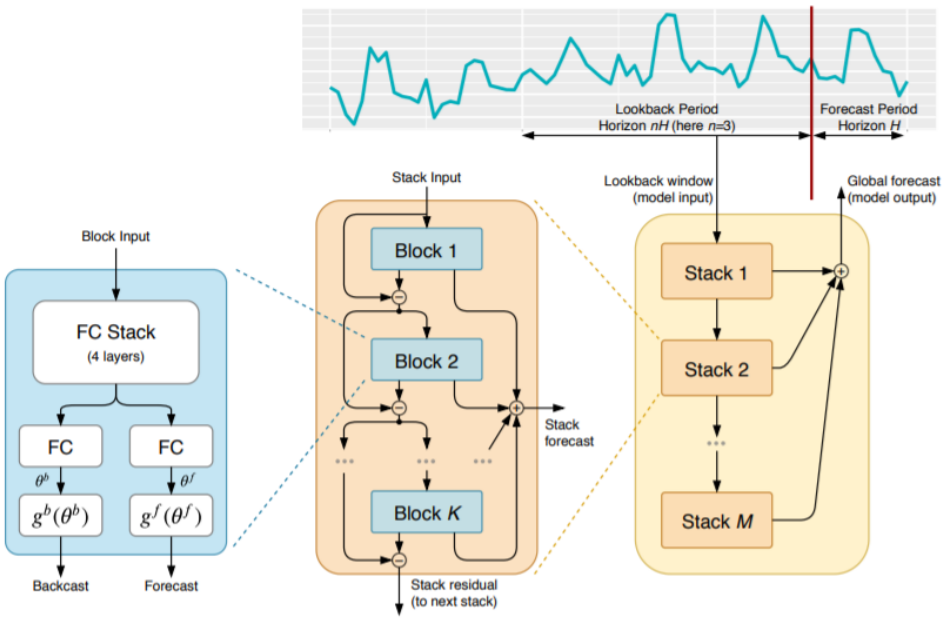
Python

unknown

Dec 15, 2022

CONTENTS

An easy, accessible way to use the NBeats model architecture in Keras.



CHAPTER ONE

INTRODUCTION

The motivation for this project was to take the NBeats model architecture defined in the original paper here: https://arxiv.org/abs/1905.10437 and reproduce it in a widely accessible form in keras. In the past few years this model has become very popular as a timer series forecasting tool, but its implementation in keras seemed elusive, without an easy-to-use, well documented option online that'd be simple for newcomers to try. This package allows anyone with basic knowledge of keras to be able to quickly use NBeats in its generic and interpretable form with just a few function calls.

To that end, KerasBeats was built with the following ideas in mind:

- It should reflect the original model architecture as closely as possible.
- It should have a simple, high level architecture that allows people to get started as quickly as possible using the familar `fit/predict` methods that everyone is already familiar with
- It should allow you to quickly and easily use it as a keras model to take advantage of the libraries existing functionality and enable its use in existing workflows

CHAPTER TWO

INSTALLATION

kerasbeats can be installed with the following line:

```
pip install keras-beats
```

CHAPTER

THREE

BASIC USAGE

The N-Beats model architecture assumes that you take a univariate time series and create training data that contains previous values for an observation at a particular point in time. For example, let's assume you have the following univariate time series:

```
# sample time series values
time_vals = [1, 2, 3, 4, 5]
```

If you were predicting one period ahead and wanted to use the previous two values in the time series as input, you want your data to be formatted like this:

```
# data formatting for N-beats
# each row represents the previous two values for the currently observed one
X = [[1, 2],
     [2, 3],
     [3, 4]]

y = [[3],
     [4],
     [5]]
```

The idea here is that `[1, 2]` were the two values that preceded `3`, `[2, 3]` were the two that preceeded `4`, and so on.

Once your input data is formatted like this then you can use `kerasbeats` in the following way:

```
from kerasbeats import NBeatsModel
mod = NBeatsModel()
mod.fit(X, y)
```

When you are finished fitting your model you can use the `predict` and `evaluate` methods, which are just wrappers on the original keras methods, and would work in the same way.

CHAPTER

FOUR

DATA PREP

Most time series data typically comes in column format, so a little data prep is usually needed before you can feed it into `kerasbeats`. You can easily do this yourself, but there are some built in functions in the `kerasbeats` package to make this a little easier.

CHAPTER

FIVE

UNIVARIATE TIME SERIES

If you have a single time series, you can use the `prep_time_series` function to get your data in the appropriate format. It works like this:

```
from kerasbeats import prep_time_series
# sample data:  a mock time series with ten values
time_vals = np.arange(10)
windows, labels = prep_time_series(lookback = 5, horizon = 1)
```

Once you are done with this the value of `windows` will be the following numpy array:

```
# training window of 5 values
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7],
       [4, 5, 6, 7, 8]])
```

The value of *labels* will be the following numpy array:

```
# the value that followed the preceeding 5
array([[5],
       [6],
       [7],
       [8],
       [9]])
```

This method accepts numpy arrays, lists, and pandas Series and DataFrames as input, but they must be one column if they are not then you'll receive an error message.

