DeepLearningPython35

Daniel Engbert, Michael Nielsen, Michal Dobrzanski

Dec 12, 2021

CONTENTS:

CHAPTER

ONE

DEEPLEARNINGPYTHON35

1.1 experiments package

1.1.1 Submodules

1.1.2 experiments.GroupsNet module

mynetExperiment.py

inherits from mynet.py, but tests the possible allowing a layer l to be partitioned into g groups. The following layer (l+1) is also paritioned into g groups, and only nodes within the same corresponding groups will have connections between them across the two layers. e.g. the first 10 nodes of layer 1, will only have connections to the first 7 nodes of layer 2. we can accomplish this easily by zeroing out (permanently) the majority of the weights between the layers, and zeroing out nabla_w entries for these zero'd weights (see self.masks).

class `experiments.GroupsNet.`**`Network`**(*sizes*)

Bases: `mynet.Network`

overload mynet.Network class

`__init__`(*sizes*)

The list `sizes` contains the number of neurons in the respective layers of the network. For example, if the list was [2, 3, 1] then it would be a three-layer network, with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron. The biases and weights for the network are initialized randomly, using a Gaussian distribution with mean 0, and variance 1. Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.

Parameters **sizes** (`list` of `int`) – list of layer sizes (e.g. *[2,3,2]*)

sizes

list of layer sizes (e.g. *[2,3,2]*)

Type `list` of `int`

num_layers

number of layers in network

Type int

biases

stores a column vector for each layer's biases (except the input layer). Initialized randomly using np.random.randn (for a gaussian distribution) example for sizes *[2,3,2]*:

$$[\begin{bmatrix}[-0.25]\\ [-0.34]\\ [0.21]\end{bmatrix}, \begin{bmatrix}[-0.88]\\ [0.47]\end{bmatrix}]$$

Type `list` of `numpy.ndarray`

`weights`

list of weights in the network (one np array for each layer)

for a given layer, col 0 = vector of weights for connections from node 0 (in cur layer) to next layer's nodes ample for sizes *[2,3,2]*:

$$[\begin{bmatrix}[-0.3, -0.01] \\ [2.94, -0.88] \\ [0.91, -1.85]\end{bmatrix}, \begin{bmatrix}[1.62, 1.21, 0.95] \\ [-0.19, -0.68, 0.61]\end{bmatrix}]$$

let $w = weights[0]$ (stores all weights coming into layer 1 from layer 0).

$w[j]$ is the list of weights coming into node j in layer 1, so $w_{jk} = w[j][k]$ is the weight between the jth neuron in layer 1, and the kth neuron in layer 0.

This ordering allows us to calculate activations with: $a' = \sigma(wa + b)$

Type `list` of `numpy.ndarray`

`updateMiniBatch`(*miniBatch*, *rate*)

Perform gradient descent using backpropogation on a single miniBatch and update the network's weights and biases.

Parameters

- **`miniBatch`** (`list` of `tuples`) – List of training inputs / expected outputs (x,y). Calculated changes to the weights and biases will be averaged across this mini batch.

  Note: to train on a single sample, set mini_batch=[(x,y)]
- **`rate`** (`float`) – Learning rate to use for training (e.g. 0.05).

1.1.3 experiments.bootyNet module

1.1.4 experiments.costPrediction module

costPrediction.py

Modified version of mynet.py to test having the network also predict its own cost TODO: use inheritance like mynetExperiment.py does…

class `experiments.costPrediction.`**`Network`**(*sizes*)

Bases: `object`

`__init__`(*sizes*)

The list `sizes` contains the number of neurons in the respective layers of the network. For example, if the list was [2, 3, 1] then it would be a three-layer network, with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron. The biases and weights for the network are initialized randomly, using a Gaussian distribution with mean 0, and variance 1. Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.

