Data Analysis and Machine Learning: Elements of machine learning

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Sep 28, 2018

Neural networks

Artificial neural networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. It is supposed to mimic a biological system, wherein neurons interact by sending signals in the form of mathematical functions between layers. All layers can contain an arbitrary number of neurons, and each connection is represented by a weight variable.

Artificial neurons

The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later been refined by others. The general idea is to mimic neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output.

This behaviour has inspired a simple mathematical model for an artificial neuron.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{1}$$

Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_i, \ldots, x_n$ received by n other neurons.

Conceptually, it is helpful to divide neural networks into four categories:

1. general purpose neural networks for supervised learning,

2. neural networks designed specifically for image processing, the most prominent example of this class being Convolutional Neural Networks (CNNs),

3. neural networks for sequential data such as Recurrent Neural Networks (RNNs), and

4. neural networks for unsupervised learning such as Deep Boltzmann Machines.

In natural science, DNNs and CNNs have already found numerous applications. In statistical physics, they have been applied to detect phase transitions in 2D Ising and Potts models, lattice gauge theories, and different phases of polymers, or solving the Navier-Stokes equation in weather forecasting. Deep learning has also found interesting applications in quantum physics. Various quantum phase transitions can be detected and studied using DNNs and CNNs, topological phases, and even non-equilibrium many-body localization. Representing quantum states as DNNs quantum state tomography are among some of the impressive achievements to reveal the potential of DNNs to facilitate the study of quantum systems.

In quantum information theory, it has been shown that one can perform gate decompositions with the help of neural. In lattice quantum chromodynamics, DNNs have been used to learn action parameters in regions of parameter space where PCA fails.

The applications are not limited to the natural sciences. There is a plethora of applications in essentially all disciplines, from the humanities to life science and medicine.

Neural network types

An artificial neural network (NN), is a computational model that consists of layers of connected neurons, or *nodes*. It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between layers. A wide variety of different NNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.

Neural networks (also called neural nets) are neural-inspired nonlinear models for supervised learning. As we will see, neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods.

Feed-forward neural networks

The feed-forward neural network (FFNN) was the first and simplest type of NN devised. In this network, the information moves in only one direction: forward through the layers.

Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of information flow. Additionally, each arrow corresponds to a weight variable, not displayed here. We observe that each node in a layer is connected to *all* nodes in the subsequent layer, making this a so-called *fully-connected* FFNN.

A different variant of FFNNs are *convolutional neural networks* (CNNs), which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation.

CNNs emulate the behaviour of neurons in the visual cortex by enforcing a *local* connectivity pattern between nodes of adjacent layers: Each node in a convolutional layer is connected only to a subset of the nodes in the previous layer, in contrast to the fully-connected FFNN. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs. They have wide applications in image and video recognition

Recurrent neural networks

So far we have only mentioned NNs where information flows in one direction: forward. *Recurrent neural networks* on the other hand, have connections between nodes that form directed *cycles*. This creates a form of internal memory which are able to capture information on what has been calculated before; the output is dependent on the previous computations. Recurrent NNs make use of sequential information by performing the same task for every element in a sequence, where each element depends on previous elements. An example of such information is sentences, making recurrent NNs especially well-suited for handwriting and speech recognition.

Other types of networks

There are many other kinds of NNs that have been developed. One type that is specifically designed for interpolation in multidimensional space is the radial basis function (RBF) network. RBFs are typically made up of three layers: an input layer, a hidden layer with non-linear radial symmetric activation functions and a linear output layer (”linear” here means that each node in the output layer has a linear activation function). The layers are normally fully-connected and there are no cycles, thus RBFs can be viewed as a type of fully-connected FFNN.

They are however usually treated as a separate type of NN due the unusual activation functions.

Multilayer perceptrons

One uses often so-called fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions.

Such networks are often called *multilayer perceptrons* (MLPs)

Why multilayer perceptrons?

According to the *Universal approximation theorem*, a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a **non-constant, bounded and monotonically-increasing continuous function**.

Note that the requirements on the activation function only applies to the hidden layer, the output nodes are always assumed to be linear, so as to not restrict the range of output values.

We note that this theorem is only applicable to an NN with *one* hidden layer. Therefore, we can easily construct an NN that employs activation functions which do not satisfy the above requirements, as long as we have at least one layer with activation functions that *do*. Furthermore, although the universal approximation theorem lays the theoretical foundation for regression with neural networks, it does not say anything about how things work in practice: A neural network can still be able to approximate a given function reasonably well without having the flexibility to fit *all other* functions.

