Solution to Q1:

1.Complete the class L2NormPenaltyNode in nodes.py.

```
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):
        ## 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
        d_w = 2 * self.l2_reg * self.w.out * self.d_out
        self.w.d_out += d_w

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

2.Attach a screenshot that shows the test results for this question.

```
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 2.7443452969133293e-09.
```

Solution to Q2:

1.Complete the class SumNode in nodes.py.

```
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

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

2.Attach a screenshot that shows the test results for this question.

```
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 2.8755647814839353e-11.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 5.263558556773282e-10.
```

Solution to Q3:

1.Complete the __init__ method in ridge_regresssion.py.

```
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.obj_reg = nodes.SquaredL2DistanceNode(a=self.prediction, b=self.y,
                                                   node_name = 'sqaure 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 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)
```

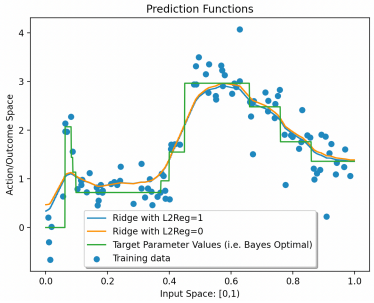
2.Report the average square error on the training set for the parameter given in the main() function.

- Parameter Setting 1 (l2_reg=1)

Epoch 1950 : Ave training loss: 0.19956220399372554

- Parameter Setting 2 (l2_reg=0)

Epoch 450 : Ave training loss: 0.20612430058004846



4. Show that $\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$, where $x = (x_1, \ldots, x_d)^T$. [Hint: Although not necessary, you might find it helpful to use the notation $\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases}$. So, for examples, $\partial_{x_j} \left(\sum_{i=1}^n x_i^2 \right) = 2x_i \delta_{ij} = 2x_j$.]

1 Solution to Q4

By the chain rule, we have:

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

For those r that r ≠ i, we have:

$$\frac{\partial y_r}{\partial W_{ij}} = x_j \delta_{ir} = 0$$

Thereupon, if and only if r = i, we would have:

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

5. Now let's vectorize this. Let's write $\frac{\partial J}{\partial y} \in \mathbb{R}^{m \times 1}$ for the column vector whose ith entry is $\frac{\partial J}{\partial y_i}$. Let's also define the matrix $\frac{\partial J}{\partial W} \in \mathbb{R}^{m \times d}$, whose ij'th entry is $\frac{\partial J}{\partial W_{ij}}$. Generally speaking, we'll always take $\frac{\partial J}{\partial A}$ to be an array of the same size ("shape" in numpy) as A. Give a vectorized expression for $\frac{\partial J}{\partial W}$ in terms of the column vectors $\frac{\partial J}{\partial y}$ and x. [Hint: Outer product.]

2 Solution to Q5

From the given rules, item in row i and column j can be expressed as:

$$\frac{\partial J}{\partial y_i} x_j$$

Hence:

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j / \rightarrow \frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} \times x$$

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

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

[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.]

3 Solution to Q6

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = W^T(\frac{\partial J}{\partial y}) \tag{6}$$

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.

4 Solution to Q7

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \frac{\partial J}{\partial y} * I = \frac{\partial J}{\partial y} \tag{7}$$

8. Show that $\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$

5 Solution to Q8

$$\frac{\partial J}{\partial A_i} = \frac{\partial J}{\partial S_i}\frac{\partial \sigma(A_i)}{\partial A_i} \tag{8}$$

$$\rightarrow \frac{\partial J}{\partial A} = \frac{\partial J}{\partial S}\frac{\partial S}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A) \tag{9}$$

Solution to Q9:

1.Complete the class AffineNode in nodes.py.

```
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 = 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 = 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]
```

2.Attach a screenshot that shows the test results for this question.

```
.DEBUG: (Parameter W1) Max rel error for partial deriv 7.269058740860952e-05.
DEBUG: (Parameter b1) Max rel error for partial deriv 1.2863089472467234e-06.
DEBUG: (Parameter W2) Max rel error for partial deriv 1.0119941207443262e-09.
```

Solution to Q10:

1.Complete the class TanhNode in nodes.py.

```
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]
```

2.Attach a screenshot that shows the test results for this question.

```
DEBUG: (Parameter b2) Max rel error for partial deriv 8.181269268308793e-10.
```

Solution to Q11:

1.Implement an MLP by completing the skeleton code in mlp_regression.py and making use of the nodes above.

```
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
        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='sqaure loss')

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

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

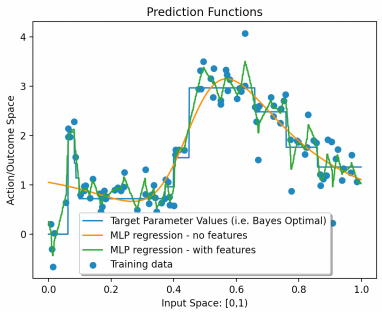
2.Run the MLP for the two settings given in the main() function and report the average training error.

- Parameter Setting 1 (non-featurized inputs)

Epoch 4950 : Ave training loss: 0.24523754780791823

- Parameter Setting 2 (featurized inputs)

Epoch 450 : Ave training loss: 0.046187354182711735



Solution to Q12:

1. Implement a Softmax Node.

```
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_z = np.zeros(self.z.d_out.shape)
        for i in range(d_z.shape[0]):
            d_z[i] = 0
            for j in range(self.d_out.shape[0]):
                if i == j:
                    d_z[i] += self.d_out[j] * (self.out[i] * (1 - self.out[j]))
                else:
                    d_z[i] += self.d_out[j] * (-self.out[i] * self.out[j])
        self.z.d_out += d_z

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

2. Attach a screenshot that shows the test result for this question.

```
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is 9.748972415428994e-09.
```

Solution to Q13:

```
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_p = np.zeros(self.y_hat.d_out.shape)
        d_p[self.y_true.out] = -1/self.y_hat.out[self.y_true.out]
        self.y_hat.d_out += d_p
        return self.d_out

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

Solution to Q14:

```
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=1):
        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")
        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.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')
        self.prediction = nodes.VectorScalarAffineNode(x = self.h,
                                                       w = self.W2,
                                                       b = self.b2,
                                                       node_name = 'VectorScalarAffine')
        self.prob = nodes.SoftmaxNode(z = self.prediction,
                                      node_name = 'softmax')
        self.objective = nodes.NLLNode(y_hat = self.prob,
                                       y_true = self.y,
                                       node_name = "NLLLoss")
        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)
```