Homework7_DS-GA1003

May 6, 2021

Name: Zhuoyuan Xu (Kallen)

NetID: zx1137

Problem 1-3

[ ]:
```
"""Computation graph node types

Nodes must implement the following methods:
__init__    - initialize node
forward     - (step 1 of backprop) retrieve output ("out") of predecessor nodes␣
 ↪(if
              applicable), update own output ("out"), and set gradient ("d_out")␣
 ↪to zero
backward    - (step 2 of backprop), assumes that forward pass has run before.
              Also assumes that backward has been called on all of the node's
              successor nodes, so that self.d_out contains the
              gradient of the graph output with respect to the node output.
              Backward computes summands of the derivative of graph output with
              respect to the inputs of the node, corresponding to paths through␣
 ↪the graph
              that go from the node's input through the node to the graph's␣
 ↪output.
              These summands are added to the input node's d_out array.
get_predecessors - return a list of the node's parents

Nodes must furthermore have a the following attributes:
node_name   - node's name (a string)
out       - node's output
d_out     - derivative of graph output w.r.t. node output

This computation graph framework was designed and implemented by
Philipp Meerkamp, Pierre Garapon, and David Rosenberg.
License: Creative Commons Attribution 4.0 International License
"""

import numpy as np
```

```python
class ValueNode(object):
    """Computation graph node having no input but simply holding a value"""
    def __init__(self, node_name):
        self.node_name = node_name
        self.out = None
        self.d_out = None

    def forward(self):
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        pass

    def get_predecessors(self):
        return []


class VectorScalarAffineNode(object):
    """ Node computing an affine function mapping a vector to a scalar."""
    def __init__(self, x, w, b, node_name):
        """
        Parameters:
        x: node for which x.out is a 1D numpy array
        w: node for which w.out is a 1D numpy array of same size as x.out
        b: node for which b.out is a numpy scalar (i.e. 0dim array)
        node_name: node's name (a string)
        """
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.x = x
        self.w = w
        self.b = b

    def forward(self):
        self.out = np.dot(self.x.out, self.w.out) + self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_x = self.d_out * self.w.out
        d_w = self.d_out * self.x.out
        d_b = self.d_out
        self.x.d_out += d_x
        self.w.d_out += d_w
        self.b.d_out += d_b
```

```
    def get_predecessors(self):
        return [self.x, self.w, self.b]


class SquaredL2DistanceNode(object):
    """ Node computing L2 distance (sum of square differences) between 2 arrays.
 ↪"""
    def __init__(self, a, b, node_name):
        """
        Parameters:
        a: node for which a.out is a numpy array
        b: node for which b.out is a numpy array of same shape as a.out
        node_name: node's name (a string)
        """
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.a = a
        self.b = b
        # Variable for caching values between forward and backward
        self.a_minus_b = None

    def forward(self):
        self.a_minus_b = self.a.out - self.b.out
        self.out = np.sum(self.a_minus_b ** 2)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_a = self.d_out * 2 * self.a_minus_b
        d_b = -self.d_out * 2 * self.a_minus_b
        self.a.d_out += d_a
        self.b.d_out += d_b
        return self.d_out

    def get_predecessors(self):
        return [self.a, self.b]


class L2NormPenaltyNode(object):
    """ Node computing l2_reg * ||w||^2 for scalars l2_reg and vector w"""
    def __init__(self, l2_reg, w, node_name):
        """
        Parameters:
        l2_reg: a numpy scalar array (e.g. np.array(.01)) (not a node)
        w: a node for which w.out is a numpy vector
```

```
        node_name: node's name (a string)
        """
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.l2_reg = np.array(l2_reg)
        self.w = w

    def forward(self):
        self.out = self.l2_reg * np.linalg.norm(self.w.out) ** 2
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_w = self.d_out * (2 * self.l2_reg * self.w.out)
        self.w.d_out += d_w
        return self.d_out

    def get_predecessors(self):
        return [self.w]


class SumNode(object):
    """ Node computing a + b, for numpy arrays a and b"""
    def __init__(self, a, b, node_name):
        """
        Parameters:
        a: node for which a.out is a numpy array
        b: node for which b.out is a numpy array of the same shape as a
        node_name: node's name (a string)
        """
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.b = b
        self.a = a

    def forward(self):
        self.out = self.a.out + self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_a = self.d_out
        d_b = self.d_out
        self.a.d_out += d_a
        self.b.d_out += d_b
```

```
    def get_predecessors(self):
        return [self.a, self.b]
```

[ ]:
```
import setup_problem
from sklearn.base import BaseEstimator, RegressorMixin
import numpy as np
import nodes_zx1137 as nodes
import graph
import plot_utils

class RidgeRegression(BaseEstimator, RegressorMixin):
    """ Ridge regression with computation graph """
    def __init__(self, l2_reg=1, step_size=.005,  max_num_epochs = 5000):
        self.max_num_epochs = max_num_epochs
        self.step_size = step_size

        # Build computation graph
        self.x = nodes.ValueNode(node_name="x") # to hold a vector input
        self.y = nodes.ValueNode(node_name="y") # to hold a scalar response
        self.w = nodes.ValueNode(node_name="w") # to hold the parameter vector
        self.b = nodes.ValueNode(node_name="b") # to hold the bias parameter␣
↪(scalar)
        self.prediction = nodes.VectorScalarAffineNode(x=self.x, w=self.w,␣
↪b=self.b,
                                                       node_name="prediction")
        # Build computation graph
        self.reg_w = nodes.L2NormPenaltyNode(l2_reg=l2_reg, w=self.w,␣
↪node_name="reg_w")
        self.obj = nodes.SquaredL2DistanceNode(a=self.y, b=self.prediction,␣
↪node_name="diff")
        self.reg_obj = nodes.SumNode(self.reg_w, self.obj, node_name="reg_obj")
        self.graph = graph.ComputationGraphFunction(inputs=[self.x],␣
↪outcomes=[self.y], parameters=[self.w,self.b], prediction=self.prediction,␣
↪objective=self.reg_obj)

    def fit(self, X, y):
        num_instances, num_ftrs = X.shape
        y = y.reshape(-1)

        init_parameter_values = {"w": np.zeros(num_ftrs), "b": np.array(0.0)}
        self.graph.set_parameters(init_parameter_values)

        for epoch in range(self.max_num_epochs):
            shuffle = np.random.permutation(num_instances)
            epoch_obj_tot = 0.0
            for j in shuffle:
```

```
                obj, grads = self.graph.get_gradients(input_values = {"x":␣
↪X[j]},
                                                      outcome_values = {"y":␣
↪y[j]})
                #print(obj)
                epoch_obj_tot += obj
                # Take step in negative gradient direction
                steps = {}
                for param_name in grads:
                    steps[param_name] = -self.step_size * grads[param_name]
                self.graph.increment_parameters(steps)

            if epoch % 50 == 0:
                train_loss = sum((y - self.predict(X,y)) **2)/num_instances
                print("Epoch ",epoch,": Ave objective=",epoch_obj_tot/
↪num_instances," Ave training loss: ",train_loss)

    def predict(self, X, y=None):
        try:
            getattr(self, "graph")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting data!
↪")

        num_instances = X.shape[0]
        preds = np.zeros(num_instances)
        for j in range(num_instances):
            preds[j] = self.graph.get_prediction(input_values={"x":X[j]})

        return preds



def main():
    data_fname = "data.pickle"
    x_train, y_train, x_val, y_val, target_fn, coefs_true, featurize =␣
↪setup_problem.load_problem(data_fname)

    # Generate features
    X_train = featurize(x_train)
    X_val = featurize(x_val)

    pred_fns = []
    x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))
    X = featurize(x)

    l2reg = 1
```

```
    estimator = RidgeRegression(l2_reg=l2reg, step_size=0.00005,␣
↪max_num_epochs=2000)
    estimator.fit(X_train, y_train)
    name = "Ridge with L2Reg="+str(l2reg)
    pred_fns.append({"name":name, "preds": estimator.predict(X) })


    l2reg = 0
    estimator = RidgeRegression(l2_reg=l2reg, step_size=0.0005,␣
↪max_num_epochs=500)
    estimator.fit(X_train, y_train)
    name = "Ridge with L2Reg="+str(l2reg)
    pred_fns.append({"name":name, "preds": estimator.predict(X) })

    # Let's plot prediction functions and compare coefficients for several fits
    # and the target function.

    pred_fns.append({"name": "Target Parameter Values (i.e. Bayes Optimal)",␣
↪"coefs": coefs_true, "preds": target_fn(x)})

    plot_utils.plot_prediction_functions(x, pred_fns, x_train, y_train,␣
↪legend_loc="best")

if __name__ == '__main__':
    main()
```

[1]:
```
%run ridge_regression.t.py
```

```
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is
6.372644513104131e-08.
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is
2.8755654572009253e-11.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is
2.8755654572009253e-11.
.DEBUG: (Parameter w) Max rel error for partial deriv 1.0832498418250635e-09.
DEBUG: (Parameter b) Max rel error for partial deriv 6.458361260311016e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.006s

OK
```

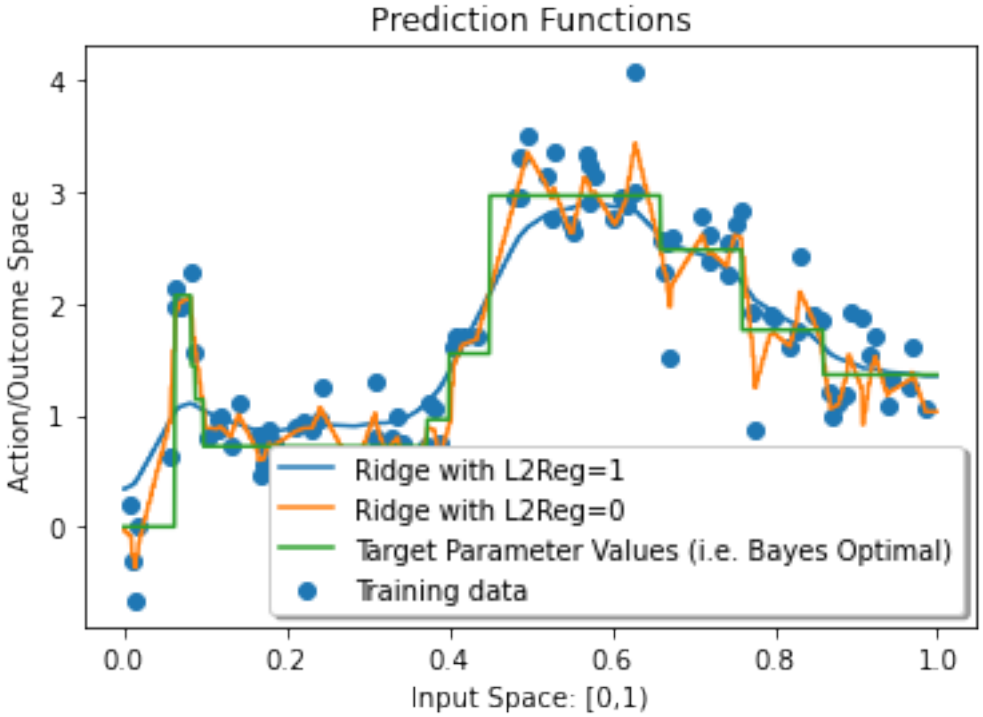
[3]:
```
%run ridge_regression.py
logging.getLogger('matplotlib.font_manager').disabled = True
```

```
Epoch  0 : Ave objective= 1.6094806213101527  Ave training loss:
0.8236515849382391
Epoch  50 : Ave objective= 0.3279977959462063  Ave training loss:
```

```
0.2417651352044313
Epoch  100 : Ave objective= 0.31649599409323076  Ave training loss:
0.21184374879140772
Epoch  150 : Ave objective= 0.31452890784244747  Ave training loss:
0.2037184825639585
Epoch  200 : Ave objective= 0.3121485084212521  Ave training loss:
0.20034182956314953
Epoch  250 : Ave objective= 0.31280373603957234  Ave training loss:
0.19877625082307304
Epoch  300 : Ave objective= 0.31248714459392785  Ave training loss:
0.19818356809561696
Epoch  350 : Ave objective= 0.3115031569492875  Ave training loss:
0.19765508280926666
Epoch  400 : Ave objective= 0.31067210531127976  Ave training loss:
0.19910064024346144
Epoch  450 : Ave objective= 0.3110554833622063  Ave training loss:
0.19774998980617353
Epoch  500 : Ave objective= 0.3102123976915999  Ave training loss:
0.19767856145932328
Epoch  550 : Ave objective= 0.30953596229096914  Ave training loss:
0.19895346084883084
Epoch  600 : Ave objective= 0.3098737660877904  Ave training loss:
0.19751472391091557
Epoch  650 : Ave objective= 0.3093150850808455  Ave training loss:
0.19817908091334666
Epoch  700 : Ave objective= 0.3083909295851297  Ave training loss:
0.19769534919964862
Epoch  750 : Ave objective= 0.3086036175478192  Ave training loss:
0.1977093852431293
Epoch  800 : Ave objective= 0.3083578705816468  Ave training loss:
0.19806412683777122
Epoch  850 : Ave objective= 0.30770838927223004  Ave training loss:
0.1980111792987666
Epoch  900 : Ave objective= 0.3078063091050808  Ave training loss:
0.1980079992986024
Epoch  950 : Ave objective= 0.3082859184800666  Ave training loss:
0.19838950013312925
Epoch  1000 : Ave objective= 0.30731243399444386  Ave training loss:
0.19856697674326151
Epoch  1050 : Ave objective= 0.30756780593877253  Ave training loss:
0.19826994478050178
Epoch  1100 : Ave objective= 0.3072806931251411  Ave training loss:
0.19830281861019144
Epoch  1150 : Ave objective= 0.3069821273999807  Ave training loss:
0.19820262441581704
Epoch  1200 : Ave objective= 0.30636961118199  Ave training loss:
0.19858780093191375
Epoch  1250 : Ave objective= 0.30721375873131973  Ave training loss:
```

```
0.19893497958744522
Epoch  1300 : Ave objective= 0.3058722447134926  Ave training loss:
0.199394484728296
Epoch  1350 : Ave objective= 0.30601599052107437  Ave training loss:
0.19928444465708506
Epoch  1400 : Ave objective= 0.30613201046323657  Ave training loss:
0.19860988095978852
Epoch  1450 : Ave objective= 0.3057938230657142  Ave training loss:
0.19875467075644532
Epoch  1500 : Ave objective= 0.3056803249217344  Ave training loss:
0.19913444351234513
Epoch  1550 : Ave objective= 0.3049570024219264  Ave training loss:
0.1999538172637944
Epoch  1600 : Ave objective= 0.30532621315949304  Ave training loss:
0.20000587406509143
Epoch  1650 : Ave objective= 0.30512372479389344  Ave training loss:
0.19926491574168348
Epoch  1700 : Ave objective= 0.30512865057760874  Ave training loss:
0.19966016044785803
Epoch  1750 : Ave objective= 0.30522086208902005  Ave training loss:
0.19964628147040353
Epoch  1800 : Ave objective= 0.3054030612989358  Ave training loss:
0.1995722944371397
Epoch  1850 : Ave objective= 0.30529700031436746  Ave training loss:
0.19954450696300843
Epoch  1900 : Ave objective= 0.30397444733770007  Ave training loss:
0.19955616786571795
Epoch  1950 : Ave objective= 0.3049864462176651  Ave training loss:
0.19938840951306924
Epoch  0 : Ave objective= 0.7480293137923937  Ave training loss:
0.40403905283902763
Epoch  50 : Ave objective= 0.12531880760265604  Ave training loss:
0.11299507360210885
Epoch  100 : Ave objective= 0.09097861100920873  Ave training loss:
0.13080076322913367
Epoch  150 : Ave objective= 0.0835764081421503  Ave training loss:
0.11156742506990712
Epoch  200 : Ave objective= 0.07411701764460152  Ave training loss:
0.07023988961800091
Epoch  250 : Ave objective= 0.07067633602316048  Ave training loss:
0.05853692783371531
Epoch  300 : Ave objective= 0.0633720787500745  Ave training loss:
0.054165150329533514
Epoch  350 : Ave objective= 0.05809762372635873  Ave training loss:
0.051974487926738956
Epoch  400 : Ave objective= 0.05668885490648498  Ave training loss:
0.04528040794802371
Epoch  450 : Ave objective= 0.051348149162253394  Ave training loss:
```

```
0.04328437440225366
```



Problem 4

Given the definition $y = Wx + b$, we can write any entry y_i in y as $y_i = W_{ij}x_j + b_i$. In the derivative

$$\frac{\partial J}{\partial W_{ij}} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial W_{ij}}$$

we can see the part

$$\frac{\partial y_r}{\partial W_{ij}} = \begin{cases} x_j & r = i \\ 0 & \text{else} \end{cases}$$

Therefore, the part of the derivative inside the summation $\frac{\partial J}{\partial y_r}\frac{\partial y_r}{\partial W_{ij}}$ has non-zero value only when $r = i$, i.e. $y_r = y_i$.

$$\frac{\partial J}{\partial W_{ij}} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$$

Problem 5

Based on our result for problem 4, the vectorized expression for $\frac{\partial J}{\partial W}$ is

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} x^T$$

where $\frac{\partial J}{\partial y}$ and x are column vectors.

Problem 6

First, the elementwise expression for the derivative of y with respect to x can be written as $\frac{\partial y_r}{\partial x_i} = W_{ri}$. Then the expression for the i-th entry $\frac{\partial J}{\partial x_i}$ is

$$\frac{\partial J}{\partial x_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\frac{\partial y_r}{\partial x_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} W_{ri} = (\frac{\partial J}{\partial y})^T W_{:,i}$$

where $W_{:,i}$ denotes the i-th column in the matrix W.

From the above expression, we can show that for the entire vector x,

$$\frac{\partial J}{\partial x} = W^T(\frac{\partial J}{\partial y})$$

Problem 7

The i-th entry in b can be written as

$$\frac{\partial J}{\partial b_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\frac{\partial y_r}{\partial b_i}$$

In this expression, $\frac{\partial y_r}{\partial b_i} = \begin{cases} 1 & r = i \\ 0 & \text{else} \end{cases}$, i.e. $\frac{\partial y_r}{\partial b_i} = \delta_{ri}$.

Thus the expression can be written as

$$\frac{\partial J}{\partial b_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\frac{\partial y_r}{\partial b_i} = \frac{\partial J}{\partial y_r}\delta_{ri} = \frac{\partial J}{\partial y_i}$$

Problem 8

Given the rule $(A \odot B)_i = A_i B_i$, for this problem, to show $\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$, we prove the dirivative of J with respect to the i-th entry of A satisfy our expression, i.e. $\frac{\partial J}{\partial A_i} = (\frac{\partial J}{\partial S} \odot \sigma'(A))_i = \frac{\partial J}{\partial S_i}\sigma'(A_i)$

Based on similar ideas from the previous problem, we can write

$$\frac{\partial J}{\partial A_i} = \sum_j \frac{\partial J}{\partial S_j}\frac{\partial S_j}{\partial A_i} = \sum_j \frac{\partial J}{\partial S_j}\delta_{ij}\sigma'(A_i) = \frac{\partial J}{\partial S_i}\sigma'(A_i)$$

Therefore, $\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$

Problem 9-11

[ ]:
```
class AffineNode(object):
    """Node implementing affine transformation (W,x,b)-->Wx+b, where W is a␣
↪matrix,
    and x and b are vectors
        Parameters:
        W: node for which W.out is a numpy array of shape (m,d)
        x: node for which x.out is a numpy array of shape (d)
```

```
        b: node for which b.out is a numpy array of shape (m) (i.e. vector of␣
↪length m)
    """
    def __init__(self, W, x, b, node_name):
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.W = W
        self.x = x
        self.b = b

    def forward(self):
        self.out = np.dot(self.W.out, self.x.out) + self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_W = np.outer(self.d_out, self.x.out)
        d_x = np.dot(self.W.out.T, self.d_out)
        d_b = self.d_out
        self.W.d_out += d_W
        self.x.d_out += d_x
        self.b.d_out += d_b

    def get_predecessors(self):
        return [self.W, self.x, self.b]


class TanhNode(object):
    """Node tanh(a), where tanh is applied elementwise to the array a
        Parameters:
        a: node for which a.out is a numpy array
    """
    def __init__(self, a, node_name):
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.a = a

    def forward(self):
        self.out = np.tanh(self.a.out)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_a = self.d_out * (1 - self.out ** 2)
        self.a.d_out += d_a
```

```
    def get_predecessors(self):
        return [self.a]
```

[ ]:
```
import matplotlib.pyplot as plt
import setup_problem
from sklearn.base import BaseEstimator, RegressorMixin
import numpy as np
import nodes_zx1137 as nodes
import graph
import plot_utils
import pdb
#pdb.set_trace()


class MLPRegression(BaseEstimator, RegressorMixin):
    """ MLP regression with computation graph """
    def __init__(self, num_hidden_units=10, step_size=.005, init_param_scale=0.
↪01, max_num_epochs = 5000):
        self.num_hidden_units = num_hidden_units
        self.init_param_scale = init_param_scale
        self.max_num_epochs = max_num_epochs
        self.step_size = step_size

        # Build computation graph
        # data
        self.x = nodes.ValueNode(node_name="x") # to hold a vector input
        self.y = nodes.ValueNode(node_name="y") # to hold a scalar response

        # parameters
        self.W1 = nodes.ValueNode(node_name="W1")
        self.b1 = nodes.ValueNode(node_name="b1")
        self.w2 = nodes.ValueNode(node_name="w2")
        self.b2 = nodes.ValueNode(node_name="b2")

        # hidden layer
        self.L = nodes.AffineNode(W=self.W1, x=self.x, b=self.b1,␣
↪node_name="affine")
        self.h = nodes.TanhNode(a=self.L, node_name="tanh")

        # prediction
        self.f = nodes.VectorScalarAffineNode(x=self.h, w=self.w2, b=self.b2,␣
↪node_name="prediction")

        # objective
        self.J = nodes.SquaredL2DistanceNode(a=self.y, b=self.f,␣
↪node_name="objective")
```

```
        self.graph = graph.ComputationGraphFunction(inputs=[self.x],␣
↪outcomes=[self.y], parameters=[self.W1,self.b1,self.w2,self.b2],␣
↪prediction=self.f, objective=self.J)

    def fit(self, X, y):
        num_instances, num_ftrs = X.shape
        y = y.reshape(-1)
        s = self.init_param_scale
        init_values = {"W1": s * np.random.standard_normal((self.
↪num_hidden_units, num_ftrs)),
                       "b1": s * np.random.standard_normal((self.
↪num_hidden_units)),
                       "w2": s * np.random.standard_normal((self.
↪num_hidden_units)),
                       "b2": s * np.array(np.random.randn()) }

        self.graph.set_parameters(init_values)

        for epoch in range(self.max_num_epochs):
            shuffle = np.random.permutation(num_instances)
            epoch_obj_tot = 0.0
            for j in shuffle:
                obj, grads = self.graph.get_gradients(input_values = {"x":␣
↪X[j]},
                                                      outcome_values = {"y":␣
↪y[j]})
                #print(obj)
                epoch_obj_tot += obj
                # Take step in negative gradient direction
                steps = {}
                for param_name in grads:
                    steps[param_name] = -self.step_size * grads[param_name]
                self.graph.increment_parameters(steps)
                #pdb.set_trace()

            if epoch % 50 == 0:
                train_loss = sum((y - self.predict(X,y)) **2)/num_instances
                print("Epoch ",epoch,": Ave objective=",epoch_obj_tot/
↪num_instances," Ave training loss: ",train_loss)

    def predict(self, X, y=None):
        try:
            getattr(self, "graph")
        except AttributeError:
```

```
            raise RuntimeError("You must train classifer before predicting data!
↪")

        num_instances = X.shape[0]
        preds = np.zeros(num_instances)
        for j in range(num_instances):
            preds[j] = self.graph.get_prediction(input_values={"x":X[j]})

        return preds



def main():
    data_fname = "data.pickle"
    x_train, y_train, x_val, y_val, target_fn, coefs_true, featurize =␣
↪setup_problem.load_problem(data_fname)

    # Generate features
    X_train = featurize(x_train)
    X_val = featurize(x_val)

    # Let's plot prediction functions and compare coefficients for several fits
    # and the target function.
    pred_fns = []
    x = np.sort(np.concatenate([np.arange(0,1,.001), x_train]))

    pred_fns.append({"name": "Target Parameter Values (i.e. Bayes Optimal)",␣
↪"coefs": coefs_true, "preds": target_fn(x)})

    estimator = MLPRegression(num_hidden_units=10, step_size=0.001,␣
↪init_param_scale=.0005,  max_num_epochs=5000)
    x_train_as_column_vector = x_train.reshape(x_train.shape[0],1) # fit␣
↪expects a 2-dim array
    x_as_column_vector = x.reshape(x.shape[0],1) # fit expects a 2-dim array
    estimator.fit(x_train_as_column_vector, y_train)
    name = "MLP regression - no features"
    pred_fns.append({"name":name, "preds": estimator.
↪predict(x_as_column_vector) })
    #plot_utils.plot_prediction_functions(x, pred_fns, x_train, y_train,␣
↪legend_loc="best")
    if (1==1):
        X = featurize(x)
        estimator = MLPRegression(num_hidden_units=10, step_size=0.0005,␣
↪init_param_scale=.01,  max_num_epochs=500)
        estimator.fit(X_train, y_train)
        name = "MLP regression - with features"
```

```
        pred_fns.append({"name":name, "preds": estimator.predict(X) })
        plot_utils.plot_prediction_functions(x, pred_fns, x_train, y_train,␣
 ↪legend_loc="best")

if __name__ == '__main__':
    main()
```

[1]:
```
%run mlp_regression.t.py
```

```
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is
2.6727224425812336e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is
1.2462746693581928e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is
1.6365789325759677e-09.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is
6.926211828825294e-09.
.DEBUG: (Parameter W1) Max rel error for partial deriv 7.757046786710294e-06.
DEBUG: (Parameter b1) Max rel error for partial deriv 6.48927428756324e-07.
DEBUG: (Parameter w2) Max rel error for partial deriv 1.7955448355231087e-09.
DEBUG: (Parameter b2) Max rel error for partial deriv 5.873348475102835e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.010s

OK
```

[4]:
```
%run mlp_regression.py
logging.getLogger('matplotlib.font_manager').disabled = True
```

```
Epoch  0 : Ave objective= 3.137516391575231  Ave training loss:
2.7222678638542765
Epoch  50 : Ave objective= 0.9453888583974991  Ave training loss:
0.9435144534800395
Epoch  100 : Ave objective= 0.9450755362110279  Ave training loss:
0.9432050024618686
Epoch  150 : Ave objective= 0.9419951828013727  Ave training loss:
0.9400330084192745
Epoch  200 : Ave objective= 0.9155739653251257  Ave training loss:
0.9128468688299803
Epoch  250 : Ave objective= 0.8241666838576367  Ave training loss:
0.8200423909145509
Epoch  300 : Ave objective= 0.7736395414396828  Ave training loss:
0.769560110586592
Epoch  350 : Ave objective= 0.7669902623365532  Ave training loss:
0.7626854449456703
Epoch  400 : Ave objective= 0.762091631250786  Ave training loss:
0.7578812526409072
```

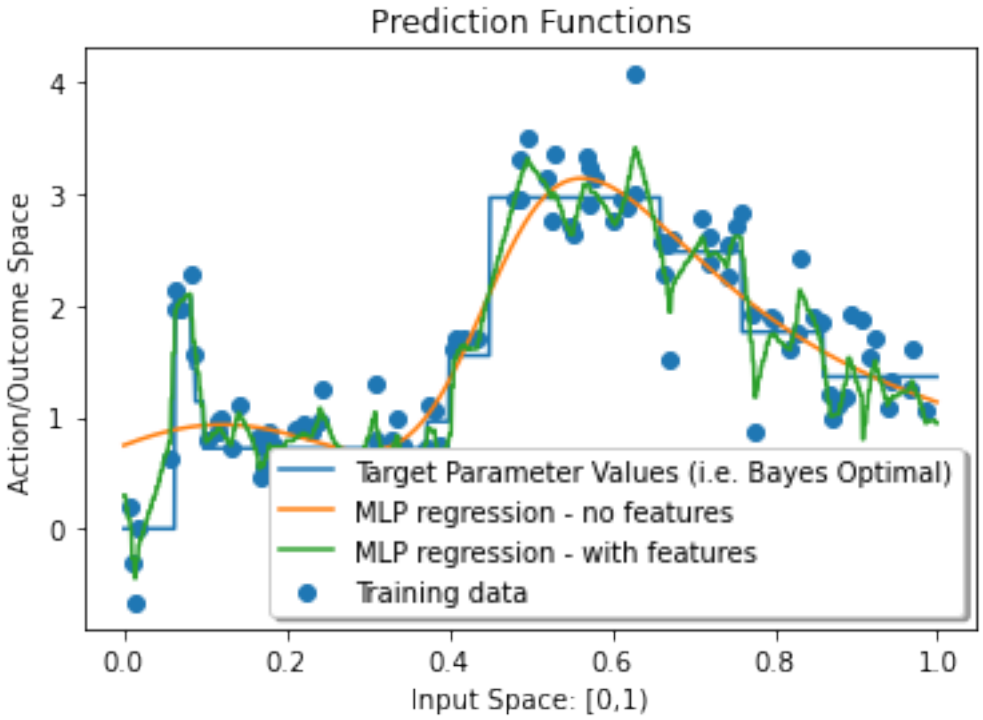
```
Epoch  450 : Ave objective= 0.7559249967010055  Ave training loss:
0.7518482998376459
Epoch  500 : Ave objective= 0.7483226554000626  Ave training loss:
0.7446265725733725
Epoch  550 : Ave objective= 0.7405672136626323  Ave training loss:
0.7365476152473991
Epoch  600 : Ave objective= 0.73165475561857  Ave training loss:
0.7279107979330908
Epoch  650 : Ave objective= 0.7229064748217219  Ave training loss:
0.7191580691895192
Epoch  700 : Ave objective= 0.7144902765253807  Ave training loss:
0.7106201108495039
Epoch  750 : Ave objective= 0.7060936010647947  Ave training loss:
0.7027625219572831
Epoch  800 : Ave objective= 0.6995418717829005  Ave training loss:
0.6955890017147723
Epoch  850 : Ave objective= 0.693231113223115  Ave training loss:
0.6893465435430121
Epoch  900 : Ave objective= 0.687716429437604  Ave training loss:
0.6837866046880056
Epoch  950 : Ave objective= 0.6827330002724764  Ave training loss:
0.6788743596875594
Epoch  1000 : Ave objective= 0.6782506072414097  Ave training loss:
0.6743233601527687
Epoch  1050 : Ave objective= 0.6729696040769835  Ave training loss:
0.669928603291718
Epoch  1100 : Ave objective= 0.6690892556428207  Ave training loss:
0.664921001954736
Epoch  1150 : Ave objective= 0.6635115290643725  Ave training loss:
0.6592624756457803
Epoch  1200 : Ave objective= 0.6569512744488532  Ave training loss:
0.6524487807364113
Epoch  1250 : Ave objective= 0.6479152382126745  Ave training loss:
0.6432642247580777
Epoch  1300 : Ave objective= 0.6364255201627868  Ave training loss:
0.6319892007535117
Epoch  1350 : Ave objective= 0.6222730125416462  Ave training loss:
0.6169863529711642
Epoch  1400 : Ave objective= 0.6040840591315946  Ave training loss:
0.5989056144803573
Epoch  1450 : Ave objective= 0.5835506401251223  Ave training loss:
0.5770374072605491
Epoch  1500 : Ave objective= 0.5578523041365594  Ave training loss:
0.5509867882316521
Epoch  1550 : Ave objective= 0.5262059791758896  Ave training loss:
0.522095832649203
Epoch  1600 : Ave objective= 0.4974530675529656  Ave training loss:
0.4899083365889365
```

```
Epoch  1650 : Ave objective= 0.4647539318248511  Ave training loss:
0.4568021973757934
Epoch  1700 : Ave objective= 0.43434791332907663  Ave training loss:
0.42571798180719483
Epoch  1750 : Ave objective= 0.4047494869542201  Ave training loss:
0.39992613129756394
Epoch  1800 : Ave objective= 0.3825566739968535  Ave training loss:
0.37478767282545794
Epoch  1850 : Ave objective= 0.3640545575789637  Ave training loss:
0.3568310761958429
Epoch  1900 : Ave objective= 0.34971594408444157  Ave training loss:
0.3434060205440905
Epoch  1950 : Ave objective= 0.3406576940241557  Ave training loss:
0.33342122675835045
Epoch  2000 : Ave objective= 0.3331512527150483  Ave training loss:
0.3271066381688972
Epoch  2050 : Ave objective= 0.3283402212664162  Ave training loss:
0.32171194860011454
Epoch  2100 : Ave objective= 0.32215689509935175  Ave training loss:
0.3164335626951301
Epoch  2150 : Ave objective= 0.31776050802016853  Ave training loss:
0.31627187077233465
Epoch  2200 : Ave objective= 0.31349884552685525  Ave training loss:
0.31172163102758416
Epoch  2250 : Ave objective= 0.31242785152280933  Ave training loss:
0.30737287215794473
Epoch  2300 : Ave objective= 0.31098707949874005  Ave training loss:
0.3036595300330174
Epoch  2350 : Ave objective= 0.3055593084871608  Ave training loss:
0.3017341644300186
Epoch  2400 : Ave objective= 0.30488093133016003  Ave training loss:
0.2990894863443083
Epoch  2450 : Ave objective= 0.29919686013255636  Ave training loss:
0.30028179999252136
Epoch  2500 : Ave objective= 0.29907330116861125  Ave training loss:
0.29551111855372736
Epoch  2550 : Ave objective= 0.2988025007675715  Ave training loss:
0.2933350394657824
Epoch  2600 : Ave objective= 0.29779170932477167  Ave training loss:
0.2915620758480712
Epoch  2650 : Ave objective= 0.2947261176645541  Ave training loss:
0.28917277916904516
Epoch  2700 : Ave objective= 0.2933139049062625  Ave training loss:
0.2880350609466849
Epoch  2750 : Ave objective= 0.2912573903645786  Ave training loss:
0.2855759136126734
Epoch  2800 : Ave objective= 0.29023533639332366  Ave training loss:
0.28391572915960905
```

```
Epoch  2850 : Ave objective= 0.2892298353882299  Ave training loss:
0.28433435587897327
Epoch  2900 : Ave objective= 0.2863975240975124  Ave training loss:
0.2808192822927724
Epoch  2950 : Ave objective= 0.2852310141917871  Ave training loss:
0.2793354166133381
Epoch  3000 : Ave objective= 0.2815364303349382  Ave training loss:
0.2784851668849465
Epoch  3050 : Ave objective= 0.28023828125662503  Ave training loss:
0.2779909956954594
Epoch  3100 : Ave objective= 0.27891149964867334  Ave training loss:
0.2750388916060075
Epoch  3150 : Ave objective= 0.2789955184929161  Ave training loss:
0.2739721488393313
Epoch  3200 : Ave objective= 0.27645933391463706  Ave training loss:
0.2734579917537181
Epoch  3250 : Ave objective= 0.27640491250548643  Ave training loss:
0.2710998317934567
Epoch  3300 : Ave objective= 0.2748622152923085  Ave training loss:
0.2697849726722456
Epoch  3350 : Ave objective= 0.2737176553260719  Ave training loss:
0.26857714620284606
Epoch  3400 : Ave objective= 0.27146613164193767  Ave training loss:
0.2689247392246583
Epoch  3450 : Ave objective= 0.2696311284706083  Ave training loss:
0.26839403720785077
Epoch  3500 : Ave objective= 0.2675941039479308  Ave training loss:
0.26558854367062484
Epoch  3550 : Ave objective= 0.2687487112596385  Ave training loss:
0.26435048762064806
Epoch  3600 : Ave objective= 0.26590618095853086  Ave training loss:
0.2647450379821079
Epoch  3650 : Ave objective= 0.2678510398591245  Ave training loss:
0.2628692717222511
Epoch  3700 : Ave objective= 0.2645966766934721  Ave training loss:
0.2615990257556361
Epoch  3750 : Ave objective= 0.26316108430692475  Ave training loss:
0.26126092293409486
Epoch  3800 : Ave objective= 0.26341760786848717  Ave training loss:
0.25905380059104904
Epoch  3850 : Ave objective= 0.2631277720515623  Ave training loss:
0.2579485570525661
Epoch  3900 : Ave objective= 0.2612898621623534  Ave training loss:
0.2588143092891282
Epoch  3950 : Ave objective= 0.2610454177904173  Ave training loss:
0.2561806270382172
Epoch  4000 : Ave objective= 0.2582925107711596  Ave training loss:
0.25539201056734767
```

```
Epoch  4050 : Ave objective= 0.2572431033759671  Ave training loss:
0.2558102322075641
Epoch  4100 : Ave objective= 0.2586842474336194  Ave training loss:
0.2535538997499953
Epoch  4150 : Ave objective= 0.2576558184767501  Ave training loss:
0.2527640328870781
Epoch  4200 : Ave objective= 0.2558839732695167  Ave training loss:
0.2523845076946139
Epoch  4250 : Ave objective= 0.2557948287001455  Ave training loss:
0.2511707437296304
Epoch  4300 : Ave objective= 0.25469554160312896  Ave training loss:
0.25063278955977186
Epoch  4350 : Ave objective= 0.25452490241521153  Ave training loss:
0.24945726881354094
Epoch  4400 : Ave objective= 0.2535865000252626  Ave training loss:
0.24878551256798875
Epoch  4450 : Ave objective= 0.25290970356054987  Ave training loss:
0.24798879683635422
Epoch  4500 : Ave objective= 0.25131362156478754  Ave training loss:
0.24850547360947003
Epoch  4550 : Ave objective= 0.2503602331422513  Ave training loss:
0.24699068361944426
Epoch  4600 : Ave objective= 0.24963334041114124  Ave training loss:
0.24702900876331393
Epoch  4650 : Ave objective= 0.24974842190562996  Ave training loss:
0.2451883736863967
Epoch  4700 : Ave objective= 0.24936992700031674  Ave training loss:
0.24448763831093995
Epoch  4750 : Ave objective= 0.24903905248332667  Ave training loss:
0.24391478582267084
Epoch  4800 : Ave objective= 0.24823475752764243  Ave training loss:
0.2432244407535074
Epoch  4850 : Ave objective= 0.24693002138529907  Ave training loss:
0.2425504303178255
Epoch  4900 : Ave objective= 0.24657090884684896  Ave training loss:
0.24212550482009576
Epoch  4950 : Ave objective= 0.2465022711516545  Ave training loss:
0.24135035321254816
Epoch  0 : Ave objective= 3.206540758880964  Ave training loss:
2.7600393335666866
Epoch  50 : Ave objective= 0.15007484988457595  Ave training loss:
0.14527634097721806
Epoch  100 : Ave objective= 0.11680875705102287  Ave training loss:
0.10942355055645007
Epoch  150 : Ave objective= 0.10051792450477723  Ave training loss:
0.08974775819159062
Epoch  200 : Ave objective= 0.08195475487224442  Ave training loss:
0.07623225666388646
```

```
Epoch  250 : Ave objective= 0.07625044503815824  Ave training loss:
0.07161626750269053
Epoch  300 : Ave objective= 0.06319844228066172  Ave training loss:
0.09582916210107607
Epoch  350 : Ave objective= 0.05629184872435556  Ave training loss:
0.052125969830460084
Epoch  400 : Ave objective= 0.05347634134934582  Ave training loss:
0.04388093422255334
Epoch  450 : Ave objective= 0.047113424587467284  Ave training loss:
0.040245012029976275
```




```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-4-9ff4424cf970> in <module>
  1 get_ipython().run_line_magic('run', 'mlp_regression.py')
----> 2 logging.getLogger('matplotlib.font_manager').disabled = True


NameError: name 'logging' is not defined
```

Problem 12-14

[ ]:
```
class SoftmaxNode(object):
    """ Softmax node
        Parameters:
        z: node for which z.out is a numpy array
    """
    def __init__(self, z, node_name):
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.z = z

    def forward(self):
        total_prob = np.sum([np.exp(self.z.out[k]) for k in range(len(self.z.
 ↪out))])
        self.out = np.array([np.exp(self.z.out[k])/total_prob for k in␣
 ↪range(len(self.z.out))])
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        derivative = np.empty([len(self.z.out), len(self.out)])
        for i in range(len(self.z.out)):
            for j in range(len(self.out)):
                if i == j:
                    derivative[i][j] = self.out[i] * (1-self.out[i])
                else:
                    derivative[i][j] = -self.out[i] * self.out[j]

        d_z = np.dot(self.d_out, derivative)
        self.z.d_out += d_z

    def get_predecessors(self):
        return [self.z]


class NLLNode(object):
    """ Node computing NLL loss between 2 arrays.
        Parameters:
        y_hat: a node that contains all predictions
        y_true: a node that contains all labels
    """
    def __init__(self, y_hat, y_true, node_name):
        self.node_name = node_name
        self.out = None
```

```
        self.d_out = None
        self.y_hat = y_hat
        self.y_true = y_true

    def forward(self):
        self.out = -np.log(self.y_hat.out[self.y_true.out])
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_ytrue = self.d_out * (-self.y_hat.out[self.y_true.out]/np.exp(-self.
↪out))
        d_yhat = np.zeros(self.y_hat.out.shape)
        d_yhat[self.y_true.out] = 1
        d_yhat = self.d_out * (-d_yhat/np.exp(-self.out))
        self.y_hat.d_out += d_yhat
        self.y_true.d_out += d_ytrue

    def get_predecessors(self):
        return [self.y_hat, self.y_true]
```

[ ]:
```
import setup_problem
from sklearn.base import BaseEstimator, RegressorMixin
try:
    from sklearn.datasets.samples_generator import make_blobs
except:
    from sklearn.datasets import make_blobs
import numpy as np
import nodes_zx1137 as nodes
import graph

def calculate_nll(y_preds, y):
    """
    Function that calculate the average NLL loss
    :param y_preds: N * C probability array
    :param y: N int array
    :return:
    """
    return np.mean(-np.log(y_preds)[np.arange(len(y)),y])


class MulticlassClassifier(BaseEstimator, RegressorMixin):
    """ Multiclass prediction """
    def __init__(self, num_hidden_units=10, step_size=.005, init_param_scale=0.
↪01, max_num_epochs = 1000, num_class=3):
        self.num_hidden_units = num_hidden_units
        self.init_param_scale = init_param_scale
```

```
        self.max_num_epochs = max_num_epochs
        self.step_size = step_size
        self.num_class = num_class

        # Build computation graph
        # data
        self.x = nodes.ValueNode(node_name="x") # to hold a vector input
        self.y = nodes.ValueNode(node_name="y") # to hold a scalar response

        # parameters
        self.W1 = nodes.ValueNode(node_name="W1")
        self.b1 = nodes.ValueNode(node_name="b1")
        self.W2 = nodes.ValueNode(node_name="W2")
        self.b2 = nodes.ValueNode(node_name="b2")

        # hidden layer
        self.L = nodes.AffineNode(W=self.W1, x=self.x, b=self.b1,␣
↪node_name="affineW1b1")
        self.h = nodes.TanhNode(a=self.L, node_name="tanh")
        self.z = nodes.AffineNode(W=self.W2, x=self.h, b=self.b2,␣
↪node_name="affineb2w2")

        # prediction
        self.f = nodes.SoftmaxNode(z=self.z, node_name="softmax")

        # objective
        self.J = nodes.NLLNode(y_hat=self.f, y_true=self.y,␣
↪node_name='objective')

        self.graph = graph.ComputationGraphFunction(inputs=[self.x],␣
↪outcomes=[self.y], parameters=[self.W1,self.b1,self.W2,self.b2],␣
↪prediction=self.f, objective=self.J)

    def fit(self, X, y):
        num_instances, num_ftrs = X.shape
        y = y.reshape(-1)
        s = self.init_param_scale
        init_values = {"W1": s * np.random.standard_normal((self.
↪num_hidden_units, num_ftrs)),
                       "b1": s * np.random.standard_normal((self.
↪num_hidden_units)),
                       "W2": np.random.standard_normal((self.num_class, self.
↪num_hidden_units)),
                       "b2": np.array(np.random.randn(self.num_class)) }
        self.graph.set_parameters(init_values)
```

```
        for epoch in range(self.max_num_epochs):
            shuffle = np.random.permutation(num_instances)
            epoch_obj_tot = 0.0
            for j in shuffle:
                obj, grads = self.graph.get_gradients(input_values = {"x":␣
↪X[j]},
                                                      outcome_values = {"y":␣
↪y[j]})
                #print(obj)
                epoch_obj_tot += obj
                # Take step in negative gradient direction
                steps = {}
                for param_name in grads:
                    steps[param_name] = -self.step_size * grads[param_name]
                self.graph.increment_parameters(steps)
                #pdb.set_trace()

            if epoch % 50 == 0:
                train_loss = calculate_nll(self.predict(X,y), y)
                print("Epoch ",epoch," Ave training loss: ",train_loss)

    def predict(self, X, y=None):
        try:
            getattr(self, "graph")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting data!
↪")

        num_instances = X.shape[0]
        preds = []
        for j in range(num_instances):
            preds.append(self.graph.get_prediction(input_values={"x":X[j]}).
↪reshape(1,-1))

        return np.concatenate(preds, axis=0)



def main():
    # load the data from HW5
    np.random.seed(2)
    X, y = make_blobs(n_samples=500, cluster_std=.25, centers=np.array([(-3,␣
↪1), (0, 2), (3, 1)]))
    training_X = X[:300]
    training_y = y[:300]
    test_X = X[300:]
    test_y = y[300:]
```

```
    # train the model
    estimator = MulticlassClassifier()
    estimator.fit(training_X, training_y)

    # report test accuracy
    test_acc = np.sum(np.argmax(estimator.predict(test_X), axis=1)==test_y)/
 ↪len(test_y)
    print("Test set accuracy = {:.3f}".format(test_acc))


if __name__ == '__main__':
    main()
```

[1]:
```
%run multiclass.t.py
```

```
C:\Users\kalle\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:143:
FutureWarning: The sklearn.datasets.samples_generator module is  deprecated in
version 0.22 and will be removed in version 0.24. The corresponding classes /
functions should instead be imported from sklearn.datasets. Anything that cannot
be imported from sklearn.datasets is now part of the private API.
  warnings.warn(message, FutureWarning)
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is
1.5385742524215873e-09.
.DEBUG: (Parameter W1) Max rel error for partial deriv 2.1791491311701626e-07.
DEBUG: (Parameter b1) Max rel error for partial deriv 5.4507464417531275e-08.
DEBUG: (Parameter W2) Max rel error for partial deriv 6.754800815066246e-08.
DEBUG: (Parameter b2) Max rel error for partial deriv 2.205053947010376e-08.
.
----------------------------------------------------------------------
Ran 2 tests in 0.012s

OK
```

[2]:
```
%run multiclass.py
```

```
Epoch  0  Ave training loss:  0.10767753468425852
Epoch  50  Ave training loss:  0.0037402729498018897
Epoch  100  Ave training loss:  0.0019509875089186069
Epoch  150  Ave training loss:  0.00131892201003299
Epoch  200  Ave training loss:  0.0009947600104512856
Epoch  250  Ave training loss:  0.0007975221227263997
Epoch  300  Ave training loss:  0.0006649220947379011
Epoch  350  Ave training loss:  0.0005697138957458585
Epoch  400  Ave training loss:  0.0004980771960410216
Epoch  450  Ave training loss:  0.00044225221211177576
Epoch  500  Ave training loss:  0.0003975450315101259
Epoch  550  Ave training loss:  0.0003609495175393885
```

```
Epoch  600  Ave training loss:  0.0003304520224436157
Epoch  650  Ave training loss:  0.0003046529432352649
Epoch  700  Ave training loss:  0.0002825495526238341
Epoch  750  Ave training loss:  0.0002634047943162139
Epoch  800  Ave training loss:  0.00024666486030361454
Epoch  850  Ave training loss:  0.0002319056839595022
Epoch  900  Ave training loss:  0.0002187970217752761
Epoch  950  Ave training loss:  0.00020707801611844357
Test set accuracy = 1.000
```