Classification Report

Release 1.0.0

Apr 01, 2020

Detailed Features

1 Features **3**

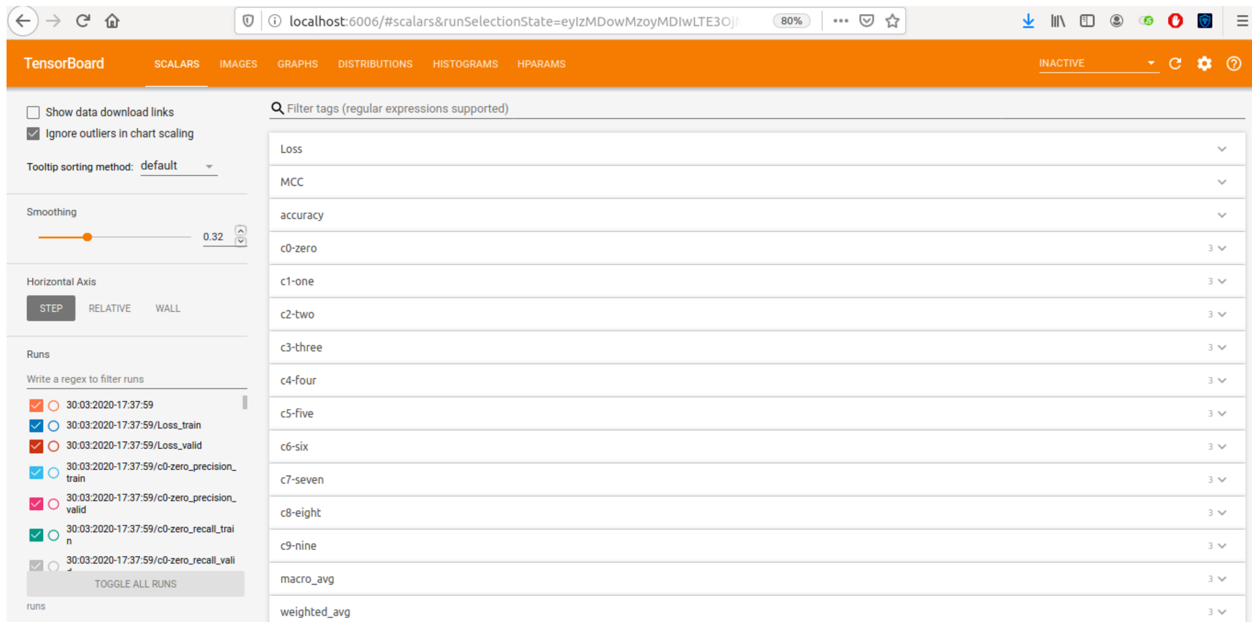
2 Detailed Features **5**
2.1 Features 5
2.2 Installation Guide 12
2.3 Examples 12
2.4 API Reference 15

3 Indices and tables **27**

Python Module Index **29**

Index **31**

Classification Report is a high-level library built on top of Pytorch which utilizes Tensorboard and scikit-learn and can be used for any classification problem. It tracks models Weight, Biases and Gradients during training and generates a detailed evaluation report for the model, all of this can be visualized on Tensorboard giving comphrensive insights. It can also be used for HyperParameter tracking which then can be utilized to compare different experiments.



CHAPTER 1

Features

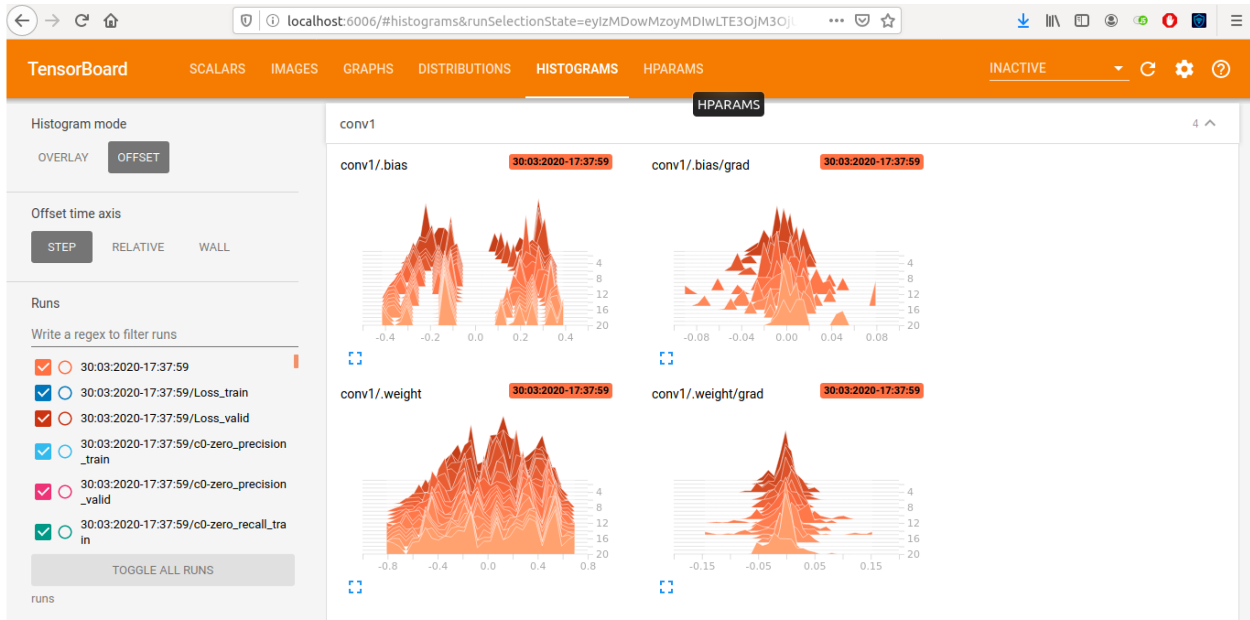
1. Model Weights, Biases and Gradients Tracking and plotting on histogram.
2. Visualizing the distribution of above described Model parameters.
3. Generating an interactive graph of the entire Model.
4. Graph of Precision, Recall and F1 Score for all the classes for each epoch.
5. Graph of Macro Avg and Weighted Avg of Precision, Recall and F1-score for each epoch.
6. Training and Validation Loss tracking for each epoch.
7. Accuracy and MCC metric tracking at each epoch.
8. Generating Confusion Matrix after certain number of epochs.
9. Bar Graph for False Positive and False Negative count for each class.
10. Scatter Plot for the predicited probabilities.
11. HyperParameter Tracking for comparing experiments.

