DS-GA 1003 - Homework 7

Eric Niblock

April 24, 2021

There is no doubt that neural networks are a very important class of machine learning models. Given the sheer number of people who are achieving impressive results with neural networks, one might think that it's relatively easy to get them working. This is a partly an illusion. One reason so many people have success is that, thanks to GitHub, they can copy the exact settings that others have used to achieve success. In fact, in most cases they can start with "pre-trained" models that already work for a similar problem, and "fine-tune" them for their own purposes. It's far easier to tweak and improve a working system than to get one working from scratch. If you create a new model, you're kind of on your own to figure out how to get it working: there's not much theory to guide you and the rules of thumb do not always work. Understanding even the most basic questions, such as the preferred variant of SGD to use for optimization, is still a very active area of research.

One thing is clear, however: If you do need to start from scratch, or debug a neural network model that doesn't seem to be learning, it can be immensely helpful to understand the low-level details of how your neural network works – specifically, back-propagation. With this assignment, you'll have the opportunity to linger on these low-level implementation details. Every major neural network type (RNNs, CNNs, Resnets, etc.) can be implemented using the basic framework we'll develop in this assignment.

To help things along, Philipp Meerkamp, Pierre Garapon, and David Rosenberg have designed a minimalist framework for computation graphs and put together some support code. The intent is for you to read, or at least skim, every line of code provided, so that you'll know you understand all the crucial components and could, in theory, create your own from scratch. In fact, creating your own computation graph framework from scratch is highly encouraged – you'll learn a lot.

To get started, please read the tutorial on the computation graph framework we'll be working with. (Note that it renders better if you view it locally.) The use of computation graphs is not specific to machine learning or neural networks. Computation graphs are just a way to represent a function that facilitates efficient computation of the function's values and its gradients with respect to inputs. The tutorial takes this perspective, and there is very little in it about machine learning, per se.

To see how the framework can be used for machine learning tasks, we've provided a full implementation of linear regression. You should start by working your way

through the `__init__` of the `LinearRegression` class in `linear_regression.py`. From there, you'll want to review the node class definitions in `nodes.py`, and finally the class `ComputationGraphFunction` in `graph.py`. `ComputationGraphFunction` is where we repackage a raw computation graph into something that's more friendly to work with for machine learning. The rest of `linear_regression.py` is fairly routine, but it illustrates how to interact with the `ComputationGraphFunction`.

As we've noted earlier in the course, getting gradient calculations correct can be difficult. To help things along, we've provided two functions that can be used to test the backward method of a node and the overall gradient calculation of a `ComputationGraphFunction`. The functions are in `test_utils.py`, and it's recommended that you review the tests provided for the linear regression implementation in `linear_regression.t.py`. (You can run these tests from the command line with `python3 linear_regression.t.py`.) The functions actually doing the testing, `test_node_backward` and `test_ComputationGraphFunction`, may seem a bit intricate, but they're implementing the exact same `gradient_checker` logic we saw in the second homework assignment.

Once you've understood how linear regression works in our framework, you're ready to start implementing your own algorithms. To help you get started, please make sure you are able to execute the following commands:

- **cd /path/to/hw7**
- **python3 linear_regression.py**
- **python3 linear_regression.t.py**

When moving to a new system, it's always good to start with something familiar. But that's not the only reason we're doing ridge regression in this homework. In ridge regression the parameter vector is "shared", in the sense that it's used twice in the objective function. In the computation graph, this can be seen in the fact that the node for the parameter vector has two outgoing edges. This sharing is common many popular neural networks (RNNs and CNNs), where it is often referred to as *parameter tying.*

The `ridge_regression.py` provides a skeleton code and `ridge_regression.t.py` is a test code, which you should eventually be able to pass.

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.**

   The following code shows the completed $\ell 2$ penalty node class,

```
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 = 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]
```

The test results for this question are included at the conclusion of problem three.

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.**

The following code shows the completed sum node class,

```
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):
            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]
```

The test results for this question are included at the conclusion of problem three.

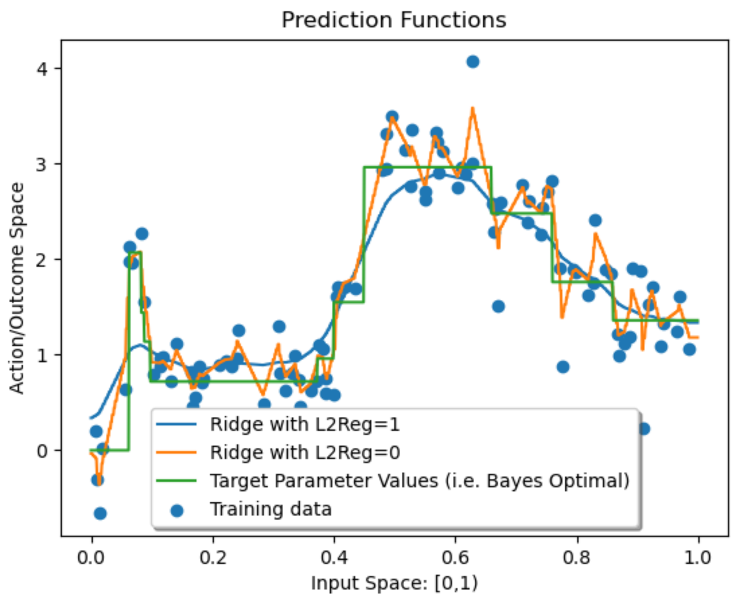
3. **Implement ridge regression with w regularized and b unregularized. Do this by completing the `__init__` method in `ridge_regression.py`, using the classes created above. When complete, you should be able to pass the tests in `ridge_regression.t.py`. Report the average square error on the training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

The following shows the completed initialization of our ridge regression class,

```
    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.input = [self.x]
            self.output = [self.y]
            self.para = [self.w, self.b]

            self.reg = nodes.SquaredL2DistanceNode(self.prediction, self.y, 'pred')
            self.norm = nodes.L2NormPenaltyNode(l2_reg, self.w, 'l2')
            self.obj = nodes.SumNode(self.reg, self.norm, 'square loss')

            self.graph = graph.ComputationGraphFunction(self.input, self.output,\
                                self.para, self.prediction, self.obj)
```

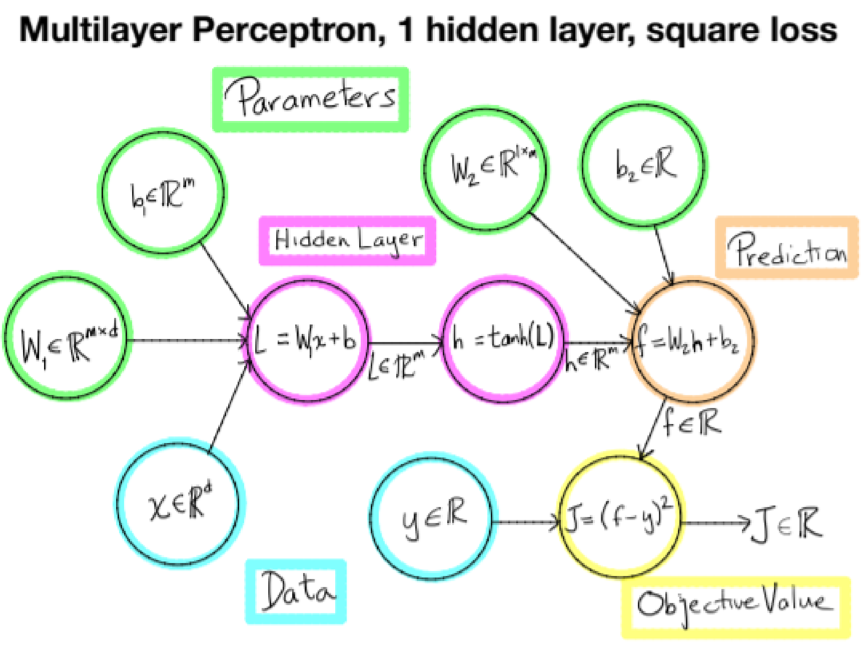
The following is the result of testing the previous nodes, and was copied from the command prompt. Additionally, we have included the plot produced from running ridge regression, as well as the average training loss after training including and not including features (0.2000 and 0.0502, respectively).

```
(base) C:\Users\Eric>python ridge_regression.t.py
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is 1.623094234596558e-08.
.DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is 5.83867125804325e-10.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is 5.83867125804325e-10.
.DEBUG: (Parameter w) Max rel error for partial deriv 2.0307927517078213e-09.
DEBUG: (Parameter b) Max rel error for partial deriv 8.80451932679026e-12.
.
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```



```
Epoch  1950 : Ave objective= 0.30335120831936047  Ave training loss:  0.19997864262061507

