1003 Hw 7: Computation Graphs, Backpropagation, and Neural Networks

rj1407

May 2019

1 Introduction

2 Computation Graph Framework

3 Ridge Regression

1. Complete the class L2NormPenaltyNode in nodes.py.

```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 scalar value >=0 (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 = l2_reg
        self.w = w

        # TODO

    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 = self.d_out * 2 * self.l2_reg * 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.

```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)
        """
        # TODO
        self.node_name = node_name
        self.out = None
        self.d_out = None
        self.a = a
        self.b = b

    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]
```

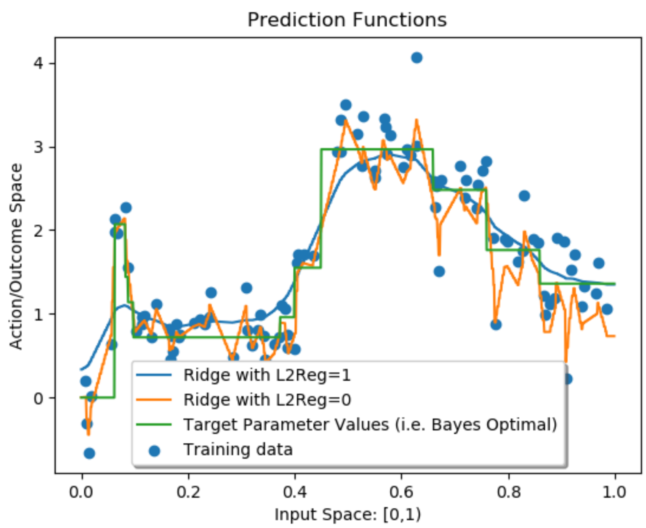
3. Implement ridge regression with w regularized and b unregularized. Do this by completing the __init__ method in the ridge_regression.py, using the classes created above. When complete, you should be able to pass the tests in ridge_regression.t.py. Report the average square error on the **training**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

   Answer:
   l2reg = 1, Ave training loss: 0.19972371048423804
   l2reg = 0, Ave training loss: 0.030071862072324693

```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
```

```python
        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")
        # TODO
        self.square_loss = nodes.SquaredL2DistanceNode(a=self.prediction, b=self.y,
                                                       node_name="square loss")
        self.l2_norm = nodes.L2NormPenaltyNode(l2_reg, self.w, node_name="l2_norm")
        self.objective = nodes.SumNode(self.square_loss, self.l2_norm, node_name="objective")
        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)

    def fit(self, X, y):
        num_instances, num_ftrs = X.shape
        y = y.reshape(-1)

        init_parameter_values = {"w": np.zeros(num_ftrs), "b": np.array(0.0)}
        self.graph.set_parameters(init_parameter_values)

        for epoch in range(self.max_num_epochs):
            shuffle = np.random.permutation(num_instances)
            epoch_obj_tot = 0.0
            for j in shuffle:
                obj, grads = self.graph.get_gradients(input_values = {"x": X[j]},
                                                      outcome_values = {"y": y[j]})
                epoch_obj_tot += obj
                # Take step in negative gradient direction
                steps = {}
                for param_name in grads:
                    steps[param_name] = -self.step_size * grads[param_name]
                    self.graph.increment_parameters(steps)

            if epoch % 50 == 0:
                train_loss = sum((y - self.predict(X,y)) **2)/num_instances
                print("Epoch ",epoch,": Ave objective=",epoch_obj_tot/num_instances," Ave training loss: ",train_loss)
```

```python

    def predict(self, X, y=None):
        try:
            getattr(self, "graph")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting data!")

        num_instances = X.shape[0]
        preds = np.zeros(num_instances)
        for j in range(num_instances):
            preds[j] = self.graph.get_prediction(input_values={"x":X[j]})

        return preds
```



4. [Optional] Create a new implementation of ridge regression that supports efficient minibatching. You will replace the the ValueNode x, which contains a vector, with a ValueNode X, which contains a matrix. The convention is that the first dimension indexes examples and the second is features (as we have always done). Many of the nodes will have to be adapted to this use case. Demonstrate its use and speedup.

4.1.1.1 $$\frac{\partial y_r}{\partial W_{ij}} = \begin{cases} x_j, & r=i \\ 0, & r\neq i \end{cases}$$

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

4.1.1.2 $$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} \otimes x = \frac{\partial J}{\partial y} x^T$$

4.1.1.3 $$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial x} = W^T \frac{\partial y}{\partial x}$$

4.1.1.4 $$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial b} = \frac{\partial J}{\partial y}$$

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

4 Multilayer Perceptron

4.1

4.2 MLP Implementation

1. 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.

```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)
    """

    ## TODO
    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 = 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_b = self.d_out
        d_x = np.matmul(np.transpose(self.W.out), self.d_out)
        d_W = np.matmul(self.d_out.reshape(-1, 1), np.transpose(self.x.out.reshape(-1,1)))
        self.b.d_out += d_b
        self.x.d_out += d_x
        self.W.d_out += d_W
        return self.d_out

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

2. Complete the class TanhNode in nodes.py. As you'll recall, $\frac{d}{dx}\tanh(x) = 1 - \tanh^2 x$. 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.

```
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
    """
    ## TODO
    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 = (1 - self.out * self.out) * self.d_out
        self.a.d_out += d_a
        return self.d_out

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

3. 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.

   Traning errors:
   no features: 0.21903644498138465
   with features: 0.027023495833501273

```
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 = 0.01
        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
        ##TODO
        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.L = nodes.AffineNode(self.W1, self.x, self.b1, node_name="L")
        self.h = nodes.TanhNode(self.L, node_name="h")
        self.prediction = nodes.VectorScalarAffineNode(self.h, self.W2, self.b2, node_name="prediction")
        self.objective = nodes.SquaredL2DistanceNode(a=self.prediction, b=self.y,
                                                     node_name="objective")
        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)

    def fit(self, X, y):
        num_instances, num_ftrs = X.shape
        y = y.reshape(-1)

        ## TODO: Initialize parameters (small random numbers -- not all 0, to break symmetry )
        s = self.init_param_scale
        init_values = {
            "W1": s * np.random.normal(0,1,size=(self.num_hidden_units, num_ftrs)),  # gaussian01
            "b1": np.zeros(self.num_hidden_units),
            "w2": s * np.random.normal(0,1,self.num_hidden_units),  # gaussian
            "b2": np.array(0.0)
        }

        self.graph.set_parameters(init_values)

        for epoch in range(self.max_num_epochs):
            shuffle = np.random.permutation(num_instances)
            epoch_obj_tot = 0.0
            for j in shuffle:
                obj, grads = self.graph.get_gradients(input_values = {"x": X[j]},
                                                      outcome_values = {"y": y[j]})
                #print(obj)
                epoch_obj_tot += obj
                # Take step in negative gradient direction
                steps = {}
```

```
                for param_name in grads:
                    steps[param_name] = -self.step_size * grads[param_name]
                    self.graph.increment_parameters(steps)

            if epoch % 50 == 0:
                train_loss = sum((y - self.predict(X,y)) **2)/num_instances
                print("Epoch ",epoch,": Ave objective=",epoch_obj_tot/num_instances," Ave training loss: ",train_loss)

    def predict(self, X, y=None):
        try:
            getattr(self, "graph")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting data!")

        num_instances = X.shape[0]
        preds = np.zeros(num_instances)
        for j in range(num_instances):
            preds[j] = self.graph.get_prediction(input_values={"x":X[j]})

        return preds
```