RSNN

August 14, 2021

[ ]:
```
from google.colab import drive
drive.mount('./drive')
```

```
Mounted at ./drive
```

[ ]:
```
import json
from tqdm.notebook import tqdm

data_path = '/content/drive/MyDrive/RSNN/'
data = json.load(open(data_path+'data.json'))
targets = json.load(open(data_path+'targets.json'))
words = json.load(open(data_path+'words.json'))
vocab = json.load(open(data_path+'vocab.json'))
```

[ ]:
```
# !pip install constant_properties_protector
# !pip install construction_requirements_integrator
# !pip install add_on_class
# !pip install matplotlib_dashboard

# !rm -rf Spiral/
# !git clone https://github.com/BehzadShayegh/Spiral

import sys
sys.path.insert(1, './Spiral/')

from spiral import (
    IntegrateAndFireSoma,
    LeakyMembrane,
    LinearDendrite,
    Axon,
    STDP,
    FullyConnectedSynapse,
    DisconnectorSynapticCover,
    RandomConnectivity,
    LeakyResponseFunction,
    ScalingResponseFunction,
    FlatResponseFunction,
    CompositeSynapticPlasticity,
```

```
    SynapticPlasticityRate,
    WeightDependentRate,
    ConvergentSynapticPlasticity,
    Network,
    OneHotEncoder,
    Object2IndexReceiver,
    KWinnersTakeAllPrinciple,
    ConstantSummationOfSynapticWeightsPrinciple,
    ConstantSummationOfLinearCoefficientsPrinciple,
    ConstantSummationOfAxonsUtilizationsPrinciple,
    KRandomClampsPrinciple,
)
from spiral.operators import *

LIF = (LeakyMembrane(IntegrateAndFireSoma))
O2IROHE = Object2IndexReceiver(OneHotEncoder)
KWLIF = KWinnersTakeAllPrinciple(LIF)
NormCoefDendrite = ConstantSummationOfLinearCoefficientsPrinciple(
    LinearDendrite
)
NormLinDendrite = ConstantSummationOfAxonsUtilizationsPrinciple(
    NormCoefDendrite
)
KCLIF = KRandomClampsPrinciple(LIF)

import torch
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[ ]:
```
batch_size = 4
scale = 500.

CONSIDER | Network(dt=1., batch=batch_size, global_plasticity=False)
###################################################
INSERT | O2IROHE(name='encoder', objects=vocab, default=vocab['<UKN>'],␣
↪unknown_exception=False)
# INSERT | KCLIF(
#     name='population',
#     shape=(110,),
#     clamps_distribution=lambda x: x.potential-x.potential.min()+1,
# )
INSERT | KWLIF(
            name='population',
            shape=(1100,),
            number_of_winners=50,
            kwinners_take_all_spare_evaluation_criteria=lambda x: x.potential
        )
###################################################
```

```
INSERT | (
    (
        FullyConnectedSynapse()
    ) | FROM | (
        Axon(
            response_function=ScalingResponseFunction(scale=scale)
        ) |OF| CONSIDERED.NETWORK['encoder']
    ) | TO | (
        NormLinDendrite(
            name='encoder_dendrite',
            plasticity=True,
            plasticity_model=CompositeSynapticPlasticity(
                synaptic_plasticities=[
                    STDP(
                        presynaptic_tagging=LeakyResponseFunction(tau=10),
                        postsynaptic_tagging=LeakyResponseFunction(tau=10),
                        ltp_rate=SynapticPlasticityRate(rate=0.1*batch_size/
↪scale),
                        ltd_rate=SynapticPlasticityRate(rate=0*batch_size/
↪scale),
                    ),
                    # ConvergentSynapticPlasticity(tau=1000),
                ]
            ),
            # maximum_weight=.5,
            # initial_weights=lambda shape: torch.rand(shape)/5,
            coefficients_sum=100,
            utilizations_sum=100,
        ) |OF| CONSIDERED.NETWORK['population']
    )
)
###################################################
# INSERT | (
#     (
#         FullyConnectedSynapse()
#     ) | FROM | (
#         Axon(
#             response_function=ScalingResponseFunction(scale=10),
#         ) |OF| CONSIDERED.NETWORK['population']
#     ) | TO | (
#         LinearDendrite(
#             name='main_dendrite',
#             plasticity=True,
#             plasticity_model=CompositeSynapticPlasticity(
#                 synaptic_plasticities=[
#                     STDP(
#                         presynaptic_tagging=LeakyResponseFunction(tau=10),
```

```
#                         postsynaptic_tagging=LeakyResponseFunction(tau=10),
#                         ltp_rate=WeightDependentRate(rate=0.1),
#                         ltd_rate=WeightDependentRate(rate=0.1),
#                     ),
#                     # ConvergentSynapticPlasticity(tau=2000),
#                 ]
#             ),
#             # initial_weights=torch.zeros,
#         ) |OF| CONSIDERED.NETWORK['population']
#     )
# )
###################################################
# INSERT | (
#     (
#         DisconnectorSynapticCover(FullyConnectedSynapse)(
#             connectivity_pattern=RandomConnectivity(
#                 rate=.1
#             )
#         )
#     ) | FROM | (
#         Axon(
#             response_function=ScalingResponseFunction(scale=100),
#         ) |OF| CONSIDERED.NETWORK['encoder']
#     ) | TO | (
#         LinearDendrite(
#             name='encoder_dendrite',
#             plasticity=False,
#             initial_weights=torch.ones,
#         ) |OF| CONSIDERED.NETWORK['population']
#     )
# )
###################################################
net = CHECKOUT | CONSIDERED.NETWORK
net.plasticity = True
net.to(device)
```

```
/usr/local/lib/python3.7/dist-packages/torch/_tensor.py:575: UserWarning:
floor_divide is deprecated, and will be removed in a future version of pytorch.
It currently rounds toward 0 (like the 'trunc' function NOT 'floor'). This
results in incorrect rounding for negative values.
To keep the current behavior, use torch.div(a, b, rounding_mode='trunc'), or for
actual floor division, use torch.div(a, b, rounding_mode='floor'). (Triggered
internally at  /pytorch/aten/src/ATen/native/BinaryOps.cpp:467.)
  return torch.floor_divide(self, other)
```

[ ]:
```
Network(
  (encoder): OneHotEncoderCoveredByObject2IndexReceiver(
```

```
    (encoder_Axon_0): Axon(
      (response_function): ScalingResponseFunction()
    )
  )
  (population):
IntegrateAndFireSomaCoveredByLeakyMembraneCoveredByKWinnersTakeAllPrinciple(
    (encoder_dendrite): LinearDendriteCoveredByConstantSummationOfLinearCoeffici
entsPrincipleCoveredByConstantSummationOfAxonsUtilizationsPrinciple(
      (_plasticity_model): CompositeSynapticPlasticity(
        (0): STDP(
          (presynaptic_tagging): LeakyResponseFunction()
          (postsynaptic_tagging): LeakyResponseFunction()
          (ltp_rate): SynapticPlasticityRate()
          (ltd_rate): SynapticPlasticityRate()
        )
      )
    )
  )
  (FullyConnectedSynapse_from_encoder_Axon_0_to_encoder_dendrite):
FullyConnectedSynapse(
    (_axon): Axon(
      (response_function): ScalingResponseFunction()
    )
    (_dendrite): LinearDendriteCoveredByConstantSummationOfLinearCoefficientsPri
ncipleCoveredByConstantSummationOfAxonsUtilizationsPrinciple(
      (_plasticity_model): CompositeSynapticPlasticity(
        (0): STDP(
          (presynaptic_tagging): LeakyResponseFunction()
          (postsynaptic_tagging): LeakyResponseFunction()
          (ltp_rate): SynapticPlasticityRate()
          (ltd_rate): SynapticPlasticityRate()
        )
      )
    )
  )
)
```

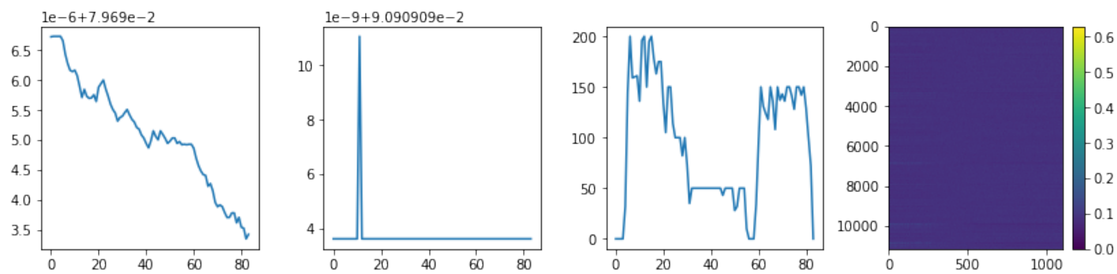
[ ]:
```
import matplotlib.pyplot as plt
from matplotlib_dashboard import MatplotlibDashboard

def plot(cli,mwi,activity,w):
  plt.figure(figsize=(14,3))
  md = MatplotlibDashboard([
          ['ci','mi','a','w'],
      ], hspace=.7, wspace=.3)

  md['ci'].plot(cli)
```

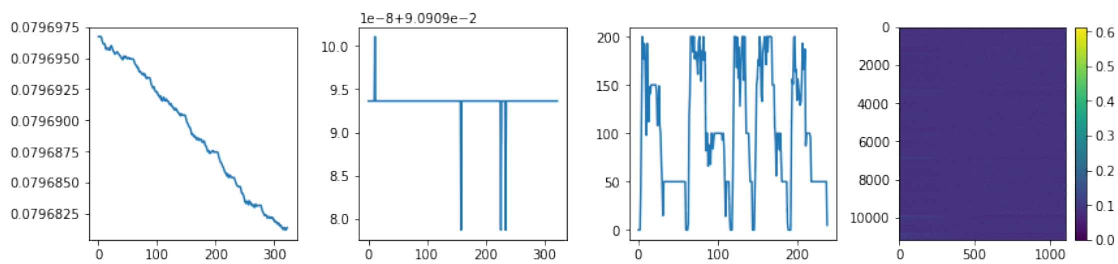
```python
  md['mi'].plot(mwi)
  md['a'].plot(activity)
  pos = md['w'].imshow(w.to('cpu').tolist(), aspect='auto')
  plt.colorbar(pos, ax=md['w'])