Epoch  450 : Ave objective= 0.05029143498226959  Ave training loss:  0.05024912502112784
```

Let's now turn to a multilayer perceptron (MLP) with a single hidden layer and a square loss. We'll implement the computation graph illustrated below:



The crucial new piece here is the nonlinear hidden layer, which is what makes the multilayer perceptron a significantly larger hypothesis space than linear prediction functions.

The multilayer perceptron consists of a sequence of "layers" implementing the following non-linear function

$$h(x) = \sigma\left(Wx + b\right),$$

where $x \in \mathbb{R}^d$, $W \in \mathbb{R}^{m\times d}$, and $b \in \mathbb{R}^m$, and where m is often referred to as the number of hidden units or hidden nodes. σ is some non-linear function, typically $\tanh$ or ReLU, applied element-wise to the argument of σ. Referring to the computation graph illustration above, we will implement this nonlinear layer with two nodes, one implementing the affine transform $L = W_1x + b_1$, and the other implementing the nonlinear function $h = \tanh(L)$. In this problem, we'll work out how to implement the backward method for each of these nodes.

In a general neural network, there may be quite a lot of computation between any given affine transformation $Wx + b$ and the final objective function value J. We will capture all of that in a function $f : \mathbb{R}^m \to \mathbb{R}$, for which $J = f(Wx + b)$. Our goal is to find the partial derivative of J with respect to each element of W, namely $\partial J/\partial W_{ij}$, as well as the partials $\partial J/\partial b_i$, for each element of b. For convenience, let $y = Wx + b$, so we can write $J = f(y)$. Suppose we have already computed the partial derivatives of J with respect to the entries of $y = (y_1, \ldots, y_m)^T$, namely $\frac{\partial J}{\partial y_i}$ for $i = 1, \ldots, m$. Then

by the chain rule, we have

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

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 $\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \textbf{else} \end{cases}$. So, for examples, $\partial_{x_j} \left(\sum_{i=1}^{n} x_i^2 \right) = 2x_i \delta_{ij} = 2x_j$.]**

Rewriting $y = Wx + b$ from vector notation yields $y_i = W_i x + b$ where W_i represents the i-th row of W. Therefore, it is clear that y_r has no dependence on W_{ij} unless $r = i$. Therefore,

$$\frac{\partial y_r}{\partial W_{ij}} = \begin{cases} x_j & i = r \\ 0 & i \neq j \end{cases} \tag{1}$$

Therefore, the sum collapses to the term that remains when $i = r$,

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j \tag{2}$$

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.]**

We have that,

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} x^T \tag{3}$$

Since we previously treated x as a $\mathbb{R}^{d \times 1}$ column vector, we use x^T. The resulting product yields an $\mathbb{R}^{m \times d}$ matrix as expected.

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.]

It is clear that,

$$\frac{\partial J}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial J}{\partial y_j} W_{j,i} = (W^T)_i \left(\frac{\partial J}{\partial y} \right) \tag{4}$$

And therefore this implies that,

$$\frac{\partial J}{\partial x} = W^T \left(\frac{\partial J}{\partial y} \right) \tag{5}$$

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.

By the chain rule,

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial b} = \frac{\partial J}{\partial y} I_m = \frac{\partial J}{\partial y} \tag{6}$$

And therefore,

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

Our nonlinear activation function nodes take an array (e.g. a vector, matrix, higher-order tensor, etc), and apply the same nonlinear transformation $\sigma : \mathbb{R} \to \mathbb{R}$ to every element of the array. Let's abuse notation a bit, as is usually done in this context, and write $\sigma(A)$ for the array that results from applying $\sigma(\cdot)$ to each element of A. If σ is differentiable at $x \in \mathbb{R}$, then we'll write $\sigma'(x)$ for the derivative of σ at x, with $\sigma'(A)$ defined analogously to $\sigma(A)$.

Suppose the objective function value J is written as $J = f(\sigma(A))$, for some function $f : S \mapsto \mathbb{R}$, where S is an array of the same dimensions as $\sigma(A)$ and A. As before, we want to find the array $\frac{\partial J}{\partial A}$ for any A. Suppose for some A we have already computed the array $\frac{\partial J}{\partial S} = \frac{\partial f(S)}{\partial S}$ for $S = \sigma(A)$. At this point, we'll want to use the chain rule to figure out $\frac{\partial J}{\partial A}$. However, because we're dealing with arrays of arbitrary shapes, it can be tricky to write down the chain rule. Appropriately, we'll use a tricky convention: We'll assume all entries of an array A are indexed by a single variable. So, for example, to sum over all entries of an array A, we'll just write $\sum_i A_i$.

8. **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. If A and B are arrays of the same shape, then their Hadamard product $A \odot B$ is an array with the same shape as A and B, and for which $(A \odot B)_i = A_i B_i$. That is, it's just the array formed by multiplying corresponding elements of A and B. Conveniently, in `numpy` if `A` and `B` are arrays of the same shape, then `A*B` is their Hadamard product.**

If we begin with the case $A \in \mathbb{R}$ (and therefore $\sigma(A) \in \mathbb{R}$), then given $J = f(\sigma(A))$, by the chain rule we have,

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial \sigma(A)} \frac{\partial \sigma(A)}{\partial A} = \frac{\partial J}{\partial S} \sigma'(A) \tag{8}$$

For an arbitrary dimension of A, we have,

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

Since every element of each matrix is equivalent, it is obvious that,

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A) \tag{10}$$

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.

Below is our implentation of the affine node,

