torch-train

Release 0.0.1

Aug 14, 2020

Contents:

torch-train provides a wrapper around the torch.nn.Module to provide scikit-learn like `fit()` and `predict()` methods.

CHAPTER 1

Installation

The most straigtforward way of installing torch-train is via pip

```
pip install torch-train
```

If you wish to stay up to date with the latest development version, you can instead download the source code. In this case, make sure that you have all the required *dependencies* installed.

1.1 Dependencies

torch-train requires the following python packages to be installed:

- pytorch: https://pytorch.org/

All dependencies should be automatically downloaded if you install torch-train via pip. However, should you want to install these libraries manually, you can install the dependencies using the requirements.txt file

```
pip install -r requirements.txt
```

Or you can install these libraries yourself

```
pip install -U torch
```

CHAPTER 2

Usage

This section gives a high-level overview of the modules implemented by torch-train. We also include a working example to guide users through the code. For detailed documentation of individual methods, we refer to the *Reference* guide.

2.1 Overview

This section explains the design of torch-train on a high level. torch-train is a network that is implemented as an extension of `torch.nn.Module`. This means it can be trained and used as any neural network module in the pytorch library.

In addition, we provide automatic methods to train and predict given data tensors using our *Module* extension. This follows a `scikit-learn` approach with `fit()`, `predict()` and `fit_predict()` methods. We refer to the *Reference* for a detailed description.

2.2 Code

To use torch-train into your own project, you can use it in place of the torch.nn.Module. Here we show some simple examples on how to use the torch-train Module in your own python code. For a complete documentation we refer to the *Reference* guide.

2.2.1 Import

To import the Module use

```
from torchtrain import Module
```

2.2.2 Working example

In this example, we create a basic torch Module and use its `fit()` and `predict()` methods to train and test. First we import `torch` and the `torchtrain` Module

```
# imports
import torch
import torch.nn as nn
from   torchtrain import Module
```

Next we create our simple network consisting of 2 layers and a softmax output function.

Note: We extend the `torchtrain.Module` instead of the `torch.nn.Module` like you normally would.

Furthermore we implement the `forward()` method to propagate through the network.

```
class MyNetwork(Module):

    def __init__(self, size_input, size_hidden, size_output):
        """Create simple network"""
        # Initialise super
        super().__init__()

        # Set layers
        self.layer_1 = nn.Linear(size_input , size_hidden)
        self.layer_2 = nn.Linear(size_hidden, size_output)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(X):
        """Forward through network"""
        # Propagate layer 1
        out = self.layer_1(X)
        # Propagate layer 2
        out = self.layer_2(out)
        # Propagate softmax layer
        out = self.softmax(out)
        # Return result
        return out
```

Now that we have created our network, we generate some random training and testing data.

```
# Generate random data
X_train = torch.rand((1024, 10))
y_train = (torch.rand(1024)*10).to(torch.int64)
X_test  = torch.rand((1024, 10))
y_test  = (torch.rand(1024)*10).to(torch.int64)
```

Finally, we create the network and invoke its `fit()` and `predict()` methods.

```
# Create network
net = MyNetwork(10, 128, 10)

# Fit network
net.fit(X_train, y_train,
    epochs        = 10,
    batch_size    = 32,
```

(continues on next page)

(continued from previous page)

```
    learning_rate = 0.01,
    criterion     = nn.NLLLoss(),
    optimizer     = optim.SGD,
    variable      = False,
    verbose       = True
)

# Predict network
y_pred = net.predict(X_test,
    batch_size = 32,
    variable   = False,
    verbose    = True
)
```

CHAPTER 3

Reference

This is the reference documentation for the classes and methods objects provided by the torch-train module.

3.1 Module

The Module class is an extension of the torch.nn.Module object. This class implements scikit-learn-like `fit()` and `predict()` methods to automatically use `nn.Module` objects for training and predicting labels. This module also automatically keeps track of the progress during fitting and predicting.

class `module.`**Module**(**args*, ***kwargs*)

Extention of nn.Module that adds fit and predict methods Can be used for automatic training.

progress

Used to track progress of fit and predict methods

Type Progress()

3.1.1 Initialization

`Module.`**__init__**(**args*, ***kwargs*)

Only calls super method nn.Module with given arguments.

3.1.2 Fit

To train a Module, we use the `fit()` method.