Mathematical model

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(u) \tag{2}$$

In an FFNN of such neurons, the *inputs* x_i are the *outputs* of the neurons in the preceding layer. Furthermore, an MLP is fully-connected, which means that each neuron receives a weighted sum of the outputs of *all* neurons in the previous layer.

Mathematical model

First, for each node i in the first hidden layer, we calculate a weighted sum u_i^1 of the input coordinates x_j,

$$u_i^1 = \sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1 \tag{3}$$

This value is the argument to the activation function f_1 of each neuron i, producing the output y_i^1 of all neurons in layer 1,

$$y_i^1 = f_1(u_i^1) = f_1\left(\sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1\right) \tag{4}$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation f_l

$$y_i^l = f_l(u_i^l) = f_l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{5}$$

where N_l is the number of nodes in layer l. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained.

Mathematical model

The output of neuron i in layer 2 is thus,

$$y_i^2 = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right) \tag{6}$$

$$= f_2\left[\sum_{j=1}^{3} w_{ij}^2 f_1\left(\sum_{k=1}^{2} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{7}$$

where we have substituted y_m^1 with. Finally, the NN output yields,

$$y_1^3 = f_3\left(\sum_{j=1}^{3} w_{1m}^3 y_j^2 + b_1^3\right) \tag{8}$$

$$= f_3\left[\sum_{j=1}^{3} w_{1j}^3 f_2\left(\sum_{k=1}^{3} w_{jk}^2 f_1\left(\sum_{m=1}^{2} w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{9}$$

Mathematical model

We can generalize this expression to an MLP with l hidden layers. The complete functional form is,

$$y_1^{l+1} = f_{l+1}\left[\sum_{j=1}^{N_l} w_{1j}^3 f_l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^2 f_{l-1}\left(\ldots f_1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\ldots\right) + b_k^2\right) + b_1^3\right] \tag{10}$$

which illustrates a basic property of MLPs: The only independent variables are the input values x_n.

Mathematical model

This confirms that an MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\vec{x} \in \mathbb{R}^n \to \vec{y} \in \mathbb{R}^m$. In our example, $n = 2$ and $m = 1$. Consequentially, the number of input and output values of the function we want to fit must be equal to the number of inputs and outputs of our MLP.

Furthermore, the flexibility and universality of a MLP can be illustrated by realizing that the expression is essentially a nested sum of scaled activation functions of the form

$$h(x) = c_1 f(c_2 x + c_3) + c_4 \tag{11}$$

where the parameters c_i are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled which is the key to the flexibility of a neural network.

Matrix-vector notation. We can introduce a more convenient notation for the activations in a NN.

Additionally, we can represent the biases and activations as layer-wise column vectors $\vec{b}_l$ and $\vec{y}_l$, so that the i-th element of each vector is the bias b_i^l and activation y_i^l of node i in layer l respectively.

We have that W_l is a $N_{l-1} \times N_l$ matrix, while $\vec{b}_l$ and $\vec{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum in becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 in

$$\vec{y}_2 = f_2(\mathrm{W}_2 \vec{y}_1 + \vec{b}_2) = f_2 \left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix} \right). \tag{12}$$

Matrix-vector notation and activation. The activation of node i in layer 2 is

$$y_i^2 = f_2\Big(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Big) = f_2 \left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2 \right). \tag{13}$$

This is not just a convenient and compact notation, but also a useful and intuitive way to think about MLPs: The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $\mathrm{W}_l \vec{y}_{l-1}$ we move forward one layer.

Activation functions. A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s). As described in, the following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

Activation functions, Logistic and Hyperbolic ones. The second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs.

Regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs. Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions Typical examples are the logistic *Sigmoid*

$$f(x) = \frac{1}{1 + e^{-x}}, \tag{14}$$

and the *hyperbolic tangent* function

$$f(x) = \tanh(x) \tag{15}$$

Relevance. The *sigmoid* function are more biologically plausible because the output of inactive neurons are zero. Such activation function are called *one-sided*. However, it has been shown that the hyperbolic tangent performs better than the sigmoid for training MLPs. has become the most popular for *deep neural networks*

```
"""The sigmoid function (or the logistic curve) is a
function that takes any real number, z, and outputs a number (0,1).
It is useful in neural networks for assigning weights on a relative scale.
The value z is the weighted sum of parameters involved in the learning algorithm."""

import numpy
import matplotlib.pyplot as plt
import math as mt

z = numpy.arange(-5, 5, .1)
sigma_fn = numpy.vectorize(lambda z: 1/(1+numpy.exp(-z)))
sigma = sigma_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, sigma)
ax.set_ylim([-0.1, 1.1])
```

```
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sigmoid function')

plt.show()

"""Step Function"""
z = numpy.arange(-5, 5, .02)
step_fn = numpy.vectorize(lambda z: 1.0 if z >= 0.0 else 0.0)
step = step_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, step)
ax.set_ylim([-0.5, 1.5])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('step function')

plt.show()

"""Sine Function"""
z = numpy.arange(-2*mt.pi, 2*mt.pi, 0.1)
t = numpy.sin(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, t)
ax.set_ylim([-1.0, 1.0])
ax.set_xlim([-2*mt.pi,2*mt.pi])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sine function')

plt.show()

"""Plots a graph of the squashing function used by a rectified linear
unit"""
z = numpy.arange(-2, 2, .1)
zero = numpy.zeros(len(z))
y = numpy.max([zero, z], axis=0)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, y)
ax.set_ylim([-2.0, 2.0])
ax.set_xlim([-2.0, 2.0])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('Rectified linear unit')

plt.show()
```

Setting up a Multi-layer perceptron model

```
from scipy import optimize

class Neural_Network(object):
```

```
def __init__(self, Lambda=0):
    #Define Hyperparameters
    self.inputLayerSize = 2
    self.outputLayerSize = 1
    self.hiddenLayerSize = 3

    #Weights (parameters)
    self.W1 = np.random.randn(self.inputLayerSize,self.hiddenLayerSize)
    self.W2 = np.random.randn(self.hiddenLayerSize,self.outputLayerSize)

    #Regularization Parameter:
    self.Lambda = Lambda

def forward(self, X):
    #Propogate inputs though network
    self.z2 = np.dot(X, self.W1)
    self.a2 = self.sigmoid(self.z2)
    self.z3 = np.dot(self.a2, self.W2)
    yHat = self.sigmoid(self.z3)
    return yHat

def sigmoid(self, z):
    #Apply sigmoid activation function to scalar, vector, or matrix
    return 1/(1+np.exp(-z))

def sigmoidPrime(self,z):
    #Gradient of sigmoid
    return np.exp(-z)/((1+np.exp(-z))**2)

def costFunction(self, X, y):
    #Compute cost for given X,y, use weights already stored in class.
    self.yHat = self.forward(X)
    J = 0.5*sum((y-self.yHat)**2)/X.shape[0] + (self.Lambda/2)*(np.sum(self.W1**2)+np.sum(self
    return J

def costFunctionPrime(self, X, y):
    #Compute derivative with respect to W and W2 for a given X and y:
    self.yHat = self.forward(X)

    delta3 = np.multiply(-(y-self.yHat), self.sigmoidPrime(self.z3))
    #Add gradient of regularization term:
    dJdW2 = np.dot(self.a2.T, delta3)/X.shape[0] + self.Lambda*self.W2

    delta2 = np.dot(delta3, self.W2.T)*self.sigmoidPrime(self.z2)
    #Add gradient of regularization term:
    dJdW1 = np.dot(X.T, delta2)/X.shape[0] + self.Lambda*self.W1

    return dJdW1, dJdW2

#Helper functions for interacting with other methods/classes
def getParams(self):
    #Get W1 and W2 Rolled into vector:
    params = np.concatenate((self.W1.ravel(), self.W2.ravel()))
    return params

def setParams(self, params):
    #Set W1 and W2 using single parameter vector:
    W1_start = 0
    W1_end = self.hiddenLayerSize*self.inputLayerSize
    self.W1 = np.reshape(params[W1_start:W1_end], \
                         (self.inputLayerSize, self.hiddenLayerSize))
```

```
        W2_end = W1_end + self.hiddenLayerSize*self.outputLayerSize
        self.W2 = np.reshape(params[W1_end:W2_end], \
                             (self.hiddenLayerSize, self.outputLayerSize))

    def computeGradients(self, X, y):
        dJdW1, dJdW2 = self.costFunctionPrime(X, y)
        return np.concatenate((dJdW1.ravel(), dJdW2.ravel()))


class trainer(object):
    def __init__(self, N):
        #Make Local reference to network:
        self.N = N

    def callbackF(self, params):
        self.N.setParams(params)
        self.J.append(self.N.costFunction(self.X, self.y))
        self.testJ.append(self.N.costFunction(self.testX, self.testY))

    def costFunctionWrapper(self, params, X, y):
        self.N.setParams(params)
        cost = self.N.costFunction(X, y)
        grad = self.N.computeGradients(X,y)
        return cost, grad

    def train(self, trainX, trainY, testX, testY):
        #Make an internal variable for the callback function:
        self.X = trainX
        self.y = trainY

        self.testX = testX
        self.testY = testY

        #Make empty list to store training costs:
        self.J = []
        self.testJ = []

        params0 = self.N.getParams()

        options = {'maxiter': 200, 'disp' : True}
        _res = optimize.minimize(self.costFunctionWrapper, params0, jac=True, method='BFGS', \
                                 args=(trainX, trainY), options=options, callback=self.callbackF)

        self.N.setParams(_res.x)
        self.optimizationResults = _res
```

