Homework 1 ML Solutions

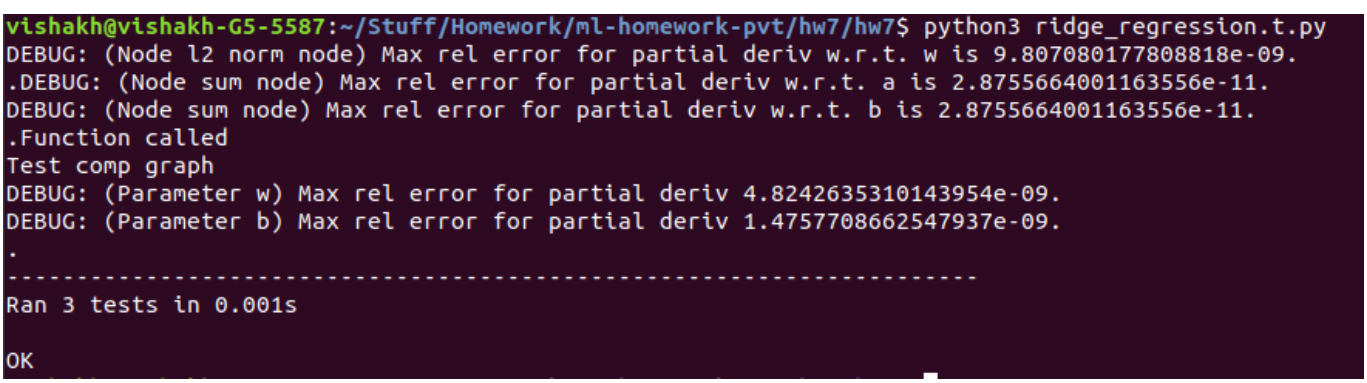
1 Ridge Regression

Q1. Complete the class L2NormPenaltyNode in nodes.py. If your code is correct, you should be able to pass test L2NormPenaltyNode in ridge_regression.t.py. Please attach a screenshot that shows the test results for this question.

The code for the node is attached below. Fig. 1 is the required image that shows the screenshot of test results for L2NormPenaltyNode from ridge_regression.t.py

Relevant Code for 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.l2_reg = np.array(l2_reg)
        self.w = w
```



```
vishakh@vishakh-G5-5587:~/Stuff/Homework/ml-homework-pvt/hw7/hw7$ python3 ridge_regression.t.py
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 9.807080177808818e-09.
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 2.8755664001163556e-11.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 2.8755664001163556e-11.
.Function called
Test comp graph
DEBUG: (Parameter w) Max rel error for partial deriv 4.8242635310143954e-09.
DEBUG: (Parameter b) Max rel error for partial deriv 1.4757708662547937e-09.
.
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Figure 1: Tests for results for Ridge Regression

```
    def forward(self):
        ## Your code
        self.out = self.l2_reg * (self.w.out @ self.w.out)
        self.d_out = np.zeros(self.out.shape)
        return self.out

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

    def get_predecessors(self):
        ## Your code
        return [self.w]
```

Q2. Complete the class SumNode in nodes.py. If your code is correct, you should be able to pass test SumNode in ridge_regression.t.py. Please attach a screenshot that shows the test results for this question.

The relevant code is attached below. Fig. 1 shows the test results for SumNode from ridge_regression.t.py.

Relevant Code for SumNode

```
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):
        # Your code
        self.out = self.a.out + self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out
```

