PySEF Documentation

Release 0.2.9

Nikolaos Passalis

Feb 05, 2018

Contents:

PySEF provides an implementation of the Similarity Embedding Framework (SEF) on top of the *PyTorch* library. *PySEF* is an easy to use Python library that exposes a *scikit-learn*-like interface, allows for easily implementing novel dimensionality reduction techniques and can efficiently handle large-scale dataset using the GPU.

CHAPTER 1

What is the Similarity Embedding Framework?

The vast majority of Dimensionality Reduction techniques rely on second-order statistics to define their optimization objective. Even though this provides adequate results in most cases, it comes with several shortcomings. The methods require carefully designed regularizers and they are usually prone to outliers. The Similarity Embedding Framework can overcome the aforementioned limitations and provides a conceptually simpler way to express optimization targets similar to existing DR techniques. Deriving a new DR technique using the Similarity Embedding Framework becomes simply a matter of choosing an appropriate target similarity matrix. A variety of classical tasks, such as performing supervised dimensionality reduction and providing out-of-of-sample extensions, as well as, new novel techniques, such as providing fast linear embeddings for complex techniques, are demonstrated.

1.1 Installation

1.1.1 Recommended (pip-based) installation

A working installation of PyTorch is required before installing PySEF. To install PyTorch, please follow the instructions given in the PyTorch site.

The recommended way to install PySEF is simply to use the *pip* package manager:

```
pip install pysef
```

All the other required dependecies will be automatically downloaded and installed.

PySEF is developed and tested on Linux (both Python 2.7 and Python 3.5 are supported). However, it is expected to run on Windows/Mac OSX as well, since all of its components are cross-platform.

1.1.2 Bleeding-edge installation

To install the latest version availabe at github, clone our repository and manually install the package:

```
git clone https://github.com/passalis/sef
cd sef
python setup.py install --user
```

1.2 Using PySEF

1.2.1 How to get started?

After installing PySEF (see *Installation*), simply import sef_dr, create a SEF object, fit it and transform your data:

```
import sef_dr
proj = sef_dr.LinearSEF(input_dimensionality=784, output_dimensionality=9)
proj.fit(data=data, target_labels=data, target='supervised', iters=10)
transformed_data = proj.transform(data)
```

The *input_dimensionality* parameter defines the dimensionality of the input data, while the *output_dimensionality* refers to the desired dimensionality of the data. Then, we can learn the projection using the *.fit()* function. The method that will be used for reducing the dimensionality of the data is specified in the *target* parameter of the *.fit()* method (PySEF provides many predefined targets/methods for dimensionality reduction, even though new methods can be also easily implemented as shown in *Extending PySEF*). Several different dimensionality reduction scenarios are discussed in the following sections (for all the conducted experiments the well-known MNIST dataset is used).

1.2.2 Using GPU acceleration

Following the PyTorch calling conventions, to use the GPU for the optimization/projection the *.cuda()* method can be used:

```
proj.cuda()
```

To move the model back to cpu, the *.cpu()* method should be called:

```
proj.cpu()
```

1.2.3 Data loading

To allow for easily evaluating and comparing different dimensionality reduction techniques we have included data loading capabilities in PySEF. Before running any of the following examples, please download the pre-extracted feature vectors from the following drobpox folder. After downloading them into a folder, e.g., let's say that we download them into the */home/nick/my_data* folder, you can easily load any of the six supported datasets as follows:

```
from sef_dr.datasets import dataset_loader
train_data, train_labels, test_data, test_labels = dataset_loader(dataset='mnist',␣
↪dataset_path='/home/nick/my_data')
train_data, train_labels, test_data, test_labels = dataset_loader(dataset='20ng',␣
↪dataset_path='/home/nick/my_data')
train_data, train_labels, test_data, test_labels = dataset_loader(dataset='15scene',␣
↪dataset_path='/home/nick/my_data')
train_data, train_labels, test_data, test_labels = dataset_loader(dataset='corel',␣
↪dataset_path='/home/nick/my_data')
```

```
train_data, train_labels, test_data, test_labels = dataset_loader(dataset='yale',␣
↪dataset_path='da/home/nick/my_datata')
train_data, train_labels, test_data, test_labels = dataset_loader(dataset='kth',␣
↪dataset_path='da/home/nick/my_datata')
```

The *MNIST dataset* and the *20NG dataset* will be automatically downloaded into the specified folder the first time that the *dataset_loader()* function will be called. Please refer to this paper for a detailed description of the evaluation setup and feature extraction process.

