Multi Layer Perceptron Implementation

Generated by Doxygen 1.8.13

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Namespace Documentation

3.1 NeuralNetwork Namespace Reference

Classes

- class NeuralNetwork

Functions

- def CrossEntropy (y_hat, y)
- def CrossEntropyGradient (y_hat, y)
- def Sigmoid (x)
- def SigmoidGradient (x)
- def to_categorical (y, num_classes=None)

Variables

- list **Costs** = [ ]
- **data** = pd.read_csv('mnist.txt', sep=" ", header=None)
- **X** = data[range(len(data.columns)-1)].as_matrix()
- **y** = data[[len(data.columns)-1]].as_matrix()
- **model** = NeuralNetwork(g=Sigmoid, dg=SigmoidGradient, L=CrossEntropy, dL=CrossEntropyGradient)
- **max_it**
- int **n** = 100
- int **max_images** = 3
- int **acc** = 0.
- int **images_plotted** = 0
- **index** = np.random.randint(len(X))
- **pred** = np.argmax(model.predict(X[index]))
- **target** = np.argmax(y[index])
- **img** = X[index].reshape((28,28))
- string **title** = target, ' missclassified as ',pred
- **interpolation**

3.1.1 Detailed Description

```
Authors:
 Carlos Brito (carlos.brito524@gmail.com)
 Laura Alonzo ()

This is a small implementation of a Fully Connected Multi
Layer Perceptron. Currently, it’s set to train over the
MNIST dataset.

To run make sure you have mnist.txt in the same folder.
After which you only need to run it with

~> $ python NeuralNetwork.py


Notes:
    - It is being trained using the whole of the database
    - We are training the method with the Conjugate Gradient Method
    - The cost function is called Cross Entropy
    - >>>> It will plot the missclassified images <<<< PLEASE
    - It will plot a graph of the cost function as time passes
```

3.1.2 Function Documentation

3.1.2.1 CrossEntropy()

```
def NeuralNetwork.CrossEntropy (
              y_hat,
              y )
```

```
Consult Wikipedia for the brief explanation of
cross entropy loss. It has a nice example for
logistic regression which is basically what w
are trying to achieve.

https://en.wikipedia.org/wiki/Cross_entropy
```

3.1.2.2 CrossEntropyGradient()

```
def NeuralNetwork.CrossEntropyGradient (
              y_hat,
              y )
```

```
Gradient of Cross Entropy with respect to
the weights.
```

3.1.2.3 Sigmoid()

```
def NeuralNetwork.Sigmoid (
              x )
```

Sigmoid function.

3.1.2.4 SigmoidGradient()

```
def NeuralNetwork.SigmoidGradient (
              x )
```

Gradient of sigmoid function.

3.1.2.5 to_categorical()

```
def NeuralNetwork.to_categorical (
              y,
              num_classes = None )
```

```
Converts a class vector (integers) to binary class matrix.
E.g. for use with categorical_crossentropy.
# Arguments
    y: class vector to be converted into a matrix
        (integers from 0 to num_classes).
    num_classes: total number of classes.
# Returns
    A binary matrix representation of the input.
```

Chapter 4

Class Documentation

4.1 NeuralNetwork.NeuralNetwork Class Reference

Public Member Functions

- def __init__ (self, g, dg, L, dL, reg=1)
- def addLayer (self, size)
- def train (self, X, y, max_it=200, epsilon_init=0.5)
- def predict (self, x)
- def init_weights (self, epsilon)
- def forward (self, X, y)
- def back_prop (self, y_hat)
- def objective (self, params, X, y)
- def callbackF (self, params)
- def getWeightGradients (self, X, y)
- def getParams (self)
- def setParams (self, params)

Public Attributes

- layers

  number of layers
- activation

  activation function
- activation_gradient

  activation gradient
- loss

  loss model
- loss_gradient

  gradient of loss
- layer_sizes

  number of nodes per each layer
- epoch

  current epoch
- reg_param

  reg param

- X

  training data
- y

  targets
- W

  weight tensor
- a

  list of vectors for linear combination at each layer
- h

  activation vectors of each layer
- weight_gradient

  list of gradients of the weigths at each layer
- **y_hat**

4.1.1 Constructor & Destructor Documentation

4.1.1.1 __init__()

```
def NeuralNetwork.NeuralNetwork.__init__ (
              self,
              g,
              dg,
              L,
              dL,
              reg = 1 )
```

```
@var l - number of layers
@var g - activation
@var dg - derivative of activation
@var L - loss
@var dL - gradient of loss
```

4.1.2 Member Function Documentation

4.1.2.1 addLayer()

```
def NeuralNetwork.NeuralNetwork.addLayer (
              self,
              size )
```

```
@param size is the size of the layer i.e.
if we want a layer of 3 nodes then we
call model.addLayer(3)
```

4.1.2.2 back_prop()

```
def NeuralNetwork.NeuralNetwork.back_prop (
              self,
              y_hat )
```

```
Algorithm to calculate the gradients of the network
Taken from Deep Learning by Ian Goodfellow, Yoshua Bengio
and Aaron Courville.

This method saves the gradients of the weights in self.weight_gradient
so you can use them for later in any method such as SGD or Batch Training

y_hat - The prediction for an input X. We only need this value to get
the error at the last layer.
```

4.1.2.3 callbackF()

```
def NeuralNetwork.NeuralNetwork.callbackF (
              self,
              params )
```

```
Helper function for the optimizer to do something at each iteration

We use this to shuffle the training set at each epoch and do some
other interactivity stuff.
```

4.1.2.4 forward()

```
def NeuralNetwork.NeuralNetwork.forward (
              self,
              X,
              y )
```

```
Forward propagation of the network
X - input matrix
y - output vector
```

4.1.2.5 getParams()

```
def NeuralNetwork.NeuralNetwork.getParams (
              self )
```

```
Get all the weights concatenated into one single 1 dimensional vector
```

4.1.2.6 getWeightGradients()

```
def NeuralNetwork.NeuralNetwork.getWeightGradients (
              self,
              X,
              y )
```

```
Concatenates and returns all the calculated weight gradients
```

4.1.2.7 init_weights()

```
def NeuralNetwork.NeuralNetwork.init_weights (
              self,
              epsilon )
```

```
initializes the weights randomly using
the shapes of adjacent layers
```

4.1.2.8 objective()

```
def NeuralNetwork.NeuralNetwork.objective (
              self,
              params,
              X,
              y )
```

```
Objective function that acts as a way of obtaining
the cost of the network on input X and gets the gradients.

params - basically all the weights concatenated into a vector
X - input
y - targets
```

4.1.2.9 predict()

```
def NeuralNetwork.NeuralNetwork.predict (
              self,
              x )
```

```
Predict a new input x by forward
propagating the network
```

4.1.2.10 setParams()

```
def NeuralNetwork.NeuralNetwork.setParams (
              self,
              params )
```

```
Reconstruct the parameters given by a unidimensional vector.
Note we use the fact that we know the size of each layer
```

4.1.2.11 train()

```
def NeuralNetwork.NeuralNetwork.train (
              self,
              X,
              y,
              max_it = 200,
              epsilon_init = 0.5 )
```

```
Trains the network using an optimizer.
Please select the method to be used in the call to optimize.minimize(method=YourChoice)
Also feel free to adjust the weight initialization parameter.
```

The documentation for this class was generated from the following file:

- NeuralNetwork.py

Index