Recognizing hand-written digits

Introduction

This notebook adapts the existing example of applying support vector classification from scikit-learn (https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html#sphx-glr-auto-examples-classification-plot-digits-classification-py) to PyRCN to demonstrate, how PyRCN can be used to classify hand-written digits.

The tutorial is based on numpy, scikit-learn and PyRCN.

In [6]:
```
import numpy as np
import time
from sklearn.base import clone
from sklearn.model_selection import train_test_split
from sklearn.model_selection import (
    ParameterGrid, RandomizedSearchCV, cross_validate)
from sklearn.utils.fixes import loguniform
from scipy.stats import uniform
from sklearn.metrics import make_scorer

from pyrcn.model_selection import SequentialSearchCV
from pyrcn.echo_state_network import ESNClassifier
from pyrcn.metrics import accuracy_score
from pyrcn.datasets import load_digits
```

Load the dataset

The dataset is already part of scikit-learn and consists of 1797 8x8 images.

We are using our dataloader that is derived from scikit-learns dataloader and returns arrays of 8x8 sequences and corresponding labels.

In [7]:
```
X, y = load_digits(return_X_y=True, as_sequence=True)
print("Number of digits: {0}".format(len(X)))
print("Shape of digits {0}".format(X[0].shape))
```

```
Number of digits: 1797
Shape of digits (8, 8)
```

Split dataset in training and test

Afterwards, we split the dataset into training and test sets. We train the ESN using 80% of the digits and test it using the remaining images.

In [8]:
```
stratify = np.asarray([np.unique(yt) for yt in y]).flatten()
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=stratify, random_state=42)
X_tr = np.copy(X_train)
y_tr = np.copy(y_train)
X_te = np.copy(X_test)
y_te = np.copy(y_test)
for k, _ in enumerate(y_tr):
    y_tr[k] = np.repeat(y_tr[k], 8, 0)
for k, _ in enumerate(y_te):
    y_te[k] = np.repeat(y_te[k], 8, 0)

print("Number of digits in training set: {0}".format(len(X_train)))
print("Shape of digits in training set: {0}".format(X_train[0].shape))
print("Number of digits in test set: {0}".format(len(X_test)))
print("Shape of digits in test set: {0}".format(X_test[0].shape))
```

```
Number of digits in training set: 1437
Shape of digits in training set: (8, 8)
Number of digits in test set: 360
Shape of digits in test set: (8, 8)
```

Set up a ESN

To develop an ESN model for digit recognition, we need to tune several hyper-parameters, e.g., input_scaling, spectral_radius, bias_scaling and leaky integration.

We follow the way proposed in the introductory paper of PyRCN to optimize hyper-parameters sequentially.

We define the search spaces for each step together with the type of search (a grid search in this context).

At last, we initialize a SeqToLabelESNClassifier with the desired output strategy and with the initially fixed parameters.

In [9]:
```
initially_fixed_params = {'hidden_layer_size': 50,
                          'input_activation': 'identity',
                          'k_in': 5,
                          'bias_scaling': 0.0,
                          'reservoir_activation': 'tanh',
                          'leakage': 1.0,
                          'bidirectional': False,
                          'k_rec': 10,
                          'wash_out': 0,
                          'continuation': False,
                          'alpha': 1e-5,
                          'random_state': 42,
                          'decision_strategy': "winner_takes_all"}

step1_esn_params = {'input_scaling': uniform(loc=1e-2, scale=1),
                    'spectral_radius': uniform(loc=0, scale=2)}
```

```
step2_esn_params = {'leakage': loguniform(1e-5, 1e0)}
step3_esn_params = {'bias_scaling': uniform(loc=0, scale=2)}
step4_esn_params = {'alpha': loguniform(1e-5, 1e0)}

kwargs_step1 = {'n_iter': 20, 'random_state': 42, 'verbose': 1, 'n_jobs': -1,
                'scoring': make_scorer(accuracy_score)
                }
kwargs_step2 = {'n_iter': 5, 'random_state': 42, 'verbose': 1, 'n_jobs': -1,
                'scoring': make_scorer(accuracy_score)
                }
kwargs_step3 = {'verbose': 1, 'n_jobs': -1,
                'scoring': make_scorer(accuracy_score)
                }
kwargs_step4 = {'n_iter': 5, 'random_state': 42, 'verbose': 1, 'n_jobs': -1,
                'scoring': make_scorer(accuracy_score)
                }

# The searches are defined similarly to the steps of a sklearn.pipeline.Pipeli
searches = [('step1', RandomizedSearchCV, step1_esn_params, kwargs_step1),
            ('step2', RandomizedSearchCV, step2_esn_params, kwargs_step2),
            ('step3', RandomizedSearchCV, step3_esn_params, kwargs_step3),
            ('step4', RandomizedSearchCV, step4_esn_params, kwargs_step4)]

base_esn = ESNClassifier(**initially_fixed_params)
```

Optimization

We provide a SequentialSearchCV that basically iterates through the list of searches that we have defined before. It can be combined with any model selection tool from scikit-learn.

In [10]:
```
sequential_search = SequentialSearchCV(base_esn,
                                       searches=searches).fit(X_tr, y_tr)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Fitting 5 folds for each of 5 candidates, totalling 25 fits
```

Use the ESN with final hyper-parameters

After the optimization, we extract the ESN with final hyper-parameters as the result of the optimization.