`Module.`**fit**(*X*, *y*, *epochs=10*, *batch_size=32*, *learning_rate=0.01*, *criterion=<sphinx.ext.autodoc.importer._MockObject object>*, *optimizer=<sphinx.ext.autodoc.importer._MockObject object>*, *variable=False*, *verbose=True*, ***kwargs*)

Train the module with given parameters

Parameters

- **X** (*torch.Tensor*) – Tensor to train with
- **y** (*torch.Tensor*) – Target tensor
- **epochs** (*int, default=10*) – Number of epochs to train with
- **batch_size** (*int, default=32*) – Default batch size to use for training
- **learning_rate** (*float, default=0.01*) – Learning rate to use for optimizer
- **criterion** (*nn.Loss, default=nn.NLLLoss()*) – Loss function to use
- **optimizer** (*optim.Optimizer, default=optim.SGD*) – Optimizer to use for training
- **variable** (*boolean, default=False*) – If True, accept inputs of variable length
- **verbose** (*boolean, default=True*) – If True, prints training progress

Returns **result** – Returns self

Return type self

3.1.3 Predict

A module can also `predict()` outputs for given inputs. Usually this method is called after fitting.

`Module.`**predict**(*X*, *batch_size=32*, *variable=False*, *verbose=True*, ***kwargs*)

Makes prediction based on input data X. Default implementation just uses the module forward(X) method, often the predict method will be overwritten to fit the specific needs of the module.

Parameters

- **X** (*torch.Tensor*) – Tensor from which to make prediction
- **batch_size** (*int, default=32*) – Batch size in which to predict items in X
- **variable** (*boolean, default=False*) – If True, accept inputs of variable length
- **verbose** (*boolean, default=True*) – If True, print progress of prediction

Returns **result** – Resulting prediction

Return type torch.Tensor

3.1.4 Fit-Predict

Sometimes we want to `fit()` and `predict()` on the same data. This can easily be achieved using the `fit_predict()` method.

`Module.`**fit_predict**(*X*, *y*, *epochs=10*, *batch_size=32*, *learning_rate=0.01*, *criterion=<sphinx.ext.autodoc.importer._MockObject object>*, *optimizer=<sphinx.ext.autodoc.importer._MockObject object>*, *variable=False*, *verbose=True*, ***kwargs*)

Train the module with given parameters

Parameters

- **X** (*torch.Tensor*) – Tensor to train with
- **y** (*torch.Tensor*) – Target tensor

- **epochs** (*int, default=10*) – Number of epochs to train with
- **batch_size** (*int, default=32*) – Default batch size to use for training
- **learning_rate** (*float, default=0.01*) – Learning rate to use for optimizer
- **criterion** (*nn.Loss, default=nn.NLLLoss*) – Loss function to use
- **optimizer** (*optim.Optimizer, default=optim.SGD*) – Optimizer to use for training
- **variable** (*boolean, default=False*) – If True, accept inputs of variable length
- **verbose** (*boolean, default=True*) – If True, prints training progress

Returns **result** – Resulting prediction

Return type torch.Tensor

3.2 Progress

The `Progress` class is used to track the progress of training and prediction.

3.2.1 Initialization

3.2.2 Reset

To restart the Progress, we use the `reset()` method. This sets the amount of items we expect to train with and the number of epochs we use for training.

3.2.3 Update

A module will update using the `update()` method, which automatically prints the progress.

Second, when we move to the next epoch we use the `update_epoch()` method.

3.3 Variable Data Loader

The Variable Data Loader class is a different implementation of the iterable torch.utils.data.DataLoader class that can handle inputs of variable lengths.

class variable_data_loader.**VariableDataLoader**(*X*, *y*, *index=False*, *batch_size=1*, *shuffle=True*)

Load data from variable length inputs

lengths
Dictionary of input-length -> input samples

Type dict()

index
If True, also returns original index

Type boolean, default=False

batch_size
Size of each batch to output

Type int, default=1

shuffle
If True, shuffle the data randomly, each yielded batch contains only input items of the same length

Type boolean, default=True

3.3.1 Initialization

VariableDataLoader.**__init__**(*X*, *y*, *index=False*, *batch_size=1*, *shuffle=True*)
Load data from variable length inputs

Parameters

- **X** (*iterable of shape=(n_samples,)*) – Input sequences Each item in iterable should be a sequence (of variable length)
- **y** (*iterable of shape=(n_samples,)*) – Labels corresponding to X
- **index** (*boolean, default=False*) – If True, also returns original index
- **batch_size** (*int, default=1*) – Size of each batch to output
- **shuffle** (*boolean, default=True*) – If True, shuffle the data randomly, each yielded batch contains only input items of the same length

3.3.2 Iterable

The VariableDataLoader is an iterable object that iterates through the entire dataset. The same object can be called multiple times due to the reset method, which automatically resets the iterable after a complete iteration. Note that it can also be set manually using the `reset()` method.

VariableDataLoader.**reset**()
Reset the VariableDataLoader

Index

Symbols

B

F

I

L

M

P

R

S

V