Deep4Cast

Jul 01, 2019

Contents

CHAPTER 1

Forecasting for decision making under uncertainty

This package is under active development. Things may change :-).

`Deep4Cast` is a scalable machine learning package implemented in `Python` and `Torch`. It has a front-end API similar to `scikit-learn`. It is designed for medium to large time series data sets and allows for modeling of forecast uncertainties.

The network architecture is based on `WaveNet`. Regularization and approximate sampling from posterior predictive distributions of forecasts are achieved via `Concrete Dropout`.

1.1 Examples

Tutorial: M4 Daily

1.2 Authors

- Toby Bischoff
- Austin Gross
- Kenneth Tran

1.3 References

- Concrete Dropout is used for approximate posterior Bayesian inference.
- Wavenet is used as encoder network.

1.3.1 Getting Started

Deep4Cast is a deep learning-based forecasting solution based on PyTorch. It can be used to build forecasters based on PyTorch models that are trained over large sets of time series.

Main Requirements

- python 3.6
- pytorch 1.0

Installation

Deep4cast can be cloned from GitHub. Before installing we recommend setting up a clean virtual environment.

From the package directory install the requirements and then the package.

1.3.2 Tutorial: M4 Daily

This notebook is designed to give a simple introduction to forecasting using the Deep4Cast package. The time series data is taken from the M4 dataset, specifically, the `Daily` subset of the data.

[1]:
```
import numpy as np
import os
import pandas as pd
import datetime as dt
import matplotlib.pyplot as plt

import torch
from torch.utils.data import DataLoader

from deep4cast.forecasters import Forecaster
from deep4cast.models import WaveNet
from deep4cast.datasets import TimeSeriesDataset
import deep4cast.transforms as transforms
import deep4cast.metrics as metrics

# Make RNG predictable
np.random.seed(0)
torch.manual_seed(0)
# Use a gpu if available, otherwise use cpu
device = ('cuda' if torch.cuda.is_available() else 'cpu')

%matplotlib inline
```

Dataset

In this section we inspect the dataset, split it into a training and a test set, and prepare it for easy consuption with PyTorch-based data loaders. Model construction and training will be done in the next section.

[2]:
```
if not os.path.exists('data/Daily-train.csv'):
    !wget https://raw.githubusercontent.com/M4Competition/M4-methods/master/Dataset/
↪Train/Daily-train.csv -P data/
```

(continues on next page)

(continued from previous page)

```
if not os.path.exists('data/Daily-test.csv'):
    !wget https://raw.githubusercontent.com/M4Competition/M4-methods/master/Dataset/
↪Test/Daily-test.csv -P data/
```

[3]:
```
data_arr = pd.read_csv('data/Daily-train.csv')
data_arr = data_arr.iloc[:, 1:].values
data_arr = list(data_arr)
for i, ts in enumerate(data_arr):
    data_arr[i] = ts[~np.isnan(ts)][None, :]
```

Divide into train and test

We use the DataLoader object from PyTorch to build batches from the test data set.

However, we first need to specify how much history to use in creating a forecast of a given length: - horizon = time steps to forecast - lookback = time steps leading up to the period to be forecast

[4]:
```
horizon = 14
lookback = 128
```

We've also found that it is not necessary to train on the full dataset, so we here select a 10% random sample of time series for training. We will evaluate on the full dataset later.

[5]:
```
import random

data_train = []
for time_series in data_arr:
    data_train.append(time_series[:, :-horizon],)
data_train = random.sample(data_train, int(len(data_train) * 0.1))
```

We follow Torchvision in processing examples using Transforms chained together by Compose.

- `Tensorize` creates a tensor of the example.
- `LogTransform` natural logarithm of the targets after adding the offset (similar to torch.log1p).
- `RemoveLast` subtracts the final value in the `lookback` from both `lookback` and `horizon`.
- `Target` specifies which index in the array to forecast.

We need to perform these transformations to have input features that are of the unit scale. If the input features are not of unit scale (i.e., of O(1)) for all features, the optimizer won't be able to find an optimium due to blow-ups in the gradient calculations.