The function contains two separate arguments:

- **horizon:** how far out into the future you want to predict. A horizon value of 1 means you are predicting one step ahead. A value of two means you are predicting two steps ahead, and so on
- **lookback:** what multiple of the *horizon* you want to use for training data. So if *horizon* is 1 and *lookback* is 5, your training window will be the previous 5 values. If *horizon* is 2 and *lookback* is 5, then your training window will be the previous 10 values.

CHAPTER

SIX

MULTIPLE TIME SERIES

You could conceivably use *kerasbeats* to learn a combination of time series jointly, assuming they shared common patterns between them.

For example, here's a simple dataset that contains two different time series in a dataframe:

```
import pandas as pd

df = pd.DataFrame()
df['label'] = ['a'] * 10 + ['b'] * 10
df['value'] = [i for i in range(10)] * 2
```

`df` would look like this in a jupyter notebook:

	label	value
0	a	0
1	a	1
2	a	2
3	a	3
4	a	4
5	a	5
6	a	6
7	a	7
8	a	8
9	a	9
10	b	0
11	b	1
12	b	2
13	b	3
14	b	4
15	b	5
16	b	6
17	b	7
18	b	8
19	b	9

This contains two separate time series, one for value `a`, and another for value `b`. If you want to prep your data so each time series for each label is turned into its corresponding training windows and labels you can use the `prep_multiple_time_series` function:

```python
from kerasbeats import prep_multiple_time_series
windows, labels = prep_multiple_time_series(df, label_col = 'label', data_col = 'value', lookback = 5, horizon = 2)
```

This function will perform the `prep_time_series` function for each unique value specified in the `label_col` column and then concatenate them together in the end, and you can then pass `windows` and `labels` into the `NBeatsModel`.

CHAPTER

SEVEN

KERASBEATS LAYER

The NBeatsModel is an abstraction over a functional keras model. You may just want to use the underlying keras primitives in your own work without the very top of the model itself.

The basic building block of kerasbeats is a custom keras layer that contains all of the N-Beats blocks stacked together. If you want access to this layer directly you can call the build_layer method:

```
from kerasbeats import NBeatsModel
model = NBeatsModel()
model.build_layer()
```

This exposes the layer attribute, which is a keras layer that can be re-used in larger, multi-faceted models if you would like.

CHAPTER

EIGHT

USING KERASBEATS AS A KERAS MODEL

Likewise, you may want to access some underlying keras functionality that's not directly available in `NBeatsModel`. In particular, when you call `fit` using the `NBeatsModel` wrapper, the `compile` step is done for you automatically.

However, if you wanted to define your own separate loss functions, or define callbacks, you can access the fully built keras model in the following way:

```
nbeats = NBeatsModel()
nbeats.build_layer()
nbeats.build_model()
```

After these two lines, you can access the `model` attribute, which will give you access to the full keras model.

So if you wanted to specify a different loss function or optimizer, you could do so easily:

```
nbeats.model.compile(loss = 'mse',
                     optimizer = tf.keras.optimizers.RMSProp(0.001))
nbeats.model.fit(windows, labels)
```

Please note that if you want to use the underlying keras model directly, you should use `nbeats.model.fit()` and not `nbeats.fit`, since it will try and compile the model for you automatically after you call it.

CHAPTER

NINE

API REFERENCE

9.1 Important Caveats

The following sections describe the different functions and classes available in `kerasbeats`. A few important notes:

- All of the default values are designed to mimic what was originally in the paper, but are not necessarily best for your project
- batch size and learning rate are parameters you can usually specify inside keras itself, but they are defined at initialization because these values were explicitly mentioned in the paper. If you want to set them yourself, you should call `NBeatsModel().build_layer().build_model()` in order to set these values in keras directly.
- If you are going to use the interpretable model, you should probably set `horizon` to at least 2. Due to the way some other parameters are specified, the matrix math typically only works if you are predicting more than one day out.

9.2 NBeatsModel

class nbeats.**NBeatsModel**(*model_type: str = 'generic'*, *lookback: int = 7*, *horizon: int = 1*, *num_generic_neurons: int = 512*, *num_generic_stacks: int = 30*, *num_generic_layers: int = 4*, *num_trend_neurons: int = 256*, *num_trend_stacks: int = 3*, *num_trend_layers: int = 4*, *num_seasonal_neurons: int = 2048*, *num_seasonal_stacks: int = 3*, *num_seasonal_layers: int = 4*, *num_harmonics: int = 1*, *polynomial_term: int = 3*, *loss: str = 'mae'*, *learning_rate: float = 0.001*, *batch_size: int = 1024*)