1.2.4 Recreating the geometry of a high dimensional space into a space with less dimensions

In unsupervised_approximation.py we demonstrate how to recreate the 50-d PCA using just 10 dimensions:

```
# Learn a high dimensional projection
proj_to_copy = PCA(n_components=50)
proj_to_copy.fit(train_data[:n_train_samples, :])
target_data = np.float32(proj_to_copy.transform(train_data[:n_train_samples, :]))

# Approximate it using the SEF and 10 dimensions
proj = LinearSEF(train_data.shape[1], output_dimensionality=10)
proj.cuda()
loss = proj.fit(data=train_data[:n_train_samples, :], target_data=target_data, target=
↪'copy', epochs=50, batch_size=128, verbose=True, learning_rate=0.001, regularizer_
↪weight=0.001)

# Evaluate the method
acc = evaluate_svm(proj.transform(train_data[:n_train_samples, :]), train_labels[:n_
↪train_samples], proj.transform(test_data), test_labels)
```

The experimental results demonstrate the ability of the proposed method to efficiently recreate the geometry of a high dimensional space into a space with less dimensions:

Method	Accuracy
PCA 10-d	82.88%
Linear SEF mimics PCA-20d	**84.87%**

1.2.5 Re-deriving similarity-based versions of well-known techniques

In supervised_reduction.py we demonstrate how to rederive similarity-based versions of well-known techniques. More specifically, a similarity-based LDA-like technique is derived:

```
proj = LinearSEF(train_data.shape[1], output_dimensionality=(n_classes - 1))
proj.cuda()
loss = proj.fit(data=train_data[:n_train, :], target_labels=train_labels[:n_train],␣
↪epochs=50, target='supervised', batch_size=128, regularizer_weight=0.001,␣
↪verbose=True)
```

The SEF-based method leads to superior results:

Method	Dimensionality	Accuracy
LDA	9d	85.66%
Linear SEF	9d	88.89%
Linear SEF	**18d**	**89.48%**

1.2.6 Providing out-of-sample extensions

In linear_outofsample.py and kernel_outofsample.py we use the SEF to provide (linear and kernel) out-of-sample extensions for the ISOMAP technique. Note that the SEF, unlike the regression-based method, is not limited by the number of dimensions of the original technique:

```
isomap = Isomap(n_components=10, n_neighbors=20)
train_data_isomap = np.float32(isomap.fit_transform(train_data[:n_train_samples, :]))
proj = LinearSEF(train_data.shape[1], output_dimensionality=10)
proj.cuda()
loss = proj.fit(data=train_data[:n_train_samples, :], target_data=train_data_isomap,␣
↪target='copy', epochs=50, batch_size=128, verbose=True, learning_rate=0.001,␣
↪regularizer_weight=0.001)
```

The results are shown in the following tables:

Method	Dimensionality	Accuracy
Linear Regression	10d	85.25%
Linear SEF	10d	85.76%
Linear SEF	**20d**	**89.48%**

Method	Dimensionality	Accuracy
Kernel Regression	10d	89.48%
Kernel SEF	10d	88.60%
Kernel SEF	**20d**	**90.88%**

