# laymon Documentation

***Release [1.0.1]***

**Shubham Gupta**

**Aug 22, 2020**

# Contents

PyLaymon or laymon is a tool for monitoring and visualizing feature maps for a neural network. Currently, the PyLaymon is limited to the pytorch framework.

CHAPTER 1

# About

PyLaymon[1] is a python based package which is used for monitoring and visualizing the layers of a neural network. The motivation behind the project was to implement a tool that assists developers envision the different layers of a neural network during the training process. This will aid the developers in better understanding the patterns learned at various layers of the neural network. Hence ease the developers in determining which layers require fine-tuning and improve the outcome expected from the neural network. This will also help developers identify any biases in the network during the training process.

Currently, the package implements the functionality to monitor and visualize the feature maps of a neural network. By default, the feature maps are displayed using the matplotlib library. However one can create their custom display classes. For more information see the Usage tab.

[1] The package only supports the Pytorch framework.

CHAPTER 2

# Installation

## 2.1 Stable release

To install PyLaymon, run this command in your terminal:

```
$ pip install laymon
```

This is the preferred method to install PyLaymon, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

## 2.2 From sources

The sources for PyLaymon can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/shubham3121/laymon
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

# Usage

## 3.1 How to use *laymon* to monitor layers

To use PyLaymon in a project:

```
# Import laymon
import laymon

# Initialize the feature map monitoring class
fMonitor = laymon.FeatureMapMonitoring()

# Add your model whose feature maps are to be monitored.
fMonitor.add_model(net)

# To start monitoring, call the `start` method during training of the model.
fMonitor.start()
```

To get the list of layers being monitored:

```
fMonitor.monitor.get_registered_observers()
```

Instead of adding the whole model, one can add specific layers, whose feature maps are to be monitored.:

```
fMonitor.add_layer(net.conv2, 'conv2')
```

To remove a layer from the monitoring:

```
fMonitor.remove_layer(net.conv2)
```

## 3.2 Example

Below is a sample example:

```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from laymon import FeatureMapMonitoring

# Creating a neural network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Initializing the neural network
net = Net()

# Initialize the feature map monitor
fMonitor = FeatureMapMonitoring()

# Add the neural network model to be monitored
fmonitoring.add_model(net)

# Add your loss and optimizers
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(1):  # loop over the dataset multiple times

running_loss = 0.0
for i, data in enumerate(trainloader, 0):
    # get the inputs; data is a list of [inputs, labels]
    inputs, labels = data

    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)

    # Start monitoring the feature maps
    fmonitoring.start()

    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

(continues on next page)

(continued from previous page)

```
    # print statistics
    running_loss += loss.item()
    if i % 2000 == 1999:    # print every 2000 mini-batches
        print('[%d, %5d] loss: %.3f' %
              (epoch + 1, i + 1, running_loss / 2000))
        running_loss = 0.0

print('Finished Training')
```

The full example notebook can be found here.

## 3.3 Custom Displays

One may feel the need to have a custom display class for visualizing the feature maps. One the applications could be displaying additional meta info or using a third partly application for visualization (e.g. Grafana) instead of the matplotlib library.

In order to accomplish this one can write his/her own custom display class and plugin it into the FeatureMapMonitoring instance.

```
# Import the Display abstract class
from laymon.interfaces import Display
from laymon import FeatureMapMonitoring

# Create your own custom class by inheriting
# and implementing the methods of the Display class.

class MyCustomDisplay(Display):
    def custom_method(self, params):
        // My Custom method

    def update_display(self, parameters, display_title):
        // Implement this method.
        // This method needs to be implemented,
        // as this method is invoked by the observers/layer object being monitored to
        // send the updated parameters to the display function
        // Your Custom Logic.
        // Calling your custom methods.


