Fit sliding window of stock markets

[2]

```
!pip install git+https://github.com/deepmind/dm-haiku
!pip install git+https://github.com/jamesvuc/jax-bayes
```

```
Successfully installed dm-haiku-0.0.10.dev0 jmp-0.0.2
Looking in indexes: https://pypi.org/simple, https://us-
python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/jamesvuc/jax-bayes
  Cloning https://github.com/jamesvuc/jax-bayes to /tmp/pip-req-build-
ab1lfcw2
  Running command git clone -q https://github.com/jamesvuc/jax-bayes
/tmp/pip-req-build-ab1lfcw2
Requirement already satisfied: absl-py>=0.9.0 in
/usr/local/lib/python3.8/dist-packages (from jax-bayes==0.1.1) (1.3.0)
Requirement already satisfied: numpy>=1.18.0 in
/usr/local/lib/python3.8/dist-packages (from jax-bayes==0.1.1) (1.21.6)
Requirement already satisfied: opt-einsum>=3.3.0 in
/usr/local/lib/python3.8/dist-packages (from jax-bayes==0.1.1) (3.3.0)
Requirement already satisfied: protobuf>=3.12.4 in
/usr/local/lib/python3.8/dist-packages (from jax-bayes==0.1.1) (3.19.6)
Requirement already satisfied: scipy>=1.5.2 in
/usr/local/lib/python3.8/dist-packages (from jax-bayes==0.1.1) (1.7.3)
Requirement already satisfied: six>=1.15.0 in
/usr/local/lib/python3.8/dist-packages (from jax-bayes==0.1.1) (1.15.0)

Requirement already satisfied: tqdm>=4.48.2 in
/usr/local/lib/python3.8/dist-packages (from jax-bayes==0.1.1) (4.64.1)
Building wheels for collected packages: jax-bayes
  Building wheel for jax-bayes (setup.py) ... done
  Created wheel for jax-bayes: filename=jax_bayes-0.1.1-py3-none-any.whl
size=1031680
sha256=e19eb05a1713067d74c24fae1af3d7293922652c6781b6f347df9a3644a852da
  Stored in directory: /tmp/pip-ephem-wheel-cache-
67s04j3k/wheels/3f/7b/9c/326882f09afedfadf20a391de383da7aaea36b633d5e17555f
Successfully built jax-bayes
Installing collected packages: jax-bayes
Successfully installed jax-bayes-0.1.1
```

[3]

```
!git clone https://github.com/stevengogogo/Bayesianneuralnet_stockmarket
```

```
Cloning into 'Bayesianneuralnet_stockmarket'...
remote: Enumerating objects: 960, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 960 (delta 3), reused 8 (delta 2), pack-reused 950
Receiving objects: 100% (960/960), 217.92 MiB | 14.87 MiB/s, done.
Resolving deltas: 100% (167/167), done.
```

[4]

```
import os
import os.path as osp
```

```python
import numpy as np
# %%
import numpy as np
np.random.seed(0)

import haiku as hk
import pandas as pd
import jax.numpy as jnp
import jax

from tqdm import tqdm, trange
from matplotlib import pyplot as plt

from jax_bayes.utils import confidence_bands
from jax_bayes.mcmc import (
    langevin_fns,
    # mala_fns,
    # hmc_fns,
)
```

Data processing

The original data set is $x_t = \{x_1, \ldots, x_{Total}\}$, the training input is a matrix with dimension $m \times s$ (where m is the capture window, and s is the number of samples). The sample is produced by shifting the original time series with lag of 2.

- Training set

$$\underbrace{\begin{bmatrix} x_{1+(t-1)T} & \cdots & x_{m+(t-1)T} \\ x_{3+(t-1)T} & \cdots & x_{2m+3+(t-1)T} \\ \vdots & \vdots & \vdots \end{bmatrix}}_{\text{m(Capture windows)}}$$

$$\underbrace{\begin{bmatrix} x_{m+(t-1)T+1} & \cdots & x_{m+(t-1)T+n} \\ x_{2m+3+(t-1)T+1} & \cdots & x_{2m+3+(t-1)T+n} \\ \vdots & \vdots & \vdots \end{bmatrix}}_{\text{n (Prediction Horizons)}}$$

- m: embedding dimension (predicting horizon)
- T: time lag

[5]

```python
data_path_base = 'Bayesianneuralnet_stockmarket/code/datasets'
def get_orig(sig, shift=2):
    return np.concatenate((sig[0,:].ravel(), sig[1:,-shift:].ravel()))
```

```
# horizon
timesteps = 5
steps_ahead = 5

# load
train = np.loadtxt(open(os.path.join(data_path_base, "MMM8_train.txt")))
train.shape
```

```
(804, 10)
```