  plt.show()
```

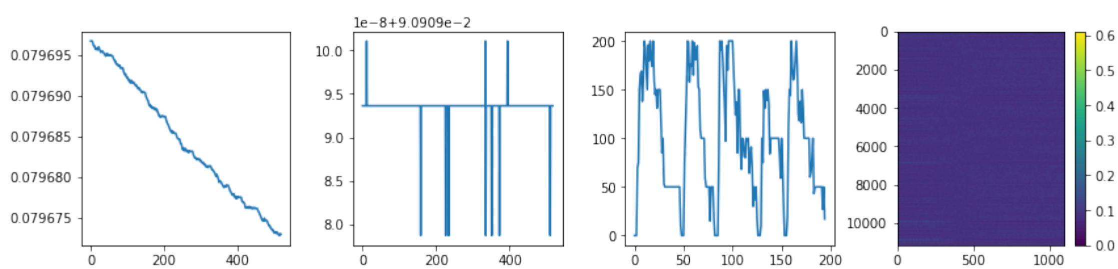
[ ]:
```python
from tqdm.notebook import tqdm
pt = 2

net.plasticity = True
net['population']['encoder_dendrite'].plasticity = True

cal_cl = lambda w: (w*(1-w)/w.numel()).sum()
# cli = [cal_cl(net['population']['population_LinearDendrite_2'].w)]
# mwi = [net['population']['population_LinearDendrite_2'].w.mean()]
cli = [cal_cl(net['population']['encoder_dendrite'].w)]
mwi = [net['population']['encoder_dendrite'].w.mean()]
activity= [0]

for e in range(1):
  for i in tqdm(range(0,len(data)//16,batch_size)):
    net.reset()
    if len(data)<i+batch_size:
      break
    subset = data[i:i+batch_size]
    max_len = max(len(d) for d in subset)
    for j in range(max_len+pt):
      words = [d[j] if j < len(d) else '<PAD>' for d in subset]
      net.progress(
          external_inputs={
              'encoder': {
                  'direct_input': words
              }
          }
      )
      cli.append(cal_cl(net['population']['encoder_dendrite'].w))
      mwi.append(net['population']['encoder_dendrite'].w.mean())
      activity.append(net['population'].spike.sum().to('cpu'))
    # for k_clamps in [1,0]:
    #   net.progress(
    #       external_inputs={
    #           'population': {
    #               'k_clamps': k_clamps
    #           },
    #           'encoder': {
    #               'direct_input': ['<PAD>' for d in subset]
```

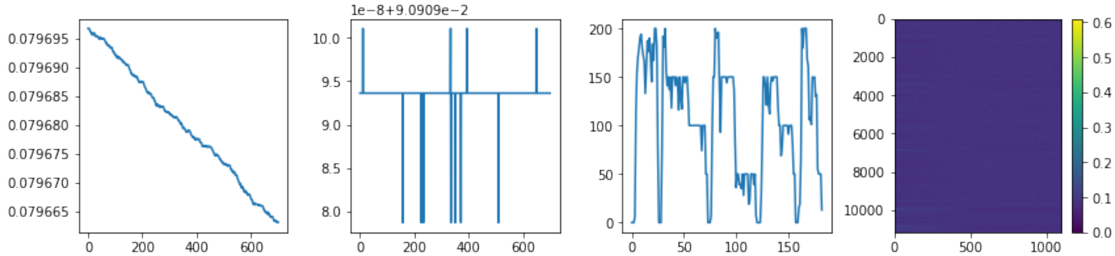
```
    #            }
    #        }
    #    )
    #    cli.append(cal_cl(net['population']['encoder_dendrite'].w))
    #    mwi.append(net['population']['encoder_dendrite'].w.mean())
    #    activity.append(net['population'].spike.sum().to('cpu'))

    if i%5==4:
      plot(cli,mwi,activity,net['population']['encoder_dendrite'].w)
      activity = []
plot(cli,mwi,activity,net['population']['encoder_dendrite'].w)
```

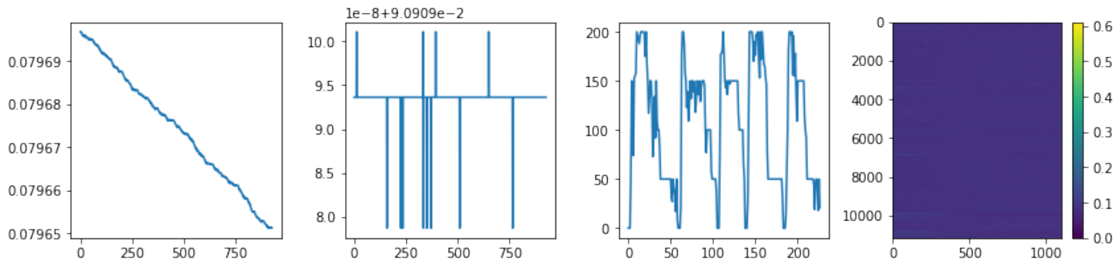
```
  0%|          | 0/313 [00:00<?, ?it/s]
```

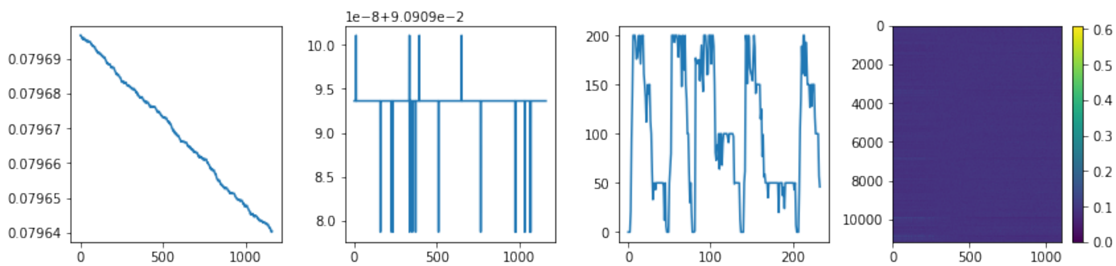


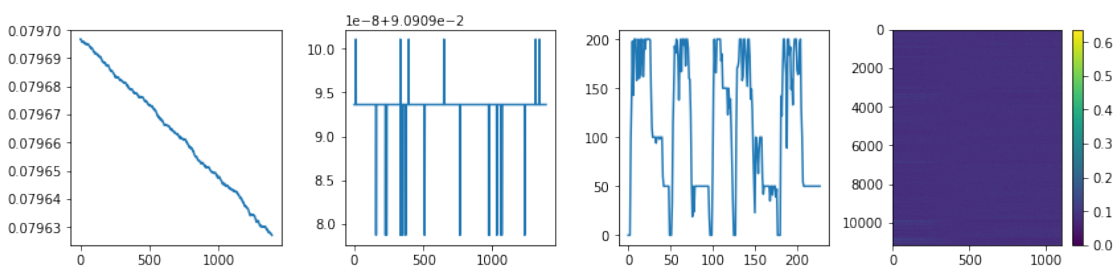


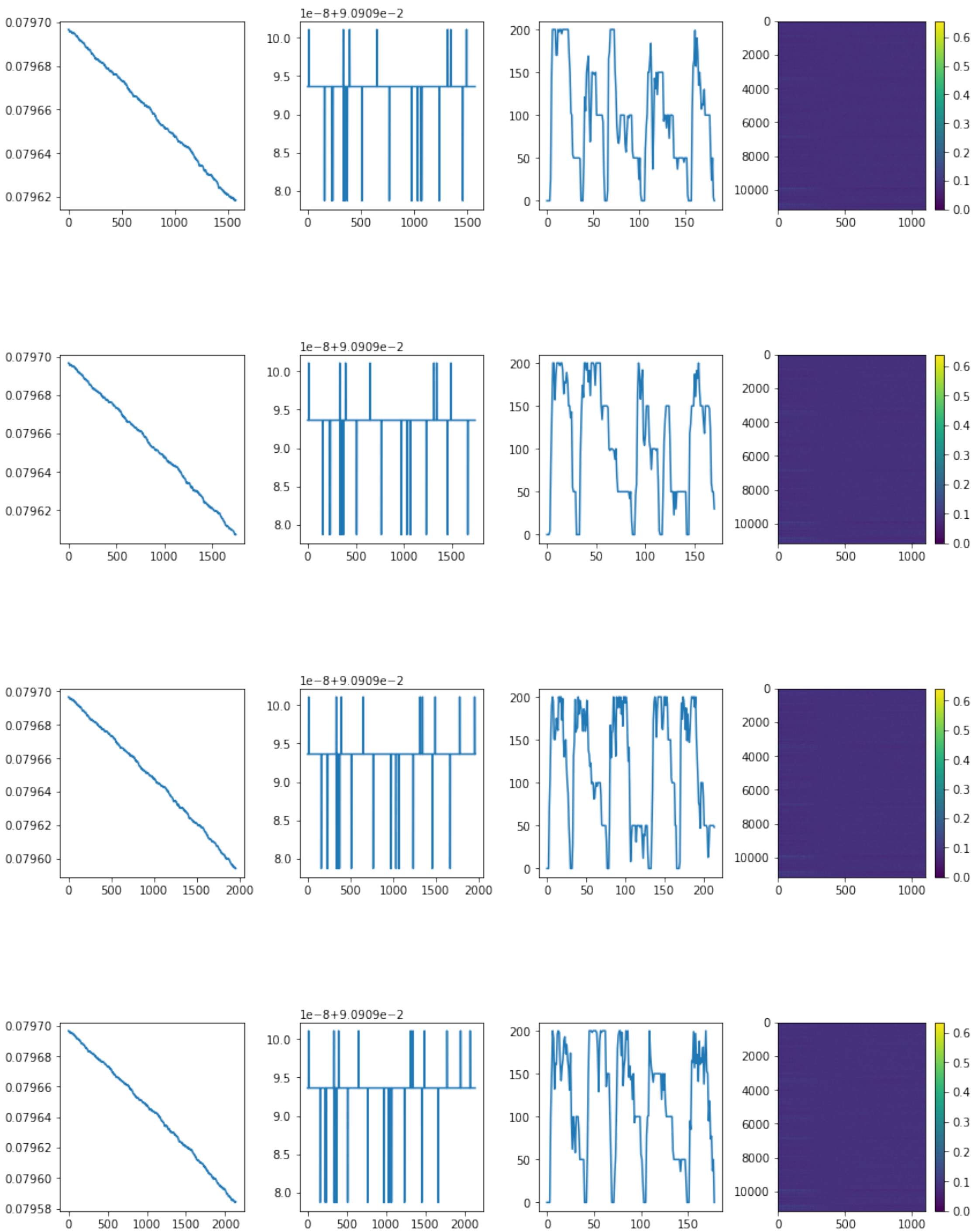


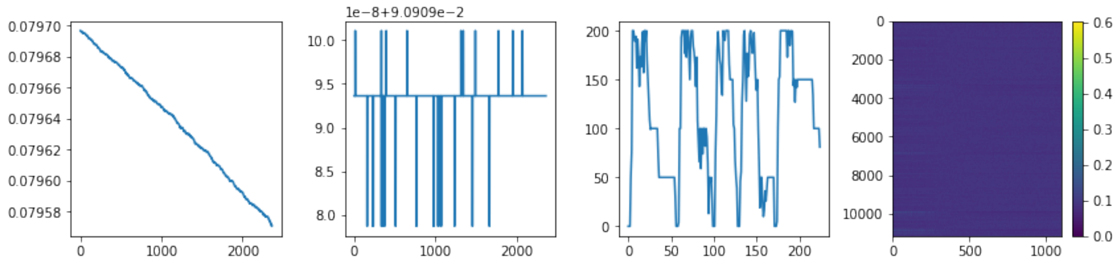

1e−8+9.0909e−2

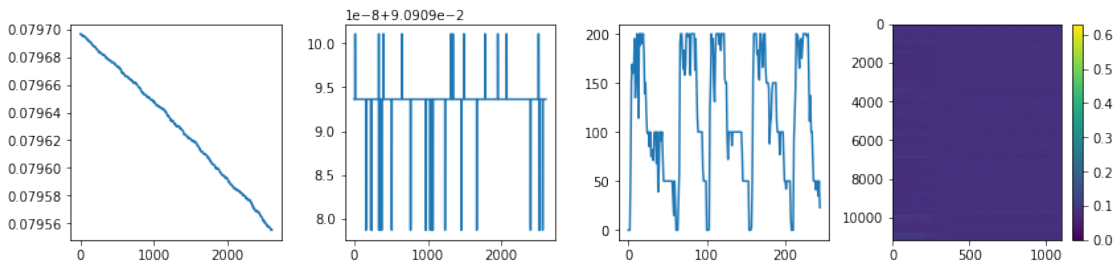

1e−8+9.0909e−2

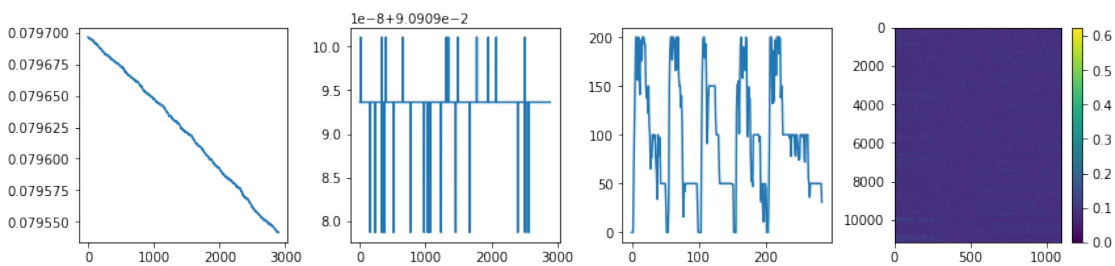

1e−8+9.0909e−2

