl.json
        config_train.json
        config_validation.json
        environment.json
    automl
        run_a
            W1_1_J1  (this a MMAR for Job 1 executed by Worker 1)
            W1_2_J3
            W2_1_J2
            W3_1_J4
            ...
        run_b
        ...
    commands
        automl.sh
        automl_train_round.sh
        set_env.sh
        train.sh
        train_finetune.sh
        train_2gpu.sh
        train_2gpu_finetune.sh
        infer.sh
        validate.sh
        export.sh
    resources
        log.config
        ...
    docs
        license.txt
        Readme.md
        ...
    models (all forms of model: checkpoint, frozen graphs, saved model, TRTIS manifest)
        model.ckpt.meta, model.ckpt.index, model.ckpt.data
        tensorboard event files
        model.frn.pb, model.trt.pb
```

Figure 6: *MMAR structure for AutoML*

Individual Hyperparameters

Following are the four types which are individual hyperparameters.

1. Continuous value type, these are of type floating-point values, for example, the learning rate.
2. Discrete value type Hyperparameters, these are of the type integer which takes a specif range of discrete values. For example, the number of splits in k-fold cross-validation.
3. Enumerate type Hyperparameters, these are a collection of discrete values of a finite

domain and unordered. For example, a set of batch sizes or they can be a set of activation functions.

4. Binary or Boolean Values, these take true or false or a value of Zero or One. For example, an indicator or flag to enable or disable a certain feature.

Conditional Hyperparameter

The framework needs to support the creation of search spaces for the parameters which depend on each other. There are the following type of dependencies

1. A hyperparameter should only be active when another hyperparameter is active.
2. Enable, Disable two hyperparameter in a mutually exclusive manner.
3. Depending on a particular hyperparameter enable or disable multiple hyperparameters

Key Components of Clara AutoML Framework

The components for Clara AutoML are an engine, controller, scheduler, executor, and handlers. High-level architecture is shown in figure 7.

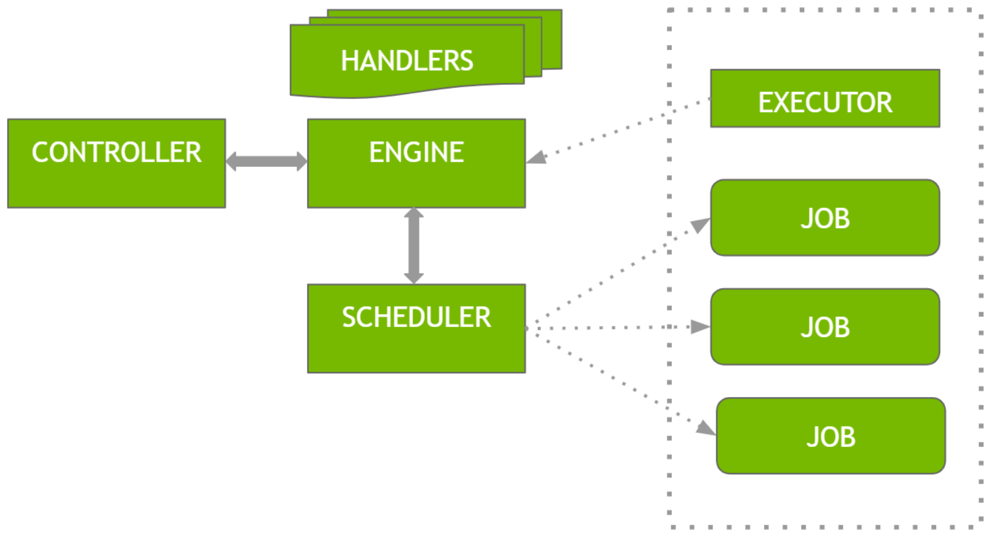

Figure 7: *High Level Architecture of Nvidia Clara AutoML*

Engine is responsible for all the coordination between the controller, schedular, and the executor. Engine is also responsible for invoking the handler hooks on the firing the events during runtime. The role of the executor is to execute the jobs scheduled by the schedular. Controller creates the search space based by implementing the algorithm for search. Some of the algorithms try to reduce the search space by adopting some heuristic stemming. The search is performed in a small batch of jobs called recommendations generated by the controller. Even for the grid search, which is a sort of brute force search, the controller tries to stop the job scheduling early if the desired result is achieved.

ReinforcementController

The Reinforcement controller is based on reinforcement learning.[8]. It separates the search space into enum subspace and float subspace. It uses reinforcement learning to generate recommendations for the float subspace. It takes these recommendations and pairs them with the enum subspace to create the final recommendations and passes them to the schedular.

Bring Your Own Components and Clara AutoML

Keep with the philosophy of “Convention over configuration” Clara, even for AutoML module provides controller, executors, schedulers, and handler. Simultaneously, all these dependencies could also be developed by the user, and DI is achieved using “config_automl.json”. The components developed by the user are called BYOC. In the case of AutoML, a complete custom AutoML system would be possible. This would help in implementing hyperparameter optimization algorithms not provided out of the box by Clara.

Clara AutoML Workflow

When the AutoML module is launched, it launchers three threads. The Engine and the controller share a thread. The scheduler runs in the second thread. Throughout the whole execution, these threads would be running. The third type of thread is for executing a run. Depending on the resources, multiple threads are launched by the scheduler.

The execution of this module starts with the executor creating the search space based on the provided configuration. This search space is handed over to the controller to generate the recommendations. These recommendations are passed on to the scheduler to launch jobs for each of the recommendations. The results are passed back to the controller to fine-tune the next set of recommendations based on the results. This forms the outer loop mentioned in section "What is the problem? Why AutoML?". This loop keeps running till all the recommendations are exhausted. This loop can also be stopped if the desired score is achieved. Each job runs the inner training loop mentioned above.

Configuring search space in "config_train.json"

Different types of hyperparameters specified in section "Types of Hyperparameters" can be configured in the "config_train.json". To configure a search space, the user needs to provide four parameters

1. "domain", it is the domain of the parameter. The supported domains are learning rate "lr", network "net", and transforms "transform".
2. the type of the hyperparameter
3. argument or the attribute
4. finally, the targets. These are the ranges of the actual values to search

In figure 8 the argument "use_amp" has boolean search space which can take value of "ture" or "false". Figure 9 shows the example for configuring search space for float value. For the float values, we need to specify the range with minimum value and maximum value. In this example, "learning_rate" takes values between 0.0001 and 0.001. Relevant portions are highlighted in the figures