Two-layer Neural Network

```
import numpy as np

#sigmoid
def nonlin(x, deriv=False):
    if (deriv==True):
        return x*(1-x)
    return 1/(1+np.exp(-x))

#input data
x=np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])

#output data
```

```
y=np.array([0,1,1,0]).T

#seed random numbers to make calculation
np.random.seed(1)

#initialize weights with mean=0
syn0=2*np.random.random((3,4))-1

for iter in range(10000):
    #forward propogation
    l0=x
    l1=nonlin(np.dot(l0,syn0))
    l1_error=y-l1
    #multiply error by slope of sigmoid at values of l1
    l1_delta=l1_error*nonlin(l1,True)
    #update weights
    syn0+=np.dot(l0.T, l1_delta)

print("Output after training: ",l1 )

import numpy as np
import random
class Network(object):

    def _init_(self, sizes):
        self.num_layers=len(sizes)
        self.sizes=sizes
        self.biases=[np.random.randn(y,1) for y in sizes[1:]]
        self.weights=[np.random.randn(y,x) for x,y in zip(sizes[:-1], sizes[1:])]

#sizes is the number of neurons in each layer
#for example, say n_1st_layer=3, n_2nd_layer=3, n_3rd_layer=1, then net=Network([3,3,1])

#The biases and weights are initialized randomly, using Gaussian distributions of mean=0, stdev=1
#z is a vector (or a np.array)

    def feedforward(self,a):
        #returns output w/ 'a' as an input
        for b, w in zip(self.biases, self.weights):
            a=sigmoid(np.dot(w,b)+b)
        return a

#Apply a Stochastic Gradient Descent (SGD) method:
    def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
        """Trains network using batches incorporating SGD. The network will be evaluated against
        test data after each epoch, with partial progress being printed out (this is useful for t
        but slows the process.)"""
        if test_data: n_test=len(test_data)
        n=len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches=[training_data[k:k+mini_batch_size] for k in xrange(o,n,mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print ("Epoch {0}: {1}/{2}".format(j, self.evaluate(test_data), n_test))
            else:
                print ("Epoch {0} complete".format(j))


    def update_mini_batch(self, mini_batch, eta):
```

```
        #updates w and b using backpropagation to a single mini batch. eta is the learning rate."
        nabla_b=[np.zeros(b.shape) for b in self.biases]
        nabla_w=[np.zeros(w.shape) for w in self.weights]
        for x,y in mini_batch:
            delta_nabla_b, delta_nabla_w=self.backprop(x,y)
            nabla_b=[nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w=[nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights=[w-(eta/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)]
        self.biases=[b-(eta/len(mini_batch))*nb for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, 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``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def evaluate(self, 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."""
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)
```

```
#Functions
def sigmoid(z):
        return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))

network=Network()

# %load neural-networks-and-deep-learning/src/mnist_loader.py
"""
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.
"""

#### Libraries
# Standard library
import pickle
import gzip

# Third-party libraries
import numpy as np

def 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.
    """
    f = gzip.open('../data/mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = cPickle.load(f)
    f.close()
    return (training_data, validation_data, test_data)

def 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."""
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)

def 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."""
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e

net=network.Network([784,30,30])
net.SGD(training_data,30,10,3,test_data=test_data)
```