1. 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.

```
!python3 ridge_regression.t.py
```

```
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 2.40251835204311e-09.
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 5.838671711157847e-10.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 1.6365788111231558e-09.
.DEBUG: (Parameter w) Max rel error for partial deriv 2.176183360312541e-09.
DEBUG: (Parameter b) Max rel error for partial deriv 3.0678572477687213e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

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

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

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

2. 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.

```
!python3 ridge_regression.t.py
```

```
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 4.83304267387949e-09.
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 5.263558043709814e-10.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 5.838672237759561e-10.
.DEBUG: (Parameter w) Max rel error for partial deriv 1.6418297217822408e-09.
DEBUG: (Parameter b) Max rel error for partial deriv 6.361722608755208e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

```python
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
        return self.d_out

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

3. Implement ridge regression with w regularized and b unregularized. Do this by completing the init method in ridge regression.py, using the classes created above. When complete, you should be able to pass the tests in ridge regression.t.py. Report the average square error on the training set for the parameter settings given in the main() function.

Ans)

- Param setting 1: Epoch 1950 : Ave objective= 0.30416333498507436 Ave training loss: 0.19975259892554026
- Param setting 2: Epoch 450 : Ave objective= 0.05068003196007712 Ave training loss: 0.0440921497856686

```
!python3 ridge_regression.t.py
```

```
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 1.5793975691809997e-09.
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 5.838671350819793e-10.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 5.838671350819793e-10.
.DEBUG: (Parameter w) Max rel error for partial deriv 1.4461955971802003e-08.
DEBUG: (Parameter b) Max rel error for partial deriv 7.521670660093729e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

```
!python3 ridge_regression.py
```

```
Epoch  1600 : Ave objective= 0.3057485785783776  Ave training loss:  0.1990621412382704
Epoch  1650 : Ave objective= 0.3054076642044352  Ave training loss:  0.19979989514480226
Epoch  1700 : Ave objective= 0.3054380428588343  Ave training loss:  0.19924823377040402
Epoch  1750 : Ave objective= 0.30518565102775874  Ave training loss:  0.1998109765222668
Epoch  1800 : Ave objective= 0.30542024692988834  Ave training loss:  0.19943023167942908
Epoch  1850 : Ave objective= 0.30286929120806494  Ave training loss:  0.20083304360117624
Epoch  1900 : Ave objective= 0.3044370169886135  Ave training loss:  0.19986403544732625
Epoch  1950 : Ave objective= 0.30416333498507436  Ave training loss:  0.19975259892554026
Epoch  0 : Ave objective= 0.6831868441647562  Ave training loss:  0.41474726450845645
Epoch  50 : Ave objective= 0.1178153688385627  Ave training loss:  0.10594259162888632
Epoch  100 : Ave objective= 0.10035638630071304  Ave training loss:  0.08708395223033598
Epoch  150 : Ave objective= 0.08501851410887146  Ave training loss:  0.07790740468137543
Epoch  200 : Ave objective= 0.07501148672587422  Ave training loss:  0.06427205386910566
Epoch  250 : Ave objective= 0.06793453829714871  Ave training loss:  0.0583967778477586
Epoch  300 : Ave objective= 0.06156154955236004  Ave training loss:  0.05614295209748719
Epoch  350 : Ave objective= 0.05799974446518799  Ave training loss:  0.05108533237386188
Epoch  400 : Ave objective= 0.052964781335534734  Ave training loss:  0.04997282714912114
Epoch  450 : Ave objective= 0.05068003196007712  Ave training loss:  0.0440921497856686
Figure(640x480)
```

```
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.l2_reg = l2_reg
        self.obj_reg = nodes.SquaredL2DistanceNode(a=self.prediction, b=self.y, node_name="square loss")
        self.obj_norm = nodes.L2NormPenaltyNode(l2_reg=self.l2_reg, w=self.w, node_name="l2 penalty")
        self.objective = nodes.SumNode(a=self.obj_reg, b=self.obj_norm, node_name="penalized sq 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)
```

4. $\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$, where $x = (x_1, \cdots, x_d)^T$

- By chain rule,

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

- Since only the item with $r=i$ is non-zero,

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

- $\frac{\partial y_i}{\partial W_{ij}} = x_j$ (as $y = Wx + b$)

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

5. $$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j = \frac{\partial J}{\partial y} \otimes x$$

---

6. $$\boxed{\frac{\partial J}{\partial x}} = \frac{\partial J}{\partial x_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \cdot \frac{\partial y_r}{\partial x_i}$$

$$\left(\text{as } y = Wx + b, \ \frac{\partial y_r}{\partial x_i} = W_{r,i} \right)$$

$$= \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \cdot W_{r,i} = (W_i)^T \cdot \frac{\partial J}{\partial y_r} = \boxed{W^T \left(\frac{\partial J}{\partial y} \right)}$$

---

7. From $y = Wx + b$, $\frac{\partial y}{\partial b} = 1$

$$\therefore \boxed{\frac{\partial J}{\partial b}} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial b} = \frac{\partial J}{\partial y} \cdot 1 = \boxed{\frac{\partial J}{\partial y}}$$