1.2.7 Performing SVM-based analysis

Finally, in svm_approximation.py an SVM-based analysis technique that mimics the similarity induced by the hyperplanes of the 1-vs-1 SVMs is used to perform DR. This method allows for using a light-weight classifier, such as the NCC, to perform fast classification:

```
# Learn an SVM
scaler = MinMaxScaler()
train_data = scaler.fit_transform(train_data)
test_data = scaler.transform(test_data)

parameters = {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]}
model = grid_search.GridSearchCV(svm.SVC(max_iter=10000, decision_function_shape='ovo
↪'), parameters, n_jobs=-1, cv=3)
model.fit(train_data[:n_train], train_labels[:n_train])

# Learn a similarity embedding
params = {'model': model, 'n_labels': np.unique(train_labels).shape[0], 'scaler':␣
↪scaler}
proj = LinearSEF(train_data.shape[1], output_dimensionality=dims)
```

```
proj.cuda()
loss = proj.fit(data=train_data[:n_train, :], target_data=train_data[:n_train, :],␣
↪target_labels=train_labels[:n_train], target='svm', target_params=params, epochs=50,
↪ learning_rate=0.001, batch_size=128, verbose=True, regularizer_weight=0.001)
```

This code repeatedly calls the SVM to calculate the similarity matrix for the samples in each batch. If the whole similarity matrix can fit into the memory, we can speed up this process by using a precomputed similarity matrix as follows:

```
from sef_dr.targets import generate_svm_similarity_matrix, sim_target_svm_precomputed

# Precompute the similarity matrix
Gt = generate_svm_similarity_matrix(train_data, train_labels, len(np.unique(train_
↪labels)), model, scaler)
params = {'Gt': Gt}

proj = LinearSEF(train_data.shape[1], output_dimensionality=dims)
proj.cuda()
loss = proj.fit(data=train_data, target_data=train_data, target_labels=train_labels,␣
↪target=sim_target_svm_precomputed, target_params=params, epochs=50, learning_rate=0.
↪001, batch_size=128, verbose=True, regularizer_weight=0.001)
```

The results are shown in the following table:

Method	Dimensionality	Accuracy
NCC - Original	784d	80.84%
NCC - Linear SEF	10d	86.50%
NCC - Linear SEF	**20d**	**86.67%**

1.3 Examples and Tutorials

We provide examples using six different datasets (15-Scene, Corel, MNIST, Yale, KTH, and 20NG) to reproduce the results obtained in the original research paper. Note that slight differences from the original paper are due to some changes (batch-based optimization, faster estimation of the scaling factor, port to PyTorch). For all the reported results a Linear SVM is trained, unless otherwise stated.

1.3.1 Prerequisites

To run the examples you have to install PySEF (please also refer to *Installation*):

```
pip install pysef
```

Before running any of the following examples, please download the pre-extracted feature vectors from the following drobpox folder into a folder named *data* and execute the script from the same root folder (or simply update the data path in the code). Refer to *Data loading* for more details.

Please also install *matplotlib*, which is also needed for some of the following examples/tutorials:

```
pip install matplotlib
```

1.3.2 Linear approximation of a high-dimensional technique

In unsupervised_approximation_multiple.py we demonstrate how to recreate the 50-d PCA using just 10 dimensions. The proposed method (abbreviated as S-PCA) is compared to the 10-d PCA method. To run the example, simply download the aforementioned file and execute it:

```
python unsupervised_approximation_multiple.py
```

The following results should be obtained:

Dataset	PCA	S-PCA
15-scene	61.94%	**67.20%**
Corel	36.18%	**38.55%**
MNIST	82.88%	**84.71%**
Yale	56.69%	**65.16%**
KTH	76.82%	**86.56%**
20NG	39.73%	**45.79%**

1.3.3 Supervised dimensionality reduction

In supervised_reduction_multiple.py we demonstrate how to perform supervised dimensionality reduction using the SEF. Two different setups are used: a) *S-LDA-1*, where the same dimensionality as the LDA method is used, and b) *S-LDA-2*, where the number of dimensions is doubled. To run the example, simply download the aforementioned file and execute it:

```
python supervised_reduction_multiple.py
```