[6]

```
pd.DataFrame(train)
```

	0	1	2	3	4	5	6
0	0.000554	0.003739	0.001985	0.000000	0.002308	0.004293	0
1	0.001985	0.000000	0.002308	0.004293	0.001846	0.004200	0
2	0.002308	0.004293	0.001846	0.004200	0.001062	0.003970	0
3	0.001846	0.004200	0.001062	0.003970	0.007847	0.011216	0
4	0.001062	0.003970	0.007847	0.011216	0.010524	0.010339	0
...	...	...	...	...	...	...	..
799	0.719476	0.720176	0.723407	0.705364	0.693515	0.701704	0
800	0.723407	0.705364	0.693515	0.701704	0.694112	0.709296	0
801	0.693515	0.701704	0.694112	0.709296	0.694166	0.692539	0
802	0.694112	0.709296	0.694166	0.692539	0.696552	0.694491	0
803	0.694166	0.692539	0.696552	0.694491	0.676650	0.692593	0

804 rows × 10 columns

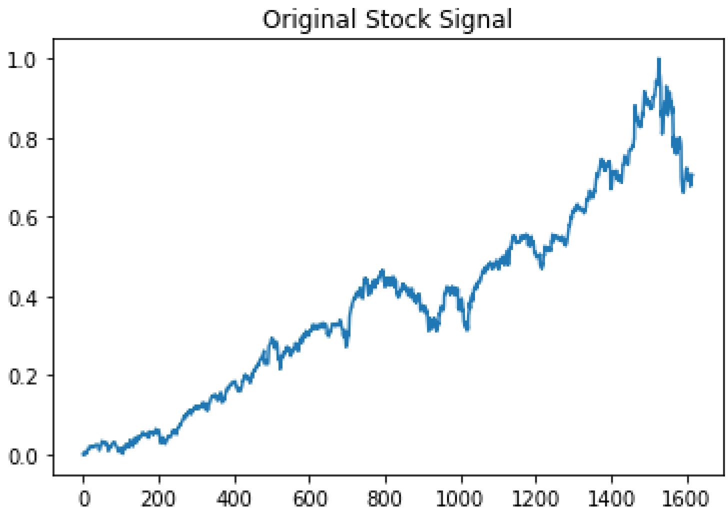
[7]

```
# original time-series
orig = get_orig(train)

pd.DataFrame(orig[0:13])
```

	0
0	0.000554
1	0.003739
2	0.001985
3	0.000000

	0
4	0.002308
5	0.004293
6	0.001846
7	0.004200
8	0.001062
9	0.003970
10	0.007847
11	0.011216
12	0.010524

[8]
```
plt.plot(orig)
plt.title("Original Stock Signal");
```



[9]
```
train.shape
```

```
(804, 10)
```

[10]
```
x_train = train[:, :timesteps]
y_train = train[:, timesteps: timesteps + steps_ahead]
xy_train = (x_train, y_train)
```

[11]
```
pd.DataFrame(x_train)
```

	0	1	2	3	4

	0	1	2	3	4

0	0.000554	0.003739	0.001985	0.000000	0.002308
1	0.001985	0.000000	0.002308	0.004293	0.001846
2	0.002308	0.004293	0.001846	0.004200	0.001062
3	0.001846	0.004200	0.001062	0.003970	0.007847
4	0.001062	0.003970	0.007847	0.011216	0.010524
...	...	...	...	...	...
799	0.719476	0.720176	0.723407	0.705364	0.693515
800	0.723407	0.705364	0.693515	0.701704	0.694112
801	0.693515	0.701704	0.694112	0.709296	0.694166
802	0.694112	0.709296	0.694166	0.692539	0.696552
803	0.694166	0.692539	0.696552	0.694491	0.676650