Parameters **`sizes`** (`list` of `int`) – list of layer sizes (e.g. *[2,3,2]*)

`sizes`

list of layer sizes (e.g. *[2,3,2]*)

Type `list` of `int`

num_layers
number of layers in network

Type int

biases
stores a column vector for each layer's biases (except the input layer). Initialized randomly using np.random.randn (for a gaussian distribution) example for sizes *[2,3,2]*:

$$[\begin{bmatrix}[-0.25]\\ [-0.34]\\ [0.21]\end{bmatrix}, \begin{bmatrix}[-0.88]\\ [0.47]\end{bmatrix}]$$

Type `list` of `numpy.ndarray`

weights
list of weights in the network (one np array for each layer)

for a given layer, col 0 = vector of weights for connections from node 0 (in cur layer) to next layer's nodes ample for sizes *[2,3,2]*:

$$[\begin{bmatrix}[-0.3, -0.01]\\ [2.94, -0.88]\\ [0.91, -1.85]\end{bmatrix}, \begin{bmatrix}[1.62, 1.21, 0.95]\\ [-0.19, -0.68, 0.61]\end{bmatrix}]$$

let $w = weights[0]$ (stores all weights coming into layer 1 from layer 0).

$w[j]$ is the list of weights coming into node j in layer 1, so $w_jk = w[j][k]$ is the weight between the jth neuron in layer 1, and the kth neuron in layer 0.

This ordering allows us to calculate activations with: $a' = \sigma(wa + b)$

Type `list` of `numpy.ndarray`

save (*filename*)
pickle data in this class as a backup to desired filename https://stackoverflow.com/a/2842727

load (*filename*)
load data from pickle file into this class to overwrite its data Incomplete: (returns a Network object)

test (*test_data*)
Return the number of test inputs for which the neural network outputs the correct result. Note that the neural network's output is assumed to be the index of whichever neuron in the final layer has the highest activation.

getOutput (*x*)
returns the output layers activations for given input (using feedforward())

feedforward (*a*)
returns list of z values and list of activations at each layer for given input

updateMiniBatch (*miniBatch, rate*)
Perform gradient descent using backpropogation on a single miniBatch and update the network's weights and biases.

Parameters

- **miniBatch** (`list` of `tuples`) – List of training inputs / expected outputs (x,y). Calculated changes to the weights and biases will be averaged across this mini batch.

  Note: to train on a single sample, set mini_batch=[(x,y)]

- **rate** (`float`) – Learning rate to use for training (e.g. 0.05).

SGD(*training_data*, *epochs*, *mini_batch_size*, *rate*, *test_data=None*, *start_epoch=1*)
Train the neural network using mini-batch stochastic gradient descent. The `training_data` is a list of tuples `(x, y)` representing the training inputs and the desired outputs. The other non-optional parameters are self-explanatory. If `test_data` is provided then the network will be evaluated against the test data after each epoch, and partial progress printed out. This is useful for tracking progress, but slows things down substantially.

backprop(*x*, *y*, *printCost=False*)
do backpropagation returns nabla_b, nabla_w (partial dertivatives of biases and weights wrt. cost function)

cost_derivative(*output_activations*, *y_initial*)
Return the vector of partial derivatives $\partial C_x/\partial a$ for the output activations. (For the quadratic cost function C in particular)

Parameters

- **output_activations** – vector of output activations
- **y_initial** – expected output vector (before the quadratic cost is appended)

quadratic_cost(*output_activations*, *y*)
Returns the quadratic cost for the output activations. :param output_activations: vector of output activations :param y: expected output vector

sigmoid(*z*)
Implements the sigmoid function. https://en.wikipedia.org/wiki/Sigmoid_function

sigmoid_prime(*z*)
Derivative of the sigmoid function.

1.1.5 experiments.runGroupsNet module

1.1.6 Module contents

1.2 mnist package

1.2.1 Submodules

1.2.2 mnist.expand_mnist module

expand_mnist.py

Take the 50,000 MNIST training images, and create an expanded set of 250,000 images, by displacing each training image up, down, left and right, by one pixel. Save the resulting file to ../data/mnist_expanded.pkl.gz.

Note that this program is memory intensive, and may not run on small systems.

`mnist.expand_mnist.`**main**()

1.2.3 mnist.mnist_average_darkness module

1.2.4 mnist.mnist_loader module

mnist_loader

A library to load the MNIST image data. For details of the data structures that are returned, see the doc strings for `load_data` and `load_data_wrapper`. In practice, `load_data_wrapper` is the function usually called by our neural network code.

`mnist.mnist_loader.`**`load_data`**`()`

Return the MNIST data as a tuple containing the training data, the validation data, and the test data. The `training_data` is returned as a tuple with two entries. The first entry contains the actual training images. This is a numpy ndarray with 50,000 entries. Each entry is, in turn, a numpy ndarray with 784 values, representing the 28 * 28 = 784 pixels in a single MNIST image. The second entry in the `training_data` tuple is a numpy ndarray containing 50,000 entries. Those entries are just the digit values (0…9) for the corresponding images contained in the first entry of the tuple. The `validation_data` and `test_data` are similar, except each contains only 10,000 images. This is a nice data format, but for use in neural networks it's helpful to modify the format of the `training_data` a little. That's done in the wrapper function `load_data_wrapper()`, see below.

`mnist.mnist_loader.`**`load_data_wrapper`**`()`

Return a tuple containing `(training_data, validation_data, test_data)`. Based on `load_data`, but the format is more convenient for use in our implementation of neural networks. In particular, `training_data` is a list containing 50,000 2-tuples `(x, y)`. `x` is a 784-dimensional numpy.ndarray containing the input image. `y` is a 10-dimensional numpy.ndarray representing the unit vector corresponding to the correct digit for `x`. `validation_data` and `test_data` are lists containing 10,000 2-tuples `(x, y)`. In each case, `x` is a 784-dimensional numpy.ndarry containing the input image, and `y` is the corresponding classification, i.e., the digit values (integers) corresponding to `x`. Obviously, this means we're using slightly different formats for the training data and the validation / test data. These formats turn out to be the most convenient for use in our neural network code.

`mnist.mnist_loader.`**`vectorized_result`**`(`*j*`)`

Return a 10-dimensional unit vector with a 1.0 in the jth position and zeroes elsewhere. This is used to convert a digit (0…9) into a corresponding desired output from the neural network.

`mnist.mnist_loader.`**`vectorToImage`**`(`*v*, *fname=''*`)`

1.2.5 mnist.mnist_svm module

1.2.6 Module contents

1.3 mynet module

1.3.1 mynet.py

A module to implement the stochastic gradient descent learning algorithm for a feedforward neural network.

More info: http://neuralnetworksanddeeplearning.com/chap1.html http://neuralnetworksanddeeplearning.com/chap2.html

`class` `mynet.`**`Network`**`(`*sizes*`)`

Bases: `object`

`__init__`(*sizes*)

The list `sizes` contains the number of neurons in the respective layers of the network. For example, if the list was [2, 3, 1] then it would be a three-layer network, with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron. The biases and weights for the network are initialized randomly, using a Gaussian distribution with mean 0, and variance 1. Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.

Parameters **`sizes`** (`list` of `int`) – list of layer sizes (e.g. *[2,3,2]*)

`sizes`

list of layer sizes (e.g. *[2,3,2]*)

Type `list` of `int`

`num_layers`

number of layers in network

Type int

`biases`

stores a column vector for each layer's biases (except the input layer). Initialized randomly using np.random.randn (for a gaussian distribution) example for sizes *[2,3,2]*:

$$[\begin{bmatrix}[-0.25]\\ [-0.34]\\ [0.21]\end{bmatrix}, \begin{bmatrix}[-0.88]\\ [0.47]\end{bmatrix}]$$

Type `list` of `numpy.ndarray`

`weights`

list of weights in the network (one np array for each layer)

for a given layer, col 0 = vector of weights for connections from node 0 (in cur layer) to next layer's nodes ample for sizes *[2,3,2]*:

$$[\begin{bmatrix}[-0.3, -0.01]\\ [2.94, -0.88]\\ [0.91, -1.85]\end{bmatrix}, \begin{bmatrix}[1.62, 1.21, 0.95]\\ [-0.19, -0.68, 0.61]\end{bmatrix}]$$

let $w = weights[0]$ (stores all weights coming into layer 1 from layer 0).