[6]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.LogTransform(targets=[0], offset=1.0),
    transforms.RemoveLast(targets=[0]),
    transforms.Target(targets=[0]),
])
```

`TimeSeriesDataset` inherits from Torch Datasets for use with Torch DataLoader. It handles the creation of the examples used to train the network using `lookback` and `horizon` to partition the time series.

The parameter 'step' controls how far apart consective windowed samples from a time series are spaced. For example, for a time series of length 100 and a setup with lookback 24 and horizon 12, we split the original time series into smaller

training examples of length 24+12=36. How much these examples are overlapping is controlled by the parameter `step` in `TimeSeriesDataset`.

[7]:
```
data_train = TimeSeriesDataset(
    data_train,
    lookback,
    horizon,
    step=1,
    transform=transform
)

# Create mini-batch data loader
dataloader_train = DataLoader(
    data_train,
    batch_size=512,
    shuffle=True,
    pin_memory=True,
    num_workers=1
)
```

Modeling and Forecasting

Temporal Convolutions

The network architecture used here is based on ideas related to WaveNet. We employ the same architecture with a few modifications (e.g., a fully connected output layer for vector forecasts). It turns out that we do not need many layers in this example to achieve state-of-the-art results, most likely because of the simple autoregressive nature of the data.

In many ways, a temporal convoluational architecture is among the simplest possible architecures that we could employ using neural networks. In our approach, every layer has the same number of convolutional filters and uses residual connections.

When it comes to loss functions, we use the log-likelihood of probability distributions from the `torch.distributions` module. This mean that if one supplues a normal distribution the likelihood of the transformed data is modeled as coming from a normal distribution.

[8]:
```
# Define the model architecture
model = WaveNet(input_channels=1,
                output_channels=1,
                horizon=horizon,
                hidden_channels=89,
                skip_channels=199,
                dense_units=156,
                n_layers=7)

print('Number of model parameters: {}.'.format(model.n_parameters))
print('Receptive field size: {}.'.format(model.receptive_field_size))

# Enable multi-gpu if available
if torch.cuda.device_count() > 1:
    print('Using {} GPUs.'.format(torch.cuda.device_count()))
    model = torch.nn.DataParallel(model)

# .. and the optimizer
optim = torch.optim.Adam(model.parameters(), lr=0.0008097436666349985)
```

(continues on next page)

(continued from previous page)

```
# .. and the loss
loss = torch.distributions.StudentT
```

```
Number of model parameters: 341347.
Receptive field size: 128.
Using 2 GPUs.
```

[9]:
```
# Fit the forecaster
forecaster = Forecaster(model, loss, optim, n_epochs=5, device=device)
forecaster.fit(dataloader_train, eval_model=True)
```

```
/home/austin/miniconda3/envs/d4cGithub/lib/python3.6/site-packages/torch/nn/parallel/
↪_functions.py:61: UserWarning: Was asked to gather along dimension 0, but all input␣
↪tensors were scalars; will instead unsqueeze and return a vector.
  warnings.warn('Was asked to gather along dimension 0, but all '
Epoch 1/5 [915731/915731 (100%)]       Loss: -1.863526 Elapsed/Remaining: 3m52s/15m30s
Training error: -2.67e+01.
Epoch 2/5 [915731/915731 (100%)]       Loss: -1.963631 Elapsed/Remaining: 11m21s/17m2s
Training error: -2.71e+01.
Epoch 3/5 [915731/915731 (100%)]       Loss: -1.983338 Elapsed/Remaining: 18m42s/
↪12m28s
Training error: -2.75e+01.
Epoch 4/5 [915731/915731 (100%)]       Loss: -1.974977 Elapsed/Remaining: 26m2s/6m30s
Training error: -2.78e+01.
Epoch 5/5 [915731/915731 (100%)]       Loss: -2.073579 Elapsed/Remaining: 33m20s/0m0s
Training error: -2.83e+01.
```

Evaluation

Before any evaluation score can be calculated, we load the held out test data.

[10]:
```
data_train = pd.read_csv('data/Daily-train.csv')
data_test = pd.read_csv('data/Daily-test.csv')
data_train = data_train.iloc[:, 1:].values
data_test = data_test.iloc[:, 1:].values

data_arr = []
for ts_train, ts_test in zip(data_train, data_test):
    ts_a = ts_train[~np.isnan(ts_train)]
    ts_b = ts_test
    ts = np.concatenate([ts_a, ts_b])[None, :]
    data_arr.append(ts)
```

[11]:
```
# Sequentialize the training and testing dataset
data_test = []
for time_series in data_arr:
    data_test.append(time_series[:, -horizon-lookback:])

data_test = TimeSeriesDataset(
    data_test,
    lookback,
    horizon,
    step=1,
    transform=transform
)
```

(continues on next page)

(continued from previous page)

```
dataloader_test = DataLoader(
    data_test,
    batch_size=1024,
    shuffle=False,
    num_workers=2
)
```

We need to transform the output forecasts. The output from the foracaster is of the form (n_samples, n_time_series, n_variables, n_timesteps). This means, that a point forcast needs to be calculated from the samples, for example, by taking the mean or the median.

[12]:
```
# Get time series of actuals for the testing period
y_test = []
for example in dataloader_test:
    example = dataloader_test.dataset.transform.untransform(example)
    y_test.append(example['y'])
y_test = np.concatenate(y_test)