# Create instance of the FeatureMapMonitoring class
f_map_monitor = FeatureMapMonitoring()

# Overwrite the display class with your custom display class.
f_map_monitor.observer_factory.display_object = MyCustomDisplay

# Now the observers/layers being monitored point to your custom display method.
```

CHAPTER 4

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 4.1 Types of Contributions

### 4.1.1 Report Bugs

Report bugs at https://github.com/shubham3121/laymon/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 4.1.4 Coding style

Please follow these coding conventions when writing code for inclusion in Scrapy:

- Unless otherwise specified, follow **PEP 8**.
- It's OK to use lines longer than 79 chars if it improves the code readability.
- Don't put your name in the code you contribute; git provides enough metadata to identify author of the code. See https://help.github.com/en/github/using-git/setting-your-username-in-git for setup instructions.

### 4.1.5 Write Documentation

PyLaymon could always use more documentation, whether as part of the official PyLaymon docs, in docstrings, or even on the web in blog posts, articles, and such.

### 4.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/shubham3121/laymon/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

### 4.1.7 Tests

Currently, tests are not implemented for this package. Feel free to implement them.

## 4.2 Get Started!

Ready to contribute? Here's how to set up *laymon* for local development.

1. Fork the *laymon* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/laymon.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv laymon
$ cd laymon/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 laymon tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

## 4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy.

CHAPTER 5

# Authors


- Shubham Gupta (LinkedIn)

# CHAPTER 6

# API Reference

This page contains auto-generated API reference documentation[1].

## 6.1 laymon

Top-level package for PyLaymon.

### 6.1.1 Submodules

#### laymon.displays

#### Module Contents

#### Classes

| | |
|---|---|
| *FeatureMapDisplay*() | A class for defining methods for displaying the parameters being monitored by a FeatureMapObserver. |

**class** laymon.displays.**FeatureMapDisplay**

Bases: *laymon.interfaces.Display*

A class for defining methods for displaying the parameters being monitored by a FeatureMapObserver.

**display_params**(*self*, *activation*, *max_subplots=5*)

Method for updating the subplots and figure with the new parameters (activation maps).

**_show**(*self*)

**update_display**(*self*, *parameters*, *display_title*)

[1] Created with sphinx-autoapi

Updates the display with the new parameters :param parameters: Tensor (activation map params) :param display_title: Title of the figure

## laymon.exceptions

### Module Contents

**exception** laymon.exceptions.**LayerRegisterException**(*layer_name*, *message=default_message*)

Bases: Exception

Exception to indicate the observer that failed to register for generating feature maps.

**default_message = Failed to register the layer:**

**__str__**(*self*)
Return str(self).

**exception** laymon.exceptions.**SingleDimensionalLayerWarning**(*layer_name*, *message=default_message*)

Bases: Warning

Warning to indicate that the layer trying to be visualized is a single dimensional layer.

**default_message = Layer is a single dimensional layer. Some of the displays might not**

**__str__**(*self*)
Return str(self).

## laymon.interfaces

### Module Contents

### Classes

| | |
|---|---|
| *Monitor*() | Abstract class to monitor the observers. |
| *Observer*() | Abstract class to describe and update the state of an observer. |
| *Display*() | Abstract class to create and update the displays attached to an observer. |
| *ObserverFactory*() | Abstract class for creating a factory to create new observers with a display attached to them. |

**class** laymon.interfaces.**Monitor**
Abstract class to monitor the observers. Methods to:

1. Add observers
2. Remove observers
3. Get the list of registered observers
4. Notify the observers when there is a change in the monitored parameters.

**__metaclass__**

**add_observer**(*self*, *observer*)

**remove_observer**(*self*, *observer*)

**notify_observers**(*self*)

**get_registered_observers**(*self*)

**class** laymon.interfaces.**Observer**
Abstract class to describe and update the state of an observer.

**__metaclass__**

**_description**

**update**(*self*, *parameters*)

**get_description**(*self*)

**class** laymon.interfaces.**Display**
Abstract class to create and update the displays attached to an observer.