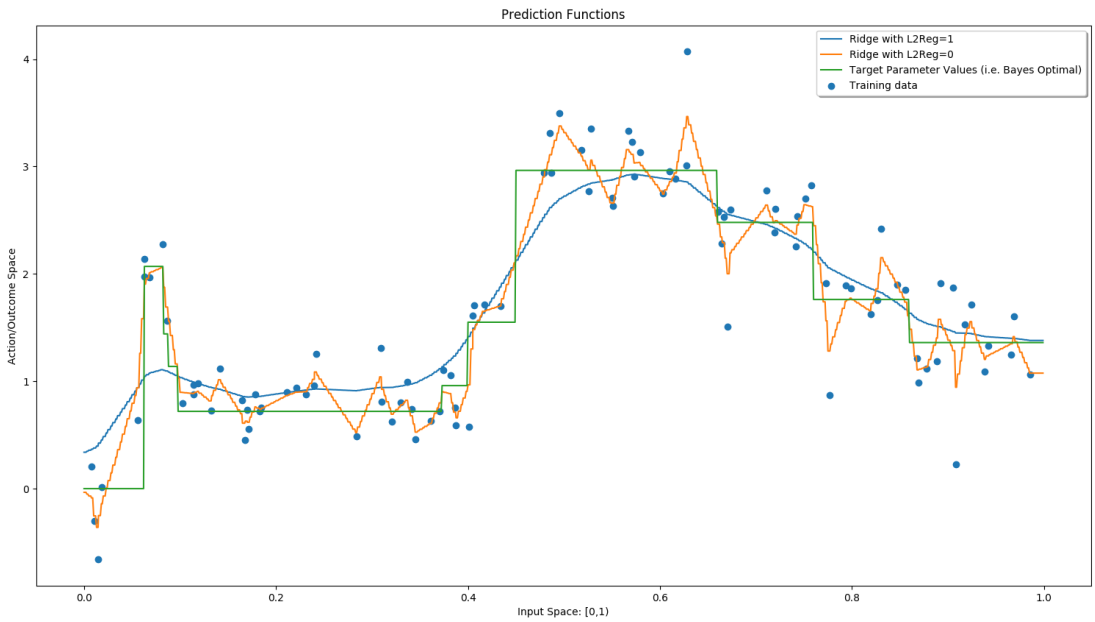


Figure 2: Output of Ridge regression for different l2 parameter values

```
def backward(self):
    # Your code
    self.a.d_out += 1 * self.d_out
    self.b.d_out += 1 * self.d_out
    return self.d_out

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

Q3. Report the average square error on the training set for the parameter settings given in the main() function.

For l2 param=1, the final training logs are:
Epoch 1950 : Ave objective= 0.30449 Ave training loss: 0.20016

For l2 param=0, the final training logs are:
Epoch 450 : Ave objective= 0.04987 Ave training loss: 0.04320

As expected, the training loss is a lot less in the case where there is no l2 regularization. As we can confirm from Fig. 2 (the output plot obtained from the script) the model overfits pretty heavily in this case resulting in very low train loss but we would prefer the smoother plot with l2 param=1.

Relevant code snippet

```python
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
        # TODO: ADD YOUR CODE HERE

        self.objective = nodes.SumNode(
                a=nodes.SquaredL2DistanceNode
                    (a=self.prediction, b=self.y,
                        node_name="loss"),
                b=nodes.L2NormPenaltyNode(l2_reg = l2_reg,
                    w=self.w, node_name = "l2"),
                        node_name = "square loss")

        self.inputs = [self.x]
        self.outcomes = [self.y]
        self.parameters = [self.w, self.b]

        self.graph =
            graph.ComputationGraphFunction(self.inputs,
                self.outcomes, self.parameters,
                    self.prediction, self.objective)
```

Logs of output from running ridge_regression.py

```
vishakh@vishakh-G5-5587:~/Stuff/Homework/ml-homework-pvt/hw7/hw7$ python3 ridge_regressi
Epoch  0 : Ave objective= 1.621870676810866  Ave training loss:  0.836931508773324
Epoch  50 : Ave objective= 0.3258384307320543  Ave training loss:  0.24213733822817474
Epoch  100 : Ave objective= 0.31544073216191054  Ave training loss:  0.2132814507319576
Epoch  150 : Ave objective= 0.31421375163969306  Ave training loss:  0.20330481101106876
```