__init__(*model_type: str = 'generic'*, *lookback: int = 7*, *horizon: int = 1*, *num_generic_neurons: int = 512*, *num_generic_stacks: int = 30*, *num_generic_layers: int = 4*, *num_trend_neurons: int = 256*, *num_trend_stacks: int = 3*, *num_trend_layers: int = 4*, *num_seasonal_neurons: int = 2048*, *num_seasonal_stacks: int = 3*, *num_seasonal_layers: int = 4*, *num_harmonics: int = 1*, *polynomial_term: int = 3*, *loss: str = 'mae'*, *learning_rate: float = 0.001*, *batch_size: int = 1024*)

Model used to create and initialize N-Beats model described in the following paper: https://arxiv.org/abs/1905.10437

inputs:

model
what model architecture to use. Must be one of ['generic', 'interpretable']

lookback
what multiplier of the forecast size you want to use for your training window

horizon
how many steps into the future you want your model to predict

num_generic_neurons
The number of neurons (columns) you want in each Dense layer for the generic block

num_generic_stacks
How many generic blocks to connect together

num_generic_layers
Within each generic block, how many dense layers do you want each one to have. If you set this number to 4, and num_generic_neurons to 128, then you will have 4 Dense layers with 128 neurons in each one

num_trend_neurons
Number of neurons to place within each Dense layer in each trend block

num_trend_stacks
number of trend blocks to stack on top of one another

num_trend_layers
number of Dense layers inside a trend block

num_seasonal_neurons
size of Dense layer in seasonal block

num_seasonal_stacks
number of seasonal blocks to stack on top on top of one another

num_seasonal_layers
number of Dense layers inside a seasonal block

num_harmonics
seasonal term to use for seasonal stack

polynomial_term
size of polynomial expansion for trend block

loss
what loss function to use inside keras. accepts any regression loss function built into keras. You can find more info here: https://keras.io/api/losses/regression_losses/

learning_rate
learning rate to use when training the model

batch_size
batch size to use when training the model

Returns
self

`build_layer()`

Initializes the Nested NBeats layer from initial parameters

attributes:

model_layer
custom keras layer that contains all of the generic, seasonal and trend layers stacked toger

Returns
self

`build_model()`

Creates keras model to use for fitting

attributes:

model
keras model that contains NBeats model layers as well as inputs, put into the keras Model class

Returns
self

`evaluate`(*y_true*, *y_pred*, ***kwargs*)

Passes predicted and true labels back to the original keras model

inputs:

y_true
numpy array or tensorflow with true labels

y_pred
numpy array or tensorflow with predicted labels

kwargs
any additional arguments you'd like to pass to the base keras model

Returns
list with specified evaluation metrics

`fit`(*X*, *y*, ***kwargs*)

Build and fit model

inputs:

X
tensor or numpy array with training windows

y
tensor or numpy array with the target values to be predicted

kwargs
any additional arguments you'd like to pass to the base keras model

attributes:

model_layer
custom keras layer that contains nested Generic, Trend, and Seasonal NBeats blocks

model
keras Model class that connects inputs to the model layer

Returns
self

`predict`(*X*, ***kwargs*)

Passes inputs back to original keras model for prediction

inputs:

X
tensor of numpy array with input data

kwargs
any additional arguments you'd like to pass to the base keras model

Returns
numpy array that contains model predictions for each sample

9.3 prep_time_series

`utilities.`**`prep_time_series`**`(`*data*, *lookback: int = 7*, *horizon: int = 1) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>*`)`

Creates windows and their corresponding labels for each unique time series in a dataset

E.g. if horizon = 2 and lookback = 3 Input: [1, 2, 3, 4, 5, 6, 7, 8] -> Output: ([1, 2, 3, 4, 5, 6], [7, 8])

Training window goes back by 3 * 2 values

inputs:

data
univariate time series you want to create windows for. Can be pandas dataframe, numpy array or list

lookback
multiple of forecast horizon that you want to use for training window

horizon
how far out into the future you want to predict

Returns
tuple with data types: (np.ndarray, np.ndarray) containing training windows and labels

9.4 prep_multiple_time_series

`utilities.`**`prep_multiple_time_series`**`(`*data*, *label_col: str*, *data_col: str*, *lookback: int = 7*, *horizon: int = 1) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>*`)`

Creates training windows for time series that are stacked on top of each other

example:

inputs = [['ar', 1]
['ar', 2], ['ar', 3], ['br', 5], ['br', 6], ['br', 7]]

outputs = [[1, 2], [[3],
[5, 6]], [7]]

It treats the values associated with 'ar' and 'br' as separate time series

inputs:

data
pandas DataFrame that has at least two columns, one that are labels for each unique time series in your dataset, and another that are the timeseries values

label_col
the name of the column that labels each time series

data_col
the column that contains the time series values

lookback
what multiple of your horizon you want your training data to be eg – a horizon of 2 and lookback of 5 creates a training window of 10

horizon
how far into the future you want to predict

Returns
tuple with data types: (np.ndarray, np.ndarray) containing training windows and labels for the concatenated time series

CHAPTER TEN

ADDITIONAL HELP

If you would like this work extended in areas that are specific to your enterprise, you may submit a request here:

PYTHON MODULE INDEX

INDEX