1003 HW6

Long Chen
lc3424@nyu.edu

April 18th, 2021

Q1

```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 * (self.w.out.T @ self.w.out)

        # create d_out with shape of self.out
        self.d_out = np.zeros(self.out.shape)

        return self.out

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

        return self.d_out

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

Output:

DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 7.593850625176783e-09.

Q2

```
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
        # create d_out with shape of self.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]
```

Output:

.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 2.8755721407399478e-11.

DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 2.8755721407399478e-11.

Q3

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

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

```
        self.w = nodes.ValueNode(node_name="w") # to hold the parameter vector
        self.b = nodes.ValueNode(node_name="b") # to hold the bias parameter (scalar)
        self.prediction = nodes.VectorScalarAffineNode(x=self.x,
                                                       w=self.w,
                                                       b=self.b,
                                                       node_name="prediction")
        # Build computation graph
        self.l2 = nodes.L2NormPenaltyNode(l2_reg=l2_reg,
                                          w=self.w,
                                          node_name='l2')
        self.loss = nodes.SquaredL2DistanceNode(a=self.prediction,
                                                b=self.y,
                                                node_name='square loss')
        self.J = nodes.SumNode(a=self.loss,
                               b=self.l2,
                               node_name='objective function')

        self.graph = graph.ComputationGraphFunction([self.x],
                                                    [self.y],
                                                    [self.w, self.b],
                                                    self.prediction,
                                                    self.J)
```

Output (only last verbose reporting):
Epoch 1950 : Ave objective= 0.3049165950410885 Ave training loss: 0.19993022516020362
Epoch 450 : Ave objective= 0.052222580267584674 Ave training loss: 0.044452983947698345

Q4

$$\frac{\delta J}{\delta W_{i,j}} = \sum_{r=1}^{m} \frac{\delta J}{\delta y_r} \frac{\delta y_r}{\delta W_{i,j}}$$

Consider, for $r \neq i$,

$$\frac{\delta y_r}{\delta W_{i,j}} = \frac{\delta (W_r x + b)}{\delta W_{i,j}} = 0.$$

And for $r = i$,

$$\frac{\delta y_r}{\delta W_{i,j}} = \frac{\delta y_i}{\delta W_{i,j}} = x_j.$$

Thus we have:

$$\frac{\delta J}{\delta W_{i,j}} = \frac{\delta J}{\delta y_i} x_j.$$

Q5

$$\frac{\delta J}{\delta W_{i,j}} = \frac{\delta J}{\delta y_i} x_j.$$

Thus we have:

$$\frac{\delta J}{\delta W} = \frac{\delta J}{\delta y} \otimes x.$$

Q6

$$\begin{aligned}\frac{\delta J}{\delta x_i} &= \sum_{r=1}^{m} \frac{\delta J}{\delta y_r} \frac{\delta y_r}{\delta x_i} \\ &= \sum_{r=1}^{m} \frac{\delta J}{\delta y_r} W_{ri} \\ &= \frac{\delta J}{\delta y} W_i\end{aligned}$$

Thus:

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

Q7

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

Q8

$$\frac{\delta J}{\delta A} = \frac{\delta J}{\delta S} \frac{\delta S}{\delta A} = \frac{\delta J}{\delta S} \circ \sigma^{'}(A)$$

Q9

```
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.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)  # this node can
    actually be in init?

        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]
```

Output:

DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 1.2532942620569942e-08.

DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 4.3652345126436525e-07.

DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 1.636578905492936e-09.

Q10

```
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.a = a
        self.node_name = node_name

    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 = self.d_out * (1 - np.power(self.out, 2))
```

```
        self.a.d_out += d

        return self.d_out

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

Output:

.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 7.874228864277552e-09.

Q11

```
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')

        # W1, b1, w2, b2
        self.W1 = nodes.ValueNode(node_name='W1')
        self.b1 = nodes.ValueNode(node_name='b1')
        self.w2 = nodes.ValueNode(node_name='w2')
        self.b2 = nodes.ValueNode(node_name='b2')

        self.affine_node = nodes.AffineNode(W=self.W1,
                                            x=self.x,
                                            b=self.b1,
                                            node_name='affine')
        self.tanh_node = nodes.TanhNode(a=self.affine_node,
                                        node_name='tanh')
        self.pred = nodes.VectorScalarAffineNode(x=self.tanh_node,
                                                 w=self.w2,
                                                 b=self.b2,
                                                 node_name='
    prediction')
        self.J = nodes.SquaredL2DistanceNode(a=self.pred,
                                             b=self.y,
                                             node_name='objective
    function')

        self.graph = graph.ComputationGraphFunction([self.x],
                                                    [self.y],
                                                    [self.W1, self.
    b1, self.w2, self.b2],
                                                    self.pred,
                                                    self.J)
```

Output (only last verbose reporting):
Epoch 4950 : Ave objective= 0.24293241030718993 Ave training loss: 0.238099094072068
Epoch 450 : Ave objective= 0.04948181058839312 Ave training loss: 0.04988023289362568