```
Epoch  200 : Ave objective= 0.31328443395106004  Ave training loss:  0.20014401662600778
Epoch  250 : Ave objective= 0.3128824267412116  Ave training loss:  0.1987682083919095
Epoch  300 : Ave objective= 0.3111889858057757  Ave training loss:  0.19788345872603952
Epoch  350 : Ave objective= 0.31133622776574954  Ave training loss:  0.19805284408984658
Epoch  400 : Ave objective= 0.31141360232120496  Ave training loss:  0.19761309632229132
Epoch  450 : Ave objective= 0.3110066605453067  Ave training loss:  0.19755346166262938
Epoch  500 : Ave objective= 0.31046794104405573  Ave training loss:  0.19748506701188492
Epoch  550 : Ave objective= 0.31053254271574576  Ave training loss:  0.19750137144350055
Epoch  600 : Ave objective= 0.309994285041916  Ave training loss:  0.19764139807340522
Epoch  650 : Ave objective= 0.30967126801215117  Ave training loss:  0.19760262020866876
Epoch  700 : Ave objective= 0.3092565859885136  Ave training loss:  0.1978534203665924
Epoch  750 : Ave objective= 0.30901707208573503  Ave training loss:  0.1975673363308178
Epoch  800 : Ave objective= 0.3089152500232622  Ave training loss:  0.19789545882774992
Epoch  850 : Ave objective= 0.3082697377707142  Ave training loss:  0.19853282924733473
Epoch  900 : Ave objective= 0.3068513396921741  Ave training loss:  0.1993629250868571
Epoch  950 : Ave objective= 0.3079754014570931  Ave training loss:  0.1980058910574052
Epoch  1000 : Ave objective= 0.30613648388939313  Ave training loss:  0.1982020990537589
Epoch  1050 : Ave objective= 0.30670802344845244  Ave training loss:  0.1988088845911796
Epoch  1100 : Ave objective= 0.3073271120864638  Ave training loss:  0.19838478957146408
Epoch  1150 : Ave objective= 0.3069801466055204  Ave training loss:  0.1986936841659954
Epoch  1200 : Ave objective= 0.3052842864759324  Ave training loss:  0.20030516927972028
Epoch  1250 : Ave objective= 0.3068290300690866  Ave training loss:  0.19851130855744742
Epoch  1300 : Ave objective= 0.3062517931405056  Ave training loss:  0.19843205936127045
Epoch  1350 : Ave objective= 0.306418475346537  Ave training loss:  0.19869480007270865
Epoch  1400 : Ave objective= 0.30635997329821657  Ave training loss:  0.1986473333895936
Epoch  1450 : Ave objective= 0.3051009554773066  Ave training loss:  0.19928197929976343
Epoch  1500 : Ave objective= 0.3057339510019018  Ave training loss:  0.1992598556105777
Epoch  1550 : Ave objective= 0.30546672369291783  Ave training loss:  0.1990595089919875
Epoch  1600 : Ave objective= 0.30595556088738857  Ave training loss:  0.1991265326562627
Epoch  1650 : Ave objective= 0.3057936276877659  Ave training loss:  0.1990271488839324
Epoch  1700 : Ave objective= 0.30524816102216207  Ave training loss:  0.1993104765954774
Epoch  1750 : Ave objective= 0.3051900560357831  Ave training loss:  0.19976851087192896
Epoch  1800 : Ave objective= 0.30429097933115457  Ave training loss:  0.2007630781978149
Epoch  1850 : Ave objective= 0.3047784558361198  Ave training loss:  0.1993760409647069
Epoch  1900 : Ave objective= 0.304372765221809  Ave training loss:  0.19976766591623427
Epoch  1950 : Ave objective= 0.30449155887560925  Ave training loss:  0.2001646825436950
Epoch  0 : Ave objective= 0.7137436806079629  Ave training loss:  0.4327838527241931
Epoch  50 : Ave objective= 0.1282091449702579  Ave training loss:  0.10687589190871949
Epoch  100 : Ave objective= 0.10268538467418745  Ave training loss:  0.08565881058115794
Epoch  150 : Ave objective= 0.08337916624153889  Ave training loss:  0.07209973473084914
Epoch  200 : Ave objective= 0.06783693968496599  Ave training loss:  0.06572630439201976
Epoch  250 : Ave objective= 0.06917949060720807  Ave training loss:  0.06078909945712467
Epoch  300 : Ave objective= 0.06120991127316033  Ave training loss:  0.05297904467191443
Epoch  350 : Ave objective= 0.05817891562887361  Ave training loss:  0.04960715142857836
Epoch  400 : Ave objective= 0.0534341760939774  Ave training loss:  0.045803026288955634
```

```
Epoch  450 : Ave objective= 0.049872199193435385  Ave training loss:  0.0432030811344943
```

2 Multilayer Perceptron

Q4. Show that $\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$, where $x = (x_1, \ldots, x_d)^T$.

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

$= \sum_{r=1, r\neq i}^{m} \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial W_{ij}} + \frac{\partial J}{\partial y_i} \frac{\partial y_i}{\partial W_{ij}}$

$= \sum_{r=1, r\neq i}^{m} \frac{\partial J}{\partial y_r} \times 0 + \frac{\partial J}{\partial y_i} \frac{\partial y_i}{\partial W_{ij}}$

$= \frac{\partial J}{\partial y_i} \frac{\partial y_i}{\partial W_{ij}}$

$= \frac{\partial J}{\partial y_i} x_j \left(y_i = \sum_{j=1}^{d} W_{ij} x_j + b_i \text{ so } \frac{\partial y_i}{\partial W_{ij}} = x_j \right)$

Q5. Give a vectorized expression for $\frac{\partial J}{\partial W}$ in terms of the column vectors $\frac{\partial J}{\partial y}$ and x.

$\frac{\partial J}{\partial y} \in \mathbb{R}^{m\times 1}$, $x \in \mathbb{R}^{d\times 1}$ and we want to find matrix $\frac{\partial J}{\partial W} \in \mathbb{R}^{m\times d}$.

In this matrix, element in row i and column j, $\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$.

So we can do this as $\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} x^T$

Q6. Show that $\frac{\partial J}{\partial x} = W^T \left(\frac{\partial J}{\partial y} \right)$

This is very similar to Q4.

$\frac{\partial J}{\partial x_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial x_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} W_{ri} \left(y_i = \sum_{j=1}^{d} W_{ij} x_j + b_i \text{ so } \frac{\partial y_i}{\partial x_k} = W_{ik} \right)$

So for x_i, we use all the elements of W_i where this is columnindexing i.e. transpose in the matrix form.

Extending to all x: $\frac{\partial J}{\partial x} = W^T \frac{\partial J}{\partial y} \left(W \in \mathbb{R}^{m\times d}, \frac{\partial J}{\partial y} \in \mathbb{R}^{m\times 1}, W^T \frac{\partial J}{\partial y} \in \mathbb{R}^{d\times 1} - y = W^T x + b, \frac{\partial y}{\partial x} = W^T \right)$

Q7. Show that $\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$, where $\frac{\partial J}{\partial b}$ is defined in the usual way.

$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial b} = \frac{\partial J}{\partial y} \times I = \frac{\partial J}{\partial y}$

This is because the b term has no coefficient so the derivative is always 1.

Q8. 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**.

$\frac{\partial J}{\partial A_i} = \frac{\partial J}{\partial S_i} \frac{d\sigma(A_i)}{dA_i} = \frac{\partial J}{\partial S_i} \sigma'(A_i)$

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

3 MLP Implementation

Q9. Complete the class AffineNode in nodes.py. Please attach a screenshot that shows the test results for this question

The code is attached below. Fig. 3 is the required image that shows the test results.

Relevant Code for Affine Node

