Deep Neural Network Hyperparameter Optimization with Orthogonal Array Tuning

Xiang Zhang, Xiaocong Chen, Lina Yao, Chang Ge, Manqing Dong

University of New South Wales, Australia

Alexia

Content

This paper presents an efficient **Orthogonal Array Tuning Method (OATM)** for **deep learning hyperparameter tuning**.

Five detailed steps and elaborate on it using two widely used deep neural network structures (Recurrent Neural Networks and Convolutional Neural Networks).

Compared to the state-of-the-art hyper-parameter tuning methods including manually (e.g., grid search and random search) and automatically (e.g., Bayesian Optimization) ones. The experiment results state that OATM can significantly save the tuning time compared to the state-of-the-art methods while preserving the satisfying performance.

“ the deep learning classification accuracy dramatically fluctuates from 32.2% to 92.6% due to the different selection of hyper-parameters”

🚀 hyper-parameter tuning method

Content – tuning methods

🚀 **hyper-parameter tuning method**

Random search	**Bayesian optimisation**	**OATM**
time-consuming; cannot converge to the global optimum	sensitive to parameters of the surrogate model; highly depending on the quality of the learning model	

Content – orthogonal array

2 Basic composition principles of Orthogonal Array

1. In the **same column** (factor), different levels have the **same appearing times**.

2. In two **randomly selected columns** (factors), different level combinations are **complete** and **balanced**. The number of Orthogonal Array rows is determined by this principle.

Row No.	Factor No.		
	Factor 1	Factor 2	Factor 3
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	2	3
6	2	3	1
7	3	1	3
8	3	2	1
9	3	3	2

Table 1: Orthogonal Array with 9 rows, 3 factors and each factor has 3 levels

Content – advantage of OATM

OATM

- Less tuning time & competitive performance;
- All possible values of all hyper-parameters are considered equally;
- Has been commonly used

Content – OATM

Hyperparameters — factors

values of the hyperparameter — levels

Experiment	Layer thickness	Air gap	Raster angle
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	2	3
6	2	3	1
7	3	1	3
8	3	2	1
9	3	3	2

Steps:

1. Determine # of to-be-tuned factors &
   # of levels of each factor
2. Construct OAT table (Weibull++, SPSS, etc…)
3. Run the experiments
4. Range analysis
5. Run the experiment with optimal hyper-parameter setting

CONTENT

CNN + BO

CNN + RS

CNN + OATM

RNN + BO

RNN + RS

RNN + OATM

EEG(electroencephalography)
28k

IMU (inertial measurement unit)
1.2m

RFID (radio frequency IDentification)
3k

[8:2]

*CNN: Convolutional Neural Networks
*RNN: Recurrent Neural Network

Table 3: Factor-Level table of RNN and CNN.

		Factor 1 (lr)	Factor 2 (λ)	Factor 3 (n_l)	Factor 4 (n_n)
RNN	**Level 1**	0.005	0.004	4	32
	Level 2	0.01	0.008	5	64
	Level 3	0.015	0.012	6	96
		Factor 1 (lr')	**Factor 2 (f')**	**Factor 3 (n'_l)**	**Factor 4 (n'_n)**
CNN	**Level 1**	0.001	[1,2]	1	64
	Level 2	0.003	[1,4]	2	128
	Level 3	0.005	[1,6]	3	192

