RosenPy

What is RosenPy?

- A complex-valued neural network library, written in Python;
- Incorporates CVNNs such as CV-FFNN (complex-valued feedforward neural network), SC-FFNN (split-complex feedforward neural network), CV-RBFNN (com-plex-valued radial basis function neural network), FC-RBFNN (fully-complex radial basis function neural network), and Deep PT-RBFNN (deep phase transmittance radial basis function neural network);
- It enables the incorporation of properties intrinsic to neural networks, such as momentum, L2 regularization, early stopping, mini-batch, and learning rate decay.

Dependencies

Python3.6+, Numpy

Features and Benefits

RosenPy is easy to use, has a fast learning curve for the end-user, and is implemented in one of the most popular programming languages available today. Additionally, the framework incorporates several features that aid in optimizing complex-valued prediction problems. In addition, the API is consistent, straightforward, extensible, and has a simple structure.

It supports five different complex-valued neural networks that the end-user can easily model and train by simple configuration of their hyperparameters.

Authors

- Ariadne Arrais Cruz – ariadne@gmail.com
- Kayol Soares Mayer – kayolmayer@gmail.com
- Dalton Soares Arantes – dalton@unicamp.br

Getting Started

Python
RosenPy is a python-based neural network library, so python must be installed on your machine. If Python is not installed, then visit the official python link – https://www.python.org/ – and download the latest version based on your OS and install it.

Python library
To avoid future release dependencies, RosenPy only uses the **NumPy** package, a basic library for Python scientific computation. If this library is not installed, then you must install it before using RosenPy.

RosenPy Installation
Currently, all RosenPy installation prerequisites have been satisfied, and installing RosenPy is straightforward. To make RosenPy work, you must clone the repository from GitHub to your local machine.

Workflow of a CVNN in the RosenPy

To comprehend the correct operation of RosenPy, it is crucial to understand the steps of deep learning, which will be briefly given below.

Collect and analyze the required data
Deep learning requires large amounts of input data to successfully learn and predict results. A first analysis on the available data provides useful information for the proper selection of the CVNN algorithm.

Choose an algorithm
Select an algorithm that best corresponds to the type of learning process and to the available input data. Five CVNN algorithms are implemented in RosenPy. There are shallow (CV-RBFNN and FC-RBFNN) and deep neural network algorithms.

Split data
It is important to split all the available data into two separate groups. The first group will be used for model learning, while the second group will be used for testing (inference phase).

Define the model
After the algorithm has been selected, we will model the network and add the necessary layers in this step. For instance, when defining the network, we will need to specify the loss function and the learning rate. Later, it will be necessary to add layers defining, for instance, neurons and activation functions. The following section will describe the RosenPy code used to model the network.

Fit the model
At this stage, with the modeled network, the actual learning process will be conducted using the training dataset. The network may need to be redesigned and retrained until the desired convergence rate is reached.

Predict result for unknown value
In addition to training and testing data, it is essential to predict the network output of unknown data. This is possible at this stage.

Architecture of RosenPy

RosenPy library can be divided into two modules:

- **`src.rosenpymodel`**; and
- **`src.rp_utils`**.

src.rosenpymodel
It contains the files that implement the complex domain neural networks. RosenPy Model is composed of the **`NeuralNetwork`** class and the Layers class.

src.rp_utils
It collects the files that implement activation functions, batch generation, cost functions, learning rate decay, regularization, and the initialization of weights and biases.

In RosenPy, the CVNN is represented by the **`NeuralNetwork`** class. In turn, the entire **`NeuralNetwork`** is composed of layers. Both CV-RBFNN and FC-RBFNN have only one layer, while the other RosenPy networks may have more than one layer.

After setting up the execution environment and choosing the proper algorithm, it must be imported into the environment, as follows:

```
import rosenpymodel.<file> as mynn
```

The **`<file>`** is one of the CVNN algorithms implemented in RosenPy, which are cvffnn, scffnn, cvrbfnn, fcrbfnn and deepptrbfnn.

Next, it is necessary to import the functionality modules, as follows:

```
from rp_utils import costFunc, actFunc, initFunc, regFunc, decayFunc,
                     batchGenFun, utils
```

The next step is model definition. We begin by initializing the **`NeuralNetwork`** class to define the neural network. The attributes of this class will be described later. A simple CV-FFNN model is defined as follows:

```
nn = mynn.CVFFNN()
```

After initializing the **`NeuralNetwork`** class, it is required to add at least one layer to the network, regardless of the algorithm type selected. If the network is shallow, only one layer will be added.