```
class AffineNode(object):
    def __init__(self, W, x, b, node_name):
        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 = self.W.out @ self.x.out
        self.out += self.b.out
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        #print(self.d_out.reshape(-1, 1).shape,
            self.x.out.reshape(-1, 1).T.shape)
        self.W.d_out += self.d_out.reshape(-1, 1) @
            self.x.out.reshape(-1, 1).T
        self.x.d_out += self.W.out.T @ self.d_out
        self.b.d_out += self.d_out * 1
        return self.d_out

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

Q10. Complete the class TanhNode in nodes.py. Please attach a screenshot that shows the test results for this question.

The code is attached below. Fig. 3 is the relevant image that shows the test outputs.

```
vishakh@vishakh-G5-5587:~/Stuff/Homework/ml-homework-pvt/hw7/hw7$ python3 mlp_regression.t.py
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 2.2194714045244017e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 4.323491985093387e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 2.804313262006903e-09.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 7.544250707846975e-09.
.Test comp graph
DEBUG: (Parameter W1) Max rel error for partial deriv 2.397262581617411e-06.
DEBUG: (Parameter b1) Max rel error for partial deriv 1.30244773863108e-07.
DEBUG: (Parameter W2) Max rel error for partial deriv 2.2292478592876663e-09.
DEBUG: (Parameter b2) Max rel error for partial deriv 4.707169664362273e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
```

Figure 3: Test outputs for mlp_regression.py

Relevant Code for TanhNode

```
class TanhNode(object):
    def __init__(self, a, node_name):
        self.node_name = node_name
        self.a = a
        self.out = None
        self.d_out = None

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

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

Q11. Run the MLP for the two settings given in the main() function and report the average training error.

The output for the two settings:
MLP - no features - Epoch 4950 : Ave objective= 0.24692 Ave training loss: 0.24260
MLP - with features - Epoch 450 : Ave objective= 0.04782 Ave training loss: 0.04242

Fig. 4 is the output from the mlp_regression.py script. With features the model gets much closer to modelling a lot of the training points because this is a more expressive setting.

Relevant Code snippet

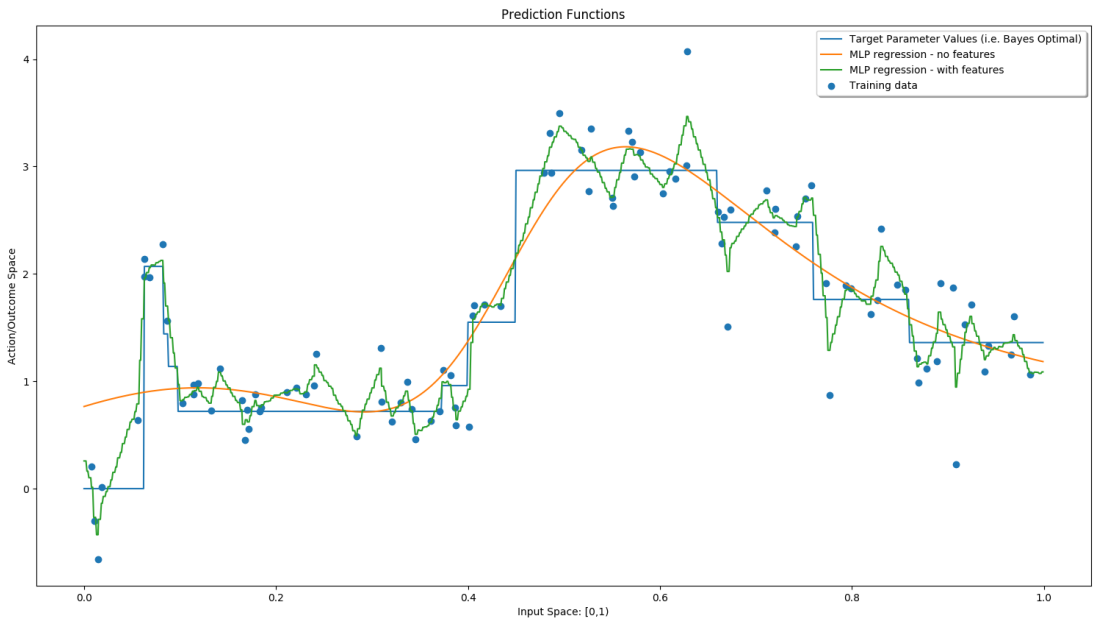


Figure 4: Output of MLP Regression script

