DeepLearningPython35

Daniel Engbert, Michael Nielsen, Michal Dobrzanski

Dec 14, 2020

CONTENTS:

CHAPTER

ONE

DEEPLEARNINGPYTHON35

1.1 bootyNet module

bootyNet.**getConvergeRate**(*history*, *largeBin*, *smallBin*, *thresHold=0.005*)

bootyNet.**gooo**(*prev=None*, *entropyYears=0*)

bootyNet.**main**()

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

1.2 expand_mnist module

1.2.1 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.

expand_mnist.**main**()

1.3 mnist_average_darkness module

1.3.1 mnist_average_darkness

A naive classifier for recognizing handwritten digits from the MNIST data set. The program classifies digits based on how dark they are — the idea is that digits like "1" tend to be less dark than digits like "8", simply because the latter has a more complex shape. When shown an image the classifier returns whichever digit in the training data had the closest average darkness.

The program works in two steps: first it trains the classifier, and then it applies the classifier to the MNIST test data to see how many digits are correctly classified.

Needless to say, this isn't a very good way of recognizing handwritten digits! Still, it's useful to show what sort of performance we get from naive ideas.

mnist_average_darkness.**avg_darknesses**(*training_data*)

Return a defaultdict whose keys are the digits 0 through 9. For each digit we compute a value which is the average darkness of training images containing that digit. The darkness for any particular image is just the sum of the darknesses for each pixel.

`mnist_average_darkness.`**`guess_digit`**`(`*image*, *avgs*`)`

Return the digit whose average darkness in the training data is closest to the darkness of `image`. Note that `avgs` is assumed to be a defaultdict whose keys are 0. . . 9, and whose values are the corresponding average darknesses across the training data.

`mnist_average_darkness.`**`main`**`()`

1.4 mnist_loader module

1.4.1 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_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_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_loader.`**`vectorToImage`**`(`*v*, *fname=''*`)`

`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.

1.5 mnist_svm module

1.5.1 mnist_svm

A classifier program for recognizing handwritten digits from the MNIST data set, using an SVM classifier.

`mnist_svm.`**`svm_baseline`**`()`

1.6 mynet module

1.6.1 network.py

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

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

Bases: object

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(*cur*, *expected*)

cost_derivative(*output_activations*, *y*)

Return the vector of partial derivatives partial C_x / partial a for the output activations.

feedforward(*a*)

getOutput(*x*)

load(*filename*)

save(*filename*)

static sigmoid(*z*)

static sigmoid_prime(*z*)

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.

updateMiniBatch(*miniBatch*, *rate*)

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 `rate` is the learning rate. note: to train on a single sample, set mini_batch=[(x,y)]

1.7 network module

1.7.1 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

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.

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`.

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

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.

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

load (*filename*)

save (*filename*)

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.

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

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

1.8 network2 module

1.8.1 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.`**CrossEntropyCost**
Bases: `object`

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.

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).

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

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.

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.

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`.

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.

feedforward(*a*)

Return the output of the network if `a` is input.

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.

save(*filename*)

Save the neural network to the file `filename`.

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.

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.

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

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

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

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

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

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

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.

1.9 network3 module

1.9.1 network3.py

A Theano-based program for training and running simple neural networks.

Supports several layer types (fully connected, convolutional, max pooling, softmax), and activation functions (sigmoid, tanh, and rectified linear units, with more easily added).

When run on a CPU, this program is much faster than network.py and network2.py. However, unlike network.py and network2.py it can also be run on a GPU, which makes it faster still.

Because the code is based on Theano, the code is different in many ways from network.py and network2.py. However, where possible I have tried to maintain consistency with the earlier programs. In particular, the API is similar to network2.py. Note that I have focused on making the code simple, easily readable, and easily modifiable. It is not optimized, and omits many desirable features.

This program incorporates ideas from the Theano documentation on convolutional neural nets (notably, http://deeplearning.net/tutorial/lenet.html), from Misha Denil's implementation of dropout (https://github.com/mdenil/dropout), and from Chris Olah (http://colah.github.io).

class network3.**ConvPoolLayer** (*filter_shape*, *image_shape*, *poolsize=(2, 2)*, *activation_fn=<theano.tensor.elemwise.Elemwise object>*)
Bases: `object`

Used to create a combination of a convolutional and a max-pooling layer. A more sophisticated implementation would separate the two, but for our purposes we'll always use them together, and it simplifies the code, so it makes sense to combine them.

set_inpt (*inpt*, *inpt_dropout*, *mini_batch_size*)

class network3.**FullyConnectedLayer**(*n_in*, *n_out*, *activation_fn=<theano.tensor.elemwise.Elemwise object>*, *p_dropout=0.0*)
Bases: object

accuracy(*y*)
Return the accuracy for the mini-batch.

set_inpt(*inpt*, *inpt_dropout*, *mini_batch_size*)

class network3.**Network**(*layers*, *mini_batch_size*)
Bases: object

SGD(*training_data*, *epochs*, *mini_batch_size*, *eta*, *validation_data*, *test_data*, *lmbda=0.0*)
Train the network using mini-batch stochastic gradient descent.

network3.**ReLU**(*z*)

class network3.**SoftmaxLayer**(*n_in*, *n_out*, *p_dropout=0.0*)
Bases: object

accuracy(*y*)
Return the accuracy for the mini-batch.

cost(*net*)
Return the log-likelihood cost.

set_inpt(*inpt*, *inpt_dropout*, *mini_batch_size*)

network3.**dropout_layer**(*layer*, *p_dropout*)

network3.**linear**(*z*)

network3.**load_data_shared**(*filename='mnist.pkl.gz'*)

network3.**size**(*data*)
Return the size of the dataset *data*.

1.10 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**()

1.11 test1 module

test1.**main**()

test1.**test**()

test1.**train**()

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

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

N

Q

R

S

T

U

V