# Get corresponding predictions
y_samples = forecaster.predict(dataloader_test, n_samples=100)
```

We calculate the symmetric MAPE.

[13]:
```
# Evaluate forecasts
test_smape = metrics.smape(y_samples, y_test)

print('SMAPE: {}%'.format(test_smape.mean()))
```

```
SMAPE: 3.1666347980499268%
```

1.3.3 Datasets

Inherits from pytorch datasets to allow use with pytorch dataloader.

class datasets.**TimeSeriesDataset**(*time_series*, *lookback*, *horizon*, *step*, *transform*, *static_covs=None*, *thinning=1.0*)

Takes a list of time series and provides access to windowed subseries for training.

Arguments:

- time_series (list): List of time series `numpy` arrays.
- lookback (int): Number of time steps used as input for forecasting.
- horizon (int): Number of time steps to forecast.
- step (int): Time step size between consecutive examples.
- transform (`transforms.Compose`): Specific transformations to apply to time series examples.
- static_covs (list): Static covariates for each item in `time_series` list.
- thinning (float): Fraction of examples to include.

1.3.4 Transformations

Transformations of the time series intended to be used in a similar fashion to torchvision.

class transforms.**Compose**(*transforms*)
Composes several transforms together.

List of transforms must currently begin with `ToTensor` and end with `Target`.

Args:

- transforms (list of `Transform` objects): list of transforms to compose.

Example:

```
>>> transforms.Compose([
>>>     transforms.ToTensor(),
>>>     transforms.LogTransform(targets=[0], offset=1.0),
>>>     transforms.Target(targets=[0]),
>>> ])
```

class transforms.**LogTransform**(*targets=None*, *offset=0.0*)
Natural logarithm of target covariate + *offset*.

$$y_i = log_e(x_i + \text{offset})$$

Args:

- offset (float): amount to add before taking the natural logarithm
- targets (list): list of indices to transform.

Example:

```
>>> transforms.LogTransform(targets=[0], offset=1.0)
```

class transforms.**RemoveLast**(*targets=None*)
Subtract final point in lookback window from all points in example.

Args:

- targets (list): list of indices to transform.

Example:

```
>>> transforms.RemoveLast(targets=[0])
```

class transforms.**Standardize**(*targets=None*)
Subtract the mean and divide by the standard deviation from the lookback.

Args:

- targets (list): list of indices to transform.

Example:

```
>>> transforms.Standardize(targets=[0])
```

class transforms.**Target**(*targets*)
Retain only target indices for output.

Args:

- targets (list): list of indices to retain.

Example:

```
>>> transforms.Target(targets=[0])
```

class transforms.**ToTensor**(*device='cpu'*)
Convert numpy.ndarrays to tensor.

Args:

- device (str): device on which to load the tensor.

Example:

```
>>> transforms.ToTensor(device='cpu')
```

1.3.5 Models

class models.**WaveNet**(*input_channels*, *output_channels*, *horizon*, *hidden_channels=64*, *skip_channels=64*, *dense_units=128*, *n_layers=7*, *n_blocks=1*, *dilation=2*)
Implements *WaveNet* architecture for time series forecasting. Inherits from pytorch Module. Vector forecasts are made via a fully-connected layer.

References:

- WaveNet: A Generative Model for Raw Audio

Arguments:

- input_channels (int): Number of covariates in input time series.
- output_channels (int): Number of target time series.
- horizon (int): Number of time steps to forecast.
- hidden_channels (int): Number of channels in convolutional hidden layers.
- skip_channels (int): Number of channels in convolutional layers for skip connections.
- dense_units (int): Number of hidden units in final dense layer.
- n_layers (int): Number of layers per Wavenet block (determines receptive field size).
- n_blocks (int): Number of Wavenet blocks.
- dilation (int): Dilation factor for temporal convolution.

Inititalize variables.

decode(*inputs: <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d011fd0>*)
Returns forecasts based on embedding vectors.

Arguments:

- inputs: embedding vectors to generate forecasts for

encode(*inputs: <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d011f98>*)
Returns embedding vectors.

Arguments:

- inputs: time series input to make forecasts for

forward(*inputs*)
Forward function.

n_parameters
Returns the number of model parameters.

receptive_field_size
Returns the length of the receptive field.

1.3.6 Forecasters

Module that handles all forecaster objects for training PyTorch models.

class forecasters.**Forecaster**(*model*, *loss*, *optimizer*, *n_epochs=1*, *device='cpu'*, *checkpoint_path='./'*, *verbose=True*)
Handles training of a PyTorch model and can be used to generate samples from approximate posterior predictive distribution.

Arguments:

- model (`torch.nn.Module`): Instance of Deep4cast *models*.
- loss (`torch.distributions`): Instance of PyTorch distribution.
- optimizer (`torch.optim`): Instance of PyTorch optimizer.
- n_epochs (int): Number of training epochs.
- device (str): Device used for training (*cpu* or *cuda*).
- checkpoint_path (str): File system path for writing model checkpoints.
- verbose (bool): Verbosity of forecaster.

embed(*dataloader*, *n_samples=100*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d2f8048>
Generate embedding vectors.

Arguments:

- dataloader (`torch.utils.data.DataLoader`): Data to make embedding vectors.
- n_samples (int): Number of forecast samples.

fit(*dataloader_train*, *dataloader_val=None*, *eval_model=False*)
Fits a model to a given a dataset.

Arguments:

- dataloader_train (`torch.utils.data.DataLoader`): Training data.
- dataloader_val (`torch.utils.data.DataLoader`): Validation data.
- eval_model (bool): Flag to switch on model evaluation after every epoch.

predict(*dataloader*, *n_samples=100*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d20feb8>
Generates predictions.

Arguments:

- dataloader (`torch.utils.data.DataLoader`): Data to make forecasts.
- n_samples (int): Number of forecast samples.

1.3.7 Metrics

Common evaluation metrics for time series forecasts.

metrics.**acd**(*data_samples*, *data_truth*, *alpha=0.05*, ***kwargs*) → float
The absolute difference between the coverage of the method and the target (0.95).

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- alpha (float): percentile to compute coverage difference

`metrics.`**`coverage`**(*data_samples*, *data_truth*, *percentiles=None*, ***kwargs*) → list

Computes coverage rates of the prediction interval.

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- percentiles (list): percentiles to calculate coverage for

`metrics.`**`mae`**(*data_samples*, *data_truth*, *agg=None*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d074668>

Computes mean absolute error (MAE)

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- agg: Aggregation function applied to sampled predictions (defaults to `np.median`).

`metrics.`**`mape`**(*data_samples*, *data_truth*, *agg=None*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d0746a0>

Computes mean absolute percentage error (MAPE)

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- agg: Aggregation function applied to sampled predictions (defaults to `np.median`).

`metrics.`**`mase`**(*data_samples*, *data_truth*, *data_insample*, *frequencies*, *agg=None*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d074748>

Computes mean absolute scaled error (MASE) as in the M4 competition.

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- data_insample (`np.array`): In-sample time series data (n_timeseries, n_variables, n_timesteps).
- frequencies (list): Frequencies to be used when calculating the naive forecast.
- agg: Aggregation function applied to sampled predictions (defaults to `np.median`).

`metrics.`**`mse`**(*data_samples*, *data_truth*, *agg=None*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d074710>

Computes mean squared error (MSE)

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- agg: Aggregation function applied to sampled predictions (defaults to `np.median`).

`metrics.`**`msis`**(*data_samples*, *data_truth*, *data_insample*, *frequencies*, *alpha=0.05*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d0747f0>

Mean Scaled Interval Score (MSIS) as shown in the M4 competition.

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- data_insample (`np.array`): In-sample time series data (n_timeseries, n_variables, n_timesteps).
- frequencies (list): Frequencies to be used when calculating the naive forecast.
- alpha (float): Significance level.

`metrics.`**`pinball_loss`**(*data_samples*, *data_truth*, *percentiles=None*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d0747b8>

Computes pinball loss.

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- percentiles (list): Percentiles used to calculate coverage.

`metrics.`**`rmse`**(*data_samples*, *data_truth*, *agg=None*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d074780>

Computes mean squared error (RMSE)

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- agg: Aggregation function applied to sampled predictions (defaults to `np.median`).

`metrics.`**`smape`**(*data_samples*, *data_truth*, *agg=None*, ***kwargs*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f019d0746d8>

Computes symmetric mean absolute percentage error (SMAPE) on the mean

Arguments:

- data_samples (`np.array`): Sampled predictions (n_samples, n_timeseries, n_variables, n_timesteps).
- data_truth (`np.array`): Ground truth time series values (n_timeseries, n_variables, n_timesteps).
- agg: Aggregation function applied to sampled predictions (defaults to `np.median`).

1.3.8 Custom Layers

Custom layers that can be used to build extended PyTorch models for forecasting.

References:

- Concrete Dropout is used for approximate posterior Bayesian inference.

class custom_layers.**ConcreteDropout**(*dropout_regularizer=1e-05*, *init_range=(0.1*, *0.3)*, *channel_wise=False*)

Applies Dropout to the input, even at prediction time and learns dropout probability from the data.

In convolutional neural networks, we can use dropout to drop entire channels using the 'channel_wise' argument.

Arguments:

- dropout_regularizer (float): Should be set to 2 / N, where N is the number of training examples.
- init_range (tuple): Initial range for dropout probabilities.
- channel_wise (boolean): apply dropout over all input or across convolutional channels.

forward(*x*)

Returns input but with randomly dropped out values.

Python Module Index

Index