8. Show $\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$, where $(A \odot B)_i = A_i B_i$

Based on definition of $\sigma'(A)$, $\frac{\partial S_i}{\partial A_i} = \sigma'(A_i)$

So, $\frac{\partial J}{\partial A_i} = \frac{\partial J}{\partial S_i} \cdot \frac{\partial S_i}{\partial A_i} = \frac{\partial J}{\partial S_i} \cdot \sigma'(A_i)$

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

9. 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.

```
!python3 mlp_regression.t.py
```

```
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 5.096644672744053e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 2.406972643688512e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 1.6365788136702465e-09.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 1.9763106114969626e-09.
.DEBUG: (Parameter W1) Max rel error for partial deriv 1.0183356954301854e-06.
DEBUG: (Parameter b1) Max rel error for partial deriv 1.4064319110385487e-07.
DEBUG: (Parameter w2) Max rel error for partial deriv 3.9176596430029976e-10.
DEBUG: (Parameter b2) Max rel error for partial deriv 1.2719200164908194e-09.
.
----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

```python
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

        return self.d_out

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

10. Complete the class TanhNode in nodes.py. As you'll recall, d tanh(x) = 1–tanh2 x. Note 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.

```
!python3 mlp_regression.t.py
```

```
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 7.20890634703604e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 2.7104551533182506e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 2.8043131637107875e-09.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 1.3223247573520519e-08.
.DEBUG: (Parameter W1) Max rel error for partial deriv 1.0234941457123598e-07.
DEBUG: (Parameter b1) Max rel error for partial deriv 7.845227157989209e-09.
DEBUG: (Parameter w2) Max rel error for partial deriv 2.1094707366925755e-09.
DEBUG: (Parameter b2) Max rel error for partial deriv 3.7026688293685366e-11.
.
----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

```python
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
        return self.d_out

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

11. 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.

Ans)

- Param setting 1: Epoch 4950 : Ave objective= 0.2517892021591148 Ave training loss: 0.2466271830311671
- Param setting 2: Epoch 450 : Ave objective= 0.04810750384711343 Ave training loss: 0.04353498392690943

```
!python3 mlp_regression.t.py
```

```
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 2.1649804391653896e-08.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 1.3375169571128545e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 5.838672493090357e-10.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 2.2602113877588296e-09.
.DEBUG: (Parameter W1) Max rel error for partial deriv 1.4024483631179634e-06.
DEBUG: (Parameter b1) Max rel error for partial deriv 5.84622549909781e-08.
DEBUG: (Parameter w2) Max rel error for partial deriv 2.365268344555844e-09.
DEBUG: (Parameter b2) Max rel error for partial deriv 5.358513545257987e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

```
!python3 mlp_regression.py
```

```
Epoch  4600 : Ave objective= 0.2552206112684144  Ave training loss:  0.2504584954429009
Epoch  4650 : Ave objective= 0.2545288778391154  Ave training loss:  0.24991999046901595
Epoch  4700 : Ave objective= 0.2536255336753047  Ave training loss:  0.2501975867031055
Epoch  4750 : Ave objective= 0.25285352999966426  Ave training loss:  0.24880916351131863
Epoch  4800 : Ave objective= 0.25310287449598295  Ave training loss:  0.24819772083835126
Epoch  4850 : Ave objective= 0.25243128503818324  Ave training loss:  0.24773243264769146
Epoch  4900 : Ave objective= 0.25182699918856594  Ave training loss:  0.24731332147361157
Epoch  4950 : Ave objective= 0.2517892021591148  Ave training loss:  0.2466271830311671
Epoch  0 : Ave objective= 3.129180955915638  Ave training loss:  2.5356880675792426
Epoch  50 : Ave objective= 0.15198897610921253  Ave training loss:  0.14127616646666227
Epoch  100 : Ave objective= 0.11617229196547293  Ave training loss:  0.10922815719456622
Epoch  150 : Ave objective= 0.09689318420383945  Ave training loss:  0.09027240887919734
Epoch  200 : Ave objective= 0.08688141536169326  Ave training loss:  0.08294853882032388
Epoch  250 : Ave objective= 0.0723006890815104  Ave training loss:  0.06625030721858471
Epoch  300 : Ave objective= 0.06747412845654321  Ave training loss:  0.05916509099607956
Epoch  350 : Ave objective= 0.06223976560659244  Ave training loss:  0.053569926499627156
Epoch  400 : Ave objective= 0.051356207027161184  Ave training loss:  0.06207849926433166
Epoch  450 : Ave objective= 0.04810750384711343  Ave training loss:  0.04353498392690943
Figure(640x480)
```

```
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
        # TODO: ADD YOUR CODE HERE

        # initialize variables (x, y, W1, b1, w2, b2)
        self.x = nodes.ValueNode(node_name="x")
        self.y = nodes.ValueNode(node_name="y")
        self.W1 = nodes.ValueNode(node_name="W1")
        self.w2 = nodes.ValueNode(node_name="w2")
        self.b1 = nodes.ValueNode(node_name="b1")
        self.b2 = nodes.ValueNode(node_name="b2")

        self.affine = nodes.AffineNode(W=self.W1, x=self.x, b=self.b1, node_name="affine")
        self.tanh = nodes.TanhNode(a=self.affine, node_name="tanh")
        self.prediction = nodes.VectorScalarAffineNode(x=self.tanh, 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)
```

12. 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.