Learning rate l2 norm coef layers# nodes#

Table 4: Range analysis of RNN

Row No.	Factor 1 (lr)	Factor 2 (λ)	Factor 3 (n_l)	Factor 4 (n_n)	Acc
1	0.005	0.004	4	32	0.875
2	0.005	0.008	5	64	0.8
3	0.005	0.012	6	96	0.521
4	0.01	0.004	5	96	0.888
5	0.01	0.008	6	32	0.797
6	0.01	0.012	4	64	0.451
7	0.015	0.004	6	64	**0.897**
8	0.015	0.008	4	96	0.335
9	0.015	0.012	5	32	0.471
R_{level1}	2.196 /3	2.66	1.661	2.143	
R_{level2}	2.136	1.932	2.159	2.148	
R_{level3}	1.703	1.443	2.215	1.744	
A_{level1}	**0.732**	**0.887**	0.554	0.714	
A_{level2}	0.712	0.644	0.720	**0.716**	
A_{level3}	0.568	0.481	**0.738**	0.581	
Lowest Acc	0.568	0.481	0.554	0.581	
Highest Acc	0.732	0.887	0.738	0.716	
Range	0.164	0.406	0.184	0.135	
Importance	$lambda > n_l > lr > n_n$				
Best Level	**Level 1**	**Level 1**	**Level 3**	**Level 2**	
Optimal Value	**0.005**	**0.004**	**6**	**64**	**0.925**

sum acc of level i

ave acc of level i

Data	Models	Methods	Optimal Factors				Metrics						
			F1	F2	F3	F4	#-Runnings	Time (s)	Acc	Prec	Recall	F-1
EEG	RNN	Grid	0.005	0.004	6	64	81	6853.6	0.9251	0.9324	0.9139	0.9231
		Random	0.01	0.008	6	32	9	766.8	0.7941	0.8003	0.7941	0.7947
		BO	0.0135	0.0049	5	32	9	703.4	0.718	0.7246	0.6474	0.6838
		Ours	0.005	0.004	6	64	9	821.9	0.925	0.9335	0.9223	0.9279
	CNN	Grid	0.005	4	3	192	81	31891.5	0.828	0.8137	0.8256	0.8269
		Random	0.003	2	1	128	9	662.8	0.7268	0.7277	0.7269	0.7266
		BO	0.001	4	3	139	9	721.9	0.7244	0.7302	0.7244	0.7263
		Ours	0.003	4	1	128	9	680.4	0.797	0.7969	0.8112	0.8003
IMU	RNN	Grid	0.005	0.004	6	96	81	3027.2	0.9936	0.9909	0.9976	0.9971
		Random	0.015	0.004	4	32	9	1008.5	0.9139	0.9209	0.9412	0.9156
		BO	0.0132	0.0041	4	48	9	1078.8	0.9872	0.9877	0.9851	0.9863
		Ours	0.005	0.004	6	64	9	1138.2	0.9913	0.9924	0.9905	0.9919
	CNN	Grid	0.003	2	1	128	81	41804.9	0.9732	0.9708	0.9708	0.9707
		Random	0.003	2	2	128	9	7089.2	0.9692	0.9691	0.9692	0.9691
		BO	0.0012	2	2	192	9	6559.7	0.9696	0.9702	0.9701	0.9701
		Ours	0.003	2	2	128	9	6809.8	0.9702	0.9699	0.9703	0.9702
RFID	RNN	Grid	0.005	0.008	6	96	81	2846.1	0.9342	0.9388	0.9201	0.9252
		Random	0.005	0.012	4	32	9	642.3	0.8891	0.9138	0.8826	0.8895
		BO	0.0142	0.0093	6	79	9	452.2	0.9071	0.8556	0.8486	0.8436
		Ours	0.005	0.008	6	64	9	497.1	0.9134	0.9138	0.9029	0.9162
	CNN	Grid	0.005	4	2	192	81	7890.8	0.9316	0.9513	0.9316	0.9375
		Random	0.005	2	1	128	9	1210.3	0.8683	0.9113	0.8684	0.8779
		BO	0.005	5	3	64	9	872.9	0.9168	0.9058	0.9194	0.9086
		Ours	0.005	4	3	192	9	980.3	0.9235	0.9316	0.9188	0.9326

Code Overview

Input data

```
# Generate dummy data (replace with your actual data)
X_train = np.random.rand(1000, 10, 84)  # (samples, timesteps, features)
y_train = np.random.randint(0, 2, size=(1000,))  # Binary labels
```

Hyperparameters

```
for lr in [0.005, 0.1, 0.15]:
    for lam in [0.004, 0.008, 0.012]:
        for n_layers in [4, 5, 6]:
            for nodes in [32, 64, 96]:
```