The following results should be obtained:

Dataset	LDA	S-LDA-1	S-LDA-2
15-scene	66.76%	75.58%	**76.98%**
Corel	37.28%	**42.58%**	42.33%
MNIST	85.66%	89.03%	**89.27%**
Yale	93.95%	92.50%	**92.74%**
KTH	90.38%	90.73%	**91.66%**
20NG	63.57%	**70.35%**	70.25%

1.3.4 Providing out-of-sample-extensions

In linear_outofsample_mutiple.py we demonstrate how to provide out-of-sample extensions for the ISOMAP technique. Two different setups are used: a) *cS-ISOMAP-1*, where the dimensionality of the projection is set to 10, and b) *cS-ISOMAP-2*, where the dimensionality of the projection is set to 20. The proposed method is compared to performing linear regression (LR). To run the example, simply download the aforementioned file and execute it:

```
python linear_outofsample_mutiple.py
```

The following results should be obtained:

Dataset	LR	cS-ISOMAP-1	cS-ISOMAP-2
15-scene	58.29%	67.26%	**69.04%**
Corel	34.93%	38.70%	**40.45%**
MNIST	85.11%	85.93%	**93.37%**
Yale	35.97%	62.09%	**82.58%**
KTH	67.20%	86.56%	**89.80%**
20NG	33.14%	41.52%	**47.97%**

Kernel extensions can be also used (kernel_outofsample_mutiple.py). The following results should be obtained:

Dataset	KR	cKS-ISOMAP-1	cKS-ISOMAP-2
15-scene	60.10%	63.89%	**68.14%**
Corel	36.22%	37.85%	**42.27%**
MNIST	89.48%	88.30%	**91.35%**
Yale	46.94%	29.84%	**62.25%**
KTH	72.31%	78.22%	**83.31%**
20NG	44.50%	41.57%	**48.81%**

1.3.5 SVM-based analysis

PySEF can be used to mimic the similarity induced by the hyperplanes of the 1-vs-1 SVMs and perform DR (svm_approximation_multiple.py). The proposed technique is combined with a lightweight Nearest Centroid Classifier. Two different setups are used: a) *S-SVM-A-1*, where the dimensionality of the projection is set to the number of classes, and b) *S-SVM-A-1*, where the dimensionality of the projection is set to twice the number of classes. To run the example, simply download the aforementioned file and execute it:

```
python svm_approximation_multiple.py
```

The following results should be obtained:

Dataset	Original	S-SVM-A-1	S-SVM-A-1
15-scene	59.67%	**74.47%**	74.10%
Corel	37.40%	**42.15%**	41.77%
MNIST	80.84%	86.71%	**86.80%**
Yale	13.95%	84.44%	**88.63%**
KTH	79.72%	92.24%	**94.09%**
20NG	60.79%	65.37%	**65.78%**

1.3.6 PySEF tutorials

To run the tutorials you have to install the Jupyter Notebook (also refer to Installing Jupyter):

```
pip install jupyter
```

Then, download the notebook tutorial you are interested in. Currently two tutorial are available: a) Supervised dimensionality reduction, and b) Defining new dimensionality reduction methods. Then, navigate to the folder that contains the notebook and start the Jupyter Notebook:

```
jupyter notebook
```

Finally, navigate to the default URL of Jupyter web app (http://localhost:8888) and select the notebook. Please make sure that you appropriately update the folder that contains the MNIST dataset when running the tutorials (refer to *Data loading* for more details, or just create an empty folder named *data* in the same root folder as the notebook and the dataset will be automatically downloaded).

1.4 Extending PySEF

1.4.1 Deriving new DR methods

The SEF allows for easily deriving novel DR techniques by simply defining the target similarity matrix. For the current implementation this can be done by defining a function that adheres to the following signature:

```python
def custom_similarity_function(target_data, target_labels, sigma, idx, target_params):
    Gt = np.zeros((len(idx), len(idx)))
    Gt_mask = np.zeros((len(idx), len(idx)))
    # Calculate the similarity target here
    return np.float32(Gt), np.float32(Gt_mask)
```

The *target_data, target_labels, sigma,* and *target_params* are passed to the *.fit()* function. During the training this function is called with a different set of indices *idx* and it is expected to return the target similarity matrix for the data that correspond to the indices defined by idx.

