mlsquare Documentation

MLSquare Foundation

Oct 20, 2021

Contents

MLSquare is an open source developer-friendly Python library, designed to make use of Deep Learning for Machine Learning developers.

Note: `mlsquare` python library is developed and maintained by MLSquare Foundation

---

In the first version we have developed **Dope**. `Dope` is aimed to provide every Machine Learning Algorithm with an equivalent DNN Implementation. The `dope` feature available in the `mlsquare` framework is aimed at making interoperability between machine learning models easier.

CHAPTER 1

Getting Started!

Setting up `mlsquare` is simple and easy

1. Create a Virtual Environment

```
virtualenv ~/.venv
source ~/.venv/bin/activate
```

2. Install `mlsquare` package

```
pip install mlsquare
```

3. Import `dope()` function from `mlsquare` and pass the `sklearn` model object.

```
>>> from mlsquare import dope
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.model_selection import train_test_split
>>> import pandas as pd
>>> from sklearn.datasets import load_diabetes

>>> model = LinearRegression()
>>> diabetes = load_diabetes()

>>> X = diabetes.data
>>> sc = StandardScaler()
>>> X = sc.fit_transform(X)
>>> Y = diabetes.target
>>> x_train, x_test, y_train, y_test =
    train_test_split(X, Y, test_size=0.60, random_state=0)

>>> m = dope(model)

>>> # All sklearn operations can be performed on m, except that the␣
↪underlying implementation uses DNN
```

(continues on next page)

(continued from previous page)

```
>>> m.fit(x_train, y_train)
>>> m.score(x_test, y_test)
```

Note: For a comprehensive tutorial please do checkout this link

CHAPTER 2

Contents

2.1 Installation Guide

This guide describes how to install *mlsquare*

On this page

2.1.1 Setting up a virtual environment

The recommended way to install `mlsquare` is to use a virtual environment created by `virtualenv`. Setup and activate a new virtual environment like this:

```
$ virtualenv envname
$ source envname/bin/activate
```

If you use the `virtualenvwrapper` scripts, type this instead:

```
$ mkvirtualenv envname
```

2.1.2 Installing the `mlsquare` package

The next step is to install `mlsquare`. The easiest way is to use *pip* to fetch the package from the Python Package Index (PyPI). This will also install the dependencies for Python.

```
(envname) $ pip install mlsquare
```

Note: Installation via `pip` installs the stable version in your environment. To install the developer version checkout the package source from GitHub and run `python setup.py install` from the directory root. Note that developer version is not stable and there are chances that code will break. If you are not sure about it, we suggest you use the stable version.

2.1.3 Testing the installation

Verify that the packages are installed correctly:

```
(envname) $ python -c 'import mlsquare'
```

If you don't see any errors, the installation was successful. Congratulations!

Next steps

Now that you successfully installed HappyBase on your machine, continue with the *User Guide* to learn how to use it.

2.2 User Guide

This user guide explores the MLSquare API and should provide you with enough information to get you started. Note that this user guide is intended as an introduction to MLSquare, not to Keras or SkLearn or any other packages in general. Readers should already have a basic understanding of the packages they were using and its API.

While the user guide does cover most features, it is not a complete reference guide. More information about the MLSquare API is available from the *API documentation*.

On this page

2.2.1 Importing the `mlsquare` module

To start using the package, we need to import the module into the python enviroment.

```
>>> import mlsquare
```

If the above command doesn't result in any errors, then the import is successful

Note: To use `mlsquare` you need *Python* 3.6 or higher

2.2.2 Load `dope()` method into the enviroment

`dope()` is the base function, that returns an implementation of a given model to its DNN implementation. Once a model is dope'd, users will be able to use the same work flow as their initial model on the dope'd object.

```
>>> from mlsquare import dope
```

2.2.3 Transpiling an existing model using *dope*

To demonstrate `dope()`, we will transpile `sklearn LinearRegression` and use the `sklearn` operations on the transpiled model.

```
>>> from sklearn.linear_model import LinearRegression
>>> model = LinearRegression()
>>> m = dope(model)

# Dope maintains the same interface as the base model package
>>> m.fit(x_train, y_train)
>>> m.score(x_test, y_test)
```

Note: `dope()` function doesn't support all the packages and the models in the package. A list of supported packages and models is available at the *Supported Modules and Models*

2.3 Developer Guide

2.3.1 Getting Started

Git and Github

Our development process heavily relies on Git and Github. If you're unfamiliar with Git or Github workflow, a good place to start would be with this guide.