804 rows × 5 columns

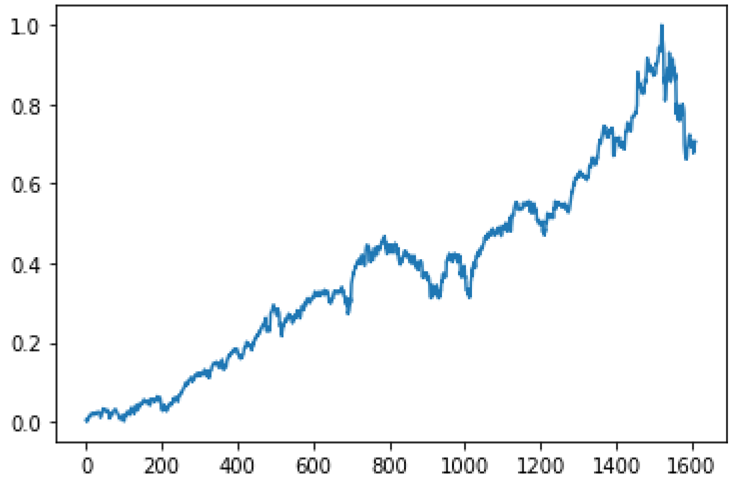
[12]
```
pd.DataFrame(y_train)
```

	0	**1**	**2**	**3**	**4**
0	0.004293	0.001846	0.004200	0.001062	0.003970
1	0.004200	0.001062	0.003970	0.007847	0.011216
2	0.003970	0.007847	0.011216	0.010524	0.010339
3	0.011216	0.010524	0.010339	0.011816	0.014355
4	0.010339	0.011816	0.014355	0.019432	0.018879
...	...	...	...	...	...
799	0.701704	0.694112	0.709296	0.694166	0.692539
800	0.709296	0.694166	0.692539	0.696552	0.694491
801	0.692539	0.696552	0.694491	0.676650	0.692593
802	0.694491	0.676650	0.692593	0.684730	0.697528
803	0.692593	0.684730	0.697528	0.705500	0.706259

804 rows × 5 columns

[13]
```
plt.plot(get_orig(y_train))
```

```
[<matplotlib.lines.Line2D at 0x7ff36bc24520>]
```




```
[14]  x_train.shape
```

```
(804, 5)
```

```
[15]  y_train.shape
```

```
(804, 5)
```

```
[16]  ## BNN training

      #could use any of the samplers modulo hyperparameters
      # sampler_fns = hmc_fns
      sampler_fns = langevin_fns
      # sampler_fns = mala_fns



      def net_fn(x):

          mlp = hk.Sequential([
              hk.Linear(10, w_init=hk.initializers.Constant(0),
                             b_init=hk.initializers.Constant(0)),
              jax.nn.sigmoid,
              hk.Linear(5,   w_init=hk.initializers.Constant(0),
                             b_init=hk.initializers.Constant(0))
              ])

          return mlp(x)
```

```
[17]  lr = 1e-4
      reg = 0.1
      lik_var = 0.5
```

```python
net = hk.transform(net_fn)
key = jax.random.PRNGKey(0)

sampler_init, sampler_propose, sampler_accept, sampler_update, sampler_ge
    sampler_fns(key, num_samples=30, step_size=lr, init_stddev=5.0)
```

[18]

```python
def logprob(params, xy):
    """ log posterior, assuming
    P(params) ~ N(0,eta)
    P(y|x, params) ~ N(f(x;params), lik_var)
    """
    x, y = xy

    preds = net.apply(params, None, x)
    log_prior = - reg * sum(jnp.sum(jnp.square(p))
                        for p in jax.tree_leaves(params))
    log_lik = - jnp.mean(jnp.square(preds - y)) / lik_var
    return log_lik + log_prior

@jax.jit
def sampler_step(i, state, keys, batch):
    # print(state)
    # input()
    params = sampler_get_params(state)
    logp = lambda params:logprob(params, batch)
    fx, dx = jax.vmap(jax.value_and_grad(logp))(params)

    fx_prop, dx_prop = fx, dx
    # fx_prop, prop_state, dx_prop, new_keys = fx, state, dx, keys
    prop_state, keys = sampler_propose(i, dx, state, keys)

    # for RK-langevin and MALA --- recompute gradients
    prop_params = sampler_get_params(prop_state)
    fx_prop, dx_prop = jax.vmap(jax.value_and_grad(logp))(prop_params)

    # for HMC
    # prop_state, dx_prop, keys = state, dx, keys
    # for j in range(5): #5 iterations of the leapfrog integrator
    #   prop_state, keys = \
    #       sampler_propose(i, dx_prop, prop_state, keys)

    #   prop_params = sampler_get_params(prop_state)
    #   fx_prop, dx_prop = jax.vmap(jax.value_and_grad(logp))(prop_params

    accept_idxs, keys = sampler_accept(
        i, fx, fx_prop, dx, state, dx_prop, prop_state, keys
    )
    state, keys = sampler_update(
        i, accept_idxs, dx, state, dx_prop, prop_state, keys
    )
```

```python
    return state, keys
```

