nimblenet Documentation

Release 0.2

Jørgen Grimnes

August 24, 2016

Contents

nimblenet is a lightweight and efficient Numpy library for creating feed forward neural networks. The library was developed with PYPY in mind and should play nicely with their super-fast JIT compiler. The networks can be trained by a variety of learning algorithms: backpropagation, resilient backpropagation, adaptive learning rate backpropagation, scaled conjugate gradient and SciPy's optimize function.

This is a list of handy links to get up and running.

- Getting Started
- Cost Functions
- Activation Functions
- Saving and loading a trained network

CHAPTER 1

Installing

```
$ pip install nimblenet
```

1.1 Dependencies

- Python 2.7
- NumPy
- SciPy (optional). This is of course a required depedency if you intend to train the network using SciPy's `optimize` function.

CHAPTER 2

Content

2.1 Getting Started

This guide will walk you through how to install nimblenet and configure a network using the library.

2.1.1 Installing

```
$ pip install nimblenet
```

Required dependencies

- Python 2.7
- NumPy

Optional dependencies

- SciPy

In order to speed up the code when using the Sigmoid activation functions, the SciPy package should also be installed. This is an optional dependency, but it is of course required if you intend to train the network using SciPy's `optimize` function.

2.1.2 Creating a Network

Once nimblenet has been installed, initializing a network is simple. The following example creates a two layered network that require two input signals.

```
from nimblenet.activation_functions import sigmoid_function
from nimblenet.neuralnet import NeuralNet

settings            = {
    "n_inputs"      : 2,
    "layers"        : [  (3, sigmoid_function), (1, sigmoid_function) ]
}

network             = NeuralNet( settings )
```

The `layers` parameter describe the topology of the network. The first tuple state that the hidden layer should have three neurons and apply the sigmoid activation function. The final tuple in the `layers` list *always* describe the number of output signals. A list of built-in activations functions are listed in Activation Functions.

Important: The final tuple in the layers list always describe the number of output signals.

The properties specified in the settings parameter are *required*. The initialization of a network is further customizable, please refer to the page Initializing a Network.

2.1.3 Training the Network

The network can be trained by a wide range of learning functions. In this quick intro, we will see how use RMSprop to fit the network to some training data.

First off, we need some dataset to fit the network to. In this guide, we will teach the network XOR. In nimblenet, a dataset is a list of *Instances*.

```
from nimblenet.data_structures import Instance
dataset = [
    # Instance( [inputs], [outputs] )
    Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance( [1,1], [0] )
]
```

The dataset above consist of four training instances with two input signals and one output signal. In general we would split the dataset into a training set and a test set, but for the XOR problem we simply specify the training and test set to be identical:

```
training_set = dataset
test_set = dataset
```

The nimblenet library also offers a selection of preprocessors to manipluate the data and make training more efficient. The preprocessors are not used in this guide, please refer to Preprocessing instead.

Before fitting the network to some training data, we need to decide which cost function we would like to optimize. There are a few cost functions already implemented in this library, and this guide will use the *Cross Entropy* cost function. However, it is easy to implement your own custom cost functions. Please refer to Cost Functions.

```
from nimblenet.cost_functions import cross_entropy_cost
cost_function = cross_entropy_cost
```

Now that we've specified a cost function, we can use RSMprop to train our network:

```
from nimblenet.learning_algorithms import *
RMSprop(
        network,                            # the network to train
        training_set,                      # specify the training set
```

```
        test_set,                             # specify the test set
        cost_function,                         # specify the cost function to calculate error

        ERROR_LIMIT             = 1e-2,        # define an acceptable error limit
        #max_iterations         = 100,         # continues until the error limit is reach if this argume
    )
```

If the training shows poor progression, you may try to gradient check the network to verify that the numerical and the analytical gradient are similar. If the gradient check fails, the math might be wrong. Refer to gradient checking here: Gradient Checking.