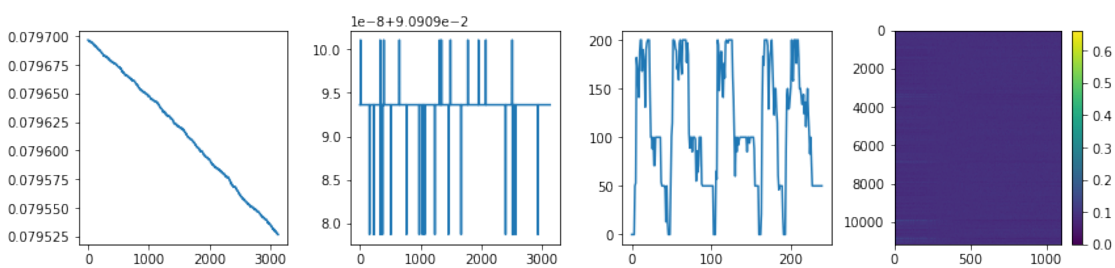

1e−8+9.0909e−2

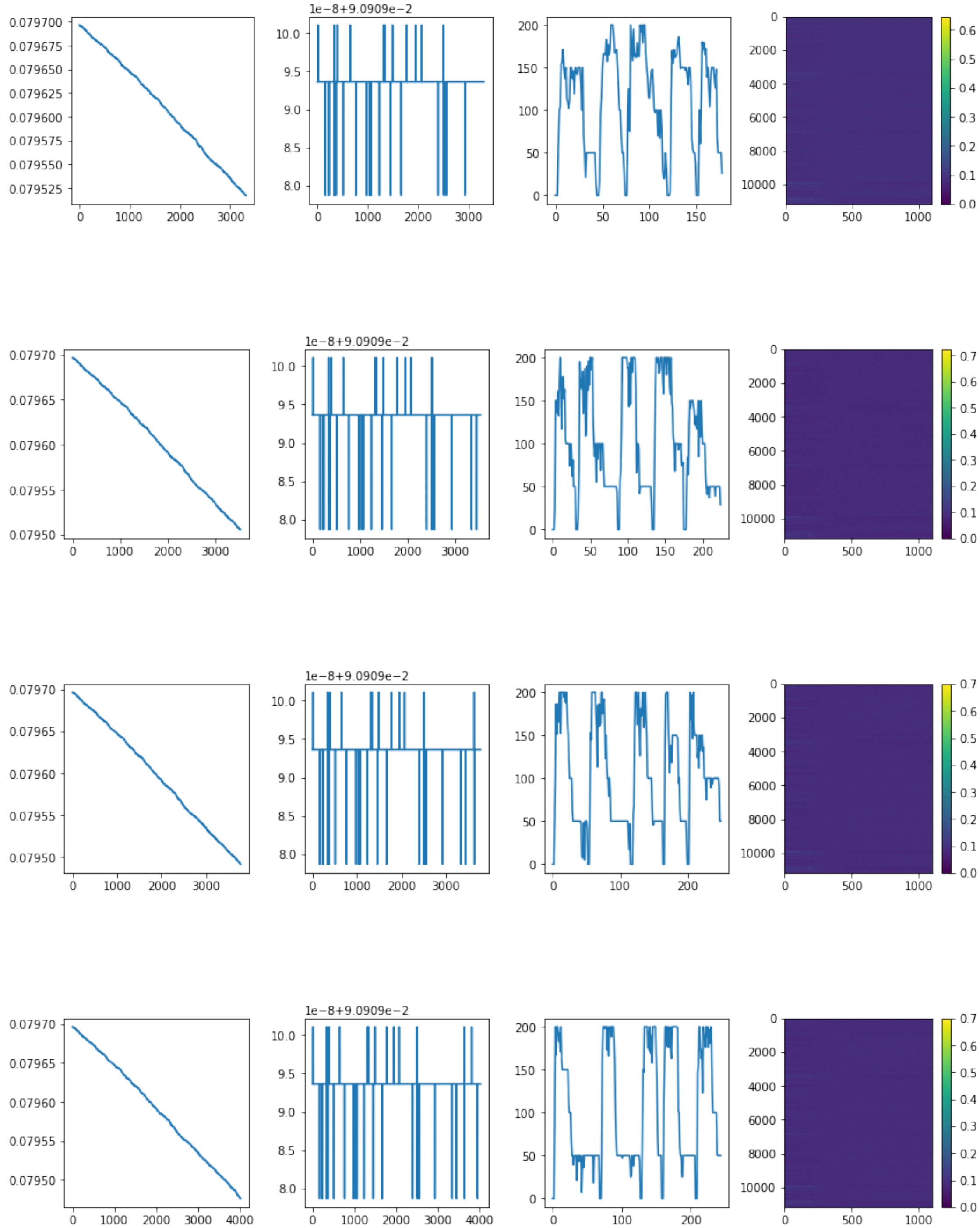

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

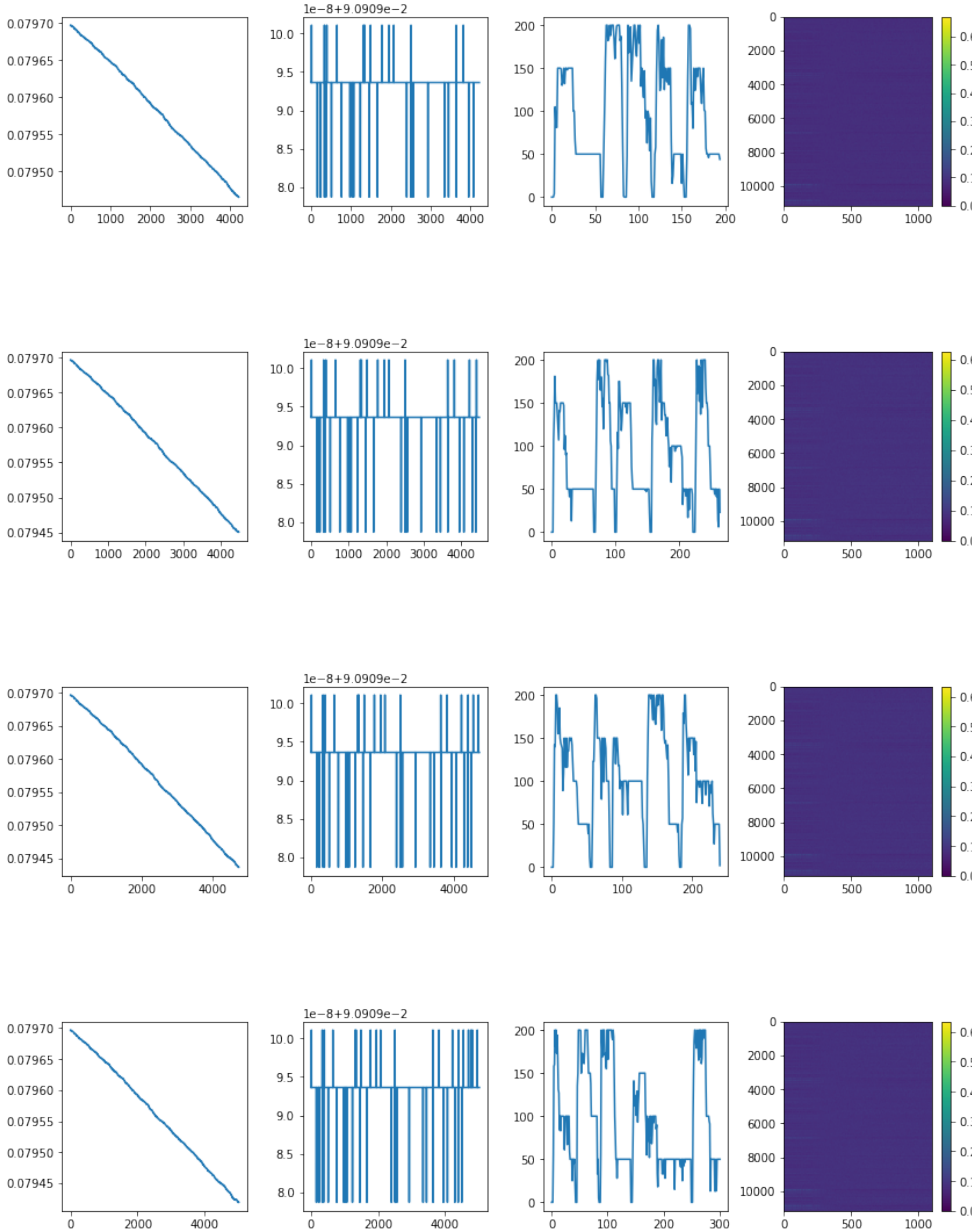

1e−8+9.0909e−2

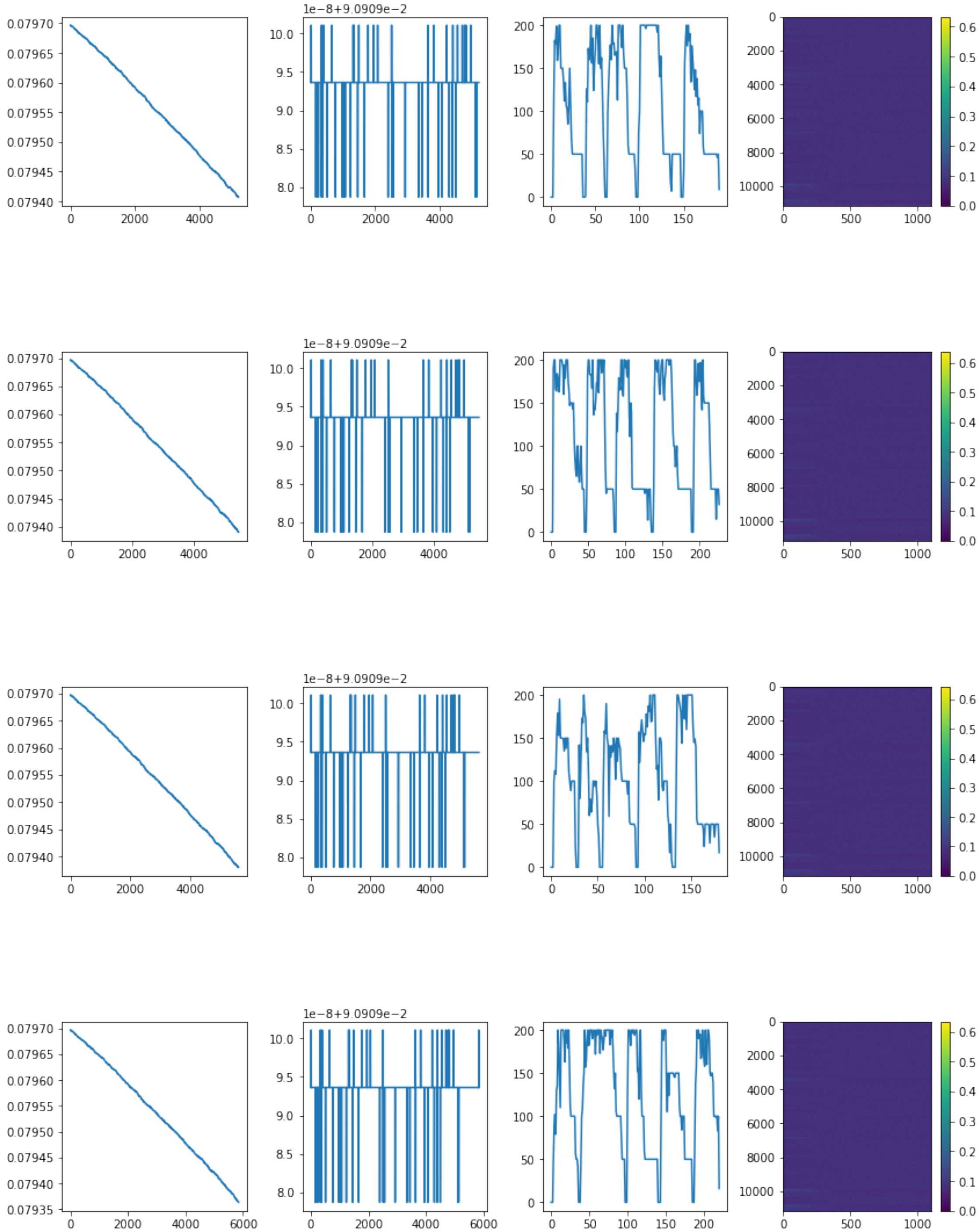

1e−8+9.0909e−2

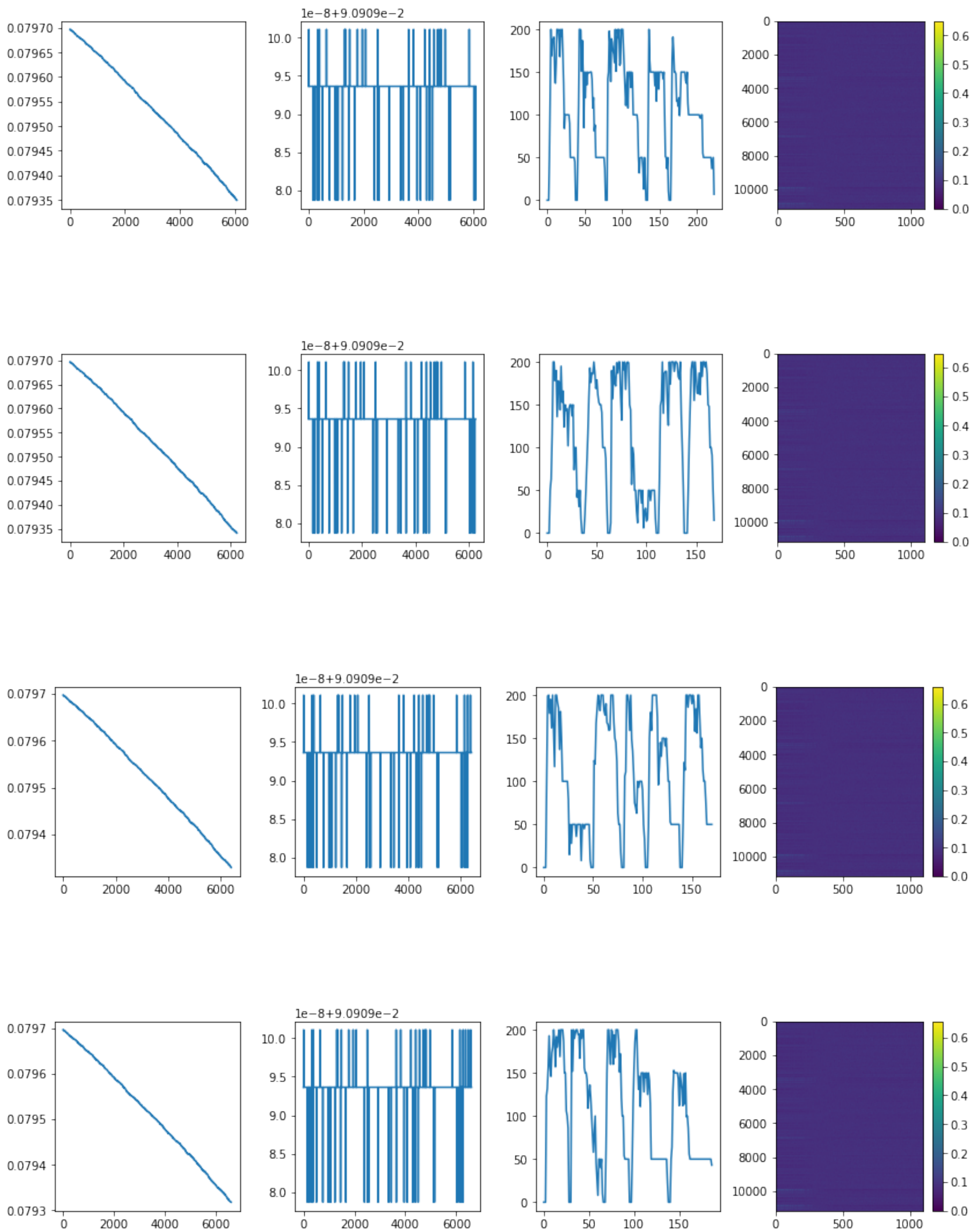

1e−8+9.0909e−2

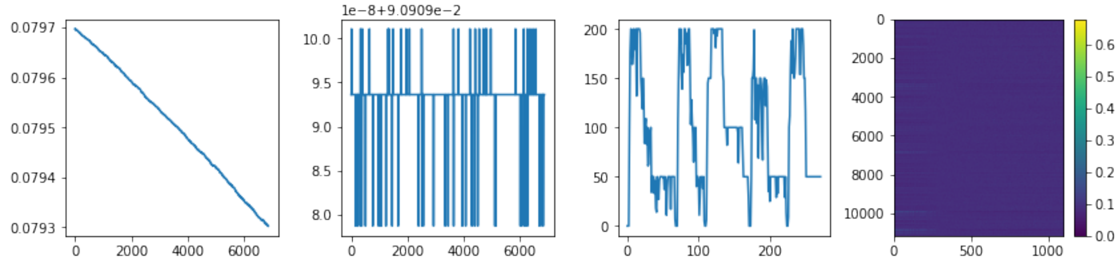

1e−8+9.0909e−2

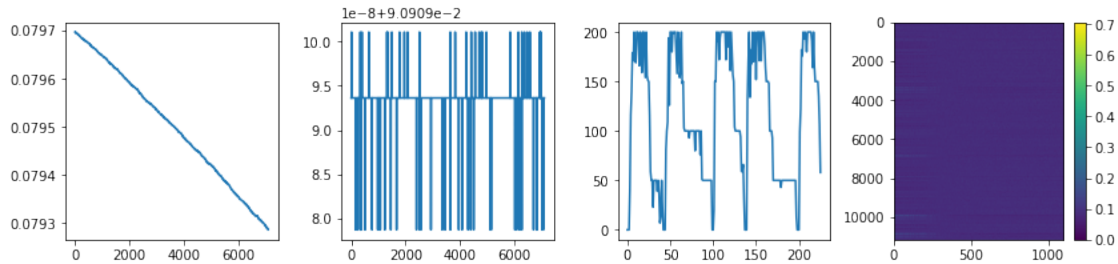

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

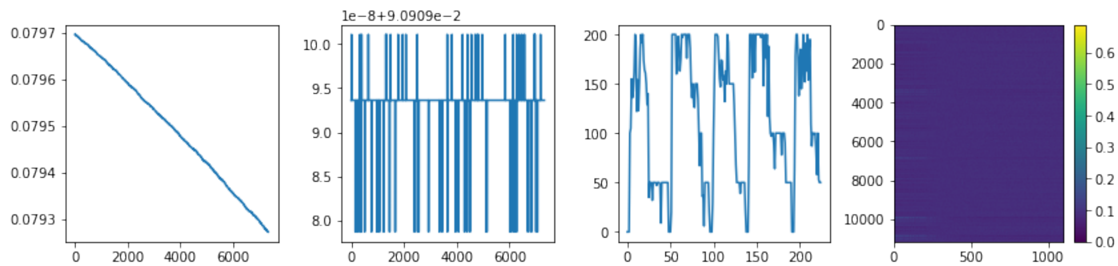

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