Code Overview –RNN

```
def rnn_run(lr, lam, n_layers, nodes):
    model = Sequential()
    # Input layer
    model.add(SimpleRNN(nodes,
                        input_shape=(X_train.shape[1], X_train.shape[2]),
                        return_sequences=(n_layers > 1),
                        kernel_regularizer=l2(lam)))

    # Hidden layers
    for i in range(1, n_layers):
        model.add(SimpleRNN(nodes,
                            return_sequences=(i != n_layers - 1),
                            kernel_regularizer=l2(lam)))

    # Output layer
    model.add(Dense(1, activation='sigmoid'))

    # Compile
    model.compile(optimizer=Adam(learning_rate=lr),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    # Train
    history = model.fit(X_train, y_train, epochs=5, verbose=0)
    # Return last epoch's accuracy
    return history.history['accuracy'][-1]
```

Code Overview–RNN

Grid Random

```
for lr in [0.005, 0.1, 0.15]:
    for lam in [0.004, 0.008, 0.012]:
        for n_layers in [4, 5, 6]:
            for nodes in [32, 64, 96]:
                print(f"Testing lr={lr}, λ={
                acc = rnn_run(lr, lam, n_lay
                print(f"→ Accuracy: {acc:.4f
```

Bayesian Optimisation

```
# -------------------- BAYESIAN OPTIMIZATION -------
print("\n🧪 Starting Bayesian Optimization...")
from skopt import gp_minimize
from skopt.space import Real, Integer
from skopt.utils import use_named_args

space = [
    Real(0.005, 0.15, name='lr'),
    Real(0.004, 0.012, name='lam'),
    Integer(4, 6, name='n_layers'),
    Integer(32, 96, name='nodes')
]

@use_named_args(space)
def objective(lr, lam, n_layers, nodes):
    acc = rnn_run(lr, lam, n_layers, nodes)
    print(f"Testing lr={lr:.4f}, λ={lam:.4f}, layers
```

Code Overview-RNN

```
# ------------------ ORTHOGONAL ARRAY ------------------
# Custom Orthogonal Array L9 (3-level, 4-parameter)
oa_combinations = [
    #   lr      lam      n_layers  nodes
    [0.005, 0.004,    4,        32],
    [0.005, 0.008,    5,        64],
    [0.005, 0.012,    6,        96],
    [0.1,   0.004,    5,        96],
    [0.1,   0.008,    6,        32],
    [0.1,   0.012,    4,        64],
    [0.15,  0.004,    6,        64],
    [0.15,  0.008,    4,        96],
    [0.15,  0.012,    5,        32],
]

# ------------------ OATM RUN ------------------
print("\n🎯 Starting RNN Orthogonal Array Tuning...")
start_oa = time.time()
best_acc_oa = 0
best_params_oa = {}

for i, (lr, lam, n_layers, nodes) in enumerate(oa_combinations):
    print(f"Testing OA Combination {i+1}: lr={lr}, lam={lam}, lay
```

Results & Analysis

RNN_BO

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` a
  super().__init__(**kwargs)
Testing lr=0.0937, λ=0.0063, layers=5, nodes=95 → Accuracy: 0.5030
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` a
  super().__init__(**kwargs)
Testing lr=0.1488, λ=0.0120, layers=4, nodes=54 → Accuracy: 0.4980

✅ Bayesian Optimization Results:
Best Accuracy: 0.5540000200271606
Best Hyperparameters: {'lr': 0.09368970827080074, 'lam': 0.004056530441757739, 'n_layers': np.int64(4), 'nodes': np.int64(66)}
Bayesian Optimization Time: 238.9990677833557
```

RNN_RS

```
→ Accuracy: 0.4870
Testing lr=0.15, λ=0.012, layers=6, nodes=96
→ Accuracy: 0.5010

Best accuracy: 0.5389999747276306
Best hyperparameters: {'lr': 0.005, 'lam': 0.008, 'n_layers': 4, 'nodes': 96}
Total tuning time: 1012.5739166736603
```

Results & Analysis

RNN + OATM (with random generated data)