To add layers to the network, the **`addLayer()`** method must be called with the appropriate parameters. For each of the CVNNs implemented in RosenPy, the **`ishape`** argument, which represents the input layer dimension, is only used in the first layer of the model, while the neurons argument represents the number of hidden nodes. The **`oshape`** is a specific argument for the RBF networks; in shallow CVNNs, as there is only one layer, the input and output dimensions and the number of hidden neurons must be specified when adding the layer; in deep PT-RBFNN, if the output dimension is not specified, the algorithm will figure out that the number of hidden neurons is the bottleneck for this layer. Simply add the neurons parameter (with the dimensionality of the hidden nodes in this layer) to the other layers of the FF networks; however, for a Deep PT-RBF, it is also possible to customize the number of neurons in the bottleneck layer (**`oshape`**) in addition to the hidden layer (**`neurons`**). If this parameter is not set, the package will assume that the number of neurons in the bottleneck layer is the same as the number of neurons in the hidden layer. In the last layer of FF networks, the output dimension is

set by the number of neurons in that layer. In deep PT-RBFNN, the output dimension can be set by (**neurons**) or (**oshape**) parameters, if they are turned on, as follows:

```
nn.addLayer(ishape=input_dim, neurons=100)
nn.addLayer(neurons=output_dim)
```

Since the model has already been defined, it can be trained by using the **fit()** method. Training occurs over epochs, as follows:

```
hist = nn.fit(input_data, output_data, epochs=2000, verbose=100)
```

Once the model is created and trained, it can be used to make predictions with the **predict()** method, as follows:

```
output_pred = nn.predict(input)
```

The NeuralNetwork class

CVNN implementations in RosenPy derive from the **`NeuralNetwork`** superclass, where the methods and parameters will be detailed below.

First, the class constructor:

```
nn = mynn.CVFFNN( cost_func=costFunc.mse,
                  learning_rate=1e-3,
                  lr_decay_method=decayFunc.none_decay,
                  lr_decay_rate=0.0,
                  lr_decay_steps=1,
                  momentum=0.0,
                  patience=np.inf)
```

The constructor class initializes the object with a default cost function and learning rate given by MSE (Mean Squared Error) and **`0.001`**, respectively. Nevertheless, during the creation of the object, the learning rate decay, momentum factor, and patience factor — parameter for the implementation of early stopping — can be specified. These optimization techniques are disabled by default.

Parameters:
- **`cost_func`, `default=costFunc.mse`**: It is used to quantify this loss during the training phase in the form of a single real number. **`init.costFunc.py`** defines the cost function;
- **`learning_rate`, `default=1e-3`**: It is a parameter used in the training of CVNN that has a small positive value, between **`0.0`** and **`1.0`**;
- **`lr_decay_method`, `default=decayFunc.none_decay`**: This is the decay method parameter, which is implemented in **`rp_utils.decayFunc.py`**.
- **`lr_decay_rate`, `default=0.0`:** It is a parameter for the method. It sets the decay rate.
- **`lr_decay_steps`, `default=1`:** It is a parameter for the staircase method.
- **`momentum`, `default=0.0`:** It sets the momentum rate.
- **`patience`, `default=numpy.inf`:** It specifies the number of epochs that the CVNN will wait to stop using it.

`addLayer()` method

```
nn.addLayer( ishape[only first layer],  neurons, oshape[CV & FC-RBF],
             weights_initializer=initFunc.random_normal,
             bias_initializer=initFunc.random_normal,
             activation=actFunc.tanh,
             reg_strength=0.0,
             lambda_init=0.1)
```

In addition to the input and output dimensions and the number of hidden neurons described in the Architecture of RosenPy section, it is possible to define the layer activation function, initialization of weights and biases, regularization L2 and, for RBF Neural Networks, the learning rate of the matrix of center vectors (gamma rate) and of the vector of variances (sigma rate), when adding layers to the network.

Parameters:

- **`ishape`:** It refers to the dimension of the input layer;
- **`neurons`:** The number of neurons in the hidden layer ;
- **`oshape`:** It refers to the dimension of the bottleneck or output layer for the RBF networks;
- **`weights_initializer`:** It defines the way to set the initial random weights;

- **bias_initializer:** It defines the way to set the initial random biases;
- **reg_strength:** It sets the regularization strength. The default value is **0.0**, which means that regularization is turned off;
- **lambda_init:** It is the initial regularization factor strength;
- **gamma_rate:** The learning rate of matrix of the center vectors (RBF networks);
- **sigma_rate:** The learning rate of the vector of variance (RBF networks).

oshape, gamma_rate *and* **sigma_rate** *are particular parameters for RBF networks.*

fit() method

```
nn.fit( x_train, y_train,
        x_val=None, y_val=None,
        epochs=100, verbose=10,
        batch_gen=batchGenFunc.batch_sequential, batch_size=None)
```

In **fit()** method the training occurs over epochs, and each epoch can be split into batches. The training process will run for a fixed number of iterations, which must be specified using the epochs argument. In addition, it is possible to specify the number of samples processed before the model is updated, simply by changing the batch size and batch gen arguments.

It is feasible to include both the validation dataset and verbose arguments. During the training, if the validation dataset is not empty, this method logs the value of the loss function for the training and validation dataset for every **n** epochs, where **n** is the verbose argument. It returns a dictionary containing an entry for each loss value in the training and validation datasets at consecutive epochs.

Parameters:

- **x_train:** Input data;
- **y_train:** Target data;
- **x_val:** Input validation data;
- **y_val:** Target validation data;
- **epochs:** Number of epochs to train the model. It is an integer number;
- **verbose:** It logs the value of the loss function for the training and validation dataset for every **n** epochs, where **n** is the verbose argument;
- **batch_gen:** Batch can be sequential or shuffled; and
- **batch_size:** Number of samples per gradient update.

predict() method

nn.predict(x)
It generates output predictions for the input samples and return a numpy array(s) of predictions.

getHistory() method

hist = nn.getHistory()

It is a method that returns a dictionary containing an entry for each loss value in the training and validation datasets at consecutive epochs to the training model.

Modules – rp_utils

The **rp_utils** folder contains the files that implement activation functions, batch generation, cost functions, learning rate decay, regularization, weights and biases initialization, and some utilities functions.

initFunc
Common initialization methods include zero, one, normal distribution, uniform distribution, Glorot normal, and Glorot uniform, as available in the CVNNlib. By default, the CVNNlib sets synaptic weights and biases with a normal distribution.

- **zeros**;
- **ones**;
- **random_normal**;
- **random_uniform**;
- **glorot_normal**; and
- **glorot_uniform**.

The weights and bias initialization are parameters to add layers to the Neural Network.

RosenPy also allows splitting the dataset into training and validation. The function is described as follows:

```
split_set(x, y, train_size=0.7, random_state=None)
```

The parameters are input (**x**) and target (**y**) data, the **size of the training dataset** (**train_size**), and the mode performs a **random split**. This function returns the training set indices for that split (**x_train** and **y_train**) and the testing set indices for that split (**x_test** and **y_test**).

actFunc
It contains the activation functions supported by the framework, which are listed below:
- tanh;
- sinh;
- atanh;
- asinh;
- tan;
- sin;
- atan;
- asin;
- acos; and
- linear.

batchFuncGen
It is especially helpful for large datasets since a considerable amount of memory would be necessary to pass all samples. Then, although the mini-batch is slower to converge, it is faster to execute than the gradient descent batch, which uses all samples simultaneously for network training.

- batch_sequential; and
- batch_shuffle.

The parameters are defined in the **fit()** method.

costFunc

RosenPy uses the mean squared error (MSE) cost function for all implemented CVNNs.

decayFunc
Three methods were implemented in RosenPy: time-based, exponential, and staircase. In the time-based method, the learning rate linearly decays with time, i.e., with the number of epochs. The learning rate exponentially decays in the exponential and staircase methods, but the staircase indicates the number of epochs in which the decay will occur.

- none_decay;
- time_based_decay;
- exponential_decay; and
- staircase.

The parameters for these methods are defined in the class initialization.

Example

XOR Complex

```
import rosenpymodel.cvffnn as cvff
from rp_utils import costFunc, actFunc, initFunc,
                     regFunc, decayFunc, batchGenFunc
import numpy as np

x = np.array([[-1.0-1.0j], [-1.0+1.0j], [1.0-1.0j], [1.0+1.0j]])
y = np.array([1, 0, 1+1.0j, 1.0j]).reshape(-1, 1)

nn = cvff.CVFFNN(cost_func=costFunc.mse,  learning_rate = 1e-2)
nn.addLayer(ishape=x.shape[1], neurons=2,
            weights_initializer=initFunc.random_normal,
            bias_initializer=initFunc.random_normal,
            activation=actFunc.tanh))

nn.addLayer(neurons=y.shape[1],
            weights_initializer=initFunc.random_normal,
            bias_initializer=initFunc.random_normal,
            activation=actFunc.tanh))

nn.fit(x, y, epochs=1000, verbose=100)

y_pred = nn.predict(x)

print(y_pred)
print(y)

print('Accuracy: {:.2f}%'.format((100*(1-np.mean(np.abs((y-y_pred)))))))
```