Slack

To get directly in touch with the team and ML Square community, you're encouraged to join our Slack channel - https://mlsquare.slack.com/

2.3.2 Setup

Forking a repository

To ensure a risk-free environment to work with, you will have to fork the mlsquare repository. Once you have forked the repository, you can call `git fetch upstream` and `git pull 'branch-name'` before you make any local. This will ensure that your local repository is up-to-date with the remote repository.

Syncing your forked repository

Please refer to this guide if you face difficulties in syncing your fork with the `mlsquare` repository.

Installing mlsquare after cloning repository

After forking and updating your local repository, you might want to do the following to install the local repository version. This will help you in testing changes you make to the repository.

```
cd path-to-local-repo/mlsquare
python setup.py develop
```

2.3.3 Adding an algorithm

This is for users interested in adding or contributing custom algorithms. Follow the below mentioned steps to contribute to the existing collection of algorithms available in mlsquare.

Where to add?

Navigate to *mlsquare.architectures* folder. Choose your primal module(say sklearn.py). The *architectures* folder consists of all existing algorithm mappings. Each .py file in this folder represents a primal module.

Implementation

1. Each algorithm is expected to be declared as a class. An algorithm should be registered in the `registry` using the `@registry.register` decorator.
2. Use the base class available in base.py as the parent class for your algorithm. Feel free to use an already existing base class(ex - glm) if it matches your algorithm's needs.
3. **The following methods and attributes are expected to implemented while creating a new model,**
    - `create_model()` - Your model's architecture lies in this method. Calling this method would return a compiled dnn model(ex - keras or pytorch model).
    - `set_params()` - The conventions followed by mlsquare in defining model parameters are mentioned below. This method should handle the "flattening" of parameters.
    - `get_params()` - Calling this method should simply return the models existing parameters.
    - `update_params()` - This method should enable updating the model parameters for an instantiated model.
    - `adapter` - This attribute should contain the adapter choice you have made for your algorithm.
    - `module_name` - The primal module name(should be a string)
    - `name` - Name that you wish the model should be reffered by.
    - `version` - If an implementation already exists for your algorithm and you wish to improve it by a different implementation, make sure you add a meaningful version number.
    - `model_params` - The parameters required to compile your model. Conventions to be followed are mentioned below.

Notes on conventions

1. Currently mlsquare supports keras as the backend for proxy models. The convention we follow is similar to that of keras with some minor changes.
2. The parameters should be defined as a dictionary of dictionaries. The first level of dict should represent each layer. Each layer should be followed by the index of the layer.
3. **Sample parameter - This sample dict shows the parameters for a keras model with 2 layer(both hidden and visible),**

```
model_params = {'layer_1': {'units': 1, 'activation': 'sigmoid'},
                'layer_2': {'activation':'softmax'}
                'optimizer': 'adam',
                'loss': 'binary_crossentropy'
                }
```

Sample implementation

1. To get started, create a base model

```
class MyBaseModel(GeneralizedLinearModel):
 def create_model(self, **kwargs):
   ## To parse your model from 'flattened' to 'nested'
   model_params = _parse_params(self._model_params, return_as='nested')

   model = Sequential()

   ## Define your model
   model.add(Dense(units=model_params['layer_1']['kernel_dim'],
                   trainable=False, kernel_initializer='random_normal',
↪# Connect with sklearn_config
                   activation=model_params['layer_1']['activation']))
   model.add(Dense(model_params['layer_2']['units'],
                   activation=model_params['layer_2']['activation']))
   model.compile(optimizer=model_params['optimizer'],
                 loss=model_params['loss'],
                 metrics=['accuracy'])

   return model
```

The above class inherits from the existing *GeneralizedLinearModel*. For most use cases, this would be sufficient, unless you wish to overwrite the *set_params()* method.

```
@registry.register
class MyModel(MyBaseModel):
    def __init__(self):
        # Import the adapter
        self.adapter = MyAdapter
        self.module_name = 'PrimalModuleName'
        self.name = 'ModelName'
        self.version = 'default'
        model_params = {'layer_1': {'units': 10,
                                    'activation': 'linear'
                                    },
                        'layer_2': {
                                    'activation': 'softmax'
```

(continues on next page)

(continued from previous page)

```
                                    },
                        'optimizer': 'adam',
                        'loss': 'categorical_hinge'}

        ## Make sure you call this method after the params are defined
        self.set_params(params=model_params, set_by='model_init')
```

Note:

1. Please make sure that you "register" your model in the registery by using the @register.registry decorator.
2. Define all mandatory attributes mention earlier in your model's *__init__()* method.
3. Set your params once you have finalized using the *set_params()* method.

Writing test cases

Please make sure that test cases are written with atleast 90% coverage for each new algorithm added. `mlsquare` utilizes `pytest` to execute test cases. Test cases should be added to the tests folder to corresponding module's file. For example, test cases for a newly added algorithm would be defined in the test_architectures.py file. Please feel free to reachout for help via our Slack channel if you face any difficulties in writing or understanding test cases.

Once you have completed your test cases, you can run them using the following commands

```
cd path-to-local-repo/mlsquare
python setup.py test
```

Creating a Pull Request

When the required additions are made and sufficient test cases are added, please raise a Pull Request. Always make sure that raise your Pull Requests to the `dev` branch. Please add `[WIP]` to the title on PRs that are not complete and is still work in progress.

Check CI and wait for review

All commits undergo an automated check by CircleCI. This ensures build checks and executes test cases.

Reviews would be done only on commits that pass the CircleCI checks.

Do not worry if the checks fail. Failing the CI checks will not close the PR by default. You can always cross-check what went wrong in CircleCI feedback and fix the errors and update the PR.

FAQs

1. **What do you mean by "transpliling" a model?** Model transpilation in mlsquare's context refers to converting a given model to it's neural network equivalent.
2. **What is a primal model?** A primal model is model that you wish to transpile to a neural network model.
3. **What is a proxy model?** The proxy model refers to the intermediate state that a primal undergoes to transpile itself to a neural network model.
4. **What is Registry and what is it used for?** *mlsquare* maintains a registry of the model mappings defined in the architectures folder. This is to ensure easy access of models at point.

2.4 API Reference

2.5 License

2.6 Contributors

- Soma S. Dhavala
- Shakkeel Ahmed
- Ravi S. Mula
- Prakash Bisht

We hear your feedback!

If you notice any issues during the usage the package `mlsquare`, please create an issue on GitHub. Before creating any issue, please check if the same issue was already created by any other user.

2.7 Creating a new Issue

1. To create a new issue, navigate to the github page of the project and create an issue from the issues column
2. Include a short title.
3. Include the error generated.
4. Include the steps to reproduce it.

2.8 Changelog

2.8.1 Version 0.1

- Support for Linear & Logistic Regression from Sklearn

2.9 Supported Modules and Models

As of the current release, `mlsquare` supports the following models from the below modules

- **`sklearn`**
  - `LinearRegression`
  - `LogisticRegression`

We are working supporting more models and modules, however if you would like us to add any module, please write to us at info[at]mlsquare.org

CHAPTER 3

External links

- Online documentation (Read the Docs)
- Downloads (PyPI)
- Source code (Github)

CHAPTER 4

Indices and tables

- genindex
- search