CHAPTER 2

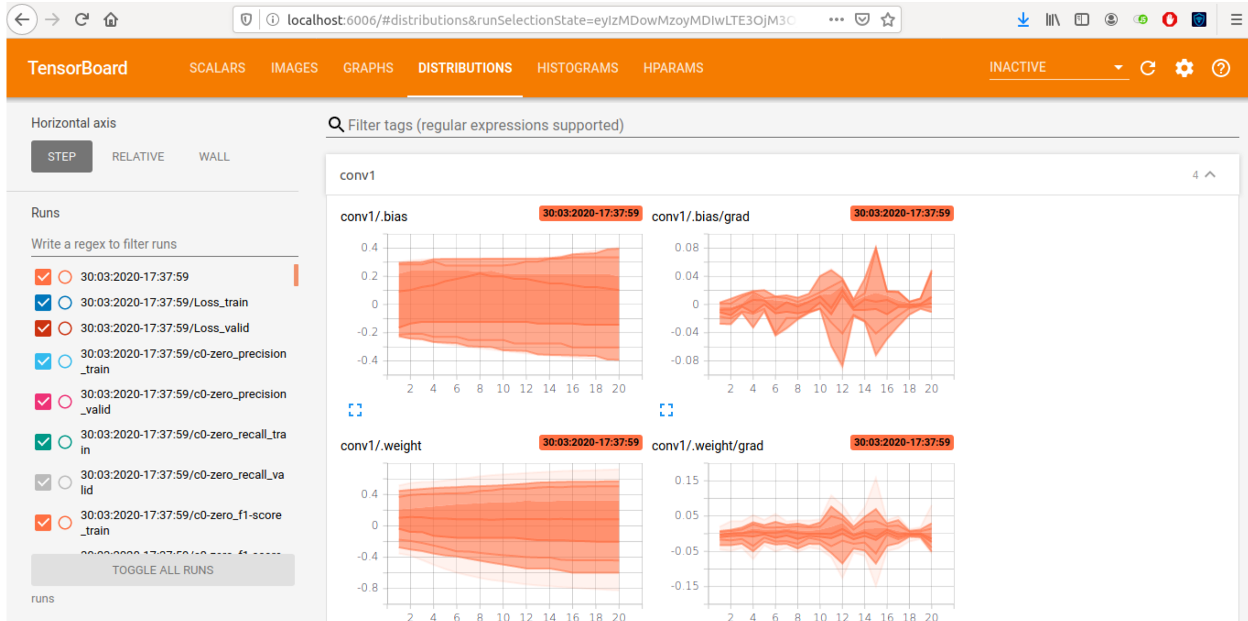
Detailed Features

2.1 Features

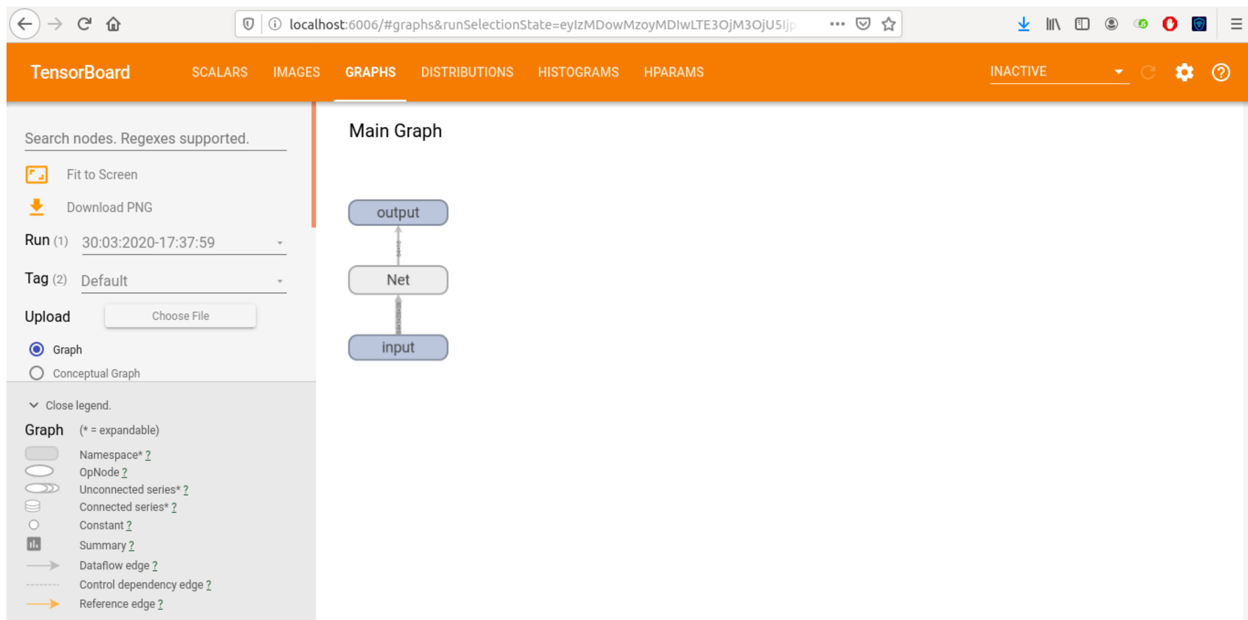
2.1.1 1. Model Weights, Biases and Gradients Tracking and plotting on histogram.



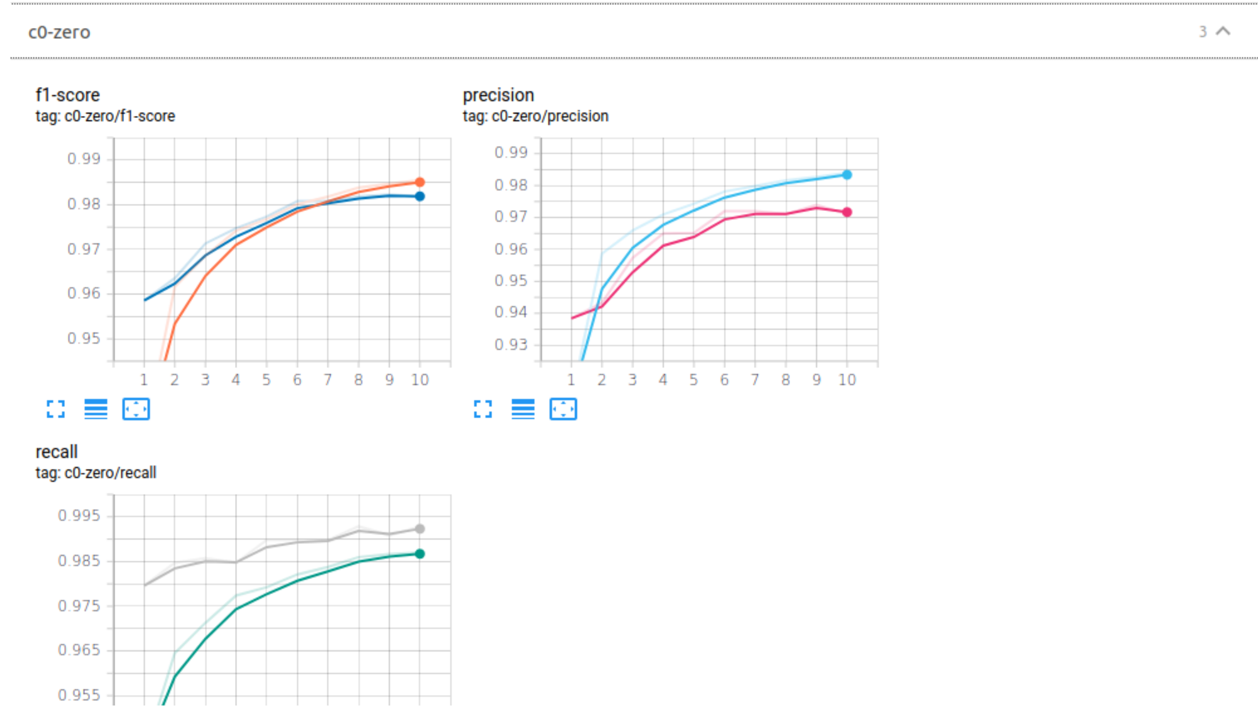
2.1.2 2. Visualizing the distribution of above described Model parameters.