```
Testing OA Combination 9: lr=0.15, lam=0.012, layers=5, nodes=32
→ Accuracy: 0.4975

✅ OATM Results:
Best Accuracy: 0.5199999809265137
Best Hyperparameters (OATM): {'lr': 0.005, 'lam': 0.004, 'n_layers': 4, 'nodes': 32}
OATM Tuning Time: 82.40417981147766 seconds
```

Results & Analysis

```
 83   POW.AF4.BetaL        2000 non-null   float64
 84   POW.AF4.BetaH        2000 non-null   float64
 85   POW.AF4.Gamma        2000 non-null   float64
 86   subject_understood   2000 non-null   float64
dtypes: float64(85), int64(2)
memory usage: 1.3 MB
```

subject_understood	count
1.0	1375
0.0	625

```
print(df.head())
```

```
   video_id  subject_id      EEG.AF3      EEG.F7      EEG.F3      EEG.FC5  \
0         0           0  4210.641113  4179.102539  4287.948730  4235.384766
1         0           0  4201.025879  4188.717773  4280.128418  4236.922852
2         0           0  4203.205078  4182.820313  4282.820313  4231.025879
3         0           0  4186.538574  4168.717773  4266.794922  4229.230957
4         0           0  4232.436035  4216.922852  4306.922852  4270.769043

       EEG.T7      EEG.P7      EEG.O1      EEG.O2  ...  POW.F8.Alpha  \
0  4207.948730  4165.000000  4135.897461  4170.000000  ...      1.583895
1  4209.615234  4152.436035  4130.128418  4149.487305  ...      1.709560
2  4207.820313  4172.436035  4131.538574  4147.948730  ...      1.873591
3  4202.179688  4155.384766  4128.333496  4151.666504  ...      2.110017
4  4217.436035  4166.538574  4155.897461  4162.820313  ...      2.462552

   POW.F8.BetaL  POW.F8.BetaH  POW.F8.Gamma  POW.AF4.Theta  POW.AF4.Alpha  \
0      0.504567      0.471979      0.138717       1.801014       1.504794
1      0.606587      0.527616      0.155580       1.859177       1.379617
2      0.795834      0.565414      0.170816       2.027946       1.283876
3      1.021118      0.579656      0.180056       2.265952       1.306188
4      1.230984      0.573620      0.181081       2.461205       1.522420

   POW.AF4.BetaL  POW.AF4.BetaH  POW.AF4.Gamma  subject_understood
0       0.258570       0.435745       0.469483                 0.0
1       0.317579       0.468416       0.642560                 0.0
2       0.441925       0.494701       0.798197                 0.0
3       0.616881       0.506062       0.886495                 0.0
4       0.822598       0.498361       0.874455                 0.0

[5 rows x 87 columns]
```

Paper use:
EEG(electroencephalography)
28k

Results & Analysis

```
 83  POW.AF4.BetaL       2000 non-null   float64
 84  POW.AF4.BetaH       2000 non-null   float64
 85  POW.AF4.Gamma       2000 non-null   float64
 86  subject_understood  2000 non-null   float64
dtypes: float64(85), int64(2)
memory usage: 1.3 MB
```

RNN +OATM (with EGG data)