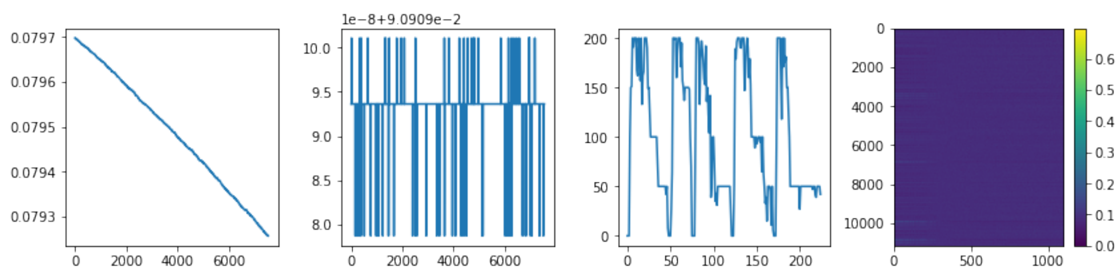

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

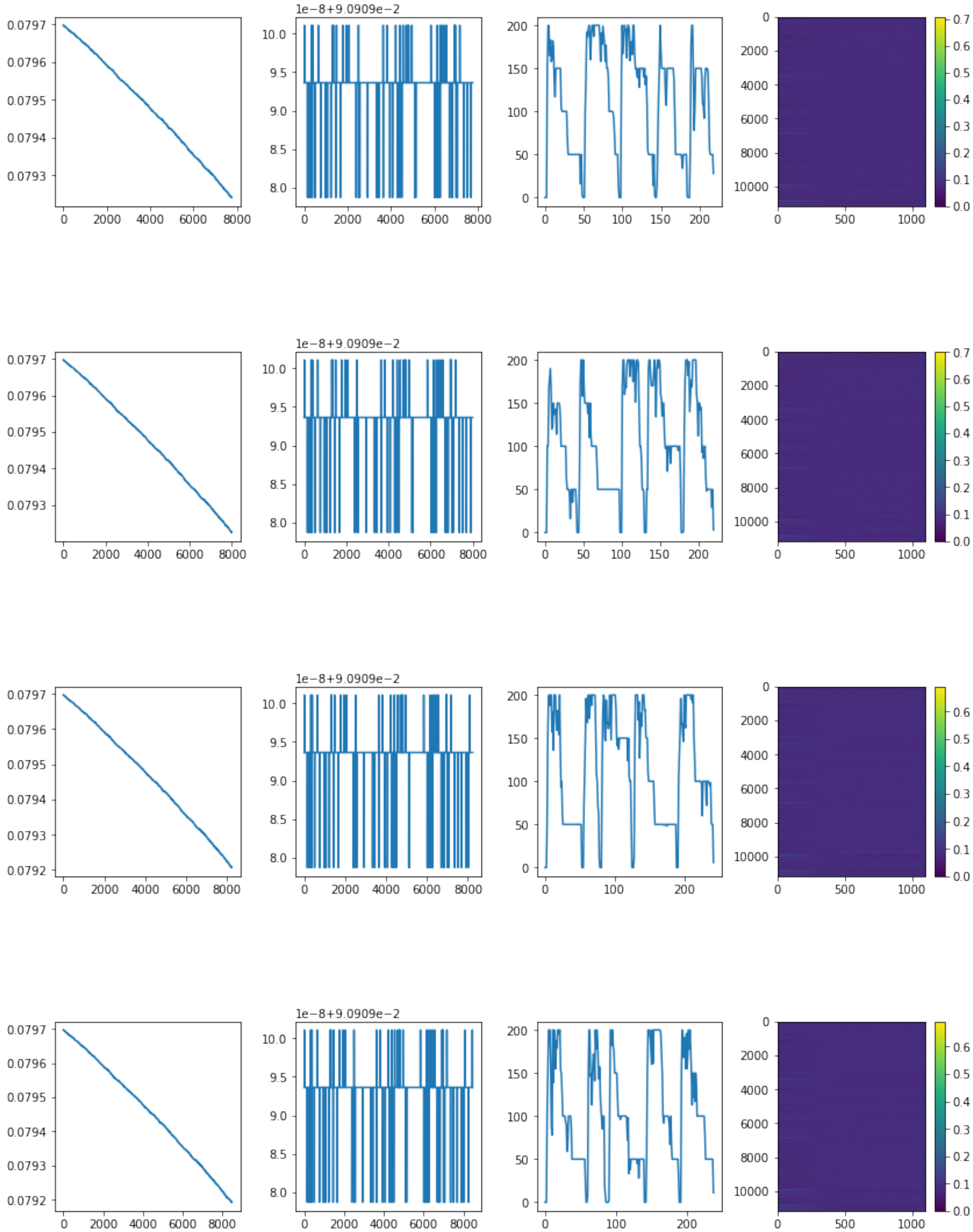

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

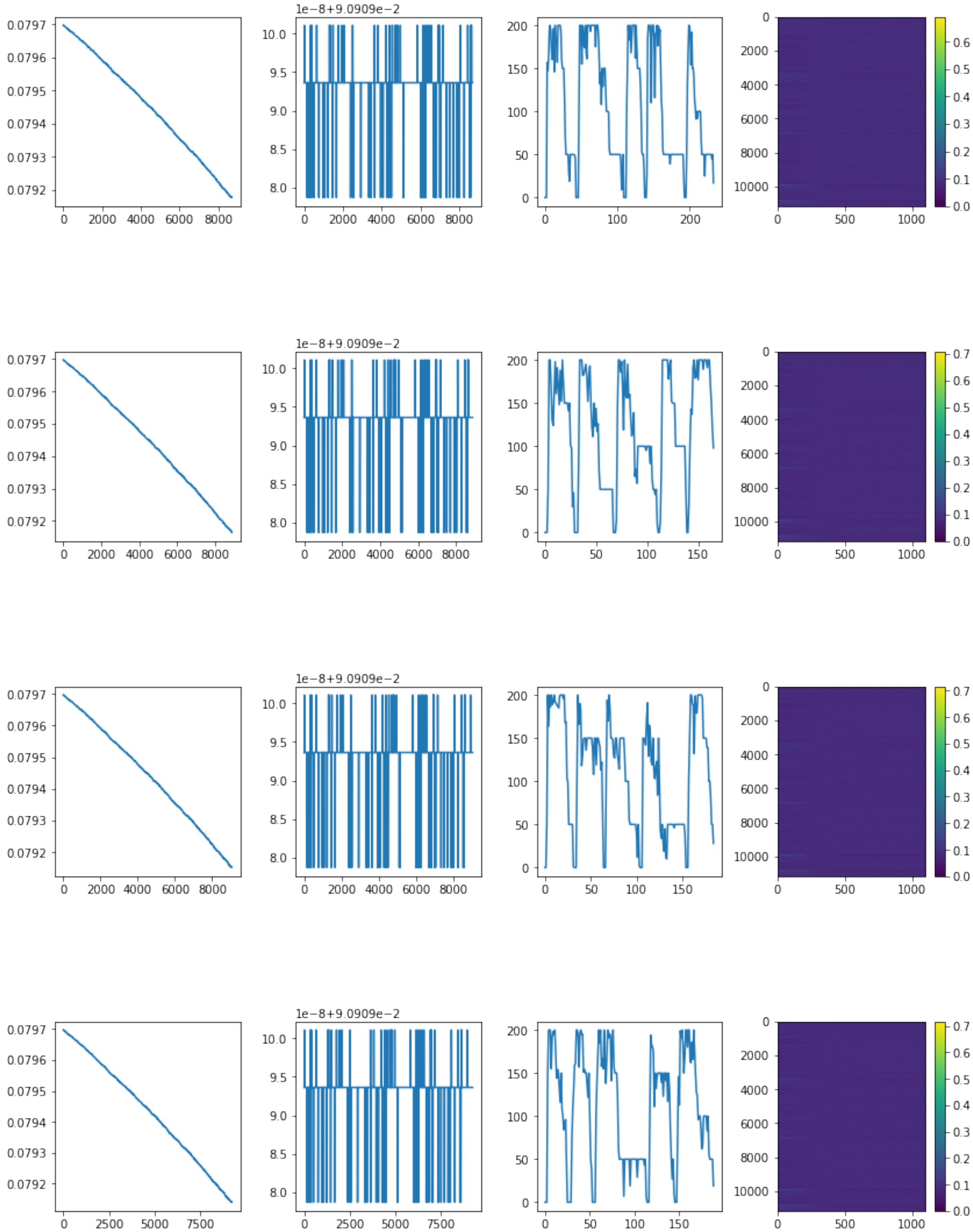

1e−8+9.0909e−2

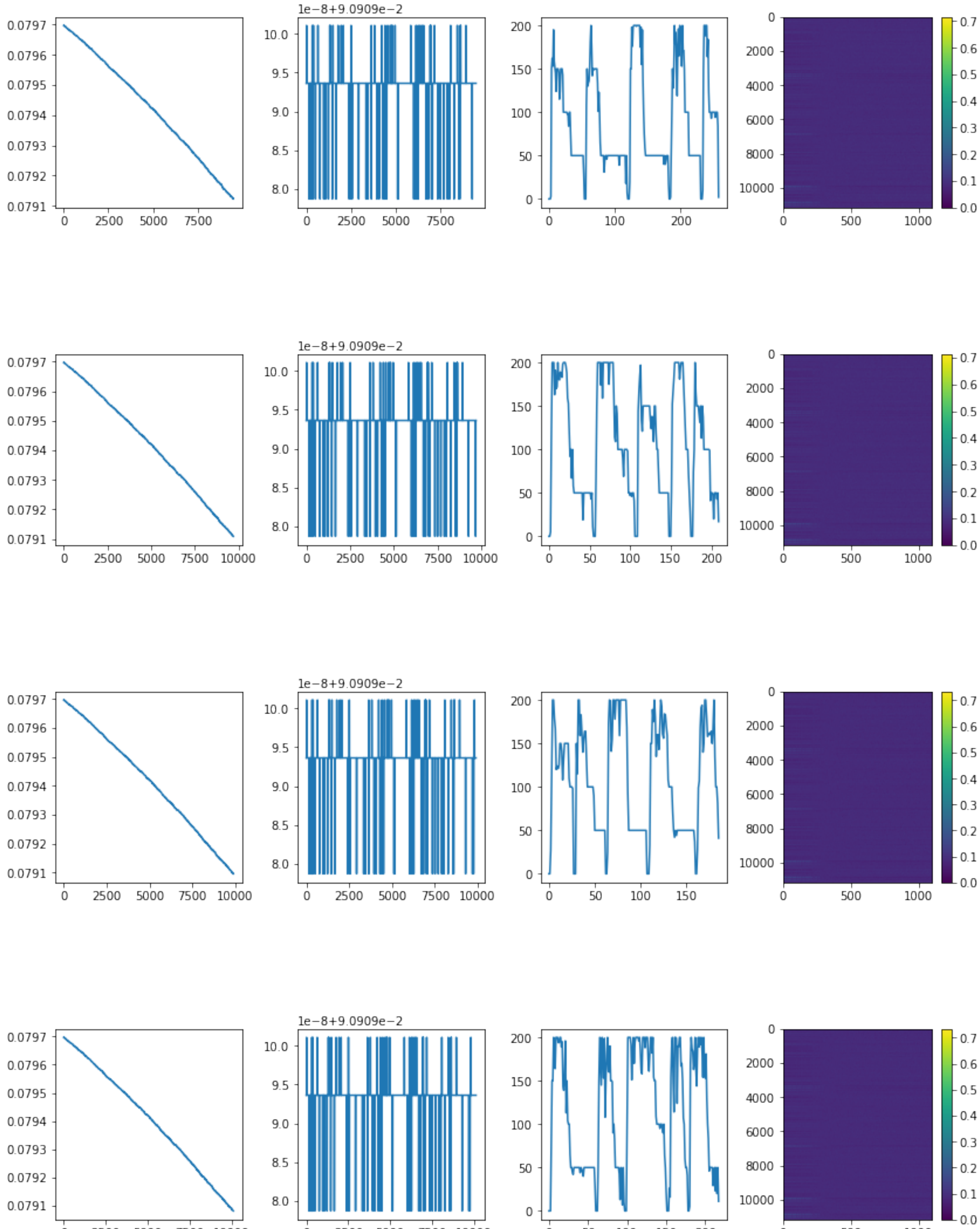

1e−8+9.0909e−2

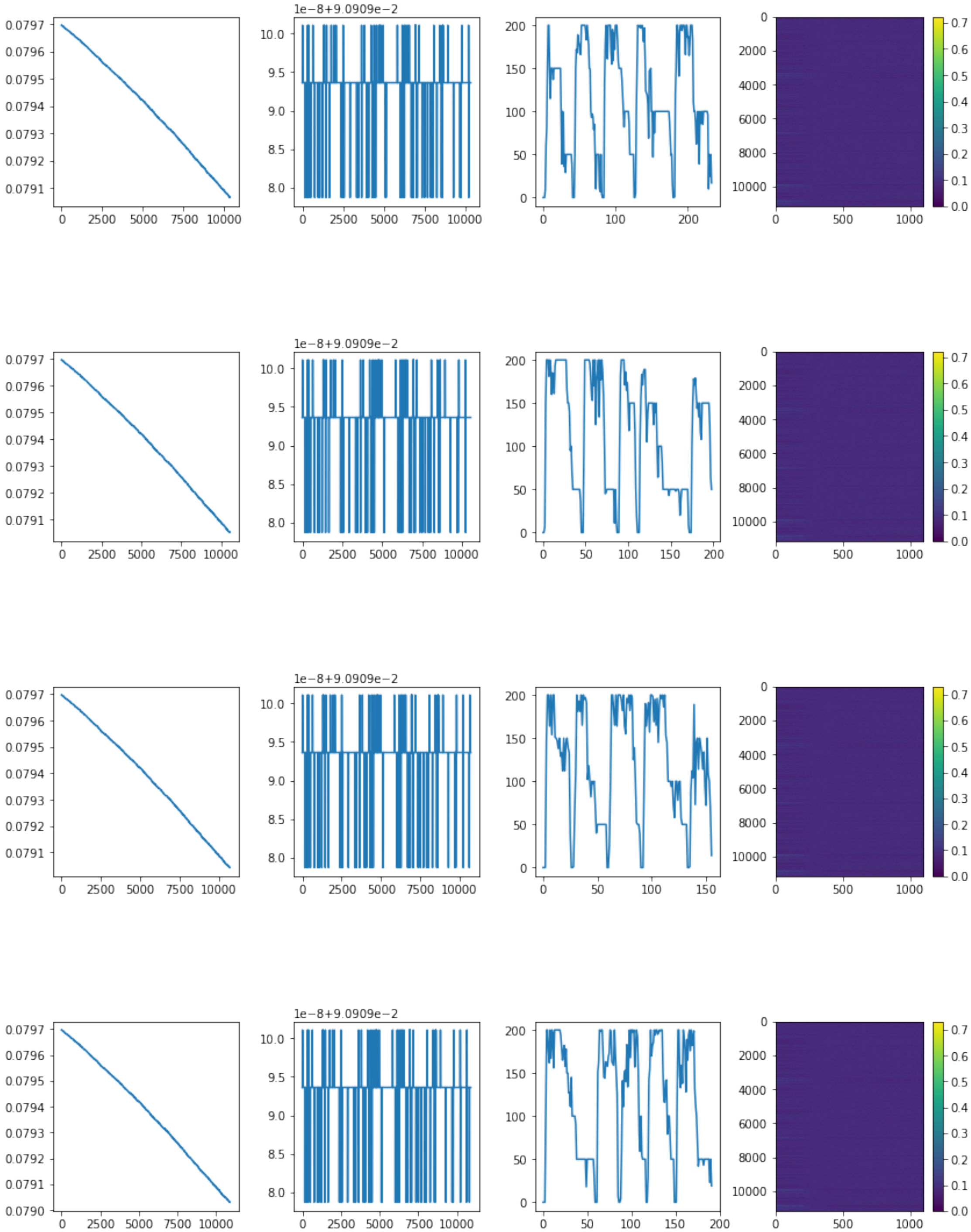

1e−8+9.0909e−2

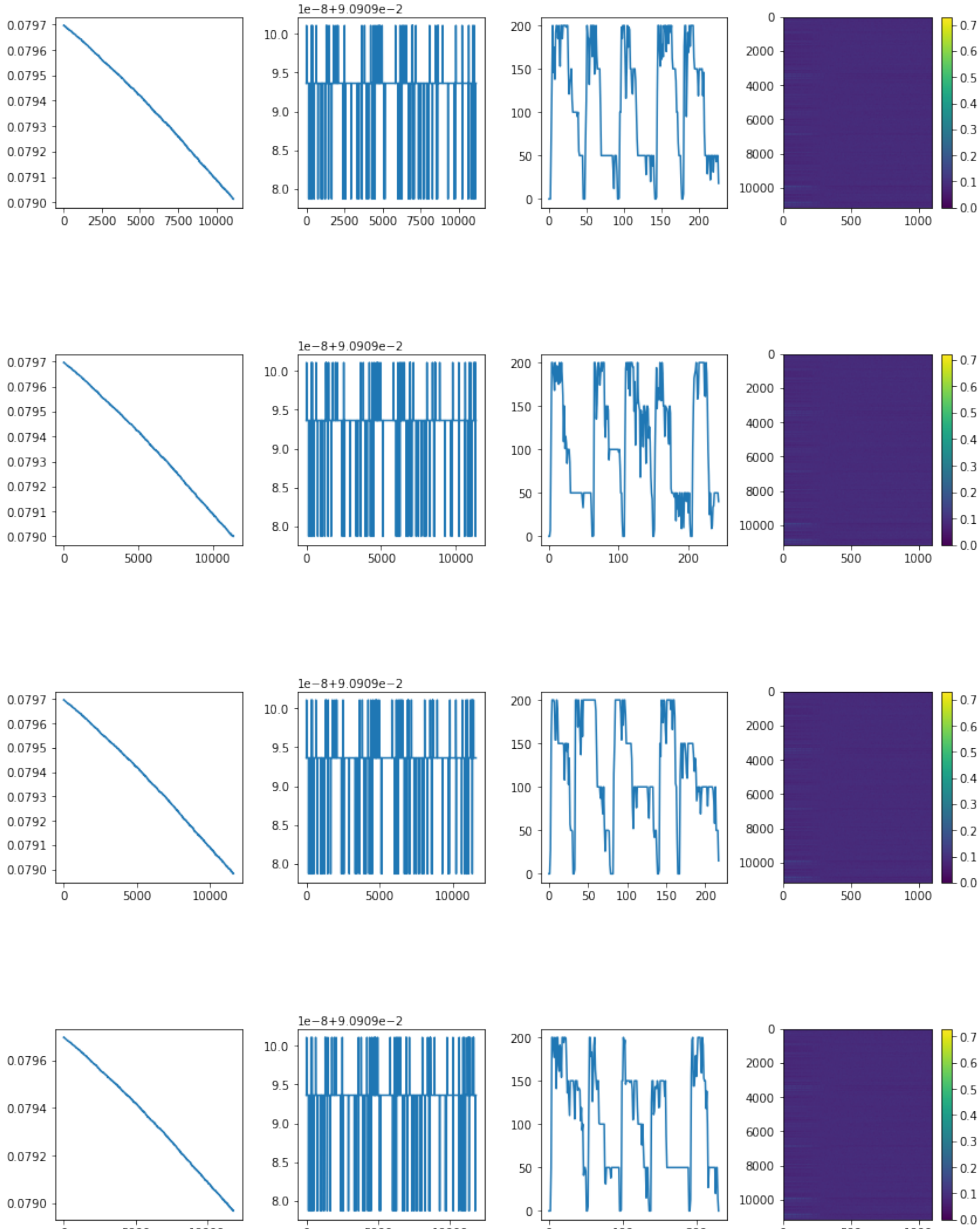

1e−8+9.0909e−2

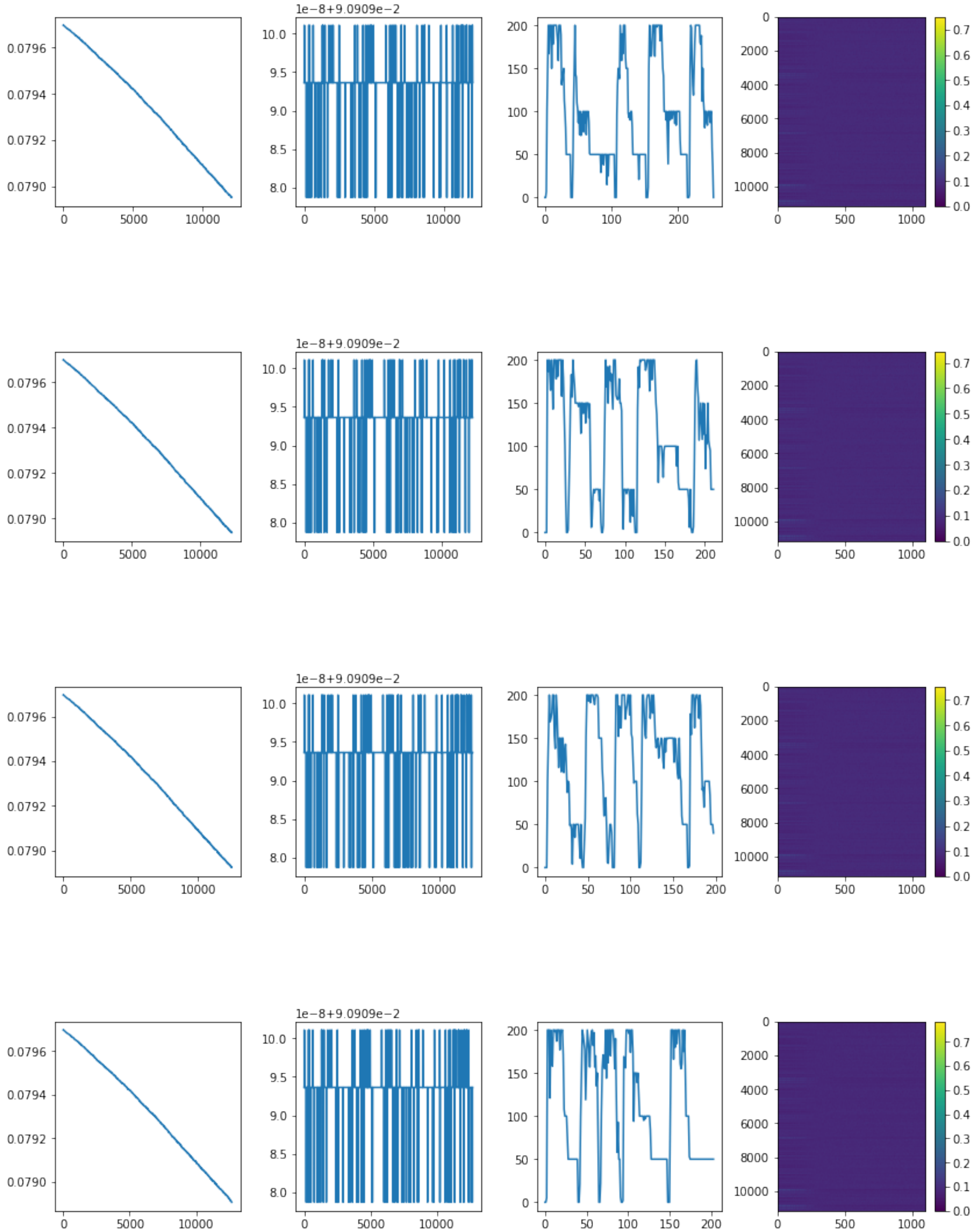

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

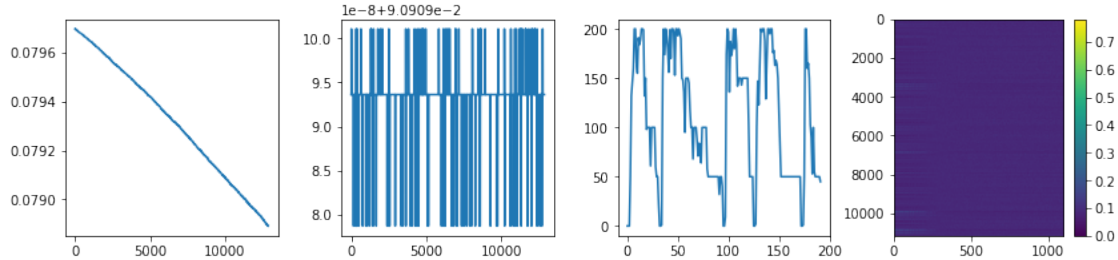

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