```
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

        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.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.intermediate = nodes.TanhNode(
```

```
            a = nodes.AffineNode(x=self.x, W=self.W1,
            b=self.b1, node_name="intermediate"),
            node_name = "tanh")

        self.prediction =
            nodes.VectorScalarAffineNode(x=self.intermediate,
            w=self.W2, b=self.b2, node_name="prediction")

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

        self.inputs = [self.x]
        self.outcomes = [self.y]
        self.parameters = [self.W1, self.b1, self.W2, self.b2]

        self.graph =
            graph.ComputationGraphFunction(self.inputs,
            self.outcomes, self.parameters,
                self.prediction, self.objective)
```

Logs for mlp_regression.py

```
vishakh@vishakh-G5-5587:~/Stuff/Homework/ml-homework-pvt/hw7/hw7$ python3 mlp_regression
Epoch  0 : Ave objective= 3.1364505162639507  Ave training loss:  2.7209483366102463
Epoch  50 : Ave objective= 0.9453489855194859  Ave training loss:  0.9434883880333981
Epoch  100 : Ave objective= 0.944836663379822  Ave training loss:  0.9429691885563833
Epoch  150 : Ave objective= 0.9396697405129335  Ave training loss:  0.9376796378579851
Epoch  200 : Ave objective= 0.9003974207335897  Ave training loss:  0.8973504367060205
Epoch  250 : Ave objective= 0.8048149516201307  Ave training loss:  0.8007340247346544
Epoch  300 : Ave objective= 0.7715105298383916  Ave training loss:  0.767309365301047
Epoch  350 : Ave objective= 0.7657952223259896  Ave training loss:  0.762216026772568
Epoch  400 : Ave objective= 0.7609187958240062  Ave training loss:  0.7571956634901443
Epoch  450 : Ave objective= 0.7552450169192526  Ave training loss:  0.7511207114970379
Epoch  500 : Ave objective= 0.747738165935662  Ave training loss:  0.7441443954786354
Epoch  550 : Ave objective= 0.7397511616922694  Ave training loss:  0.7357875450486253
Epoch  600 : Ave objective= 0.7307474329262439  Ave training loss:  0.7270566663390469
Epoch  650 : Ave objective= 0.7217269331985525  Ave training loss:  0.7179271202775843
Epoch  700 : Ave objective= 0.7127093017570467  Ave training loss:  0.708871176469325
Epoch  750 : Ave objective= 0.7038057802270512  Ave training loss:  0.7002150020873218
Epoch  800 : Ave objective= 0.6960467025362445  Ave training loss:  0.6921983436197695
Epoch  850 : Ave objective= 0.6882156705414997  Ave training loss:  0.6849385418355325
Epoch  900 : Ave objective= 0.6818828718110039  Ave training loss:  0.6783401610165158
Epoch  950 : Ave objective= 0.6761546157695464  Ave training loss:  0.6720927515911721
Epoch  1000 : Ave objective= 0.6691197034830219  Ave training loss:  0.6659376464080967
Epoch  1050 : Ave objective= 0.6630986483683894  Ave training loss:  0.6592774103294721
```