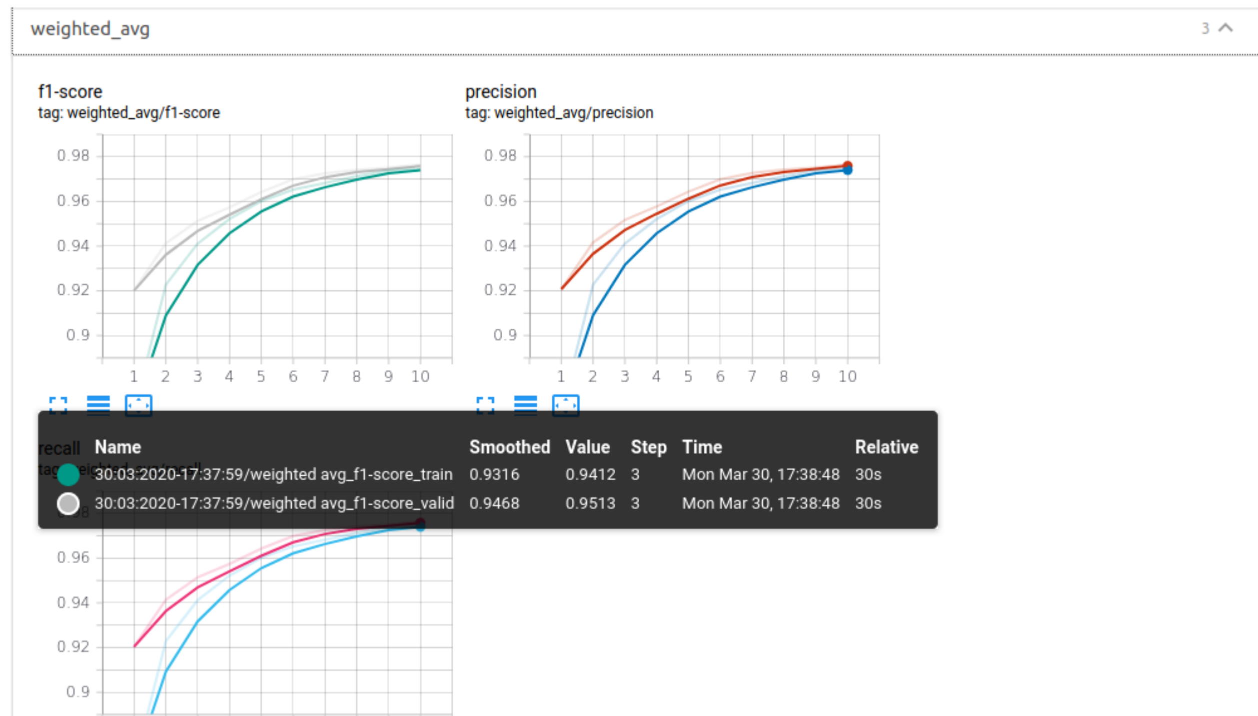
2.1.3 3. Generating an interactive graph of the entire Model



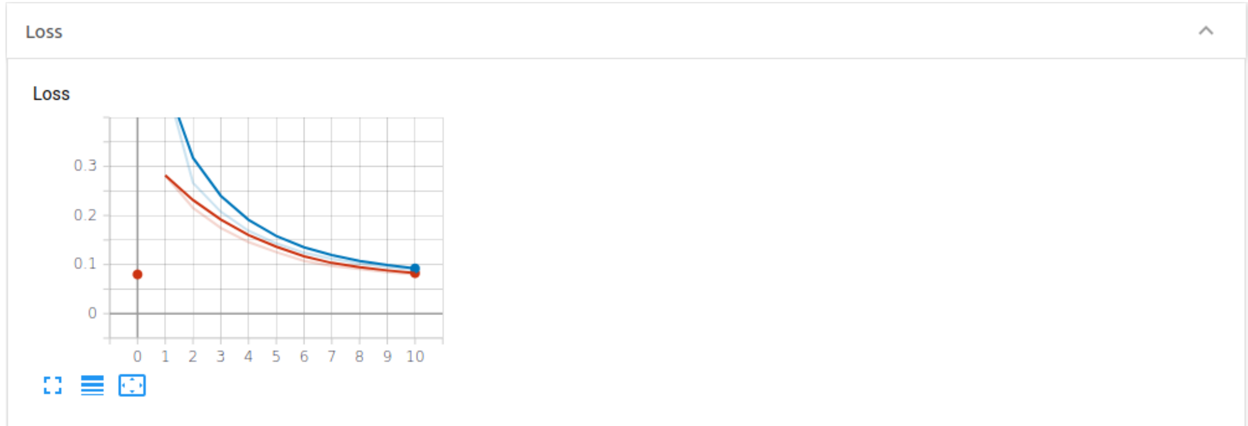
2.1.4 4. Graph of Precision, Recall and F1 Score for all the classes for each epoch.



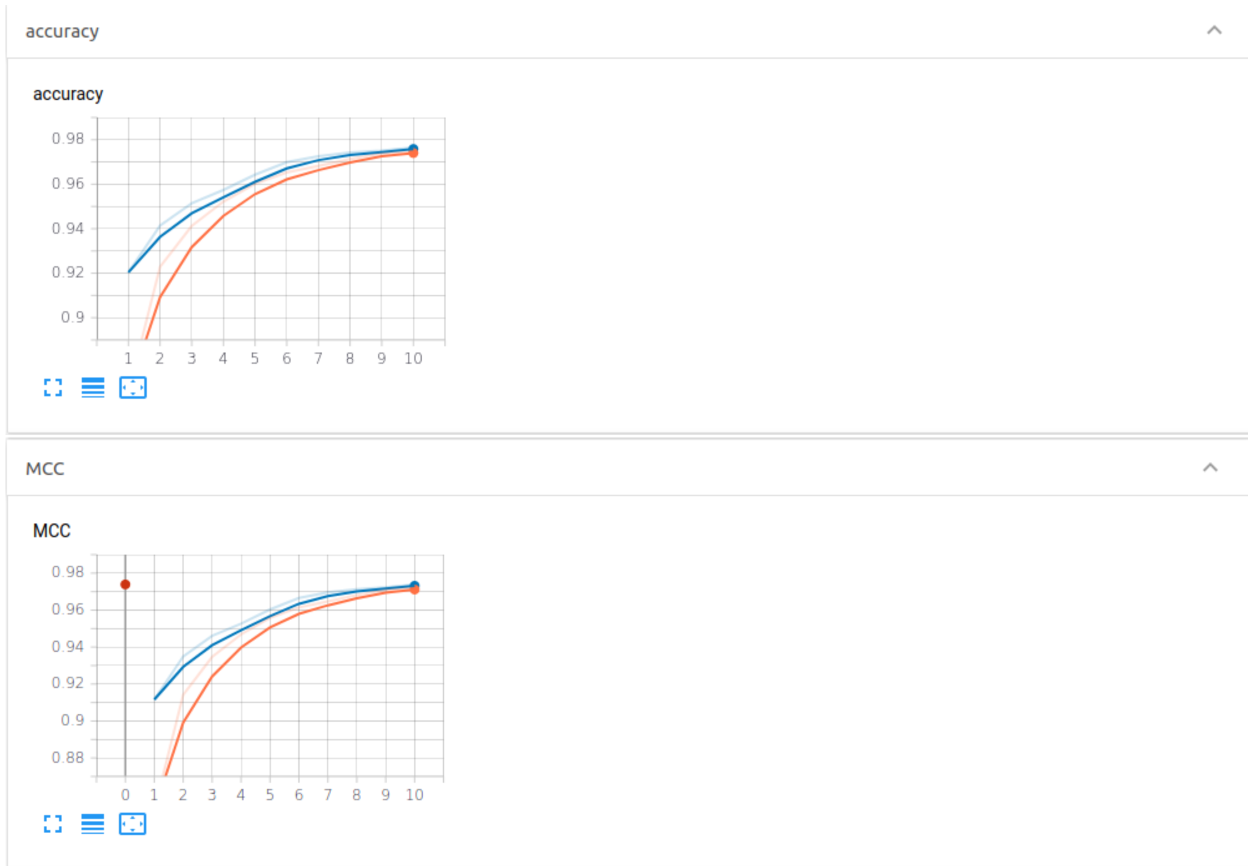
2.1.5 5. Graph of Macro Avg and Weighted Avg of Precision, Recall and F1-score for each epoch.