$w[j]$ is the list of weights coming into node j in layer 1, so $w_{jk} = w[j][k]$ is the weight between the jth neuron in layer 1, and the kth neuron in layer 0.

This ordering allows us to calculate activations with: $a' = \sigma(wa + b)$

Type `list` of `numpy.ndarray`

`save`(*filename*)

pickle data in this class as a backup to desired filename https://stackoverflow.com/a/2842727

`load`(*filename*)

load data from pickle file into this class to overwrite its data Incomplete: (returns a Network object)

`test`(*test_data*)

Return the number of test inputs for which the neural network outputs the correct result. Note that the neural network's output is assumed to be the index of whichever neuron in the final layer has the highest activation.

`testQuadraticCost`(*test_data*)

Return the quadratic cost of the provided test_data. Where test_data is a list of 2-tuples, the first element being an input vector to the network, and the second as the ground-truth output vector.

getOutput (*x*)
returns the output layers activations for given input (using feedforward())

feedforward (*a*)
returns list of z values and list of activations at each layer for given input

updateMiniBatch (*miniBatch*, *rate*)
Perform gradient descent using backpropogation on a single miniBatch and update the network's weights and biases.

Parameters

- **miniBatch** (`list` of `tuples`) – List of training inputs / expected outputs (x,y). Calculated changes to the weights and biases will be averaged across this mini batch.

  Note: to train on a single sample, set mini_batch=[(x,y)]

- **rate** (`float`) – Learning rate to use for training (e.g. 0.05).

SGD (*training_data*, *epochs*, *mini_batch_size*, *rate*, *test_data=None*, *start_epoch=1*)
Train the neural network using mini-batch stochastic gradient descent. The `training_data` is a list of tuples `(x, y)` representing the training inputs (a vector) and the desired outputs (as an integer index, OR as a vector). The other non-optional parameters are self-explanatory. If `test_data` is provided then the network will be evaluated against the test data after each epoch, and partial progress printed out. This is useful for tracking progress, but slows things down substantially.

backprop (*x*, *y*)
do backpropagation returns nabla_b, nabla_w (partial dertivatives of biases and weights wrt. cost function)

cost_derivative (*output_activations*, *y*)
Return the vector of partial derivatives $\partial C_x / \partial a$ for the output activations. (For the quadratic cost function C in particular)

Parameters

- **output_activations** – vector of output activations
- **y** – expected output vector

sigmoid (*z*)
Implements the sigmoid function. https://en.wikipedia.org/wiki/Sigmoid_function

sigmoid_prime (*z*)
Derivative of the sigmoid function.

1.4 network module

1.4.1 network.py

IT WORKS

A module to implement the stochastic gradient descent learning algorithm for a feedforward neural network. Gradients are calculated using backpropagation. Note that I have focused on making the code simple, easily readable, and easily modifiable. It is not optimized, and omits many desirable features.

class network.**Network** (*sizes*)
Bases: `object`

__init__ (*sizes*)
The list `sizes` contains the number of neurons in the respective layers of the network. For example, if

the list was [2, 3, 1] then it would be a three-layer network, with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron. The biases and weights for the network are initialized randomly, using a Gaussian distribution with mean 0, and variance 1. Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.

save (*filename*)

load (*filename*)

feedforward (*a*)
Return the output of the network if `a` is input.