```
Epoch  1100 : Ave objective= 0.6561370385629894  Ave training loss:  0.651621879994213
Epoch  1150 : Ave objective= 0.6473237253912493  Ave training loss:  0.6423589496393136
Epoch  1200 : Ave objective= 0.6354863564496168  Ave training loss:  0.6308566890764866
Epoch  1250 : Ave objective= 0.6214174928767402  Ave training loss:  0.6158786427415515
Epoch  1300 : Ave objective= 0.6039347358731062  Ave training loss:  0.597457916650355
Epoch  1350 : Ave objective= 0.5821182595285667  Ave training loss:  0.575144409681316
Epoch  1400 : Ave objective= 0.5556863873546084  Ave training loss:  0.5490180820544659
Epoch  1450 : Ave objective= 0.5270058828139301  Ave training loss:  0.5204223645066516
Epoch  1500 : Ave objective= 0.4938556620685716  Ave training loss:  0.4877682512755683
Epoch  1550 : Ave objective= 0.46343470846680107  Ave training loss:  0.4546185126259296
Epoch  1600 : Ave objective= 0.43150748657442706  Ave training loss:  0.4242119409959569
Epoch  1650 : Ave objective= 0.4044682778490046  Ave training loss:  0.39610204449329794
Epoch  1700 : Ave objective= 0.37932068196871227  Ave training loss:  0.3735853934856591
Epoch  1750 : Ave objective= 0.3625589794005787  Ave training loss:  0.3557038165216707
Epoch  1800 : Ave objective= 0.35047161359823414  Ave training loss:  0.3421362916101367
Epoch  1850 : Ave objective= 0.338365381043005  Ave training loss:  0.33481219886021535
Epoch  1900 : Ave objective= 0.33221298858510023  Ave training loss:  0.3265601955603678
Epoch  1950 : Ave objective= 0.3284335445412415  Ave training loss:  0.3205868464104362
Epoch  2000 : Ave objective= 0.32589911666998916  Ave training loss:  0.3171669223133322
Epoch  2050 : Ave objective= 0.3211639160735931  Ave training loss:  0.31351090810224114
Epoch  2100 : Ave objective= 0.3161252164303498  Ave training loss:  0.31248858227293075
Epoch  2150 : Ave objective= 0.31571342452140116  Ave training loss:  0.3088075428170896
Epoch  2200 : Ave objective= 0.31232546212521406  Ave training loss:  0.3063852207510138
Epoch  2250 : Ave objective= 0.31020650570337127  Ave training loss:  0.3042940641972317
Epoch  2300 : Ave objective= 0.3088351132784633  Ave training loss:  0.3021176686680571
Epoch  2350 : Ave objective= 0.30674333019008915  Ave training loss:  0.3004501210502638
Epoch  2400 : Ave objective= 0.30412727062120753  Ave training loss:  0.2988944657010897
Epoch  2450 : Ave objective= 0.3012567519994473  Ave training loss:  0.2976713179795796
Epoch  2500 : Ave objective= 0.29778541898780814  Ave training loss:  0.2985313033161670
Epoch  2550 : Ave objective= 0.2970737651747959  Ave training loss:  0.2942740247217457
Epoch  2600 : Ave objective= 0.2957520981122271  Ave training loss:  0.292368676750087
Epoch  2650 : Ave objective= 0.29593977742432664  Ave training loss:  0.2899102989768989
Epoch  2700 : Ave objective= 0.29139529806749337  Ave training loss:  0.2900406949437
Epoch  2750 : Ave objective= 0.29289604501569083  Ave training loss:  0.2871000780066514
Epoch  2800 : Ave objective= 0.2908954090874926  Ave training loss:  0.28560262137814324
Epoch  2850 : Ave objective= 0.2886681795856042  Ave training loss:  0.28380412773802116
Epoch  2900 : Ave objective= 0.28750855853847346  Ave training loss:  0.2825308480722757
Epoch  2950 : Ave objective= 0.28577240746619637  Ave training loss:  0.2810051167263867
Epoch  3000 : Ave objective= 0.28499905436993683  Ave training loss:  0.2791438534025513
Epoch  3050 : Ave objective= 0.282566408244302  Ave training loss:  0.27820109539546745
Epoch  3100 : Ave objective= 0.2809913115137915  Ave training loss:  0.27714135858183886
Epoch  3150 : Ave objective= 0.28014576929507756  Ave training loss:  0.275121107697778
Epoch  3200 : Ave objective= 0.27615739503888054  Ave training loss:  0.2750522140596840
Epoch  3250 : Ave objective= 0.2773907709834304  Ave training loss:  0.2726208376014607
Epoch  3300 : Ave objective= 0.2763374702587285  Ave training loss:  0.2712890418789844
```