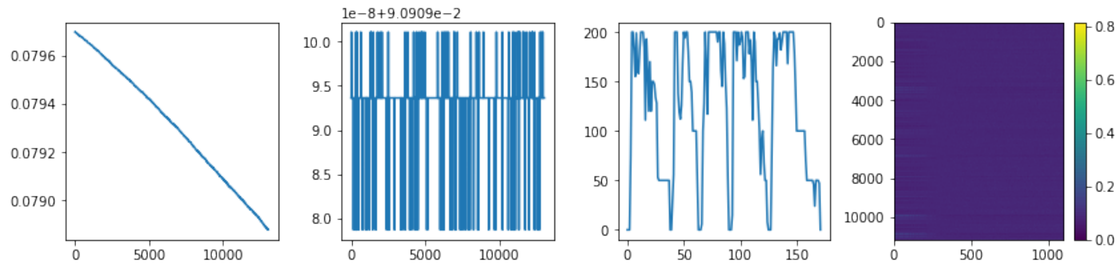

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

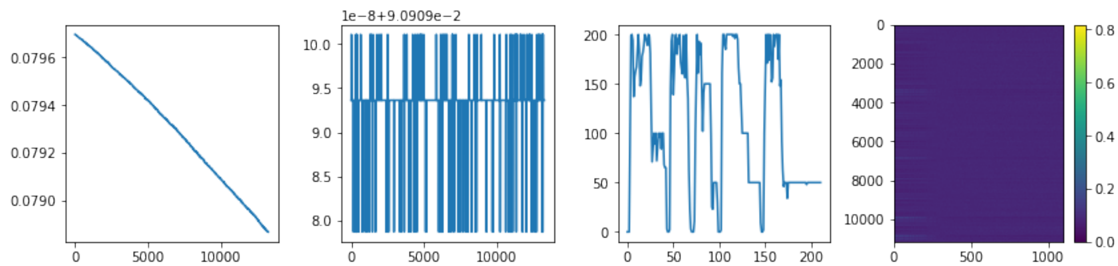

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

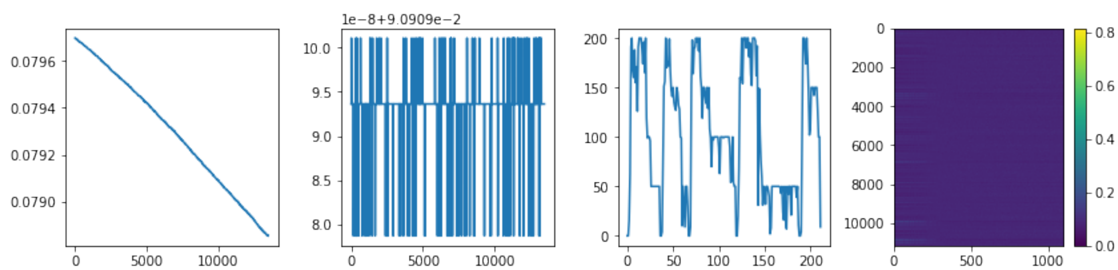

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2

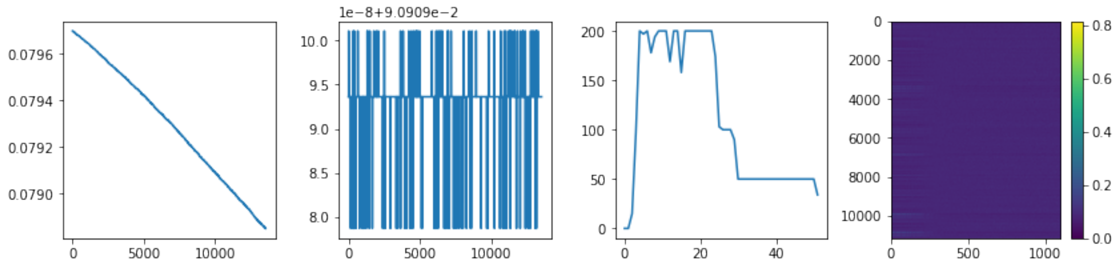

1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2
1e−8+9.0909e−2


0.0796
0.0794
0.0792
0.0790
0 5000 10000
1e−8+9.0909e−2
10.0 9.5 9.0 8.5 8.0
200 150 100 50 0
0 50 100 150 200
0 2000 4000 6000 8000 10000
0 500 1000
0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0


0.0796
0.0794
0.0792
0.0790
0 5000 10000
1e−8+9.0909e−2
10.0 9.5 9.0 8.5 8.0
200 150 100 50 0
0 50 100 150
0 2000 4000 6000 8000 10000
0 500 1000
0.8 0.6 0.4 0.2 0.0


0.0796
0.0794
0.0792
0.0790
0 5000 10000
1e−8+9.0909e−2
10.0 9.5 9.0 8.5 8.0
200 150 100 50 0
0 50 100 150 200
0 2000 4000 6000 8000 10000
0 500 1000
0.8 0.6 0.4 0.2 0.0


0.0796
0.0794
0.0792
0.0790
0 5000 10000
1e−8+9.0909e−2
10.0 9.5 9.0 8.5 8.0
200 150 100 50 0
0 50 100 150 200
0 2000 4000 6000 8000 10000
0 500 1000
0.8 0.6 0.4 0.2 0.0



[ ]:
```
import json