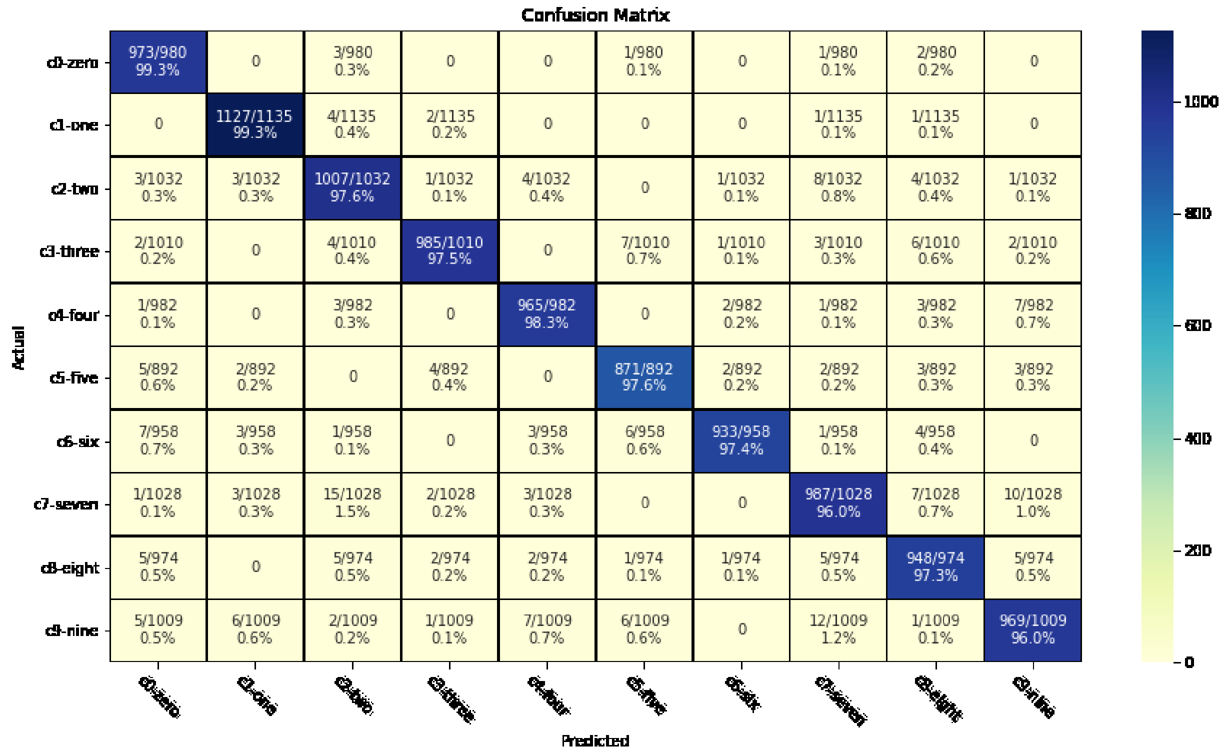
2.1.6 6. Training and Validation Loss tracking for each epoch.



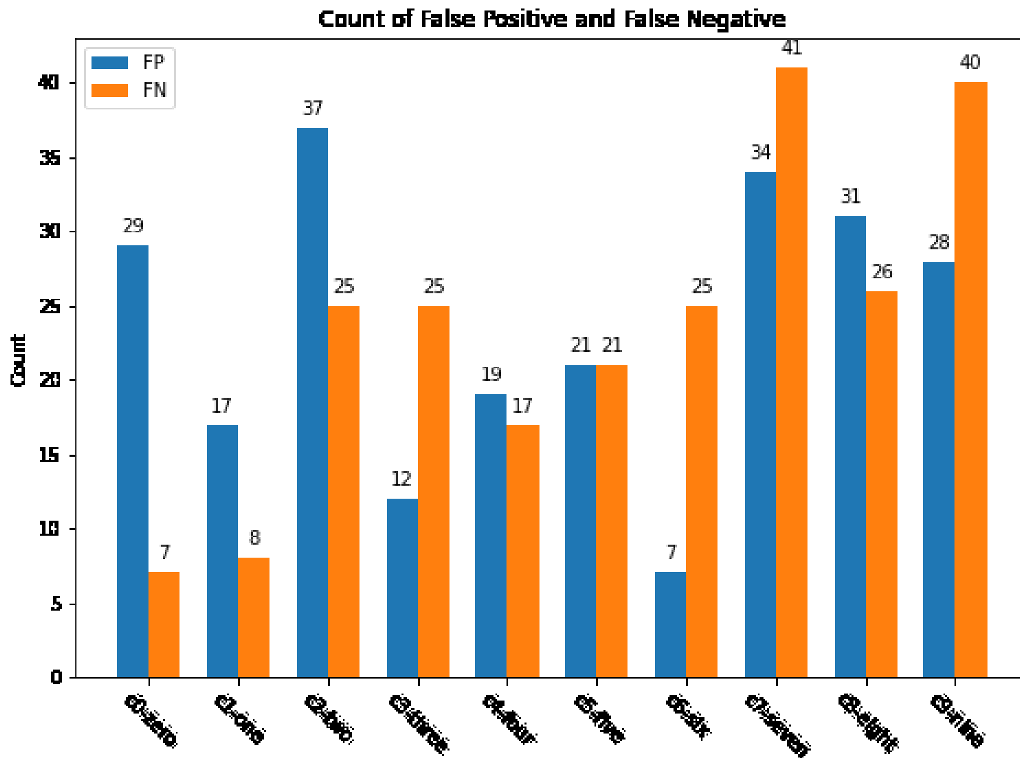
2.1.7 7. Accuracy and MCC metric tracking at each epoch.