```
Epoch  3350 : Ave objective= 0.2741152950279376  Ave training loss:  0.2704641622746684
Epoch  3400 : Ave objective= 0.2737742368629361  Ave training loss:  0.2687399486329265
Epoch  3450 : Ave objective= 0.2727529038942671  Ave training loss:  0.26756564560855656
Epoch  3500 : Ave objective= 0.27192695881229  Ave training loss:  0.26648096147317296
Epoch  3550 : Ave objective= 0.2677506576238848  Ave training loss:  0.2674705335967561
Epoch  3600 : Ave objective= 0.2691634881224019  Ave training loss:  0.26456876589929884
Epoch  3650 : Ave objective= 0.2677820494350292  Ave training loss:  0.2631551634350602
Epoch  3700 : Ave objective= 0.26614462461053173  Ave training loss:  0.2621207368917163
Epoch  3750 : Ave objective= 0.26596176508433883  Ave training loss:  0.2614726340114273
Epoch  3800 : Ave objective= 0.26484639488878725  Ave training loss:  0.2601519646783216
Epoch  3850 : Ave objective= 0.2637146587635012  Ave training loss:  0.25914912110808297
Epoch  3900 : Ave objective= 0.2629062560616895  Ave training loss:  0.25832865881321215
Epoch  3950 : Ave objective= 0.2619784718968105  Ave training loss:  0.2575716263719976
Epoch  4000 : Ave objective= 0.25923975279958716  Ave training loss:  0.2567253954374721
Epoch  4050 : Ave objective= 0.25998916558627544  Ave training loss:  0.2555524908757764
Epoch  4100 : Ave objective= 0.2593302161778874  Ave training loss:  0.25465794216563614
Epoch  4150 : Ave objective= 0.25822696812573315  Ave training loss:  0.2537746386044512
Epoch  4200 : Ave objective= 0.25791324271126576  Ave training loss:  0.2531519067577993
Epoch  4250 : Ave objective= 0.25688535495636106  Ave training loss:  0.2521257992041187
Epoch  4300 : Ave objective= 0.25533821632918674  Ave training loss:  0.2516921424074300
Epoch  4350 : Ave objective= 0.2551976358634548  Ave training loss:  0.25072275416238876
Epoch  4400 : Ave objective= 0.25424810261105635  Ave training loss:  0.2501939938485384
Epoch  4450 : Ave objective= 0.25257324116372837  Ave training loss:  0.2504399874938385
Epoch  4500 : Ave objective= 0.25239289256911934  Ave training loss:  0.2491278804129672
Epoch  4550 : Ave objective= 0.25257735021671046  Ave training loss:  0.2478245056448298
Epoch  4600 : Ave objective= 0.2517838986146544  Ave training loss:  0.2470063847996806
Epoch  4650 : Ave objective= 0.25086555807365174  Ave training loss:  0.2466701912584022
Epoch  4700 : Ave objective= 0.25039909357775697  Ave training loss:  0.2456687951539131
Epoch  4750 : Ave objective= 0.24954804437038422  Ave training loss:  0.2452392726076424
Epoch  4800 : Ave objective= 0.24913174899605697  Ave training loss:  0.2444678245039379
Epoch  4850 : Ave objective= 0.24841503459722653  Ave training loss:  0.2438004457683581
Epoch  4900 : Ave objective= 0.2480002169572863  Ave training loss:  0.24322682547061322
Epoch  4950 : Ave objective= 0.24692937905429677  Ave training loss:  0.2426041517138278
Epoch  0 : Ave objective= 3.2224283850519786  Ave training loss:  2.7278214384038875
Epoch  50 : Ave objective= 0.14349383064499607  Ave training loss:  0.15607846507007303
Epoch  100 : Ave objective= 0.11942725310405011  Ave training loss:  0.11202285856115159
Epoch  150 : Ave objective= 0.10175773866869177  Ave training loss:  0.09292349136780018
Epoch  200 : Ave objective= 0.08484068247783876  Ave training loss:  0.08052545630545957
Epoch  250 : Ave objective= 0.07843325930745354  Ave training loss:  0.08389156708892452
Epoch  300 : Ave objective= 0.07103307834741796  Ave training loss:  0.05722722996505229
Epoch  350 : Ave objective= 0.06113033256583599  Ave training loss:  0.05205135765004742
Epoch  400 : Ave objective= 0.05708166571659867  Ave training loss:  0.04749425488986014
Epoch  450 : Ave objective= 0.04782026716809111  Ave training loss:  0.04242840843306306
```

Q12. Implement a Softmax node.

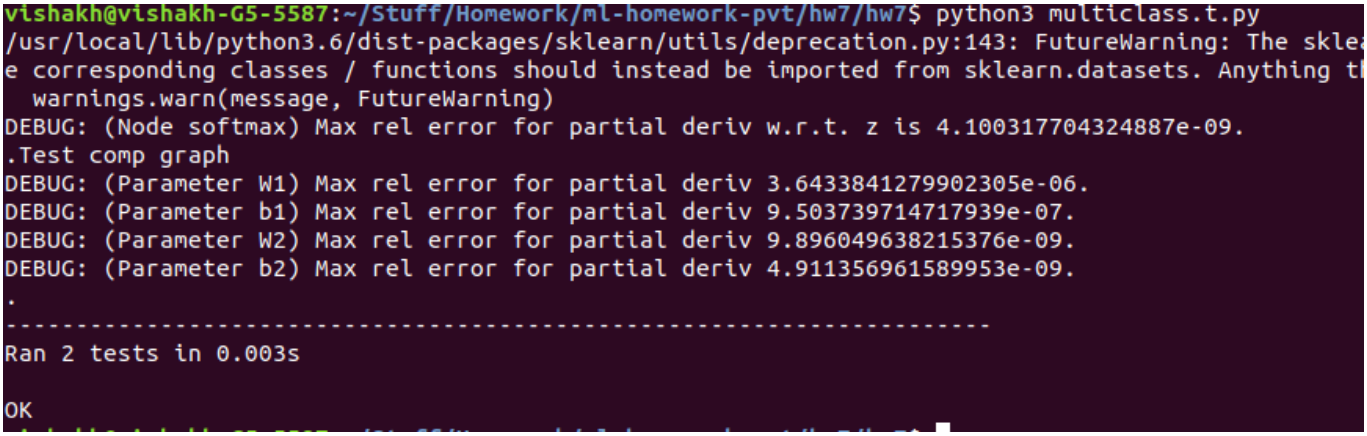
The code is attached below. A screenshot of the test output is Fig. 5

Relevant Code

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

    def forward(self):
        self.out =
            np.exp(self.z.out)/np.sum(np.exp(self.z.out))
                #np.tanh(self.a.out)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        denom = np.sum(np.exp(self.z.out))
        for i in range(len(self.z.out)):
            i_exp = np.exp(self.z.out[i])
            for j in range(len(self.z.out)):
                j_exp = np.exp(self.z.out[j])
                j_grad = j_exp
                if i != j:
                    j_grad = 0
                division_rule = (j_grad * denom - j_exp *
                    i_exp) / denom**2
                self.z.d_out[i] += self.d_out[j] *
                    division_rule
        return self.d_out

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

Q13. Implement a negative log-likelihood loss node for multiclass classification.

The relevant code is attached below. Fig. 5 is a screenshot of the test outputs.