```
"epochs": 2,
        "num_training_epoch_per_valid": 20,
        "train_summary_recording_interval": 10,
        "use_scanning_window": false,
        "multi_gpu": false,
        "learning_rate": 1e-3,
        "use_amp": false,
        "dynamic_input_shape": true,

        "search": [
        {
        "args": ["use_amp"],
        "type": "enum",
        "domain": "net",
        "targets": [[true], [false]]
        }
        ],
```

Figure 8: *Boolean Hyperparameter Example*

```
"epochs": 1,
        "num_training_epoch_per_valid": 20,
        "train_summary_recording_interval": 10,
        "multi_gpu": false,
        "learning_rate": 1e-3,
        "use_amp": false,
        "dynamic_input_shape": false,
        "search": [
        {
        "domain": "lr",
        "args": ["learning_rate"],
        "type": "float",
        "targets": [0.0001,0.001]
        }
        ],
```

Figure 9: *Float Hyperparameter Example*

Clara Examples

Examples are available at this github link

Figure 10 shows the example for configuring search space for a list of floating value. In this example, "poly_power" takes values 0.9 and 0.99. All the above examples are for configuring a single hyperparameter. In figure 11 conditional hyperparameter configuration is shown. In this example either "mySearchLoss1" or "mySearchLoss2"are enabled. They are mutually exclusive.

```
"lr_policy": {
        "name": "ReducePoly",
        "args": {
        "poly_power": 0.99
        },
        "search": [
        {
        "type": "float",
        "args": ["poly_power"],
        "targets": [0.9,0.99],
        "domain": "lr"
        }
        ]
        },
```

Figure 10: *Float Enum Hyperparameter Example*