weights = net['population']['encoder_dendrite'].w.to('cpu').tolist()
json.dump(weights, open('weights.json','w'))
```

[ ]:
```
import json

weights = json.load(open('weights (1).json'))
net['population']['population_LinearDendrite_2']._w = torch.as_tensor(weights).
↪to(device)
```

[ ]:
```
net.plasticity = False
net['population']['encoder_dendrite'].plasticity = False

pt = 2

train_vectors = []
for i in tqdm(range(0,len(data)//10,batch_size)):
  net.reset()
  if len(data)<i+batch_size:
    break
  subset = data[i:i+batch_size]
  vectors = torch.zeros(net['population'].spike.shape)
  max_len = max(len(d) for d in subset)
  for j in range(max_len+pt):
    words = [d[j] if j < len(d) else '<PAD>' for d in subset]
    net.progress(
        external_inputs={
              # 'population': {
              #     'k_clamps': 1
              # },
            'encoder': {
                'direct_input': words
            }
        }
    )
```

```
    vector = net['population'].spike.detach().clone().to('cpu')
    for i,d in enumerate(subset):
      if j>len(d)+pt:
        vector[i] = 0
    vectors += vector
  for i,d in enumerate(subset):
    vector = vectors[i] / (len(d)+pt)
    train_vectors.append(vector.tolist())
```

```
  0%|          | 0/500 [00:00<?, ?it/s]
```

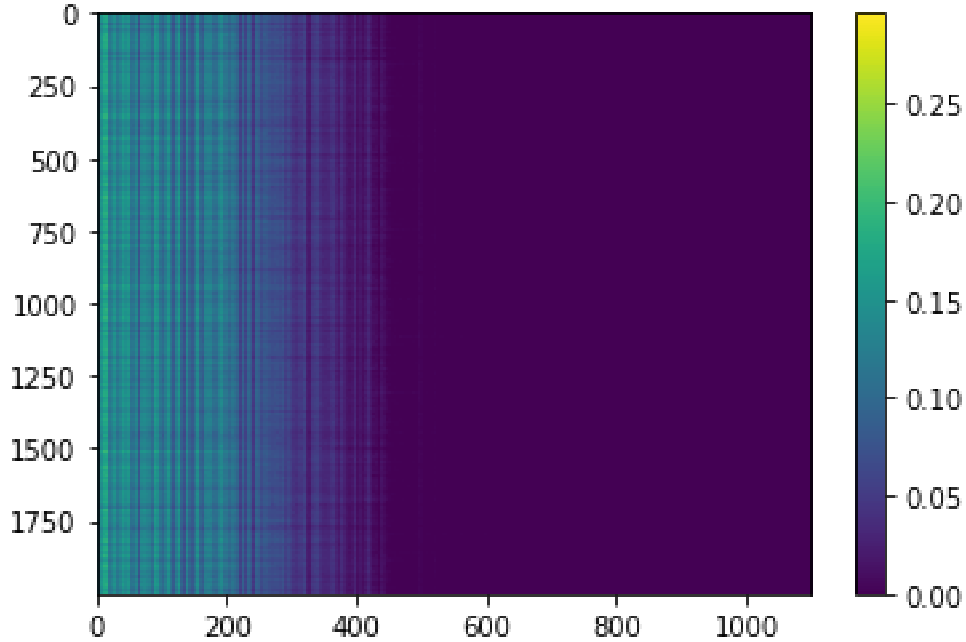
[ ]:
```
json.dump(train_vectors, open('train_vectors.json','w'))
```

[ ]:
```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0).fit(train_vectors, targets[:
 ↪len(train_vectors)])
clf.score(train_vectors, targets[:len(train_vectors)])
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:940:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

[ ]: 0.5775

[ ]:
```
import matplotlib.pyplot as plt
plt.imshow(train_vectors, aspect='auto')
plt.colorbar()
plt.show()
```



[ ]:
```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0).fit(train_vectors, targets[:
 ↪len(train_vectors)])
clf.score(train_vectors, targets[:len(train_vectors)])
```

[ ]:
```
import matplotlib.pyplot as plt
plt.imshow(train_vectors, aspect='auto')
plt.colorbar()
plt.show()
```

[ ]:
```
test_data = json.load(open(data_path+'test_data.json'))

test_vectors = []
for i in tqdm(range(0,len(test_data)//10,batch_size)):
  net.reset()
  if len(test_data)<i+batch_size:
    break
  subset = test_data[i:i+batch_size]
  vectors = torch.zeros(net['population'].spike.shape)
  max_len = max(len(d) for d in subset)
  for j in range(max_len+10):
    words = [d[j] if j < len(d) else '<PAD>' for d in subset]
    net.progress(
        external_inputs={
            'encoder': {
```

```
                'direct_input': words
            }
        }
    )
    vector = net['population'].spike.detach().clone().to('cpu')
    for i,d in enumerate(subset):
      if j>len(d)+10:
        vector[i] = 0
    vectors += vector
  for i,d in enumerate(subset):
    vector = vectors[i] / (len(d)+10)
    test_vectors.append(vector.tolist())
```

[ ]:
```
test_targets = json.load(open(data_path+'test_targets.json'))
clf.score(test_vectors, test_targets[:len(test_vectors)])
```

[ ]:
```

```