SGD (*training_data*, *epochs*, *mini_batch_size*, *eta*, *test_data=None*, *start_epoch=1*)
Train the neural network using mini-batch stochastic gradient descent. The `training_data` is a list of tuples `(x, y)` representing the training inputs and the desired outputs. The other non-optional parameters are self-explanatory. If `test_data` is provided then the network will be evaluated against the test data after each epoch, and partial progress printed out. This is useful for tracking progress, but slows things down substantially.

update_mini_batch (*mini_batch*, *eta*)
Update the network's weights and biases by applying gradient descent using backpropagation to a single mini batch. The `mini_batch` is a list of tuples `(x, y)`, and `eta` is the learning rate.

backprop (*x*, *y*)
Return a tuple `(nabla_b, nabla_w)` representing the gradient for the cost function C_x. `nabla_b` and `nabla_w` are layer-by-layer lists of numpy arrays, similar to `self.biases` and `self.weights`.

evaluate (*test_data*)
Return the number of test inputs for which the neural network outputs the correct result. Note that the neural network's output is assumed to be the index of whichever neuron in the final layer has the highest activation.

cost_derivative (*output_activations*, *y*)
Return the vector of partial derivatives partial C_x / partial a for the output activations.

`network.`**sigmoid** (*z*)
The sigmoid function.

`network.`**sigmoid_prime** (*z*)
Derivative of the sigmoid function.

1.5 network2 module

1.5.1 network2.py

An improved version of network.py, implementing the stochastic gradient descent learning algorithm for a feedforward neural network. Improvements include the addition of the cross-entropy cost function, regularization, and better initialization of network weights. Note that I have focused on making the code simple, easily readable, and easily modifiable. It is not optimized, and omits many desirable features.

class `network2.`**QuadraticCost**
Bases: `object`

static fn (*a*, *y*)
Return the cost associated with an output `a` and desired output `y`.

static delta (*z*, *a*, *y*)
Return the error delta from the output layer.

class `network2.`**CrossEntropyCost**
Bases: `object`

static fn (*a*, *y*)
Return the cost associated with an output `a` and desired output `y`. Note that np.nan_to_num is used to ensure numerical stability. In particular, if both `a` and `y` have a 1.0 in the same slot, then the expression (1-y)*np.log(1-a) returns nan. The np.nan_to_num ensures that that is converted to the correct value (0.0).

static delta (*z*, *a*, *y*)
Return the error delta from the output layer. Note that the parameter `z` is not used by the method. It is included in the method's parameters in order to make the interface consistent with the delta method for other cost classes.

class `network2.`**Network** (*sizes*, *cost=<class 'network2.CrossEntropyCost'>*)
Bases: `object`

__init__ (*sizes*, *cost=<class 'network2.CrossEntropyCost'>*)
The list `sizes` contains the number of neurons in the respective layers of the network. For example, if the list was [2, 3, 1] then it would be a three-layer network, with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron. The biases and weights for the network are initialized randomly, using `self.default_weight_initializer` (see docstring for that method).

default_weight_initializer ()
Initialize each weight using a Gaussian distribution with mean 0 and standard deviation 1 over the square root of the number of weights connecting to the same neuron. Initialize the biases using a Gaussian distribution with mean 0 and standard deviation 1.

Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.

large_weight_initializer ()
Initialize the weights using a Gaussian distribution with mean 0 and standard deviation 1. Initialize the biases using a Gaussian distribution with mean 0 and standard deviation 1.

Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers.

This weight and bias initializer uses the same approach as in Chapter 1, and is included for purposes of comparison. It will usually be better to use the default weight initializer instead.

feedforward (*a*)
Return the output of the network if `a` is input.

SGD (*training_data*, *epochs*, *mini_batch_size*, *eta*, *lmbda=0.0*, *evaluation_data=None*, *monitor_evaluation_cost=False*, *monitor_evaluation_accuracy=False*, *monitor_training_cost=False*, *monitor_training_accuracy=False*, *early_stopping_n=0*)
Train the neural network using mini-batch stochastic gradient descent. The `training_data` is a list of tuples `(x, y)` representing the training inputs and the desired outputs. The other non-optional parameters are self-explanatory, as is the regularization parameter `lmbda`. The method also accepts `evaluation_data`, usually either the validation or test data. We can monitor the cost and accuracy on either the evaluation data or the training data, by setting the appropriate flags. The method returns a tuple containing four lists: the (per-epoch) costs on the evaluation data, the accuracies on the evaluation data, the costs on the training data, and the accuracies on the training data. All values are evaluated at the end of each training epoch. So, for example, if we train for 30 epochs, then the first element of the tuple will be a 30-element list containing the cost on the evaluation data at the end of each epoch. Note that the lists are empty if the corresponding flag is not set.