[19]

```python
# initialization
params = net.init(jax.random.PRNGKey(42), x_train)
sampler_state, sampler_keys = sampler_init(params)
```

```
/usr/local/lib/python3.8/dist-packages/haiku/_src/initializers.py:69:
UserWarning: Explicitly requested dtype float64 requested in astype is not
available, and will be truncated to dtype float32. To enable more dtypes,
set the jax_enable_x64 configuration option or the JAX_ENABLE_X64 shell
environment variable. See https://github.com/google/jax#current-gotchas for
more.
  return jnp.broadcast_to(jnp.asarray(self.constant), shape).astype(dtype)
```

[20]

```python
params['linear']['w'].shape
```

```
(5, 10)
```

[21]

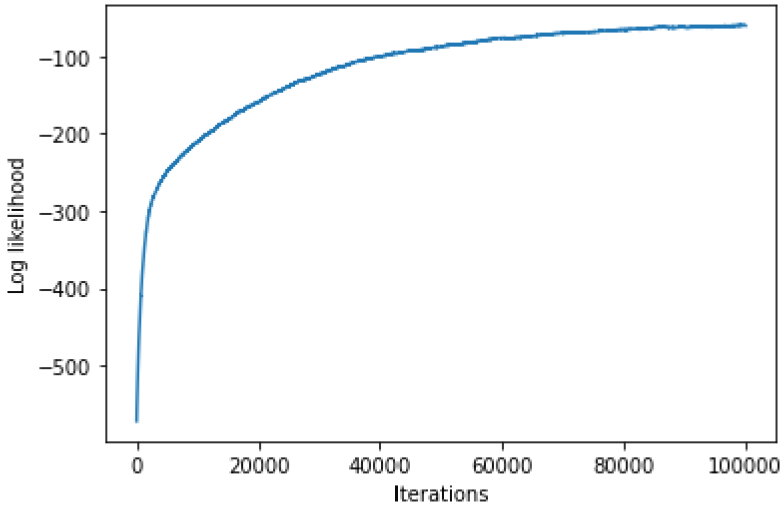
```python
#do the sampling
niter = 100000
train_logp = np.zeros(niter)
for step in trange(niter):
    # Training log
    sampler_params = sampler_get_params(sampler_state)
    logp = lambda params:logprob(params, xy_train)
    train_logp[step] = jnp.mean(jax.vmap(logp)(sampler_params))

    sampler_state, sampler_keys = \
        sampler_step(step, sampler_state, sampler_keys, xy_train)


sampler_params = sampler_get_params(sampler_state)
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]<ipython-input-18-
34c999bbf60f>:10: FutureWarning: jax.tree_leaves is deprecated, and will be
removed in a future release. Use jax.tree_util.tree_leaves instead.
  for p in jax.tree_leaves(params))
100%|██████████| 100000/100000 [22:19<00:00, 74.65it/s]
```

[22]

```python
# Training log
ftn, axtn = plt.subplots()
axtn.plot(train_logp)
axtn.set_xlabel("Iterations")
axtn.set_ylabel("Log likelihood")
#ftn.savefig("../img/training_MALA_{}-iter.pdf".format(niter))
```

```
Text(0, 0.5, 'Log likelihood')
```



[23]

```
outputs = jax.vmap(net.apply, in_axes=(0, None, None))(sampler_params, No
outputs.shape
```

```
(30, 804, 5)
```

[24]

```
pred_lines = np.array([ get_orig(outputs[i,:,:]) for i in range(0, output
pred_lines.shape
```

```
(30, 1611)
```

[25]

```
ms = jnp.mean(pred_lines, axis=0)
ss = jnp.std(pred_lines, axis=0)

lower, uper = ms-ss, ms+ss
```

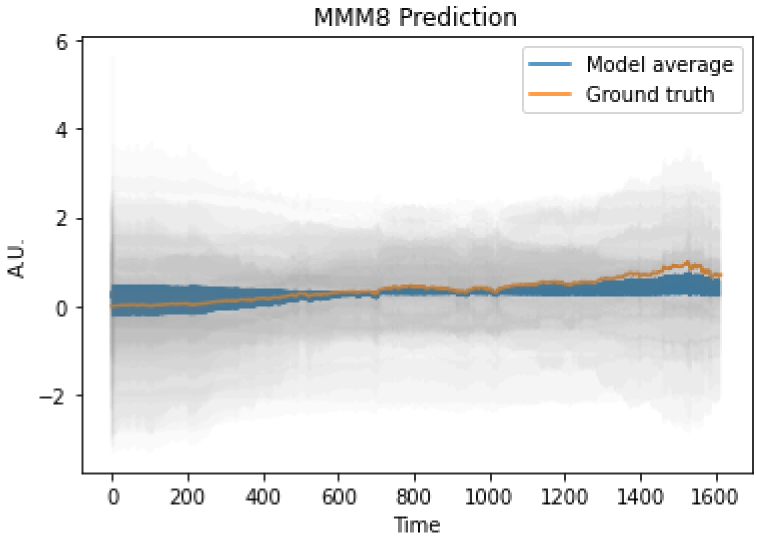
[26]

```
x = jax.device_put(outputs)
```