```
"search": [
        {
        "domain": "transform",
        "type": "enum",
        "args": ["mySearchLoss1","mySearchLoss2"],
        "targets": [[true,false],[false,true]]
        },
        {
        "domain": "transform",
        "type": "enum",
        "args": ["mySearchOptimizer1","mySearchOptimizer2"],
        "targets": [[true,false],[false,true]]
        }
        ],
        "train": {
        "loss":[
        {
        "name": "Dice",
        "apply": {"@disabled": "mySearchLoss1"},
        "args": {
        "skip_background": true
        }
        },
        {
        "name": "Focal",
        "apply": {"@disabled": "mySearchLoss2"},
        "args": {
        "skip_background": true
        }
        }]
```

Figure 11: *Conditional Hyperparameter, Enable mutually Exclusive Example*

Integrating with MLFlow

Clara AutoML generates multiple jobs. The output for all the runs is organized in the MMAR under the directory automl. Clara does not offer a clean way to organize all the job runs for analysis.

> "MLflow is a popular open source platform for managing ML development, including experiment tracking, reproducibility, and deployment."
>
> in Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. by[9]

To organize all the AutoML Jobs, NVidia Clara is integrated with MLFlow. This integration is achieved by using a custom handler.

Handlers in Clara AutoML

Handlers are based on the Event-Driven Programming paradigm. The code in the handler is fired during the occurrence of specific events during AutoML runtime. Clara AutoML Supports the following events.

1. recommendations_available - This is fired when the recommendations are available.
2. startup - This is fired at the start of the AutoML module.
3. shutdown - This is fired when there are no more recommendations available.
4. start_job - This is fired at the start of a job run.
5. round_ended - This is fired when one round of recommendations is completed.
6. end_job - This is fired at the end of a job run.

MLFlow Integration

The strategy for the integration of Clara and MLFlow is to use two events. (1) “startup” event and (2) “end_job” event. At the beginning of the module, the experiment is set up as shown in figure 12. At the end of each job, the run was added to the experiment created in the startup event as shown in figure 13.

```
def startup(self, ctx: Context):
        mmar_root = os.getenv('MMAR_ROOT')
        mmar_root_components = mmar_root.split(os.sep)
        experiment_id = mmar_root_components[mmar_root_components.index('MMARs')-1]

        mlflow.set_tracking_uri("xxxx")
        mlClient = mlflow.tracking.MlflowClient()
        self.id = mlClient.create_experiment(name='AM'+str(experiment_id))
        mlClient.set_experiment_tag(self.id, 'mlflow.note.content', 'This is Experiment 3')
```

Figure 12: *MLFlow integration code in the startup event*

```
def end_job(self, ctx: Context):

        job_name = ctx.get_prop(ContextKey.JOB_NAME)
        parms = ctx.get_prop(ContextKey.CONCRETE_SEARCH_VALUE)
        with self.update_lock:
            with mlflow.start_run(experiment_id = self.id, run_name=job_name) as run:

                mlflow.set_tag("Description","This is a cool job with job name as "+job_name)
                mlflow.set_tag("mlflow.note.content","This is a cool job with job name as "+job_name)
                for k, v in parms.items():
                    par = k.split(":")[1]
                    attr = getattr(v, "__getitem__", None)
                    if attr is not None:
                        v1=v[0]
                    else:
                        v1=v
                    print("par=", par, " val=", v1)
                    mlflow.log_param(par, v)
                score = ctx.get_prop(ContextKey.SCORE)
                print("score =",score)
                print(mlflow.active_run().info.run_id)
                print(mlflow.active_run().info.artifact_uri)
                print(mlflow.get_artifact_uri())
                print(mlflow.get_tracking_uri())
                print(self.id)
                print ("MLFLOW added ")
                print("___________________________")
        # mlflow.end_run()

        return
```

Figure 13: *MLFlow Integration code in the end_job event*

References

[1] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. page 25.

[2] James Bergstra, R Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. page 10.

[3] Frank Hutter and Joaquin Vanschoren. Automatic Machine Learning (AutoML): A Tutorial. page 60.

[4] Wayne Wei. Windmaple/awesome-AutoML, February 2021.

[5] Gustavo Pabon and Mario Leyton. Tackling Algorithmic Skeleton's Inversion of Control. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 42–46, Munich, Germany, February 2012. IEEE. ISBN 978-1-4673-0226-5. doi: 10.1109/PDP.2012.86.

[6] The Zen of Python. https://zen-of-python.info/.

[7] Catalog | NVIDIA NGC. https://ngc.nvidia.com/catalog/collections?orderBy=scoreDESC&pageNumber=0&qu

[8] Dong Yang, Holger Roth, Ziyue Xu, Fausto Milletari, Ling Zhang, and Daguang Xu. Searching Learning Strategy with Reinforcement Learning for 3D Medical Image Segmentation. *arXiv:2006.05847 [cs]*, June 2020.

[9] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, pages 1–4, Portland OR USA, June 2020. ACM. ISBN 978-1-4503-8023-2. doi: 10.1145/3399579.3399867.