```
Epoch 5/9
691/691 ━━━━━━━━━━━━━━━━━━━━ 108s 156ms/step - accuracy: 0.9989 - loss: 0.0035 - val_accuracy: 0.9996 - val_loss: 0.0014
Epoch 6/9
691/691 ━━━━━━━━━━━━━━━━━━━━ 106s 154ms/step - accuracy: 1.0000 - loss: 7.7271e-05 - val_accuracy: 0.9998 - val_loss: 8.4756e-04
Epoch 7/9
691/691 ━━━━━━━━━━━━━━━━━━━━ 142s 154ms/step - accuracy: 1.0000 - loss: 2.3762e-05 - val_accuracy: 0.9998 - val_loss: 8.1429e-04
Epoch 8/9
691/691 ━━━━━━━━━━━━━━━━━━━━ 142s 154ms/step - accuracy: 1.0000 - loss: 1.3162e-05 - val_accuracy: 0.9998 - val_loss: 8.2574e-04
Epoch 9/9
691/691 ━━━━━━━━━━━━━━━━━━━━ 108s 156ms/step - accuracy: 1.0000 - loss: 8.4991e-06 - val_accuracy: 0.9998 - val_loss: 8.2899e-04
--------------------------------------------------
RNN Config: lr=0.0010, λ=0.0040, layers=4, nodes=64
Test Accuracy: 0.9993
Time taken: 1098.64 seconds
```

Code Overview –CNN

CNN_RS & CNN_BO

```python
for lr in [0.001, 0.01, 0.05]:
    for lam in [0.001, 0.01, 0.02]:
        for n_layers in [2, 3]:
            for filters in [32, 64]:
```

```python
# ------------------ BAYESIAN OPTIMIZATION ------------
print("\n🧪 Starting CNN Bayesian Optimization...")
space = [
    Real(1e-4, 0.1, name='lr'),
    Real(1e-4, 0.05, name='lam'),
    Integer(2, 4, name='n_layers'),
    Integer(32, 128, name='filters')
]

@use_named_args(space)
def objective(**params):
    acc = cnn_run(params['lr'], params['lam'], params['
```

CNN_OATM

```python
# ------------------ ORTHOGONAL ARRAY ------------------
# Custom Orthogonal Array L9 (3-level, 4-parameter)
oa_combinations = [
    #   lr      lam     n_layers  filters
    [0.001, 0.001,    2,        32],
    [0.001, 0.01,     3,        64],
    [0.001, 0.02,     4,        128],
    [0.01,  0.001,    3,        128],
    [0.01,  0.01,     4,        32],
    [0.01,  0.02,     2,        64],
    [0.05,  0.001,    4,        64],
    [0.05,  0.01,     2,        128],
    [0.05,  0.02,     3,        32],
]

# ------------------ OATM RUN ------------------
```

Code Overview – CNN

```
# ------------------ COMMON CNN TRAINING FUNCTION ------------------
def cnn_run(lr, lam, n_layers, filters):
    filters = int(filters)
    n_layers = int(n_layers)

    model = Sequential()
    model.add(Conv1D(filters, kernel_size=3, activation='relu', padding='same',
                     input_shape=(X_train.shape[1], X_train.shape[2]),
                     kernel_regularizer=l2(lam)))

    for _ in range(1, n_layers):
        model.add(Conv1D(filters, kernel_size=3, activation='relu', padding='same',
                         kernel_regularizer=l2(lam)))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    model.compile(optimizer=Adam(learning_rate=lr),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    history = model.fit(X_train, y_train, epochs=5, verbose=0)
    return history.history['accuracy'][-1]
```

Results & Analysis

CNN_BO