[27]

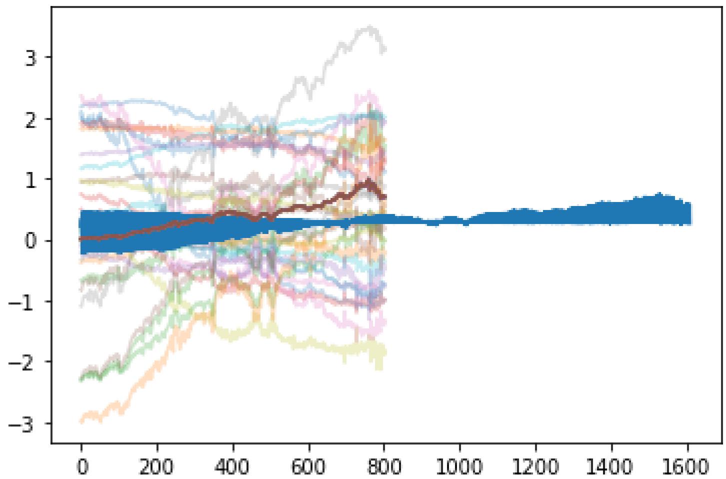
```
fmma, axmma = plt.subplots()
axmma.plot(ms, label="Model average")
axmma.plot(orig, label="Ground truth")
for i in range(outputs.shape[0]):
        axmma.plot(get_orig(outputs[i,:,:]), color="gray",alpha=0.04)
axmma.set_xlabel("Time")
axmma.set_ylabel("A.U.")
axmma.set_title("MMM8 Prediction")
axmma.legend()
#fmma.savefig("../img/prediction_MALA_{}-iter.pdf".format(niter))
```

```
<matplotlib.legend.Legend at 0x7ff3582de490>
```



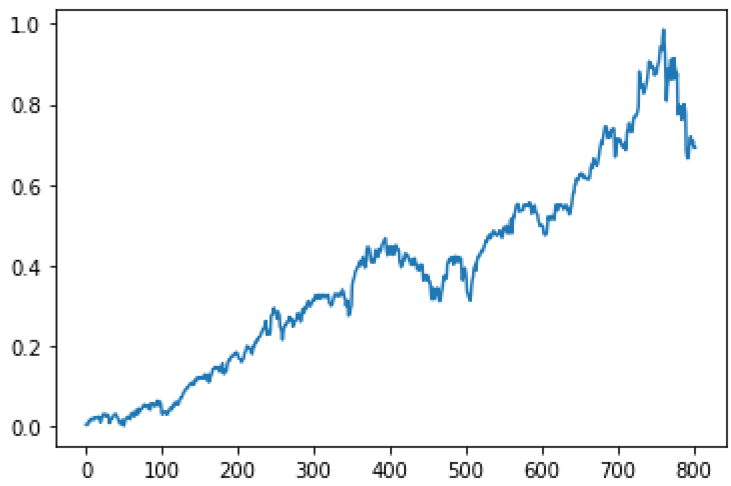

```
[28]  f, ax = plt.subplots(1)
      for i in range(outputs.shape[0]):
              ax.plot(outputs[i,:,0], alpha=0.25)
      ax.plot(ms, label="Mean")
      ax.plot(y_train)
```

```
[<matplotlib.lines.Line2D at 0x7ff3580b31f0>,
 <matplotlib.lines.Line2D at 0x7ff3580b3280>,
 <matplotlib.lines.Line2D at 0x7ff3580b3340>,
 <matplotlib.lines.Line2D at 0x7ff3580b3400>,
 <matplotlib.lines.Line2D at 0x7ff3580b34c0>]
```



```
[29]  plt.plot(y_train[:,0])
```

```
[<matplotlib.lines.Line2D at 0x7ff358013610>]
```


1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500
600
700
800