update_mini_batch (*mini_batch*, *eta*, *lmbda*, *n*)
Update the network's weights and biases by applying gradient descent using backpropagation to a single mini batch. The `mini_batch` is a list of tuples `(x, y)`, `eta` is the learning rate, `lmbda` is the regularization parameter, and `n` is the total size of the training data set.

backprop (*x*, *y*)
Return a tuple `(nabla_b, nabla_w)` representing the gradient for the cost function C_x. `nabla_b` and `nabla_w` are layer-by-layer lists of numpy arrays, similar to `self.biases` and `self.weights`.

accuracy (*data*, *convert=False*)
Return the number of inputs in `data` for which the neural network outputs the correct result. The neural network's output is assumed to be the index of whichever neuron in the final layer has the highest activation.

The flag `convert` should be set to False if the data set is validation or test data (the usual case), and to True if the data set is the training data. The need for this flag arises due to differences in the way the results `y` are represented in the different data sets. In particular, it flags whether we need to convert between the different representations. It may seem strange to use different representations for the different data sets. Why not use the same representation for all three data sets? It's done for efficiency reasons – the program usually evaluates the cost on the training data and the accuracy on other data sets. These are different types of computations, and using different representations speeds things up. More details on the representations can be found in mnist_loader.load_data_wrapper.

total_cost (*data*, *lmbda*, *convert=False*)
Return the total cost for the data set `data`. The flag `convert` should be set to False if the data set is the training data (the usual case), and to True if the data set is the validation or test data. See comments on the similar (but reversed) convention for the `accuracy` method, above.

save (*filename*)
Save the neural network to the file `filename`.

`network2.`**load** (*filename*)
Load a neural network from the file `filename`. Returns an instance of Network.

`network2.`**vectorized_result** (*j*)
Return a 10-dimensional unit vector with a 1.0 in the j'th position and zeroes elsewhere. This is used to convert a digit (0...9) into a corresponding desired output from the neural network.

`network2.`**sigmoid** (*z*)
The sigmoid function.

`network2.`**sigmoid_prime** (*z*)
Derivative of the sigmoid function.

1.6 network3 module

1.7 test module

Testing code for different neural network configurations. Adapted for Python 3.5.2

Usage in shell: python3.5 test.py

Network (network.py and network2.py) parameters: 2nd param is epochs count 3rd param is batch size 4th param is learning rate (eta)

Author: Michał Dobrzański, 2016 dobrzanski.michal.daniel@gmail.com

`test.`**main** ()

1.8 test1 module

`test1.`**`main`**`()`

`test1.`**`visualizeResults`**`(`*dataSet*, *fname='backups/latest.pkl'*, *saveSuccesses=False*`)`

`test1.`**`train`**`()`

`test1.`**`test`**`()`

1.9 tests package

1.9.1 Submodules

1.9.2 tests.conftest module

`tests.conftest.`**`get_dummy_training_data`**`(`*inputSize*, *outputSize*, *count*`)`
creates a list of randomized (dummy) training data (X, Y) of the desired dimentions.

1.9.3 tests.test_network1 module

test_network1.py

unit test mynet.py against network.py (the provided implementation)

`tests.test_network1.`**`test_feed_forward`**`()`
test that feedforward() behaves the same across the two implemenations

`tests.test_network1.`**`test_backprop`**`()`
test that backprop() behaves the same across the two implemenations

`tests.test_network1.`**`test_update_mini_batch`**`()`
test that update_mini_batch() behaves the same across the two implemenations

`tests.test_network1.`**`test_SGD`**`()`
test that SGD() behaves the same across the two implemenations

1.9.4 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

L

M

W