**__metaclass__**

**update_display**(*self*, *parameters*, *display_title*)

**class** laymon.interfaces.**ObserverFactory**
Abstract class for creating a factory to create new observers with a display attached to them. The *display_object* stores a display class used to visualize an observer's parameters/states.

**__metaclass__**

**display_object**

**create**(*self*, *observer*, *observer_name*)

## laymon.monitor

### Module Contents

### Classes

| | |
|---|---|
| *ObserverHookObject*(kwargs) | AN object used to store: |
| *FeatureMapMonitor*() | A monitor type class for visualizing the feature maps of a neural network. |

**class** laymon.monitor.**ObserverHookObject**(*kwargs*)
AN object used to store: 1. The observer object which is being hooked 2. Parameters being monitored 3. Handler of the hooked layer

**class** laymon.monitor.**FeatureMapMonitor**
Bases: *laymon.interfaces.Monitor*

A monitor type class for visualizing the feature maps of a neural network.

**add_observer**(*self*, *layer_observer*)

1. Creates a layer observer object.
2. Hooks the layer to capture the activation map of the layer.
3. Adds the observer object to the list of monitored observers.

**Parameters** **layer_observer** – Observer object

**remove_observer**(*self*, *layer_observer=None*, *layer_name=None*)

**If the layer observer/layer name is in the list of monitored observers:**

1. Unhook the layer.
2. Remove the layer observer from the list of monitored observers.

**Parameters**

- **layer_observer** – Layer Observer.
- **layer_name** – Name of the layer being monitored.

**Returns** True/False based on whether the observer by deleted or not.

**static _is_layer_single_dim**(*layer*)
Checks if the layer is a single dimensional layer

**_get_activation_map**(*self*, *layer_name*)
Hooks the layer to capture activation maps for the given layer and return the handler to the hook

**notify_observers**(*self*)
Updates all the observers being monitored with the new parameters

**get_registered_observers**(*self*)
Returns the list of observers being monitored.

## laymon.monitoring

### Module Contents

### Classes

| | |
|---|---|
| *FeatureMapMonitoring*() | A wrapper class for adding a model or model layers for monitoring the feature maps |

**class** laymon.monitoring.**FeatureMapMonitoring**
Bases: object

A wrapper class for adding a model or model layers for monitoring the feature maps during training of the model.

**add_layer**(*self*, *layer*, *layer_name*)
Adds the layer whose feature maps are to be monitored. :param layer: pyTorch layer :param layer_name: (str) name of the layer :return: the observer object of the layer being monitored

**_add_layer**(*self*, *layer*, *layer_name*)
Create a observer class of the layer to be monitored and adds it to the list of observers being monitored. :param layer: pyTorch layer :param layer_name: (str) name of the layer :return: Observer object of the layer

**_remove_layer**(*self*, *layer_name*)
Removes an observer from the list of observers being monitored. :param layer_name: name of the observer :return: True if observer was deleted, else False

**remove_layer**(*self*, *layer_name*)
Remove the layer from list of layer being monitored. :param layer_name: str (name of the layer) :return: True if the layer is deleted from the list of monitored objects, else False

**add_model**(*self*, *model*)
Registers all the layers a pyTorch model whose activations maps are to monitored. :param model: :return:

**start**(*self*)
Starts monitoring the feature maps of the registered layers/model.

### laymon.observers

#### Module Contents

#### Classes

| | |
|---|---|
| *FeatureMapObserver*(layer, layer_name, update_display) | An class used to create observers that are used to monitor the feature maps of the given layer. |
| *FeatureMapObserverFactory*() | A factory type class to create a FeatureMapObserver for the given layer |

**class** laymon.observers.**FeatureMapObserver**(*layer*, *layer_name*, *update_display*)
Bases: *laymon.interfaces.Observer*

An class used to create observers that are used to monitor the feature maps of the given layer.

**update**(*self*, *parameters*)
Update the display attached to the observer with the new parameters/activations. :param parameters: Tensor :return: None

**get_layer_name**(*self*)
Returns the layer name being observed.

**get_layer**(*self*)
Returns the layer object being observed.

**class** laymon.observers.**FeatureMapObserverFactory**
Bases: *laymon.interfaces.ObserverFactory*

A factory type class to create a FeatureMapObserver for the given layer