```
vishakh@vishakh-G5-5587:~/Stuff/Homework/ml-homework-pvt/hw7/hw7$ python3 multiclass.t.py
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:143: FutureWarning: The skle
e corresponding classes / functions should instead be imported from sklearn.datasets. Anything t
  warnings.warn(message, FutureWarning)
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is 4.100317704324887e-09.
.Test comp graph
DEBUG: (Parameter W1) Max rel error for partial deriv 3.6433841279902305e-06.
DEBUG: (Parameter b1) Max rel error for partial deriv 9.503739714717939e-07.
DEBUG: (Parameter W2) Max rel error for partial deriv 9.896049638215376e-09.
DEBUG: (Parameter b2) Max rel error for partial deriv 4.911356961589953e-09.
.
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Figure 5: Tests for multiclass

Relevant Code:

```
class NLLNode(object):
    def __init__(self, y_hat, y_true, node_name):

        self.node_name = node_name
        self.y_hat = y_hat
        self.y_true = y_true
        self.out = None
        self.d_out = None

    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):
        #print(self.y_hat.d_out)
        #print(self.y_hat.d_out[0])
        for i in range(len(self.y_hat.out)):
            if i == self.y_true.out:
                self.y_hat.d_out[i] = self.d_out *
                    -1/self.y_hat.out[i]
            else:
                self.y_hat.d_out[i] = 0
        self.y_true.d_out = 0
        return self.d_out

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

Q14. Implement a MLP for multiclass classification 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.

Fig. 5 is the image with test cases passed.

Relevant Code Snippet

```
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
        # TODO: add your code here
        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.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.intermediate = nodes.TanhNode(
                a = nodes.AffineNode(x=self.x, W=self.W1,
                    b=self.b1, node_name="intermediate"),
                        node_name = "tanh")

        self.intermediate2 =
            nodes.AffineNode(x=self.intermediate, W=self.W2,
                b=self.b2, node_name="intermediate2")

        self.prediction = nodes.SoftmaxNode(z =
            self.intermediate2, node_name = "prediction")
```

```
        self.objective = nodes.NLLNode(y_hat=self.prediction,
            y_true=self.y, node_name="nll loss")

        self.inputs = [self.x]
        self.outcomes = [self.y]
        #print(self.y)
        self.parameters = [self.W1, self.b1, self.W2, self.b2]

        self.graph =
            graph.ComputationGraphFunction(self.inputs,
            self.outcomes, self.parameters, self.prediction,
            self.objective)
```

Logs for multiclass.py

```
vishakh@vishakh-G5-5587:~/Stuff/Homework/ml-homework-pvt/hw7/hw7$ python3 multiclass.py
Epoch  0  Ave training loss:  0.10767753468425852
Epoch  50  Ave training loss:  0.0037402729498018884
Epoch  100  Ave training loss:  0.0019509875089186038
Epoch  150  Ave training loss:  0.0013189220100329915
Epoch  200  Ave training loss:  0.000994760010451283
Epoch  250  Ave training loss:  0.0007975221227263982
Epoch  300  Ave training loss:  0.0006649220947379003
Epoch  350  Ave training loss:  0.0005697138957458559
Epoch  400  Ave training loss:  0.0004980771960410228
Epoch  450  Ave training loss:  0.0004422522121117765
Epoch  500  Ave training loss:  0.00039754503151012623
Epoch  550  Ave training loss:  0.0003609495175393885
Epoch  600  Ave training loss:  0.00033045202244361534
Epoch  650  Ave training loss:  0.00030465294323526527
Epoch  700  Ave training loss:  0.0002825495526238348
Epoch  750  Ave training loss:  0.000263404794316215
Epoch  800  Ave training loss:  0.00024666486030361416
Epoch  850  Ave training loss:  0.0002319056839595011
Epoch  900  Ave training loss:  0.0002187970217752761
Epoch  950  Ave training loss:  0.0002070780161184421
Test set accuracy = 1.000
```