2.1.4 Using the Network

After the training has completed, we can verify the training by forward propagating some input data in the network. Since the network is written using matrices, we can forward propagate multiple input instances at once. In contrast to the instances generated when training the network, these instance will only be created with a single parameter (the input signal). The following code tests the output of two instances:

```
prediction_set = [ Instance([0,1]), Instance([1,0]) ]
print network.predict( prediction_set )
>> [[ 0.99735413]
    [ 0.99735378]]
```

The prediction method returns a 2D NumPy array with shape `[n_samples, n_outputs]`. The first dimension of the list contain the outputs from the corresponing Instance.

2.1.5 Putting it all together

```
from nimblenet.activation_functions import sigmoid_function
from nimblenet.cost_functions import cross_entropy_cost
from nimblenet.learning_algorithms import RMSprop
from nimblenet.data_structures import Instance
from nimblenet.neuralnet import NeuralNet


dataset         = [
    Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance( [1,1], [0] )
]

settings        = {
    "n_inputs" : 2,
    "layers"   : [  (5, sigmoid_function), (1, sigmoid_function) ]
}

network         = NeuralNet( settings )
training_set    = dataset
test_set        = dataset
cost_function   = cross_entropy_cost


RMSprop(
        network,                               # the network to train
        training_set,                         # specify the training set
        test_set,                             # specify the test set
        cost_function,                         # specify the cost function to calculate error
```

```
        ERROR_LIMIT             = 1e-2,     # define an acceptable error limit
        #max_iterations         = 100,      # continues until the error limit is reach if this argume
    )
```

2.2 Initializing a Network

In nimblenet, a neural network is configured according to a dict of parameters specified upon initialization.

```
from nimblenet.neuralnet import NeuralNet
network = NeuralNet({
    "n_inputs" : 2,
    "layers"   : [ (1, sigmoid_function) ],
})
```

Important: The final tuple in the layers list always describe the number of output signals.

2.2.1 Parameters

The two dict keys `n_inputs` and `layers` are required. However, the network is further customizable through specifying any of the following dict parameters:

- `n_inputs` the number of input signals
- `layers` the topology of the network
- `initial_bias_value` the input signal from the bias node will be initialized to this value
- `weights_low` the lower bound on weight value during the random initialization
- `weights_high` the upper bound on weight value during the random initialization

2.2.2 Example

```
from nimblenet.neuralnet import NeuralNet
settings            = {
    # Required settings
    "n_inputs"              : 2,        # Number of network input signals
    "layers"                : [  (3, sigmoid_function), (1, sigmoid_function) ],
                                        # [ (number_of_neurons, activation_function) ]
                                        # The last pair in the list dictate the number of output sig

    # Optional settings
    "initial_bias_value"    : 0.0,
    "weights_low"           : -0.1,     # Lower bound on the initial weight value
    "weights_high"          : 0.1,      # Upper bound on the initial weight value
}
network = NeuralNet( settings )
```

2.3 Saving and loading a trained network

2.3.1 Save

A network can be easily saved to a file:

```
from nimblenet.neuralnet import NeuralNet
# Create a network
network = NeuralNet({
    "n_inputs" : 2,
    "layers"   : [ (1, sigmoid_function) ],
})
# Save the network to disk
network.save_network_to_file( "%s.pkl" % "filename" )
```

In addition to doing this explicitly, all of the learning algorithms also offer the possibility to save the network after the training has completed. This is done by passing the named parameter `save_trained_network = True` when calling the learning function:

```
RMSprop( ..., save_trained_network = False ) # omitted parameters for readability
```

This will promt the user whether to save the network or not, upon completion of the training.

2.3.2 Load

If you have saved a network to a file, you can easily load the network back up by calling:

```
from nimblenet.neuralnet import NeuralNet
network = NeuralNet.load_network_from_file( "%s.pkl" % "filename" )
```

2.4 Preprocessing

This is a work in progress. However, the library has already implementations a few of the most used preprocessing techniques.

A preprocessor can be constructed by combining any number of these techniques, and is intended allow maximum configurability.

2.4.1 Usage

First, we need to import `construct_preprocessor`. This will take care of combining our preprocessors:

```
from nimblenet.preprocessing import construct_preprocessor
```

Next, we import the preprocessors we'd like to apply:

```
from nimblenet.preprocessing import replace_nan, standarize
```

Then, we combine the preprocessors. This is done by sending a list of preprocessors in addition to the dataset which we would like to fit the preprocessors againts. Note: this dataset should be the combined set of training, test and validation data.

```
preprocess = construct_preprocessor( dataset, [
                    ( replace_nan, {"replace_with": 0 }),
                    standardize
               ])
```

This constructed preprocessor can now be applied to your datasets. Let's take a look at how we can apply this to the XOR dataset:

```
from nimblenet.data_structures import Instance
dataset                 = [ Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Inst
preprocess              = construct_preprocessor( dataset, [
                                  ( replace_nan, {"replace_with": 0 }),
                                  standardize
                             ])
preprocessed_dataset = preprocess( dataset )
```

Remember that if using a preprocessor before training the network, you will have to use the same preprocessor before using the network to predict based on new input signals.

Important: The dataset given to `construct_preprocessor` should be the combined set of training, test and validation data.

2.4.2 Available preprocessors

Standardize

```
from nimblenet.preprocessing import standarize
```

Has no parameters.

Replace *NaN*

```
from nimblenet.preprocessing import replace_nan
```

Takes an optional parameter `replace_with`. By default, it replaces *NaN* with the mean of the given input signal.

In order to replace *NaN* with zero:

```
from nimblenet.preprocessing import construct_preprocessor, replace_nan
from nimblenet.data_structures import Instance

dataset    = [ Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance( [1,1
preprocess = construct_preprocessor( dataset, [
                    ( replace_nan, {"replace_with": 0 }),
               ])
```

Subtract Mean

```
from nimblenet.preprocessing import subtract_mean
```

Has no parameters.

Normalize

```
from nimblenet.preprocessing import normalize
```

Has no parameters.

Whiten

```
from nimblenet.preprocessing import whiten
```

Takes an optional parameter `epsilon`. By default, epsilon equals `1e-5`.

In order to redefine epsilon to e.g 0.5:

```
from nimblenet.preprocessing import construct_preprocessor, whiten
from nimblenet.data_structures import Instance

dataset    = [ Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance( [1,1
preprocess = construct_preprocessor( dataset, [
                ( whiten, {"epsilon": 0.5 }),
            ])
```

2.5 Gradient Checking

Gradient checking great method for debugging neural networks. The main challenge with implementing these networks, is to get the gradient calculations correct. To verify the analytically computed gradients that are used during gradient descent, we can compare these gradients to numerically calculated gradients. This is called gradient checking.

Warning: Gradient checking against a large dataset is *very* slow.

Important: If the gradient check fails, it will query the user whether to abort or continue executing the script.

2.5.1 Usage

Checking the gradient of a network requires both a *dataset* and a specific *cost function*.

```
network = NeuralNet( ... ) # parameters omitted for readability
network.check_gradient( dataset, cost_function )
```

The following code snippet is a complete example on how to perform gradient checking:

```
from nimblenet.activation_functions import binary_cross_entropy_cost
from nimblenet.cost_functions import cross_entropy_cost
from nimblenet.data_structures import Instance
from nimblenet.neuralnet import NeuralNet

cost_function  = binary_cross_entropy_cost
dataset        = [ Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance(
settings       = {
    "n_inputs" : 2,
    "layers"   : [  (2, sigmoid_function), (1, sigmoid_function) ]
}

network        = NeuralNet( settings )
network.check_gradient( dataset, cost_function )
```

2.6 Activation Functions

The some of the most popular activation functions has already been implemented in nimblenet. However, it is very easy to specify your own activation function as described in *Arbitrary Activation Functions*.