**display_object**

**create**(*self*, *layer*, *layer_name*)
Create a FeatureMapObserver for the given layer and attaches the display function for the layer being monitored. :param layer: :param layer_name: :return:

## 6.1.2 Package Contents

#### Classes

| | |
|---|---|
| *FeatureMapMonitoring*() | A wrapper class for adding a model or model layers for monitoring the feature maps |

Continued on next page

Table 6 – continued from previous page

| | | |
|---|---|---|
| *FeatureMapObserver*(layer, layer_name, update_display) | | An class used to create observers that are used to monitor the feature maps of the given layer. |
| *FeatureMapDisplay*() | | A class for defining methods for displaying the parameters being monitored by a FeatureMapObserver. |
| *FeatureMapMonitor*() | | A monitor type class for visualizing the feature maps of a neural network. |

**class** laymon.**FeatureMapMonitoring**

Bases: object

A wrapper class for adding a model or model layers for monitoring the feature maps during training of the model.

**add_layer**(*self*, *layer*, *layer_name*)

Adds the layer whose feature maps are to be monitored. :param layer: pyTorch layer :param layer_name: (str) name of the layer :return: the observer object of the layer being monitored

**_add_layer**(*self*, *layer*, *layer_name*)

Create a observer class of the layer to be monitored and adds it to the list of observers being monitored. :param layer: pyTorch layer :param layer_name: (str) name of the layer :return: Observer object of the layer

**_remove_layer**(*self*, *layer_name*)

Removes an observer from the list of observers being monitored. :param layer_name: name of the observer :return: True if observer was deleted, else False

**remove_layer**(*self*, *layer_name*)

Remove the layer from list of layer being monitored. :param layer_name: str (name of the layer) :return: True if the layer is deleted from the list of monitored objects, else False

**add_model**(*self*, *model*)

Registers all the layers a pyTorch model whose activations maps are to monitored. :param model: :return:

**start**(*self*)

Starts monitoring the feature maps of the registered layers/model.

**class** laymon.**FeatureMapObserver**(*layer*, *layer_name*, *update_display*)

Bases: *laymon.interfaces.Observer*

An class used to create observers that are used to monitor the feature maps of the given layer.

**update**(*self*, *parameters*)

Update the display attached to the observer with the new parameters/activations. :param parameters: Tensor :return: None

**get_layer_name**(*self*)

Returns the layer name being observed.

**get_layer**(*self*)

Returns the layer object being observed.

**class** laymon.**FeatureMapDisplay**

Bases: *laymon.interfaces.Display*

A class for defining methods for displaying the parameters being monitored by a FeatureMapObserver.

**display_params**(*self*, *activation*, *max_subplots=5*)

Method for updating the subplots and figure with the new parameters (activation maps).

**_show**(*self*)

**update_display**(*self*, *parameters*, *display_title*)
Updates the display with the new parameters :param parameters: Tensor (activation map params) :param display_title: Title of the figure

**class** laymon.**FeatureMapMonitor**
Bases: *laymon.interfaces.Monitor*

A monitor type class for visualizing the feature maps of a neural network.

**add_observer**(*self*, *layer_observer*)

1. Creates a layer observer object.
2. Hooks the layer to capture the activation map of the layer.
3. Adds the observer object to the list of monitored observers.

**Parameters** **layer_observer** – Observer object

**remove_observer**(*self*, *layer_observer=None*, *layer_name=None*)

**If the layer observer/layer name is in the list of monitored observers:**

1. Unhook the layer.
2. Remove the layer observer from the list of monitored observers.

**Parameters**

- **layer_observer** – Layer Observer.
- **layer_name** – Name of the layer being monitored.

**Returns** True/False based on whether the observer by deleted or not.

**static _is_layer_single_dim**(*layer*)
Checks if the layer is a single dimensional layer

**_get_activation_map**(*self*, *layer_name*)
Hooks the layer to capture activation maps for the given layer and return the handler to the hook

**notify_observers**(*self*)
Updates all the observers being monitored with the new parameters

**get_registered_observers**(*self*)
Returns the list of observers being monitored.

laymon.**__author__ = Shubham Gupta**

laymon.**__email__ = shubhamgupta3121@gmail.com**

laymon.**__version__ = 1.0.0**

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## l

# Index

## Symbols



## A



## C



## D

## F



## G



## L



## M



## N



## O



## P



## R



## S



## U