FuseMedML: a framework for accelerated discovery in machine learning based biomedicine

Alex Golts¹*, Moshe Raboh¹*, Yoel Shoshan¹*, Sagi Polaczek¹, Simona Rabinovici-Cohen¹, and Efrat Hexter¹

¹ IBM Research - Haifa, Israel ¶ Corresponding author * These authors contributed equally.

Summary

Machine Learning is at the forefront of scientific progress in Healthcare and Medicine. To accelerate scientific discovery, it is important to have tools that allow progress iterations to be collaborative, reproducible, reusable and easily built upon without “reinventing the wheel” for each task.

FuseMedML, or fuse, is a Python framework designed for accelerated Machine Learning (ML) based discovery in the medical domain. It is highly flexible and designed for easy collaboration, encouraging code reuse. Flexibility is enabled by a generic data object design where data is kept in a nested (hierarchical) Python dictionary (NDict), allowing to efficiently process and fuse information from multiple modalities. Functional components allow to specify input and output keys, to be read from and written to the nested dictionary.

Easy code reuse is enabled through key components implemented as standalone packages under the main fuse repo using the same design principles. These include fuse.data - a flexible data processing pipeline, fuse.dl - reusable Deep Learning (DL) model architecture components and loss functions, and fuse.eval - a library for evaluating ML models.

Statement of need

Medical research often involves multiple modalities (e.g., imaging, clinical data, biochemical representations) and tasks (e.g., classification, segmentation, clinical condition prediction). In our experience working on numerous such projects, we have identified three key challenges: 1. Setting up or implementing a new baseline model can be time-consuming, even when similar projects have already been completed by the same lab. 2. Transferring individual components across projects can be difficult, leading to researchers frequently “reinventing the wheel.” 3. Collaborating between projects across modalities and domains, such as imaging and molecules, is often challenging.

To address these challenges, FuseMedML was developed with the goal of simplifying and streamlining medical research projects.

Before open sourcing it, we used fuse internally in multiple research projects (Raboh, Levanony, et al., 2022), (Rabinovici-Cohen, Tlusty, et al., 2022), (Rabinovici-Cohen, Fernández, et al., 2022), (Jubran et al., 2021), (Tlusty et al., 2021), (Golts et al., 2022), (Barros et al.) and experienced significant improvement in development time, reusability and collaboration. We were also able to meaningfully measure our progress and statistical significance of our results with off-the-shelf fuse.eval components that facilitate metrics’ confidence interval calculation and model comparison. These tools have enabled us to organize two challenges as part of the 2022 International Symposium on Biomedical Imaging (ISBI) (Raboh, Golts, et al., 2022), (Pati et al., 2022).

State of the field

FuseMedML is a comprehensive machine learning library that focuses on the biomedical domain. It offers a range of tools covering the entire development process, including data preparation, model training, and evaluation. Built on top of popular machine learning frameworks such as PyTorch (Paszke et al., 2019) and PyTorch Lightning (Falcon & The PyTorch Lightning team, 2019), FuseMedML also includes flexible domain-specific capabilities to complement these frameworks. Overall, FuseMedML aims to facilitate machine learning discoveries within the healthcare and life science sectors. One way in which fuse can complement PyTorch is through its generic design concept (See Figure 1) of storing arbitrary types of data in a specialized nested dictionary. This is a key driver of flexibility, allowing minimal code modifications when moving building blocks between different projects. Concretely, fuse has a dataset class that extends the PyTorch dataset, and a model wrapper class that enables PyTorch models to operate on batch_dicts rather than tensors.

In the case of PyTorch Lightning, fuse integrates with it directly as it builds upon its comprehensive trainer class, also allowing users to define their models and data modules in PyTorch Lightning style, with flexible levels of customizability.

There are existing PyTorch-based ML libraries that similarly to fuse cater to researchers in the biomedical domain. Two examples of such prominent libraries are MONAI (Cardoso et al., 2022) and PyHealth (Zhao et al., 2021). MONAI is primarily focused on medical imaging applications. PyHealth on the other hand mainly focuses on health records data. fuse is designed to support different types of medical data and multimodal use cases involving imaging, clinical and biochemical data.

As with generic ML frameworks like PyTorch and PyTorch Lightning, fuse can also coexist with the more specific libraries like MONAI, PyHealth or others. A user may opt to borrow parts from different libraries and complement them with components from fuse. As another example, a user may want to use the data ops of fuse which are generic and flexible, or its data caching mechanism, which allows to separate processing into a static and dynamic pipelines, controlling the desired stages to be cached.

Figure 1: This figure illustrates FuseMedML’s design concept. A fuse component is instantiated with input and output keys. These keys refer to the sample_dict, the basic data sample structure of fuse represented by a special nested Python dictionary called “NDict”.

Packages

fuse.data

FuseMedML’s data package is designed for building a flexible and powerful data pipeline with reusable building blocks called ops. See Figure 2 for a simple example for how such a building block can be used across different projects.