```
!python3 multiclass.t.py
```

```
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is 3.224197197211726e-08.
.DEBUG: (Parameter W1) Max rel error for partial deriv 0.0034522232406777017.
DEBUG: (Parameter b1) Max rel error for partial deriv 4.931291984539323e-06.
DEBUG: (Parameter W2) Max rel error for partial deriv 8.025725451101171e-08.
DEBUG: (Parameter b2) Max rel error for partial deriv 2.444586428981455e-08.
.
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

```python
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):
        I = np.eye(self.z.out.shape[0])
        d_z = self.d_out @ (self.out.reshape(-1,1) * (I-self.out.reshape(-1,1).T))
        self.z.d_out += d_z
        return self.d_out

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

13. 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.

```
! python3 multiclass.t.py
```

```
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is 4.0720297226290016e-08.
.DEBUG: (Parameter W1) Max rel error for partial deriv 0.003758631664432458.
DEBUG: (Parameter b1) Max rel error for partial deriv 4.427123495313254e-06.
DEBUG: (Parameter W2) Max rel error for partial deriv 9.513048846801781e-08.
DEBUG: (Parameter b2) Max rel error for partial deriv 1.2250136576683586e-08.
.
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

```python
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.y_hat = y_hat
        self.y_true = y_true
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.log_likelihood_elements = None

    def forward(self):
        self.log_likelihood_elements = self.y_true.out*np.log(self.y_hat.out) + (1-self.y_true.out)*np.log(1-self.y_hat.out)
        self.out = -np.sum(self.log_likelihood_elements)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        d_y_hat = self.d_out * ((-self.y_true.out/self.y_hat.out) + (1-self.y_true.out)/(1-self.y_hat.out))
        self.y_hat.d_out += d_y_hat

        return self.d_out

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

14. 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.

Ans)

- test result:
  Epoch 950 Ave training loss: 1.098610481902228
  Test set accuracy = 0.440

```
! python3 multiclass.t.py
```

```
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is 2.4707612958850724e-08.
.DEBUG: (Parameter W1) Max rel error for partial deriv 2.2246845867534564e-07.
DEBUG: (Parameter b1) Max rel error for partial deriv 2.8713153278798485e-07.
DEBUG: (Parameter W2) Max rel error for partial deriv 5.967296793011328e-08.
DEBUG: (Parameter b2) Max rel error for partial deriv 1.7622812282215924e-08.
.
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

```
! python3 multiclass.py
```

```
Epoch  0  Ave training loss:  1.173826640097864
Epoch  50  Ave training loss:  1.098614266195899
Epoch  100  Ave training loss:  1.0986221645634051
Epoch  150  Ave training loss:  1.098613562726454
Epoch  200  Ave training loss:  1.0986136410229712
Epoch  250  Ave training loss:  1.0986135631744594
Epoch  300  Ave training loss:  1.0986188138298667
Epoch  350  Ave training loss:  1.0986080992626166
Epoch  400  Ave training loss:  1.0986176393463343
Epoch  450  Ave training loss:  1.0986102050118365
Epoch  500  Ave training loss:  1.0986138470209137
Epoch  550  Ave training loss:  1.0986122212214515
Epoch  600  Ave training loss:  1.0986165994094421
Epoch  650  Ave training loss:  1.0986140826848703
Epoch  700  Ave training loss:  1.0986148684954136
Epoch  750  Ave training loss:  1.0986227325072
Epoch  800  Ave training loss:  1.098614166197705
Epoch  850  Ave training loss:  1.0986102419111892
Epoch  900  Ave training loss:  1.0986107134024037
Epoch  950  Ave training loss:  1.098610481902228
Test set accuracy = 0.440
```

```
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

        # initialize variables (x, y, W1, b1, w2, b2)
        self.x = nodes.ValueNode(node_name="x")
        self.y = nodes.ValueNode(node_name="y")
        self.W1 = nodes.ValueNode(node_name="W1")
        self.W2 = nodes.ValueNode(node_name="W2")
        self.b1 = nodes.ValueNode(node_name="b1")
        self.b2 = nodes.ValueNode(node_name="b2")

        self.affine = nodes.AffineNode(W=self.W1, x=self.x, b=self.b1, node_name="affine")
        self.tanh = nodes.TanhNode(a=self.affine, node_name="tanh")
        self.affine2 = nodes.AffineNode(W=self.W2, x=self.tanh, b=self.b2, node_name="affine")
        self.prediction = nodes.SoftmaxNode(z=self.affine2, node_name="prediction")
        self.objective = nodes.NLLNode(y_hat=self.prediction, y_true=self.y, node_name="NLL")

        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)
```