```
✅ Best CNN Bayesian Accuracy: 0.6200000047683716
Best Hyperparameters (Bayesian):
  lr = 0.00026
  lam = 0.00105
  n_layers = 4
  filters = 127
Bayesian Optimization Time: 108.28684377670288 seconds
```

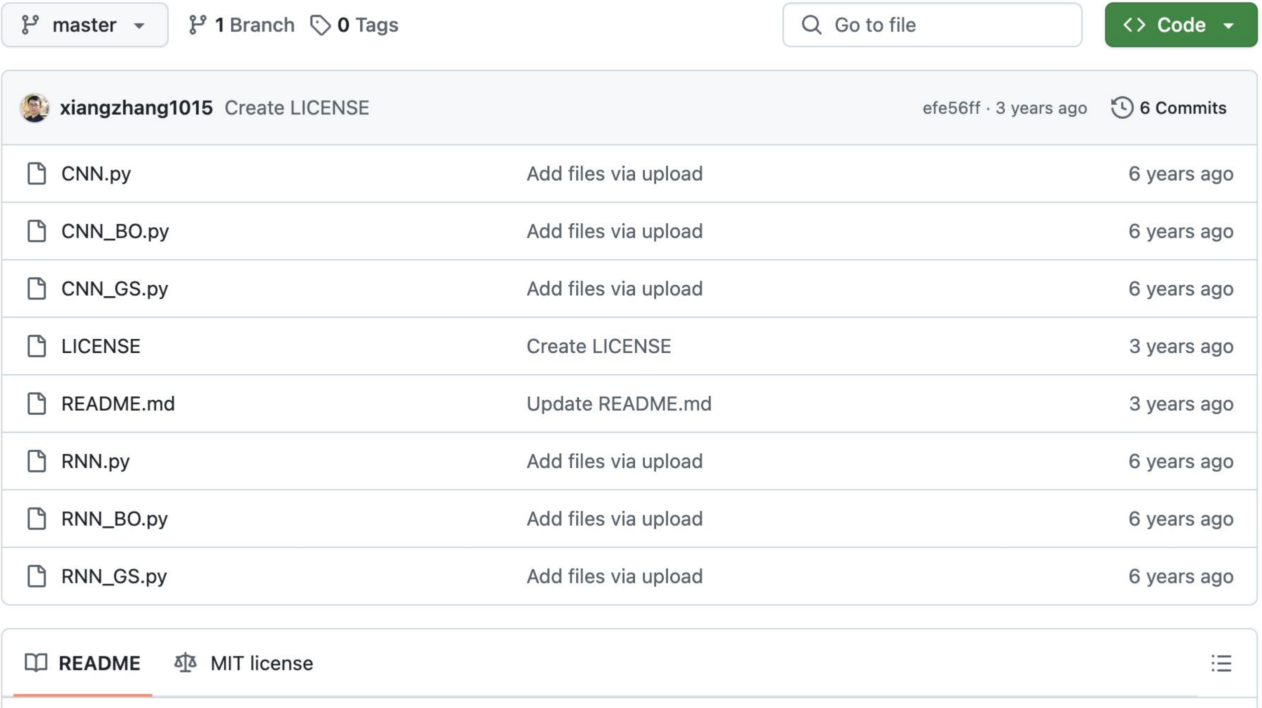
CNN_OATM

```
Testing OA Combination 7: lr=0.05, lam=0.001,
→ Accuracy: 0.5225
Testing OA Combination 8: lr=0.05, lam=0.01, l
→ Accuracy: 0.5225
Testing OA Combination 9: lr=0.05, lam=0.02, l
→ Accuracy: 0.4725

✅ Best CNN OATM Accuracy: 0.7774999737739563
Best Hyperparameters (OATM): {'lr': 0.001, 'la
OATM Tuning Time: 52.97719669342041 seconds
```

CNN_RS

```
Testing CNN: lr=0.05, λ=0.02, layers=3, filters=64
→ Accuracy: 0.4762

✅ Best CNN Grid Search Accuracy: 0.762499988079071
Best Hyperparameters (Grid): {'lr': 0.001, 'lam': 0.
Grid Search Time: 175.24910259246826 seconds
```

Challenges and Modifications



Challenges and Modifications -1

xiangzhang1015 Add files via upload

Eg. RNN_BO

Code | Blame | 26 lines (21 loc) · 822 Bytes | Raw

```python
from RNN import *
from bayes_opt import BayesianOptimization
import time


# define optimization function
# input hyperparameters needed to be adjust
def rnn_bo(lr, lam, n_layers, nodes):
    acc = rnn_run(lr=lr, lam=lam, n_layers=int(n_layers), nodes=int(nodes))
    return acc


# specify the values range of required hyperparameters
bo = BayesianOptimization(rnn_bo,
                          {
                              "lr": (0.005, 0.15),
                              "lam": (0.004, 0.012),
                              "n_layers": (4, 6),
                              "nodes": (32, 96)
                          })

num_iter = 7
init_points = 2
start_time = time.time()
bo.maximize(init_points=init_points, n_iter=num_iter)
print("Running time of the optimization is ", time.time() - start_time)
```