In [11]:
```
base_esn = sequential_search.best_estimator_
```

In [12]:
```
base_esn.get_params()
```

Out[12]:
```
{'bias_scaling': 1.7698288197982674,
 'bias_shift': 0.0,
 'hidden_layer_size': 50,
 'input_activation': 'identity',
 'input_scaling': 0.06808361216819946,
 'input_shift': 0.0,
 'k_in': 5,
 'predefined_bias_weights': None,
 'predefined_input_weights': None,
 'random_state': 42,
 'sparsity': 0.2,
 'bidirectional': False,
 'k_rec': 10,
 'leakage': 0.009846738873614563,
 'predefined_recurrent_weights': None,
 'reservoir_activation': 'tanh',
 'spectral_radius': 1.7323522915498704,
 'alpha': 6.026889128682509e-05}
```

In [13]:
```
sequential_search.all_cv_results_["step4"]
```

Out[13]:
```
{'mean_fit_time': array([0.99171114, 0.99734344, 0.99260564, 0.99950948, 0.958
91252]),
 'std_fit_time': array([0.0621232 , 0.04251586, 0.03642196, 0.03933799, 0.0442
4365]),
 'mean_score_time': array([0.27029257, 0.27011256, 0.26387949, 0.28059196, 0.2
7726517]),
 'std_score_time': array([0.01596187, 0.02623838, 0.01508273, 0.02181204, 0.02
077122]),
 'param_alpha': masked_array(data=[0.0007459343285726546, 0.5669849511478852,
                    0.045705630998014515, 0.009846738873614563,
                    6.026889128682509e-05],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'alpha': 0.0007459343285726546},
  {'alpha': 0.5669849511478852},
  {'alpha': 0.045705630998014515},
  {'alpha': 0.009846738873614563},
  {'alpha': 6.026889128682509e-05}],
 'split0_test_score': array([0.64930556, 0.43619792, 0.57421875, 0.62803819,
0.66666667]),
 'split1_test_score': array([0.65147569, 0.46354167, 0.59939236, 0.640625  ,
0.66102431]),
 'split2_test_score': array([0.63937282, 0.42857143, 0.56358885, 0.59843206,
0.6533101 ]),
 'split3_test_score': array([0.6445993 , 0.46907666, 0.57012195, 0.60060976,
0.6511324 ]),
 'split4_test_score': array([0.6141115 , 0.44642857, 0.54747387, 0.58449477,
0.63980836]),
 'mean_test_score': array([0.63977297, 0.44876325, 0.57095916, 0.61043996, 0.6
5438837]),
 'std_test_score': array([0.01348917, 0.01550538, 0.01688581, 0.02066306, 0.00
915568]),
 'rank_test_score': array([2, 5, 4, 3, 1], dtype=int32)}
```

Test the ESN

Finally, we increase the reservoir size and compare the impact of uni- and bidirectional ESNs. Notice that the ESN strongly benefit from both, increasing the reservoir size and from the bi-directional working mode.

In [ ]:
```
param_grid = {'hidden_layer_size': [50, 100, 200, 400, 500],
              'bidirectional': [False, True]}

print("CV results\tFit time\tInference time\tAccuracy score\tSize[Bytes]")
for params in ParameterGrid(param_grid):
    esn_cv = cross_validate(clone(base_esn).set_params(**params), X=X_train, y=
                            scoring=make_scorer(accuracy_score), n_jobs=-1)
    t1 = time.time()
    esn = clone(base_esn).set_params(**params).fit(X_train, y_train)
    t_fit = time.time() - t1
    t1 = time.time()
    esn_par = clone(base_esn).set_params(**params).fit(X_train, y_train, n_job
    t_fit_par = time.time() - t1
    mem_size = esn.__sizeof__()
    t1 = time.time()
    acc_score = accuracy_score(y_test, esn.predict(X_test))
    t_inference = time.time() - t1
    print("{0}\t{1}\t{2}\t{3}\t{4}".format(esn_cv, t_fit, t_inference, acc_sco
                                           mem_size))
```

```
CV results      Fit time        Inference time  Accuracy score  Size[Bytes]
{'fit_time': array([1.05878973, 0.99830127, 1.01817036, 0.97793221, 1.1108930
1]), 'score_time': array([0.29641557, 0.30827832, 0.26319551, 0.22847462, 0.26
747322]), 'test_score': array([0.91666667, 0.89236111, 0.87456446, 0.8815331 ,
0.87456446])}   1.3643665313720703      0.30077600479125977     0.894444444444
4445    29892
{'fit_time': array([ 9.63227057, 10.61319089,  9.59326506,  9.00197959, 10.632
12609]), 'score_time': array([0.26419258, 0.24499059, 0.27687764, 0.2585876 ,
0.26462984]), 'test_score': array([0.9375    , 0.92013889, 0.91986063, 0.91289
199, 0.90243902])}      13.251605749130249      0.3381636142730713      0.9305
555555555556    99092
{'fit_time': array([11.74328637, 11.56316018, 12.22108722, 11.40854406, 10.392
79962]), 'score_time': array([0.27444959, 0.46049619, 0.40587139, 0.30505157,
0.27671957]), 'test_score': array([0.94444444, 0.94444444, 0.93031359, 0.93031
359, 0.91637631])}      11.908453226089478      0.3247661590576172      0.9555
555555555556    357492
```

In [ ]:

In [ ]: