0.1 HW 7 Joby George

0.2 DS GA 1003 Due 5/6/22

1 Problems 1 - 3:

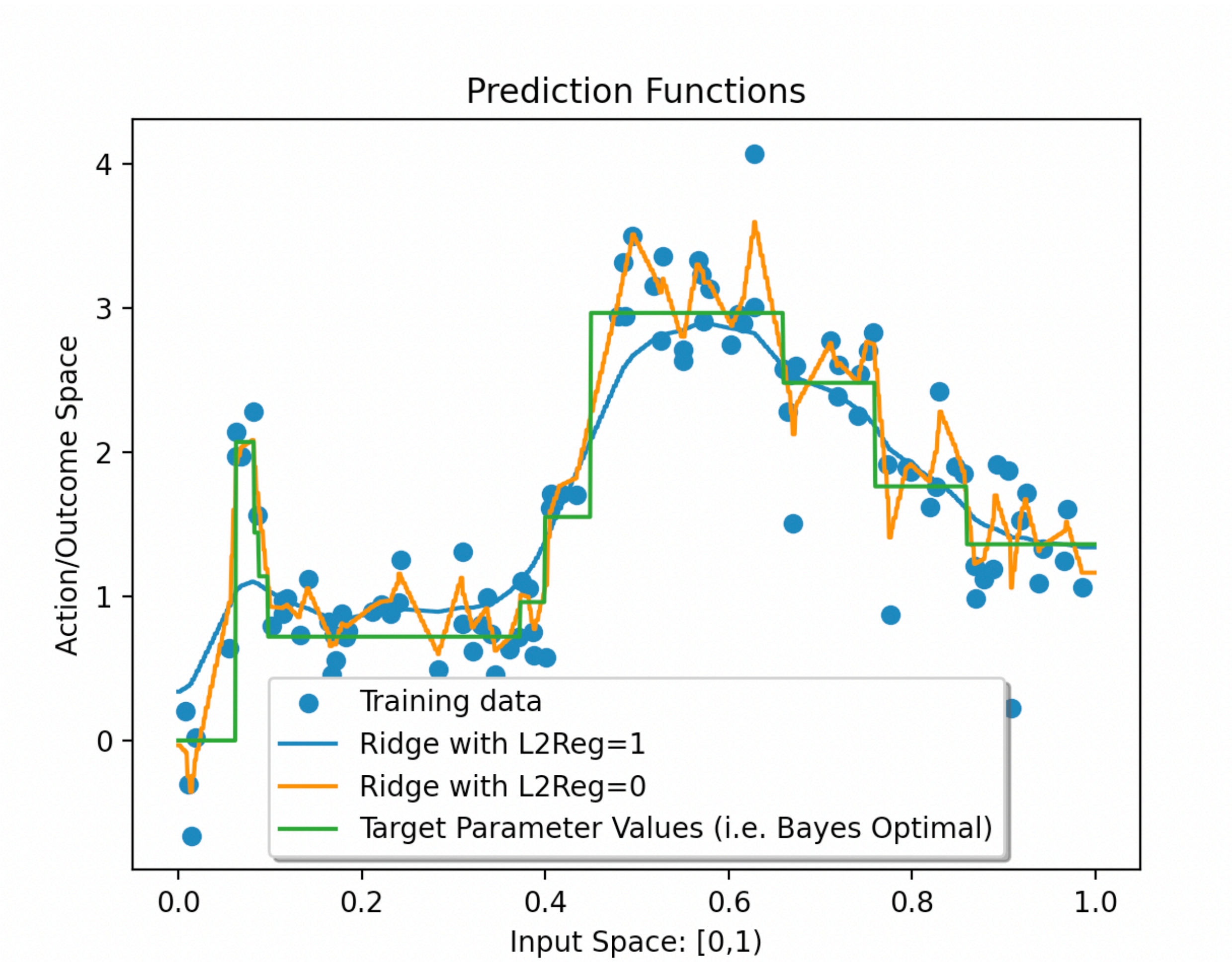
Make updates to the nodes.py and ridge_regression.py functions, running the script via terminal, paste the screenshot of the resulting test to prove you have fixed the scripts.

Done below.

```
(base) jobygeorge@10-19-35-86 hw7 % python3 ridge_regression.t.py
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 1.4649875
9792306e-09.
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 5.8386715466
11587e-10.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 5.83867154661
1587e-10.
.DEBUG: (Parameter w) Max rel error for partial deriv 5.464895182951349e-09.
DEBUG: (Parameter b) Max rel error for partial deriv 8.144792010002911e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
(base) jobygeorge@10-19-35-86 hw7 %
```

1.1 Output of ridge_regression.py

The graph generated by running *ridge_regression.py* is produced below:



The average square errors reported for the two lambdas were: .2 after 1950 epochs using $\lambda = 0$, and .056 after 450 epochs using $\lambda = 1$

In [ ]:
```
## Problem 1  L2NormPenaltyNode

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.out = self.l2_reg = np.array(l2_reg)
        self.w = w

    def forward(self):
```

```python
        self.out = self.l2_reg * np.dot(self.w.out, self.w.out)
        self.d_out = np.zeros(self.out.shape)
        return(self.out)

    def backward(self):
        self.w.d_out = 2*self.l2_reg*self.d_out*self.w.out
        pass

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

## Problem 2 SumNode
class SumNode(object):
    """ Node computing a + b, for numpy arrays a and b"""
    """
    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)
    """
    def __init__(self, a, b, node_name):
        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):
        self.a.d_out += self.d_out
        self.b.d_out += self.d_out
        return self.d_out
    def get_predecessors(self):
        return([self.a, self.b])

## Problem 3 Graph
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")
        self.y = nodes.ValueNode(node_name="y")
        self.w = nodes.ValueNode(node_name="w")
        self.b = nodes.ValueNode(node_name="b")

        self.prediction = nodes.VectorScalarAffineNode(
            x=self.x,
            w=self.w,
            b=self.b,
            node_name="prediction")
```

```python
        # Build computation graph
        residual = nodes.SquaredL2DistanceNode(
            a=self.prediction,
            b=self.y,
            node_name="square loss")

        regularization =  nodes.L2NormPenaltyNode(
            l2_reg,
            self.w,
            node_name='L2NormPenalty')

        self.objective = nodes.SumNode(
            residual,
            regularization, node_name = 'loss')

        self.graph = graph.ComputationGraphFunction(
            inputs = [self.x],
            outcomes= [self.y],
            parameters=[self.w,self.b],
            prediction=self.prediction,
            objective = self.objective)
```

executed in 131ms, finished 16:05:41 2022-05-06

2 Problem 4:

Show that $\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$ where $x = (x_1, \ldots, x_d)^T$

2.1 Problem 4 answer:

We know:

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

$$y_i = W_{ij} @ x + b_i \tag{2}$$

Therefore:

$$\frac{\partial y_i}{\partial W_{ij}} = x_j \tag{3}$$

Where x_j is the jth element of x. Re-expressing this we get:

$$\frac{\partial J}{\partial W_{ij}} = (\frac{\partial J}{\partial y_i} x_j)_{ij} \tag{4}$$

Where x_j is the jth entry of x and $(\frac{\partial J}{\partial y_i} x_j)_{ij}$ is the i jth entry in the m x d matrix.

2.1.1 Q.E.D.

3 Problem 5

Give a vectorized expression for $\frac{\partial J}{\partial W}$ in terms of the column vectors $\frac{\partial J}{\partial y}$ and x.

3.1 Problem 5 Answer

Looking at an example of the matrix from problem four, we see that the first entries by row would be:

$$[\frac{\partial J}{\partial W_{11}}]_{11} = \frac{\partial J}{\partial y_1} * x_1 \tag{5}$$

and continuing we see:

$$[\frac{\partial J}{\partial W_{12}}]_{12} = \frac{\partial J}{\partial y_1} * x_2 \tag{6}$$

This means the first row of our matrix is the first entry of $\frac{\partial J}{\partial y_1} \otimes$ x.

To get the ith row of $\frac{\partial J}{\partial W_i}$ all we have to do is take $y_i \otimes$ x.

Rather than keeping this in a non-vectorized format, we can simplify $\frac{\partial J}{\partial W}$ to be:

$$\frac{\partial J}{\partial y} \otimes x_j \tag{7}$$

3.1.1 Q.E.D

4 Problem 6

In the usual way, define $\frac{\partial J}{\partial x} \in R^d$, whose i'th entry is $\frac{\partial J}{\partial x_i}$. Show that

$$\frac{\partial J}{\partial x} = W^T \left(\frac{\partial J}{\partial y} \right) \tag{8}$$

Note, if x is just data, technically we won't need this derivative. However, in a multilayer perceptron, x may actually be the output of a previous hidden layer, in which case we will need to propagate the derivative through x as well.

4.1 Problem 6 answer:

$$\frac{\partial J}{\partial x_i} = \frac{\partial J}{\partial y} * \frac{\partial y}{\partial x_i} \tag{9}$$

The rationale behind taking the entire $\frac{\partial J}{\partial y}$ when we change any given x_i in our data, we observe that it would impact all m elements in W at column i, due to the nature of matrix vector multiplication.

We observe that $\frac{\partial y}{\partial x_i}$ is the ith column of W, meaning:

$$\frac{\partial y}{\partial x_i} = W_i^T \frac{\partial J}{\partial y} \tag{10}$$

For the whole vector $\frac{\partial y}{\partial x}$ we can solve by taking:

$$\frac{\partial y}{\partial x} = W^T \frac{\partial J}{\partial y} \tag{11}$$

4.1.1 Q.E.D.

5 Problem 7

Show that $\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$ where $\frac{\partial J}{\partial b}$ is defined in the usual way.

5.1 Problem 7 answer

By chain rule:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} * \frac{\partial y}{\partial b} \tag{12}$$

since Y = Wx + b $\frac{\partial y}{\partial b}$ = 1 therefore:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \tag{13}$$

5.1.1 Q.E.D

6 Problem 8

Show that $\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$, where we're using $\odot$ to represent the **Hadamard product**.

If A and B are arrays of the same shape, then their Hadamardproduct $A \odot B$ is an array with the same shape as A and B, and for which $(A \odot B)_i = A_i . B_i$. That is, it's just the array formed by multiplying corresponding elements of A and B. Conveniently, in numpy if A and B are arrays of the same shape, then A*B is their Hadamard product.

6.1 Problem 8 Answer

We know that the derivative of σ(a) is σ'(a). Using the chain rule we can express:

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} * \frac{\partial S}{\partial A} \tag{14}$$

The derivative of S with respect to A is exactly σ'(a)

What does σ'(a) represent? Since σ is an element wise transformation, each element in matrix A is multiplied by a scalar at the same index in S. σ' also completes this same element operation, meaning what we have is

Therefore what we have is:

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A) \tag{15}$$

where $\odot$ is the Hadamard product

6.1.1 Q.E.D

7 Problem 9 - 11

Complete the class AffineNode in nodes.py. Be sure to propagate the gradient with respect to x as well, since when we stack these layers, x will itself be the output of another node that depends on our optimization parameters. If your code is correct, you should be able to pass test AffineNode in mlp regression.t.py. Please attach a screenshot that shows the test results for this question.

Complete the class TanhNode in nodes.py. As you'll recall, d tanh(x) = 1−tanh2 x. Note dx that in the forward pass, we'll already have computed tanh of the input and stored it in self.out. So make sure to use self.out and not recalculate it in the backward pass. If your code is correct, you should be able to pass test TanhNode in mlp regression.t.py. Please attach a screenshot that shows the test results for this question.

Implement an MLP by completing the skeleton code in mlp regression.py and making use of the nodes above. Your code should pass the tests provided in mlp regression.t.py. Note that to break the symmetry of the problem, we initialize our weights to small random values, rather than all zeros, as we often do for convex optimization problems. Run the MLP for the two settings given in the main() function and report the average training error. Note that with an MLP, we can take the original scalar as input, in the hopes that it will learn nonlinear features on its own, using the hidden layers. In practice, it is quite challenging to get such a neural network to fit as well as one where we provide features.

In [ ]:
```
# Problem 9
class AffineNode(object):
    """Node implementing affine transformation (W,x,b)-->Wx+b, where W is a
    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
    """

    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 = self.W.out @ self.x.out + self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        self.x.d_out = np.dot(self.W.out.T, self.d_out)
        self.b.d_out = self.d_out
        self.W.d_out = np.outer(self.d_out, self.x.out)
        return self.d_out
```

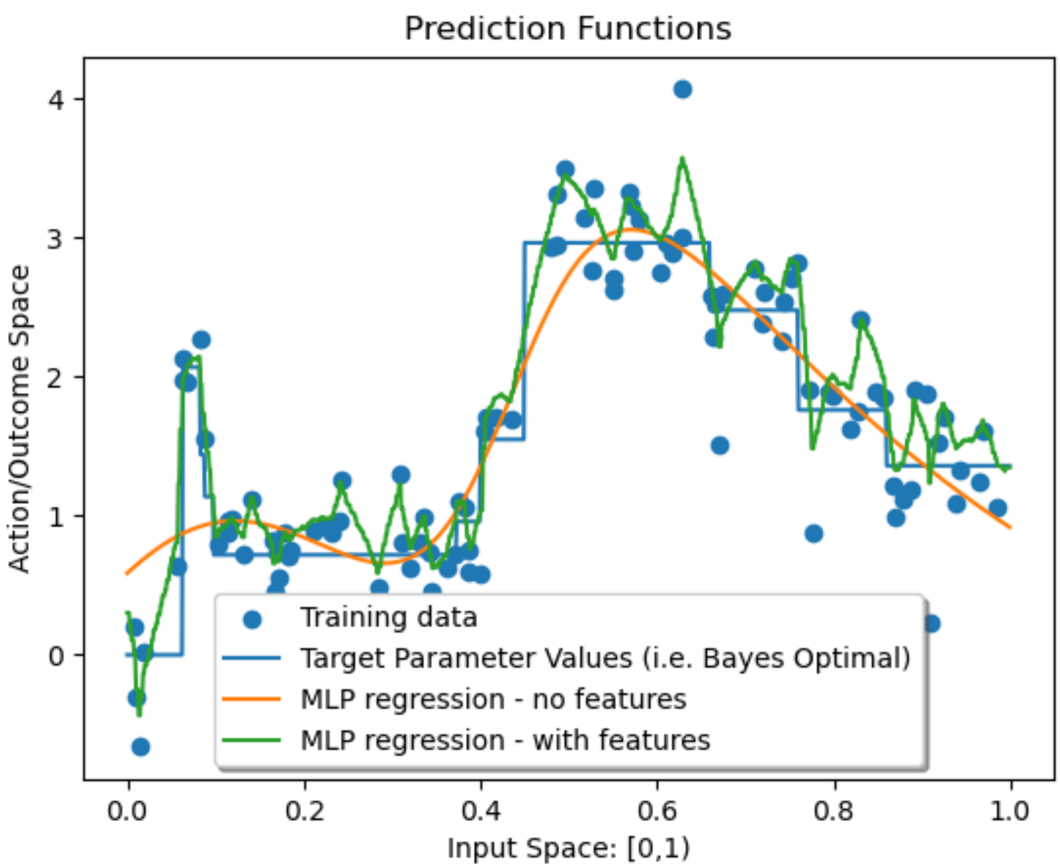
```python
    def get_predecessors(self):
        return([self.W, self.x, self.b])

# Problem 10


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):
        self.a.d_out = self.d_out*(1-self.out**2)
        return self.d_out

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

    pass

# Problem 11


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
        """"""
        first set of logic, use Affine node to get
        L in the hw, then apply tan_h to it
        """"
        self.x = nodes.ValueNode(node_name="x")  # to hold a vector input
        # to hold the parameter vector
        self.W1 = nodes.ValueNode(node_name="W1")
        # to hold the bias parameter (scalar)
        self.b1 = nodes.ValueNode(node_name="b1")
        self.affine = nodes.AffineNode(
```

```
            x=self.x, W=self.W1, b=self.b1, node_name='affine')
        self.tan_h = nodes.TanhNode(a=self.affine, node_name='tan_h')

        """
        second set of logic, use the
        VectorScalarAffineNode to get the ultimate
        output and then calculate the L2 loss
        """
        self.w2 = nodes.ValueNode(
            node_name="w2")  # to hold the parameter vector
        # to hold the bias parameter (scalar)
        self.b2 = nodes.ValueNode(node_name="b2")
        self.prediction = nodes.VectorScalarAffineNode(
            x=self.tan_h, w=self.w2, b=self.b2,
            node_name="prediction")

        self.y = nodes.ValueNode(node_name="y")  # to hold a scalar respon

        self.objective = nodes.SquaredL2DistanceNode(
            a=self.prediction, b=self.y,  node_name="square loss")

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

7.1 Screenshot mlp_regression.t.py for 9 and 10

```
(base) jobygeorge@10-19-35-86 HW7 % python3 mlp_regression.t.py
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 5.1378587571396975e-08.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 3.6546295504792983e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 1.6365789043253243e-09.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 5.045213092974115e-09.
.DEBUG: (Parameter W1) Max rel error for partial deriv 1.4544565841302641e-08.
DEBUG: (Parameter b1) Max rel error for partial deriv 1.957650618278275e-09.
DEBUG: (Parameter w2) Max rel error for partial deriv 9.887629139605405e-08.
DEBUG: (Parameter b2) Max rel error for partial deriv 6.936189287954398e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.006s

OK
(base) jobygeorge@10-19-35-86 HW7 %
```

7.2 Screenshot of ml_regression.py



Average training error was: .2385 for the first parameter and 0.0428 for the second. (Trained on 4950 and 450 epochs, respectively)

8 Problem 12 -14

Implement a Softmax node. We provided skeleton code for class SoftmaxNode in nodes.py. If your code is correct, you should be able to pass test SoftmaxNode in multiclass.t.py. Please attach a screenshot that shows the test results for this question.

Implement a negative log-likelihood loss node for multiclass classification. We provided skeleton code for class NLLNode in nodes.py. The test code for this question is combined with the test code for the next question.

Implement a MLP for multi-classclassification by completing the skeleton code in multiclass.py. Your code should pass the tests in test multiclass provided in multiclass.t.py. Please attach a screenshot that shows the test results for this question.

In [ ]:

```
# Problem 12
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):
        self.out = np.exp(self.z.out) / np.sum(np.exp(self.z.out))
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        #d f(z) / d(z)  = p(1-p)
        temp = []
        for prob in self.out:
            temp.append(prob * (1 - prob))
        self.diag = temp
        self.temp2 = -1 * np.outer(self.out, self.out)
        np.fill_diagonal(self.temp2, np.array(self.diag))

        dz = self.d_out.T @ self.temp2
        self.z.d_out += dz

        return self.d_out

    def get_predecessors(self):
        #print(self.z.d_out)
        return ([self.z])


#Problem 13
class NLLNode(object):
    """ Node computing NLL loss between 2 arrays.
```

```python
        Parameters:
        y_hat: a node that contains a vector, for a single
        x's probability prediction
        y_true: a node that's out is a single value, corresponding
        to the true class value of x_i. Used as an index

        Interestingly enough, maybe b/c we're doing SGD
        the shape of Y_HAT is R^k and Y_true R

    """
    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.a = self.y_hat.out[self.y_true.out]
        self.out = np.mean(-np.log(self.a))
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        #inspiration
        # https://stats.stackexchange.com/questions/309427/softmax-with-

        #initialize an array that is of same shape as y_hat.out
        #we'll use this for y_hat.d_out

        temp_mat = np.zeros_like(self.y_hat.out)

        #now the derivative of log(y_hat_true.out[y])
        #this means we take the recipricol of y_hat_true[y] and multiply
        #it by -1

        dz = -1 * (self.a**-1)
        #both of the dz are equal to each other, included both for
        #transparency
        dz = -1 * (self.y_hat.out[self.y_true.out]**-1)

        #now the only entry in our array that has a value, is when
        #class = y, so we update our array at index y_i by setting
        #it to dz
        temp_mat[self.y_true.out] = dz
        self.y_hat.d_out = self.d_out * temp_mat

        #don't need y_true.d_out b/c it's an index
        return (self.d_out)

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


#Problem 14
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
        self.x = nodes.ValueNode(node_name="x")  # inputs
        self.y = nodes.ValueNode(node_name="y")  # vector of class label
        self.W1 = nodes.ValueNode(
            node_name="W1")  # to hold the parameter vector
        self.b1 = nodes.ValueNode(
            node_name="b1")  # to hold the bias parameter (scalar)
        self.W2 = nodes.ValueNode(
            node_name="W2")  # to hold the parameter vector
        self.b2 = nodes.ValueNode(
            node_name="b2")  # to hold the bias parameter (scalar)
        self.affine = nodes.AffineNode(x=self.x,
                                       W=self.W1,
                                       b=self.b1,
                                       node_name='affine')
        self.tan_h = nodes.TanhNode(a=self.affine, node_name='tan_h')
        """
        second set of logic, use the VectorScalarAffineNode to get the u
        output and then calculate the L2 loss
        """
        self.Z = nodes.AffineNode(x=self.tan_h,
                                  W=self.W2,
                                  b=self.b2,
                                  node_name='Z')
        self.prediction = nodes.SoftmaxNode(z=self.Z, node_name="predict

        self.objective = nodes.NLLNode(y_hat=self.prediction,
                                       y_true=self.y,
                                       node_name='NLL')
        self.graph = graph.ComputationGraphFunction(
            inputs=[self.x],
            outcomes=[self.y],
            parameters=[self.W1, self.b1, self.W2, self.b2],
            prediction=self.prediction,
            objective=self.objective)
```

8.1 Screenshots for Problem 12 and 13

```
(base) jobygeorge@10-19-35-86 HW7 % python3 multiclass.t.py
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is 1.684183134568
5728e-08.
.DEBUG: (Parameter W1) Max rel error for partial deriv 4.3128959556168814e-05.
DEBUG: (Parameter b1) Max rel error for partial deriv 3.1405329566481014e-05.
DEBUG: (Parameter W2) Max rel error for partial deriv 5.721898631132528e-09.
DEBUG: (Parameter b2) Max rel error for partial deriv 2.758474616989107e-09.
.
----------------------------------------------------------------------
Ran 2 tests in 0.006s

OK
```

8.2 Screenshot for Problem 14

```
(base) jobygeorge@10-19-35-86 HW7 % python3 multiclass.py
Epoch  0  Ave training loss:  0.10767753468425854
Epoch  50  Ave training loss:  0.003740272949801889
Epoch  100  Ave training loss:  0.0019509875089186053
Epoch  150  Ave training loss:  0.0013189220100329906
Epoch  200  Ave training loss:  0.0009947600104512845
Epoch  250  Ave training loss:  0.0007975221227264001
Epoch  300  Ave training loss:  0.0006649220947379011
Epoch  350  Ave training loss:  0.0005697138957458585
Epoch  400  Ave training loss:  0.0004980771960410213
Epoch  450  Ave training loss:  0.00044225221211177576
Epoch  500  Ave training loss:  0.0003975450315101266
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:  0.00026340479431621313
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:  0.00020707801611844284
Test set accuracy = 1.000
```