2.6.1 Usage

Using the various activation functions is as easy as importing the desired activation function and using it when declaring the network topology. Below is an example of how to use the Sigmoid activation function in a simple neural network.

```
from nimblenet.activation_functions import sigmoid_function
from nimblenet.neuralnet import NeuralNet

settings       = {
    "n_inputs" : 2,                          # Two input signals

    # Using the sigmoid activation function in a layer
    "layers"   : [  (1, sigmoid_function) ] # A single layer neural network with one output signal
}

network        = NeuralNet( settings )
```

A network may of course use different activation functions at each layer:

```
from nimblenet.activation_functions import sigmoid_function, tanh_function
from nimblenet.neuralnet import NeuralNet

settings        = {
    "n_inputs" : 2,                         # Two input signals

    # Using both tanh and sigmoid activation functions
    "layers"   : [  (2, tanh_function), (1, sigmoid_function) ] # A two layered neural network with
}

network         = NeuralNet( settings )
```

2.6.2 List of cost functions

Sigmoid function

```
from nimblenet.activation_functions import sigmoid_function
```

Tanh function

```
from nimblenet.activation_functions import tanh_function
```

Softmax function

```
from nimblenet.activation_functions import softmax_function
```

Elliot function

The Elliot function is a fast approximation to the Sigmoid activation function.

```
from nimblenet.activation_functions import elliot_function
```

Symmetric Elliot function

The Symmetric Elliot function is a fast approximation to the tanh activation function.

```
from nimblenet.activation_functions import symmetric_elliot_function
```

ReLU function

```
from nimblenet.activation_functions import ReLU_function
```

LReLU function

This is the leaky rectified linear activation function.

```
from nimblenet.activation_functions import LReLU_function
```

Linear function

```
from nimblenet.activation_functions import linear_function
```

Softplus function

```
from nimblenet.activation_functions import softplus_function
```

Softsign function

```
from nimblenet.activation_functions import softsign_function
```

2.6.3 Arbitrary Activation Functions

It is easy to write your own, custom activation functions. A activation function takes the required form:

```
def activation_function( signal, derivative = False ):
    ...
```

The `signal` parameter is a NumPy matrix with shape `[n_samples, n_outputs]`. When the `derivative` flag is true, the activation function is expected to return the partial derivation of the function.

As an example, we can look at how the tanh activation function is implemented:

```
def tanh_function( signal, derivative=False ):
    squashed_signal = np.tanh( signal )

    if derivative:
        return 1 - np.power( squashed_signal, 2 )
    else:
        return squashed_signal
```

How to

Lets define a custom cost function and use it when training the network:

```
from nimblenet.learning_algorithms import backpropagation
from nimblenet.cost_functions import sum_squared_error
from nimblenet.data_structures import Instance
from nimblenet.neuralnet import NeuralNet
import numpy as np

def custom_activation_function( signal, derivative = False ):
    # This activation function amounts to a ReLU layer
    if derivative:
        return (signal > 0).astype(float)
```

```
    else:
        return np.maximum( 0, signal )
#end

dataset         = [ Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance(
settings        = {
    "n_inputs" : 2,

    # This is where we apply our custom activation function:
    "layers"   : [  (2, custom_activation_function) ]
}

network         = NeuralNet( settings )
training_set    = dataset
test_set        = dataset
cost_function   = sum_squared_error

backpropagation(
        network,                # the network to train
        training_set,           # specify the training set
        test_set,               # specify the test set
        cost_function           # specify the cost function to optimize
    )
```

2.7 Learning Algorithms

This library offers a wide range of analytical learning algorithms. These algorithms have been implemented in Python using NumPy and its matrices for efficiency:

- *Backpropagation Variations*
  - *Vanilla Backpropagation*
  - *Classical Momentum*
  - *Nesterov Momentum*
  - *RMSprop*
  - *Adagrad*
  - *Adam*
- *Additional Learning Algorithms*
  - *Resilient Backpropagation*
  - *Scaled Conjugate Gradient*
  - *SciPy's Optimize*