Each op class’s __call__ function gets as an input a sample_dict, a dictionary that stores all the necessary information about a sample processed so far. Typically, an op’s constructor gets keys that specify what it should consider in sample_dict and where to store the output. Similarly, a minibatch is represented by a batch_dict.

A special kind of ops are “Meta ops”. They can be thought of as a form of wrapper op around

a regular, lower level op or function, to help achieve a special behavior such as repeating that low level op, applying it with random values and more. “Meta ops” also help avoid writing boilerplate code.

A data pipeline may consist of a static_pipeline and a dynamic_pipeline. The output of the static_pipeline can be cached to optimize running time and GPU utilization. The dynamic_pipeline is responsible for “online” processing that we don’t want to cache, such as random augmentations. An instance of a fuse dataset class, which inherits from the PyTorch dataset class is then created from defined static and dynamic pipelines.

The data package also includes generic utilities such as a PyTorch based sampler enabling batch class balancing and a tool for splitting data into folds according to predefined criteria.

![Project A and Project B](example.png)

**Figure 2**: In this example a medical image loader is the fuse component reused in projects A and B. Different projects can have different formats for their data samples, but they can all use OpMedicalImageLoader by providing the appropriate key names when calling it. In Project B the same key name is used for the input and output, resulting in the loaded image data overriding the image paths in the updated sample.

**fuse.dl**

FuseMedML’s DL package works with PyTorch models, only modifying them to interact with a batch_dict. For training, fuse.dl utilizes PyTorch-Lightning, either through an already made LightningModuleDefault class that inherits from Pytorch-Lightning’s LightningModule class, or by allowing users who seek maximal customizability to implement their own custom LightningModule and operate in close resemblance to the standard PyTorch-Lightning workflow or use alternative training loop implementations.

fuse.dl also offers generic core DL components such as model architectures and losses, implemented in fuse style. See an example model architecture definition in Figure 3.
Figure 3: In this example a model architecture is defined using the ModelMultiHead class. It contains a 3D ResNet backbone represented by the BackboneResnet3D class and a 3D classification head represented by the Head3D class. Note the user can define a list of heads, to support a multi task use case. The inputs to the backbone and classification heads are defined in the fuse style described earlier, using the batch_dict key names with the relevant data. This enables easy reuse of similar model architectures between projects.

**fuse.eval**

FuseMedML’s evaluation package is a standalone library for evaluating machine learning models using various performance metrics and comparing the results between models. It offers advanced capabilities such as a generic confidence interval wrapper for any metric, a generic one-versus-all extension for converting any binary metric to a multi-class scenario, and metrics for comparing models while considering statistical significance. The package also includes model calibration tools and a pipeline for combining a sequence of metrics with possible dependencies. In addition, the evaluation package supports automatic per-fold evaluation and subgroup analysis, and can handle large data sets through batching and multiprocessing. See Figure 4 for an example of an evaluation metric pipeline that can be reused across projects.

```python
ModelMultiHead(
    conv_inputs=('data.input.img',),
    backbone=BackboneResnet3D(in_channels=1),
    heads=[
        Head3D(head_name='classification',
               mode='classification',
               conv_inputs=["model.backbone_features", 512])
    ]
)
```

**Figure 4:** In this example a pipeline of evaluation metric components is shown. It consists of two metrics: the Area Under the receiver operating characteristic Curve and the Area Under the Precision-Recall Curve. Both metrics are wrapped with a Confidence Interval (CI) metric, resulting in a lower and upper bound for each metric. The metrics are executed by an instance of the EvaluatorDefault class, the basic fuse.eval class that combines input sources, evaluates using the specified metrics, generates a report and returns a dictionary with all the metrics results.

```python
data = {"pred": [0.1, 0.2, 0.6, 0.7, 0.8, ...],
       "target": [0, 0, 1, 1, ...] }
metrics = OrderedDict(
    [("auc", CI(MetricAUCROC(pred="pred", target="target"), stratum="target")),
     ("auc_pr", CI(MetricAUCCPR(pred="pred", target="target"), stratum="target"))
     ]
)
evaluator = EvaluatorDefault()
results = evaluator.eval(ids=None, data=data, metrics=metrics)
```

**Figure 4:** In this example a pipeline of evaluation metric components is shown. It consists of two metrics: the Area Under the receiver operating characteristic Curve and the Area Under the Precision-Recall Curve. Both metrics are wrapped with a Confidence Interval (CI) metric, resulting in a lower and upper bound for each metric. The metrics are executed by an instance of the EvaluatorDefault class, the basic fuse.eval class that combines input sources, evaluates using the specified metrics, generates a report and returns a dictionary with all the metrics results.

Extensions

The core technology of FuseMedML and its component packages is general, while domain-specific functionality is contained within extensions. One such extension, fuse-imaging, is currently available and extends the FuseMedML data package with operations useful for medical imaging, as well as implementations of public medical datasets.

References