For example, let's define a function that sets a target similarity of 0.8 for the samples that belong to the same class, and 0.1 for the samples that belong to different classes:

```python
def sim_target_supervised(target_data, target_labels, sigma, idx, target_params):

    cur_labels = target_labels[idx]
    N = cur_labels.shape[0]

    N_labels = len(np.unique(cur_labels))

    Gt, mask = np.zeros((N, N)), np.zeros((N, N))

    for i in range(N):
        for j in range(N):
            if cur_labels[i] == cur_labels[j]:
                Gt[i, j] = 0.8
                mask[i, j] = 1
            else:
                Gt[i, j] = 0.1
                mask[i, j] = 0.8 / (N_labels - 1)

    return np.float32(Gt), np.float32(mask)
```

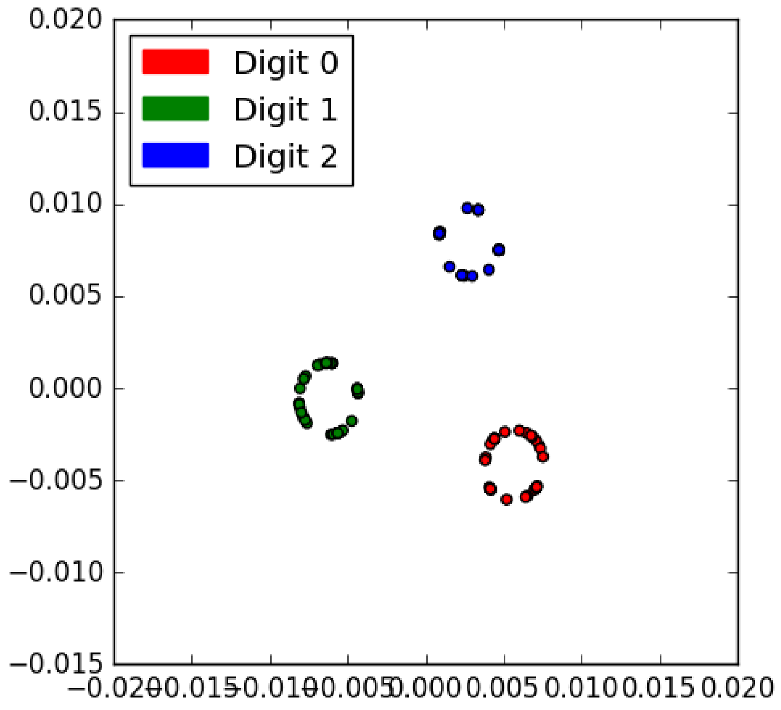
Note that we also appropriately set the weighting mask to account for the imbalance between the intra-class and inter-class samples. It is important to remember to work only with the current batch (using the *idx*) and not use the whole training set (that is always passed to *target_data/target_labels*). You can find more target function examples in sef_dr/targets.py.

The target that we just defined tries to place the samples of the same class close together (but not to collapse them into the same point), as well as to repel samples of different classes (but still maintain as small similarity between them). Of course this problem is ill-posed in the 2-D space (when more than 3 points per class are used), but let's see what happens!

Let's overfit the projection:

```
proj = KernelSEF(train_data, train_data.shape[0], 2, sigma=1, learning_rate=0.0001,␣
↪regularizer_weight=0)
proj.fit(train_data, target_labels=train_labels, target=sim_target_supervised,␣
↪iters=500, verbose=True)
train_data = proj.transform(train_data)
```

and visualize the results:



Close enough! The samples of the same class have been arranged in circles, while the circles of different classes are almost equidistant to each other!