To shorten the code examples given below, the following code snippet is implicitly called before executing the examples:

```
from nimblenet.activation_functions import sigmoid_function
from nimblenet.cost_functions import cross_entropy_cost
from nimblenet.data_structures import Instance
from nimblenet.neuralnet import NeuralNet

dataset         = [
    Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance( [1,1], [1] )
]
```

```
settings        = {
    "n_inputs" : 2,
    "layers"   : [ (1, sigmoid_function) ]
}

network         = NeuralNet( settings )
training_set    = dataset
test_set        = dataset
cost_function   = cross_entropy_cost
```

Important: The *dropout* regularization strategy is applicable to all backpropagation variations and adaptive learning rate methods. The scaled conjugate gradient, SciPy's `optimize`, and resilient backpropagation does not support this regularization.

2.7.1 Backpropagation Variations

These are the common parameters accepted by the following learning algorithms along with their default value:

```
learning_algorithm(
    # Required parameters
    network,                     # the neural network instance to train
    training_set,                # the training dataset
    test_set,                    # the test dataset
    cost_function,               # the cost function to optimize

    # Optional parameters
    ERROR_LIMIT          = 1e-3, # Error tolerance when terminating the learning
    max_iterations       = (),   # Regardless of the achieved error, terminate after max_iterations e
    batch_size           = 0,    # Set the batch size. 0 implies using the entire training_set as a b
    input_layer_dropout  = 0.0,  # Dropout fraction of the input layer
    hidden_layer_dropout = 0.0,  # Dropout fraction of in the hidden layer(s)
    print_rate           = 1000, # The epoch interval to print progression statistics
    save_trained_network = False # Whether to ask the user if they would like to save the network aft
)
```

Vanilla Backpropagation

This example show how to train your network using the vanilla backpropagation algorithm. The optional common parameters has been skipped for brevity, but the algorithm conforms to *common backpropagation variables*.

```
from nimblenet.learning_algorithms import backpropagation
backpropagation(
        # Required parameters
        network,                     # the neural network instance to train
        training_set,                # the training dataset
        test_set,                    # the test dataset
        cost_function,               # the cost function to optimize
    )
```

Classical Momentum

This example show how to train your network using backpropagation with classical momentum. The optional common parameters has been skipped for brevity, but the algorithm conforms to *common backpropagation variables*.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import backpropagation_classical_momentum
backpropagation_classical_momentum(
        # Required parameters
        network,                     # the neural network instance to train
        training_set,                # the training dataset
        test_set,                    # the test dataset
        cost_function,               # the cost function to optimize

        # Classical momentum backpropagation specific, optional parameters
        momentum_factor = 0.9
    )
```

Nesterov Momentum

This example show how to train your network using backpropagation with Nesterov momentum. The optional common parameters has been skipped for brevity, but the algorithm conforms to *common backpropagation variables*.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import backpropagation_nesterov_momentum
backpropagation_nesterov_momentum(
        # Required parameters
        network,                     # the neural network instance to train
        training_set,                # the training dataset
        test_set,                    # the test dataset
        cost_function,               # the cost function to optimize

        # Nesterov momentum backpropagation specific, optional parameters
        momentum_factor = 0.9
    )
```

RMSprop

This example show how to train your network using RMSprop. The optional common parameters has been skipped for brevity, but the algorithm conforms to *common backpropagation variables*.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import RMSprop
RMSprop(
        # Required parameters
        network,                     # the neural network instance to train
        training_set,                # the training dataset
        test_set,                    # the test dataset
        cost_function,               # the cost function to optimize

        # RMSprop specific, optional parameters
        decay_rate   = 0.99,
        epsilon      = 1e-8
    )
```

Adagrad

This example show how to train your network using Adagrad. The optional common parameters has been skipped for brevity, but the algorithm conforms to *common backpropagation variables*.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import adagrad
adagrad(
        # Required parameters
        network,                     # the neural network instance to train
        training_set,                # the training dataset
        test_set,                    # the test dataset
        cost_function,               # the cost function to optimize

        # Adagrad specific, optional parameters
        epsilon     = 1e-8
    )
```

Adam

This example show how to train your network using Adam. The optional common parameters has been skipped for brevity, but the algorithm conforms to *common backpropagation variables*.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import Adam
Adam(
        # Required parameters
        network,                     # the neural network instance to train
        training_set,                # the training dataset
        test_set,                    # the test dataset
        cost_function,               # the cost function to optimize

        # Adam specific, optional parameters
        beta1       = 0.9,
        beta2       = 0.999,
        epsilon     = 1e-8
    )
```

2.7.2 Additional Learning Algorithms

Resilient Backpropagation

This example show how to train your network using resilient backpropagation. This is the iRprop+ variation of resilient backpropagation.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import resilient_backpropagation
resilient_backpropagation(
    # Required parameters
    network,                     # the neural network instance to train
    training_set,                # the training dataset
    test_set,                    # the test dataset
    cost_function,               # the cost function to optimize
```

```
    # Resilient backpropagation specific, optional parameters
    weight_step_max      = 50.,
    weight_step_min      = 0.,
    start_step           = 0.5,
    learn_max            = 1.2,
    learn_min            = 0.5,

    # Optional parameters
    ERROR_LIMIT          = 1e-3, # Error tolerance when terminating the learning
    max_iterations       = (),   # Regardless of the achieved error, terminate after max_iterations e
    print_rate           = 1000, # The epoch interval to print progression statistics
    save_trained_network = False # Whether to ask the user if they would like to save the network aft
)
```

Scaled Conjugate Gradient

This example show how to train your network using scaled conjugate gradient. This algorithm has been implemented according to Scaled Conjugate Gradient for Fast Supervised Learning authored by Martin Møller.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import scaled_conjugate_gradient
scaled_conjugate_gradient(
    # Required parameters
    network,                     # the neural network instance to train
    training_set,                # the training dataset
    test_set,                    # the test dataset
    cost_function,               # the cost function to optimize

    # Optional parameters
    ERROR_LIMIT          = 1e-3, # Error tolerance when terminating the learning
    max_iterations       = (),   # Regardless of the achieved error, terminate after max_iterations e
    print_rate           = 1000, # The epoch interval to print progression statistics
    save_trained_network = False # Whether to ask the user if they would like to save the network aft
)
```

SciPy's Optimize

This example show how to train your network using SciPy's `optimize` function. This learning algorithm requires SciPy to be installed.

Named variables are shown together with their default value.

```
from nimblenet.learning_algorithms import scipyoptimize
scipyoptimize(
    # Required parameters
    network,                     # the neural network instance to train
    training_set,                # the training dataset
    test_set,                    # the test dataset
    cost_function,               # the cost function to optimize

    # SciPy Optimize specific, optional parameters
    method               = "Newton-CG", # The method name correspond to the method names accepted by

    # Optional parameters
```

```
    save_trained_network = False # Whether to ask the user if they would like to save the network aft
)
```

2.8 Cost Functions

A the most popular and applicable cost functions has already been implemented in this library, and are listed below. However, it is very easy to specify your own cost functions as described in *Arbitrary Cost Functions*.

Warning: The Softmax Categorical Cross Entropy cost function is required when using a softmax layer in the network topology.

2.8.1 Usage

Using the various cost functions is as easy as only importing the desired cost function and passing it to the decided learning function. Below is an example of how to use the Cross Entropy cost function when training using the vanilla backpropagation algorithm.

```
from nimblenet.cost_functions import binary_cross_entropy_cost
from nimblenet.activation_functions import sigmoid_function
from nimblenet.learning_algorithms import backpropagation
from nimblenet.data_structures import Instance
from nimblenet.neuralnet import NeuralNet

dataset        = [ Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance(
settings       = {
    "n_inputs" : 2,
    "layers"   : [  (2, sigmoid_function) ]
}

network        = NeuralNet( settings )
training_set   = dataset
test_set       = dataset
cost_function  = binary_cross_entropy_cost

backpropagation(
        network,              # the network to train
        training_set,         # specify the training set
        test_set,             # specify the test set

        # This is where we specify the cost function to optimize:
        cost_function         # specify the cost function to calculate error
    )
```

2.8.2 List of cost functions

Sum Squared Error

```
from nimblenet.cost_functions import sum_squared_error
```

Binary Cross Entropy

```
from nimblenet.cost_functions import binary_cross_entropy_cost
```

Softmax Categorical Cross Entropy

This cost function is **required** when including a softmax layer in your network topology.

```
from nimblenet.cost_functions import softmax_categorical_cross_entropy_cost
```

Hellinger Distance

```
from nimblenet.cost_functions import hellinger_distance
```

2.8.3 Arbitrary Cost Functions

It is easy to optimize your own, custom cost functions. A cost function has the required form:

```
def custom_cost_function(
            outputs,            # the signal emitted from the network
            targets,            # the target values we would like the network to output
            derivative = False  # whether the cost function should return its derivative
        ):
    ...
```

The `outputs` and `targets` parameters are NumPy matrices with shape `[n_samples, n_outputs]`.

As an example, we can look at how the Sum Squared Error function is implemented:

```
def sum_squared_error( outputs, targets, derivative = False ):
    if derivative:
        return outputs - targets
    else:
        return 0.5 * np.mean(np.sum( np.power(outputs - targets,2), axis = 1 ))
```

Important: Observe that we calculate the mean of the error, per singal, across the input instances fed into the network. This detail is important to remember in order to get the derivatives correct.

How to

Lets define a custom cost function and use it when training the network:

```
from nimblenet.activation_functions import sigmoid_function
from nimblenet.learning_algorithms import backpropagation
from nimblenet.data_structures import Instance
from nimblenet.neuralnet import NeuralNet
import numpy as np

def custom_cost_function( outputs, targets, derivative = False ):
    if derivative:
        return outputs - targets
    else:
        return 0.5 * np.mean(np.sum( np.power(outputs - targets,2), axis = 1 ))
#end

dataset         = [ Instance( [0,0], [0] ), Instance( [1,0], [1] ), Instance( [0,1], [1] ), Instance(
settings        = {
    "n_inputs" : 2,
    "layers"   : [  (2, sigmoid_function) ]
}

network         = NeuralNet( settings )
training_set    = dataset
test_set        = dataset
cost_function   = custom_cost_function

backpropagation(
        network,                # the network to train
        training_set,           # specify the training set
        test_set,               # specify the test set

        # This is where we specify the cost function to optimize:
        cost_function           # specify the cost function to calculate error
    )
```

2.9 Using the Network

Nimblenet is implemented using matrices rather than for-loops. This allow more efficient computation, and also enables the network to forward propagate multiple input instances at once.

In contrast to the instances generated when training the network:

```
from nimblenet.data_structures import Instance
dataset = [
    # Instance( [inputs], [outputs] )
    Instance( [0,0], [0] ), ...
]
```

the instances used during prediction need only to be instantiated with a single parameter (the input signal):

```
from nimblenet.data_structures import Instance
dataset = [
    # Instance( [inputs] )
    Instance( [0,0] ), ...
]
```

The following code calculates the output from two instances:

```
prediction_set = [ Instance([0,1]), Instance([1,0]) ]
print network.predict( prediction_set )
>> [[ 0.99735413]
    [ 0.99735378]]
```

The prediction method returns a 2D NumPy array with shape `[n_samples, n_outputs]`. That means each row in the output matrix correspond to an input Instance. The first row of the output matrix, is the output generated from the first instance. Refer to the the expected output below:

```
prediction_set = [ Instance([0,1]) ]
print network.predict( prediction_set )
>> [[ 0.99735413]]
```

2.10 Support

Have you spotted a bug, or run into inconsistencies in the documentation? Please report the issue at Github.

Support

Have you spotted a bug, or run into inconsistencies in the documentation? Please report the issue at Github.