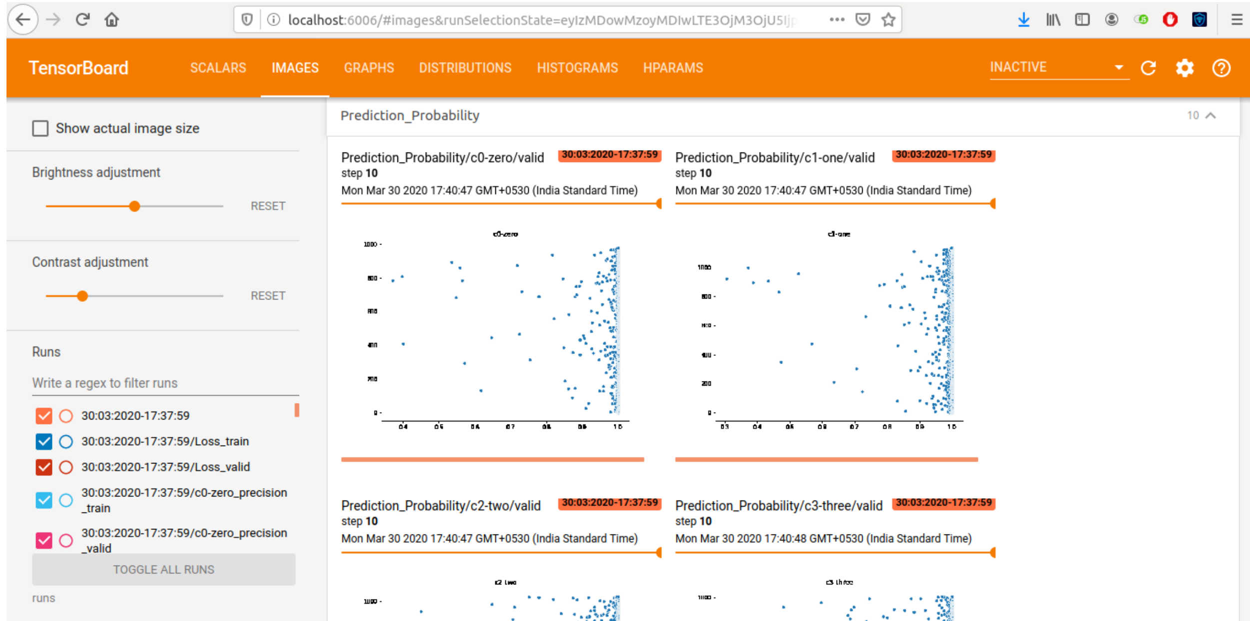
2.1.8 8. Generating Confusion Matrix after certain number of epochs.



2.1.9 9. Bar Graph for False Positive and False Negative count for each class.



2.1.10 10. Scatter Plot for the predicited probabilities.



2.1.11 11. Hyparameter Tracking for comparing experiments.

Features Detailed description of the features with pictures of tensorbaord visualization.

2.2 Installation Guide

2.2.1 Installing Classification Report

Classification Report library runs on python 3.6 and greater.

```
pip install classification-report
```

Things that are good to know

classification-report library is written in pure python and depends on a few key python packages

1. Pytorch, An open source machine learning framework that accelerates the path from research prototyping to production deployment.
2. Tensorboard,TensorBoard provides the visualization and tooling needed for machine learning experimentation.
3. Numpy, NumPy is the fundamental package for scientific computing with Python.
4. Seaborn, Seaborn is a Python data visualization library based on matplotlib.
5. Matplotlib, Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
6. scikit-learn, Machine Learning in Python

2.2.2 Installing from Source

```
git clone https://github.com/aman5319/Classification-Report
cd Classification-Report
```

```
pip install -e .
```

2.2.3 Developing Classification Report

For developing classification report you can install the library with all development dependencies but first clone the repo.

```
git clone https://github.com/aman5319/Classification-Report
cd Classification-Report
```

```
pip install -e ".[dev]"
```

2.3 Examples

2.3.1 Creating Config

There are multiple scenarios where we need to create configuration like in context of deep learning

1. Model Config

2. Training Config
3. Inference Config

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> model_config = Config(number_layers=3, num_head=32)
>>> model_config.number_layers
3
>>> training_config.lr
0.1
```

The benefit of using these config files are the can be `saved as JSON file` and can be `loaded from a JSON file`

Save Configuration

```
>>> training_config.save_config_json("training_config.json")
Configuration Saved
```

Load Configuration

```
>>> training_config = Config.load_config_json("training_config.json") # Execute the␣
↪saving code first.
>>> training_config.lr
```

For other methods in Config check the API references.

2.3.2 Creating HyperParameter

Since we can have many configs. It is necessary to combine all of them in place so that they can all be in one place and one can easily track all the `HyperParameter` to compare against multiple experiments.

`HyperParameter` class inherits from `Config` class so it supports all the method of `Config`.

```
>>> from classification_report import Config, HyperParameters
>>> model_config = Config(**{'hid_dim': 512,'n_layers': 8,'n_heads': 8,'pf_dim': 2048,
↪'dropout': 0.1})
>>> training_config = Config(num_epochs=15, max_lr=0.09, batch_size=64)
>>> inference_config = Config(batch_size=16)
>>> hyper = HyperParameters(model_config = model_config, training_config=training_
↪config,
infer_config=inference_config)
>>> hyper.model_config.hid_dim
512
>>> hyper.training_config.num_epochs
15
```

Similar to `Config` we can also save and load this `HyperParameter` which is a collection of configs.

while saving the `HyperParameter` it writes only in `one JSON` file.

```
>>> hyper.save_config_json("hyper.json")
Configuration Saved
>>> h = HyperParameters.load_config_json("hyper.json")
Saved Configuration Loaded
>>> h.training_config.num_epochs
15
```

2.3.3 Creating Report

There are demo_notebooks which can be executed to see the full potential of the this library.

Simple MNIST with Simple Reporting Example

Simple MNIST with Detail Reporting Example

This two notebook covers almost all the features of the library but to see details of it head over to API References

2.3.4 Visualising Report

As the report is generated on the fly while the model is training. All the visualization can be seen using tensorboard.

Whenever this library is executed a `runs` folder is created on the top-level and tensorboard uses that runs folder track.

This runs folder contains all the experiment and can be used to compare different experiments and can be shared among your teammates for studying.

The ideal way to visualize is to first execute the tensorboard from the same directory level from where all the notebooks are created.

```
tensorboard --logdir=runs
```

```
~/Projects/Personal/classification_report/demo_notebooks$ ls
MNIST_Example_Detail.ipynb  MNIST_Example.ipynb  runs
/Projects/Personal/classification_report/demo_notebooks$ tensorboard --logdir=runs
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.1.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

2.3.5 Demo

Goole Colab (Recommended)

Just open this notebook on colab and view the entire tensorboard visualization. - Simple Mnist Simple Reporting Visualize on colab

Manually Visualize a pre-existing experiment on your local.

Install the library After installing the library execute these commands.

```
git clone https://github.com/aman5319/Classification-Report # clone library
cd Classification-Report/demo_notebook # change the directory
tensorboard --logdir=runs # execute tensorboard
```

2.4 API Reference

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

2.4.1 classification_report

Submodules

classification_report.config

Module Contents

class classification_report.config.**Config**(***kwargs*)

Bases: object

This class can be used to create and store data for an type of configuration.

Parameters ****kwargs** – Arbitrary keyword arguments.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> model_config = Config(number_layers=3, num_head=32)
>>> model_config.number_layers
3
```

__setattr__(*self, name, value*)

__getattr__(*self, name*)

__delattr__(*self, name*)

update(*self, **kwargs*)

To update the config class attributes values.

This will add new attribute for a non existing attribute in the Config class or replace the value with a new a value for an existing attribute.

Parameters ****kwargs** – Arbitrary keyword arguments.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.lr
0.1
>>> training_config.update(lr=0.2,precision:16)
>>> training_config.lr
0.2
```

append_values(*self, **kwargs*)

This method can be used to append values to an existing attribute.

[1] Created with sphinx-autoapi

For Example if using Lr Scheduler then this can be use to track all lr values by appending in a list.

Note: The attribute should be prexisiting.

Parameters `**kwargs` – Arbitrary keyword arguments.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.lr
0.1
>>> training_config.append_values(lr=0.2)
>>> training_config.lr
[0.1,0.2]
>>> training_config.append_values(lr=[0.3,0.4])
>>> [0.1,0.2,0.3,0.4]
```

`__str__`(*self*)

`save_config_json`(*self, path: str*)

Save the Configuration in json format.

Parameters `path` – The file path to save json file.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.save_config_json("training_config.json")
Configuration Saved
```

`classmethod load_config_json`(*cls, path: str*)

Loading the saved configuration.

Parameters `path` – The file from json config will be loaded.

Returns A Config Class is returned with attributes set from json file

Return type *Config*

Examples::

```
>>> from classification_report import Config
>>> training_config = Config.load_config_json("training_config.json") #␣
↪Execute the saving code first.
>>> training_config.lr
0.1
```

`get_dict_repr`(*self*)

Returns The Dictionary representation of Config class

Return type dict

Example::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.get_dict_repr()
{"lr":0.1,"batch_size":32,"device":"GPU"}
```

class classification_report.config.**HyperParameters**(***configs*)

Bases: *classification_report.config.Config*

It stores collections of Config in one place. It inherits the Config class.

Parameters ****config** – Arbitary Config objects

Raises AssertionError – Pass only Config class object.

Examples::

```
>>> from classification_report import Config, HyperParameters
>>> model_config = Config(**{'hid_dim': 512,'n_layers': 8,'n_heads': 8,'pf_dim
↪': 2048,'dropout': 0.1})
>>> training_config = Config(num_epochs=15, max_lr=0.09, batch_size=64)
>>> inference_config = Config(batch_size=16)
>>> hyper = HyperParameters(model_config = model_config, training_
↪config=training_config,infer_config=inference_config)
>>> hyper.model_config.hid_dim
512
>>> hyper.save_config_json("hyper.json")
Configuration Saved
>>> hyper = HyperParameters.load_config_json("hyper.json")
Saved Configuration Loaded
```

static flatten(*d: dict*, *parent_key: str = ''*, *sep: str = '_'*)

Flatten the nested dictionary using recusrion.

Parameters

- **d** – The dictionary to flatten.
- **parent_key** – The parent key.
- **sep** – The sep to be used to join parent_key with nested_key

Returns Flattened dictionary.

Return type dict

classification_report.report

Module Contents

class classification_report.report.**Report**(*classes: TrainType*, *dir_name: str = None*)

Generating Report for classification model by tracking Model training and giving different types of metrics to evaluate the Model.

For any classification problem during Model's training it is very important to track Model's Weight Biases and Gradients. After training the important part is the model evaluation where we evaluate the model performance. This Report class simplify the evaluation part where all the evaluation metrics are automatically generated for the model.It uses Tensorboard to visualize all these.

write_a_batch (*self*, *loss: LossType*, *batch_size: int*, *actual: ActualType*, *prediction: PredictionType*, *train: bool = True*)

This methods records the batch information during train and val phase.

During training and validation record the loss, batch actual labels and predicted labels.

Note: For prediction don't pass raw_logits pass softmax output.

Parameters

- **loss** – The batch loss.
- **batch_size** – The batch size on which the loss was calculated. The batch_size may change during last iteration so calculate batch_size from data.
- **actual** – The actual labels.
- **prediction** – The predicted labels.
- **train** – True signifies training mode and False Validation Mode.

Returns `Report` class instance.

update_actual_prediction (*self*, *actual: ActualType*, *prediction: PredictionType*, *train_type: TrainType*)

Stores actual and predicted labels seperately for training and validation and after every batch call the values are appended. :param actual: The actual labels. :param prediction: The predicted labels. :param train_type: The labels belong to *train* or *valid*.

Returns Report class instances.

update_loss (*self*, *loss: LossType*, *batch_size: int*, *train_type: TrainType*)

Accumlates loss for every batch seperately for training and validation.

Parameters

- **loss** – The batch loss.
- **batch_size** – The batch size on which the loss was calculated. The batch_size may change during last iteration so calculate batch_size from data.
- **train_type** – The Labels belong to *train* or *valid*.

Returns `Report` class instance.

update_data_iter (*self*, *batch_size: int*, *train_type: TrainType*)

Accumlates the iteration count and data point count for training and validation.

Parameters

- **batch_size** – The batch size on which the loss was calculated. The batch_size may change during last iteration so calculate batch_size from data.
- **train_type** – The Labels belong to *train* or *valid*.

Returns `Report` class instance.

plot_an_epoch (*self*, *detail: bool = False*)

Plot an epoch method simplifies ploting standard things which are needed to be plotted after an epoch completion for granular control use this which *detail = False* and call other methods on top of it.

Parameters **detail** – whether to use detail mode or not.

Returns `Report` class instance.

init_data_storage(*self*)
Clean the data storage units after every epoch.

change_data_type(*self*, *data: LossType*, *required_data_type: LossType*)
Change the data type of input to required data type.

Parameters

- **data** – Input data type.
- **required_data_type** – Change the data type to given format, can be either *np* or *f*.

Returns The data in required data type.

close(*self*)
Close the tensorboard writer object.

After calling this method report will not track anything.

write_to_tensorboard(*self*)
This methods call various other method which write on tensorboard.

plot_loss(*self*)
Plots loss at the end of the epoch.

Returns *Report* class instance.

plot_model(*self*, *model: torch.nn.Module*, *data: torch.Tensor*)
Plot model graph.

Parameters

- **model** – The model architecture.
- **data** – The input to the model.

Returns *Report* class instance.

plot_confusion_matrix(*self*, *at_which_epoch*)
Plots confusion matrix.

Parameters **at_which_epoch** – After how many epochs the confusion matrix should be plotted. For example if the model is trained for 10 epochs and you want to plot confusion matrix after every 5 epoch then the input to this method will be 5.

Returns *Report* class instance.

plot_precision_recall(*self*)
Plots Precision Recall F1-score graph for all Classes with Weighted Average and Macro Average.

Returns *Report* class instance.

plot_missclassification_count(*self*, *at_which_epoch*)
Plot Misclassification Count for each class.

Bar graph for False Positive and False Negative Count.

Parameters **at_which_epoch** – After how many epochs the Misclassification Count should be plotted. For example if the model is trained for 10 epochs and you want to plot this after every 5 epoch then the input to this method will be 5.

Returns *Report* class instance.

calculate_fp_fn(*self*, *actual: ActualType*, *pred: PredictionType*)
Calculates False Postive and False Negative Count per class.

Parameters

- **actual** – The actual labels.
- **pred** – The predicted Labels.

Returns *Report* class instance.

plot_mcc (*self*)
Plots Mathews Correlation Coefficient.

Returns *Report* class instance.

plot_pred_prob (*self*, *at_which_epoch: int*)
Plots scatter plot for the predicted probabilites for each class.

Parameters **at_which_epoch** – After how many epochs the predicted probabilites should be plotted. For example if the model is trained for 10 epochs and you want to plot this after every 5 epoch then the input to this method will be 5.

Returns *Report* class instance.

plot_model_data_grad (*self*, *at_which_iter: int*)
Plot Histogram and Distribution for each layers of model Weights, Bias and Gradients.

Parameters **at_which_iter** – After how many iteration this should be plotted. The ideal way to plot this to plot after every one-half or one-third of the train_iterator.

Returns *Report* class instance.

Examples::

```
>>> report.plot_model_data_grad(at_which_iter = len(train_iterator)/2)
```

plot_hparams (*self*, *config: HyperParameters*)
Plot Hyper parameters for the model. This method should be called once training is over.

Parameters **config** – Hyperparameter Configs.

Returns *Report* class instance.

classification_report.return_types

Module Contents

classification_report.return_types.**LossType**

classification_report.return_types.**TrainType**

classification_report.utils

Module Contents

classification_report.utils.**convert_prob_to_label** (*data: np.ndarray*) → np.ndarray
Convert Probability to labels

Parameters **data** – (np.ndarray): Probability output of softmax. The size of data (batch_size, num_classes)

Returns Returns pobability converted to labels by finding maximum in last dimension. The output shape is (batch_size,)

Return type np.ndarray

classification_report.version

Module Contents

classification_report.version.**__version__ = 1.0.0**

Package Contents

class classification_report.**Config**(***kwargs*)

Bases: object

This class can be used to create and store data for an type of configuration.

Parameters ****kwargs** – Arbitrary keyword arguments.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> model_config = Config(number_layers=3, num_head=32)
>>> model_config.number_layers
3
```

__setattr__(*self*, *name*, *value*)

__getattr__(*self*, *name*)

__delattr__(*self*, *name*)

update(*self*, ***kwargs*)

To update the config class attributes values.

This will add new attribute for a non existing attribute in the Config class or replace the value with a new a value for an existing attribute.

Parameters ****kwargs** – Arbitrary keyword arguments.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.lr
0.1
>>> training_config.update(lr=0.2,precision:16)
>>> training_config.lr
0.2
```

append_values(*self*, ***kwargs*)

This method can be used to append values to an existing attribute.

For Example if using Lr Scheduler then this can be use to track all lr values by appending in a list.

Note: The attribute should be prexisiting.

Parameters `**kwargs` – Arbitrary keyword arguments.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.lr
0.1
>>> training_config.append_values(lr=0.2)
>>> training_config.lr
[0.1,0.2]
>>> training_config.append_values(lr=[0.3,0.4])
>>> [0.1,0.2,0.3,0.4]
```

`__str__`(*self*)

`save_config_json`(*self*, *path: str*)

Save the Configuration in json format.

Parameters `path` – The file path to save json file.

Examples::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.save_config_json("training_config.json")
Configuration Saved
```

`classmethod load_config_json`(*cls*, *path: str*)

Loading the saved configuration.

Parameters `path` – The file from json config will be loaded.

Returns A Config Class is returned with attributes set from json file

Return type *Config*

Examples::

```
>>> from classification_report import Config
>>> training_config = Config.load_config_json("training_config.json") #␣
↪Execute the saving code first.
>>> training_config.lr
0.1
```

`get_dict_repr`(*self*)

Returns The Dictionary representation of Config class

Return type dict

Example::

```
>>> from classification_report import Config
>>> training_config = Config(lr=0.1, batch_size=32, device="GPU")
>>> training_config.get_dict_repr()
{"lr":0.1,"batch_size":32,"device":"GPU"}
```

class classification_report.**HyperParameters**(***configs*)

Bases: *classification_report.config.Config*

It stores collections of Config in one place. It inherits the Config class.

Parameters ****config** – Arbitary Config objects

Raises AssertionError – Pass only Config class object.

Examples::

```
>>> from classification_report import Config, HyperParameters
>>> model_config = Config(**{'hid_dim': 512,'n_layers': 8,'n_heads': 8,'pf_dim
↪': 2048,'dropout': 0.1})
>>> training_config = Config(num_epochs=15, max_lr=0.09, batch_size=64)
>>> inference_config = Config(batch_size=16)
>>> hyper = HyperParameters(model_config = model_config, training_
↪config=training_config,infer_config=inference_config)
>>> hyper.model_config.hid_dim
512
>>> hyper.save_config_json("hyper.json")
Configuration Saved
>>> hyper = HyperParameters.load_config_json("hyper.json")
Saved Configuration Loaded
```

static flatten(*d: dict*, *parent_key: str = ''*, *sep: str = '_'*)

Flatten the nested dictionary using recusrion.

Parameters

- **d** – The dictionary to flatten.
- **parent_key** – The parent key.
- **sep** – The sep to be used to join parent_key with nested_key

Returns Flattened dictionary.

Return type dict

class classification_report.**Report**(*classes: TrainType*, *dir_name: str = None*)

Generating Report for classification model by tracking Model training and giving different types of metrics to evaluate the Model.

For any classification problem during Model's training it is very important to track Model's Weight Biases and Gradients. After training the important part is the model evaluation where we evaluate the model performance. This Report class simplify the evaluation part where all the evaluation metrics are automatically generated for the model.It uses Tensorboard to visualize all these.

write_a_batch(*self*, *loss: LossType*, *batch_size: int*, *actual: ActualType*, *prediction: PredictionType*, *train: bool = True*)

This methods records the batch information during train and val phase.

During training and validation record the loss, batch actual labels and predicted labels.

Note: For prediction don't pass raw_logits pass softmax output.

Parameters

- **loss** – The batch loss.

- **batch_size** – The batch size on which the loss was calculated. The batch_size may change during last iteration so calculate batch_size from data.
- **actual** – The actual labels.
- **prediction** – The predicted labels.
- **train** – True signifies training mode and False Validation Mode.

Returns `Report` class instance.

update_actual_prediction (*self*, *actual: ActualType*, *prediction: PredictionType*, *train_type: TrainType*)

Stores actual and predicted labels seperately for training and validation and after every batch call the values are appended. :param actual: The actual labels. :param prediction: The predicted labels. :param train_type: The labels belong to *train* or *valid*.

Returns Report class instances.

update_loss (*self*, *loss: LossType*, *batch_size: int*, *train_type: TrainType*)

Accumlates loss for every batch seperately for training and validation.

Parameters

- **loss** – The batch loss.
- **batch_size** – The batch size on which the loss was calculated. The batch_size may change during last iteration so calculate batch_size from data.
- **train_type** – The Labels belong to *train* or *valid*.

Returns `Report` class instance.

update_data_iter (*self*, *batch_size: int*, *train_type: TrainType*)

Accumlates the iteration count and data point count for training and validation.

Parameters

- **batch_size** – The batch size on which the loss was calculated. The batch_size may change during last iteration so calculate batch_size from data.
- **train_type** – The Labels belong to *train* or *valid*.

Returns `Report` class instance.

plot_an_epoch (*self*, *detail: bool = False*)

Plot an epoch method simplifies ploting standard things which are needed to be plotted after an epoch completion for granular control use this which *detail* = *False* and call other methods on top of it.

Parameters **detail** – whether to use detail mode or not.

Returns `Report` class instance.

init_data_storage (*self*)

Clean the data storage units after every epoch.

change_data_type (*self*, *data: LossType*, *required_data_type: LossType*)

Change the data type of input to required data type.

Parameters

- **data** – Input data type.
- **required_data_type** – Change the data type to given format, can be either *np* or *f*.

Returns The data in required data type.

close (*self*)
Close the tensorboard writer object.

After calling this method report will not track anything.

write_to_tensorboard (*self*)
This methods call various other method which write on tensorboard.

plot_loss (*self*)
Plots loss at the end of the epoch.

Returns `Report` class instance.

plot_model (*self*, *model: torch.nn.Module*, *data: torch.Tensor*)
Plot model graph.

Parameters

- **model** – The model architecture.
- **data** – The input to the model.

Returns `Report` class instance.

plot_confusion_matrix (*self*, *at_which_epoch*)
Plots confusion matrix.

Parameters **at_which_epoch** – After how many epochs the confusion matrix should be plotted. For example if the model is trained for 10 epochs and you want to plot confusion matrix after every 5 epoch then the input to this method will be 5.

Returns `Report` class instance.

plot_precision_recall (*self*)
Plots Precision Recall F1-score graph for all Classes with Weighted Average and Macro Average.

Returns `Report` class instance.

plot_missclassification_count (*self*, *at_which_epoch*)
Plot Misclassification Count for each class.

Bar graph for False Positive and False Negative Count.

Parameters **at_which_epoch** – After how many epochs the Misclassification Count should be plotted. For example if the model is trained for 10 epochs and you want to plot this after every 5 epoch then the input to this method will be 5.

Returns `Report` class instance.

calculate_fp_fn (*self*, *actual: ActualType*, *pred: PredictionType*)
Calculates False Postive and False Negative Count per class.

Parameters

- **actual** – The actual labels.
- **pred** – The predicted Labels.

Returns `Report` class instance.

plot_mcc (*self*)
Plots Mathews Correlation Coefficient.

Returns `Report` class instance.

plot_pred_prob (*self*, *at_which_epoch: int*)
Plots scatter plot for the predicted probabilites for each class.

Parameters **at_which_epoch** – After how many epochs the predicted probabilites should be plotted. For example if the model is trained for 10 epochs and you want to plot this after every 5 epoch then the input to this method will be 5.

Returns *Report* class instance.

plot_model_data_grad(*self*, *at_which_iter: int*)

Plot Histogram and Distribution for each layers of model Weights, Bias and Gradients.

Parameters **at_which_iter** – After how many iteration this should be plotted. The ideal way to plot this to plot after every one-half or one-third of the train_iterator.

Returns *Report* class instance.

Examples::