```python
import time
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
from skopt import gp_minimize
from skopt.space import Real, Integer
from skopt.utils import use_named_args

# Generate dummy data
X_train = np.random.rand(1000, 10, 84)
y_train = np.random.randint(0, 2, size=(1000,))

# Common model training function
def rnn_run(lr, lam, n_layers, nodes):
    nodes = int(nodes)  # 👈 ensure nodes is an integer
    n_layers = int(n_layers)  # 👈 ensure n_layers is an integer

    model = Sequential()
    model.add(SimpleRNN(nodes,
                        input_shape=(X_train.shape[1], X_train.shape[2]),
                        return_sequences=(n_layers > 1),
                        kernel_regularizer=l2(lam)))

    for i in range(1, n_layers):
        model.add(SimpleRNN(nodes,
                            return_sequences=(i != n_layers - 1),
                            kernel_regularizer=l2(lam)))

    model.add(Dense(1, activation='sigmoid'))

    model.compile(optimizer=Adam(learning_rate=lr),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    history = model.fit(X_train, y_train, epochs=5, verbose=0)
    return history.history['accuracy'][-1]



  # -------------------- BAYESIAN OPTIMIZATION --------------------
  print("\n🧪 Starting Bayesian Optimization...")
  from skopt import gp_minimize
  from skopt.space import Real, Integer
  from skopt.utils import use_named_args
```

Challenges and Modifications – 2 & 3

- Datasets are not provided

- Timing consuming

```
Testing lr=0.15, λ=0.012, layers=6, nodes=32
Testing lr=0.15, λ=0.012, layers=6, nodes=64
Testing lr=0.15, λ=0.012, layers=6, nodes=96
Running time of the optimization is  1092.134437084198
```

Input data

```
# Generate dummy data (replace with your actual data)
X_train = np.random.rand(1000, 10, 84)  # (samples, timesteps, features)
y_train = np.random.randint(0, 2, size=(1000,))  # Binary labels
```

Critical Evaluation– code

Code:

- Clear structure
- Concise and easy to understand

- Cannot run(have to modify)
- Dataset is not provided

3.5/5

xiangzhang1015 Add files via upload

Code Blame 26 lines (21 loc) · 822 Bytes Raw

```python
from RNN import *
from bayes_opt import BayesianOptimization
import time


# define optimization function
# input hyperparameters needed to be adjust
def rnn_bo(lr, lam, n_layers, nodes):
    acc = rnn_run(lr=lr, lam=lam, n_layers=int(n_layers), nodes=int(nodes))
    return acc


# specify the values range of required hyperparameters
bo = BayesianOptimization(rnn_bo,
                          {
                              "lr": (0.005, 0.15),
                              "lam": (0.004, 0.012),
                              "n_layers": (4, 6),
                              "nodes": (32, 96)
                          })

num_iter = 7
init_points = 2
start_time = time.time()
bo.maximize(init_points=init_points, n_iter=num_iter)
print("Running time of the optimization is ", time.time() - start_time)
```

Critical Evaluation – Methodology

🔁 **Efficiency**	Evaluates far fewer combinations than grid search while maintaining good coverage.
⚡ **Speed**	Much faster than grid search

Critical Evaluation – Methodology

⛓️ **No adaptive learning**	OATM treats each run independently.
⚙️ **Fixed designs**	Limited to predefined OA designs
❌ **Discrete only**	OATM is designed for categorical or discrete levels.

Critical Evaluation – Methodology

Row No.	Factor No.		
	Factor 1	Factor 2	Factor 3
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	2	3
6	2	3	1
7	3	1	3
8	3	2	1
9	3	3	2

Table 1: Orthogonal Array with 9 rows, 3 factors and each factor has 3 levels

Critical Evaluation – Improvement

1. Hybrid Tuning Approach – OATM + BO/GS
2. Adaptive OA – Iterative orthogonal arrays


THANK
YOU