```
>>> report.plot_model_data_grad(at_which_iter = len(train_iterator)/2)
```

plot_hparams(*self*, *config: HyperParameters*)

Plot Hyper parameters for the model. This method should be called once training is over.

Parameters **config** – Hyperparameter Configs.

Returns *Report* class instance.

classification_report.**convert_prob_to_label**(*data: np.ndarray*) → np.ndarray

Convert Probability to labels

Parameters **data** – (np.ndarray): Probability output of softmax. The size of data (batch_size, num_classes)

Returns Returns pobability converted to labels by finding maximum in last dimension. The output shape is (batch_size,)

Return type np.ndarray

classification_report.**__version__ = 1.0.0**

CHAPTER 3

Indices and tables

- search

Python Module Index

c

classification_report, 15
classification_report.config, 15
classification_report.report, 17
classification_report.return_types, 20
classification_report.utils, 20
classification_report.version, 21

Index

Symbols

__delattr__() (*classification_report.Config method*), 21
__delattr__() (*classification_report.config.Config method*), 15
__getattr__() (*classification_report.Config method*), 21
__getattr__() (*classification_report.config.Config method*), 15
__setattr__() (*classification_report.Config method*), 21
__setattr__() (*classification_report.config.Config method*), 15
__str__() (*classification_report.Config method*), 22
__str__() (*classification_report.config.Config method*), 16
__version__ (*in module classification_report*), 26
__version__ (*in module classification_report.version*), 21

A

append_values() (*classification_report.Config method*), 21
append_values() (*classification_report.config.Config method*), 15

C

calculate_fp_fn() (*classification_report.Report method*), 25
calculate_fp_fn() (*classification_report.report.Report method*), 19
change_data_type() (*classification_report.Report method*), 24
change_data_type() (*classification_report.report.Report method*), 19
classification_report (*module*), 15
classification_report.config (*module*), 15
classification_report.report (*module*), 17
classification_report.return_types (*module*), 20
classification_report.utils (*module*), 20
classification_report.version (*module*), 21
close() (*classification_report.Report method*), 24
close() (*classification_report.report.Report method*), 19
Config (*class in classification_report*), 21
Config (*class in classification_report.config*), 15
convert_prob_to_label() (*in module classification_report*), 26
convert_prob_to_label() (*in module classification_report.utils*), 20

F

flatten() (*classification_report.config.HyperParameters static method*), 17
flatten() (*classification_report.HyperParameters static method*), 23

G

get_dict_repr() (*classification_report.Config method*), 22
get_dict_repr() (*classification_report.config.Config method*), 16

H

HyperParameters (*class in classification_report*), 22
HyperParameters (*class in classification_report.config*), 17

I

init_data_storage() (*classification_report.Report method*), 24
init_data_storage() (*classification_report.report.Report method*), 18

L

load_config_json() (*classification_report.Config class method*), 22
load_config_json() (*classification_report.config.Config class method*), 16
LossType (*in module classification_report.return_types*), 20

P

plot_an_epoch() (*classification_report.Report method*), 24
plot_an_epoch() (*classification_report.report.Report method*), 18
plot_confusion_matrix() (*classification_report.Report method*), 25
plot_confusion_matrix() (*classification_report.report.Report method*), 19
plot_hparams() (*classification_report.Report method*), 26
plot_hparams() (*classification_report.report.Report method*), 20
plot_loss() (*classification_report.Report method*), 25
plot_loss() (*classification_report.report.Report method*), 19
plot_mcc() (*classification_report.Report method*), 25
plot_mcc() (*classification_report.report.Report method*), 20
plot_missclassification_count() (*classification_report.Report method*), 25
plot_missclassification_count() (*classification_report.report.Report method*), 19
plot_model() (*classification_report.Report method*), 25
plot_model() (*classification_report.report.Report method*), 19
plot_model_data_grad() (*classification_report.Report method*), 26
plot_model_data_grad() (*classification_report.report.Report method*), 20
plot_precision_recall() (*classification_report.Report method*), 25
plot_precision_recall() (*classification_report.report.Report method*), 19
plot_pred_prob() (*classification_report.Report method*), 25
plot_pred_prob() (*classification_report.report.Report method*), 20

R

Report (*class in classification_report*), 23
Report (*class in classification_report.report*), 17

S

save_config_json() (*classification_report.Config method*), 22
save_config_json() (*classification_report.config.Config method*), 16

T

TrainType (*in module classification_report.return_types*), 20

U

update() (*classification_report.Config method*), 21
update() (*classification_report.config.Config method*), 15
update_actual_prediction() (*classification_report.Report method*), 24
update_actual_prediction() (*classification_report.report.Report method*), 18
update_data_iter() (*classification_report.Report method*), 24
update_data_iter() (*classification_report.report.Report method*), 18
update_loss() (*classification_report.Report method*), 24
update_loss() (*classification_report.report.Report method*), 18

W

write_a_batch() (*classification_report.Report method*), 23
write_a_batch() (*classification_report.report.Report method*), 17
write_to_tensorboard() (*classification_report.Report method*), 25
write_to_tensorboard() (*classification_report.report.Report method*), 19