```
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.W = W
        self.x = x
        self.b = b
        self.node_name = node_name

        self.out = None
        self.d_out = None

    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):
        dW = np.outer(self.d_out, self.x.out)
        dx = self.d_out.T@self.W.out
        db = self.d_out

        self.W.d_out += dW
        self.x.d_out += dx
        self.b.d_out += db

        return(self.d_out)

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

The test results for this problem are included at the end of problem eleven.

10. 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. 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.

Below is our implementation of the hyperbolic tangent node,

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

        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):
        da = (1-(self.out)**2)*self.d_out

        self.a.d_out += da

        return(self.d_out)

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

The test results for this problem are included at the end of problem eleven.

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

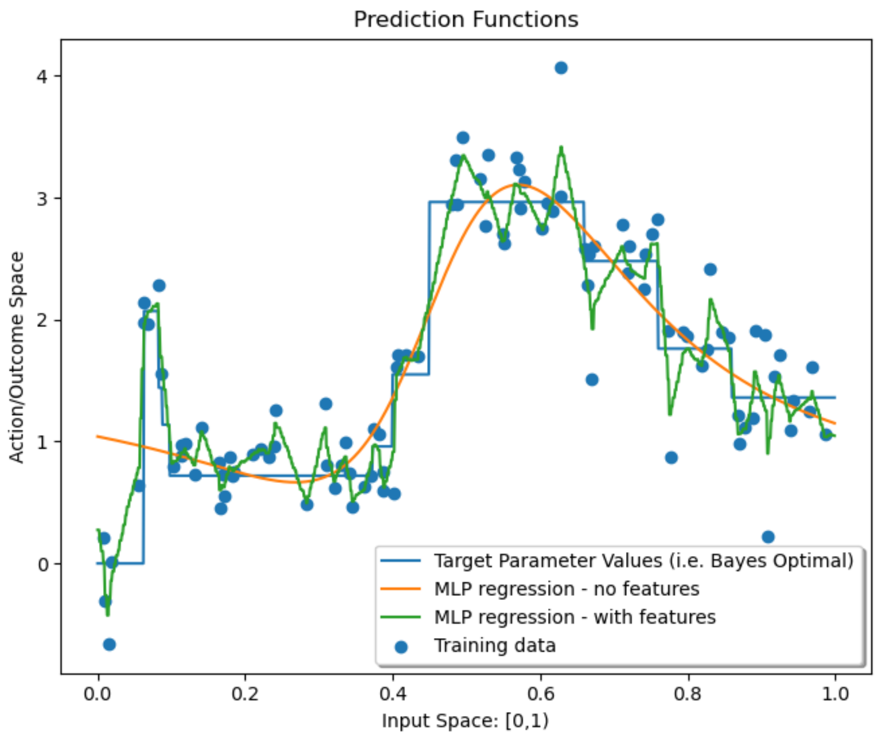
`mlp_regression.t.py`. Note that to break the symmetry of the problem, we initialize our weights to small random values, rather than all zeros, as we often do for convex optimization problems. Run the MLP for the two settings given in the `main()` function and report the average training error. Note 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.

Below is our completion of our MLP initialization,

```python
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
        self.x = nodes.ValueNode(node_name='x')
        self.W = nodes.ValueNode(node_name='W1')
        self.w = nodes.ValueNode(node_name='w2')
        self.B = nodes.ValueNode(node_name='b1')
        self.b = nodes.ValueNode(node_name='b2')
        self.y = nodes.ValueNode(node_name='y')
        self.inp = [self.x]
        self.outp = [self.y]


        self.affine = nodes.AffineNode(self.W,self.x,self.B,'aff')
        self.t = nodes.TanhNode(self.affine, 'tanh')
        self.pred = nodes.VectorScalarAffineNode(self.t, self.w, self.b, 'pred')

        self.obj = nodes.SquaredL2DistanceNode(self.pred, self.y, 'obj')

        self.para = [self.W, self.B, self.w, self.b]

        self.graph = graph.ComputationGraphFunction(self.inp, self.outp, self.para,
                                                    self.pred, self.obj)
```

The following is the result of testing the previous nodes, and was copied from the command prompt. Additionally, we have included the plot produced from running MLP, as well as the average training loss after training with and without features (0.2636 and 0.0406, respectively).

```
(base) C:\Users\Eric>python mlp_regression.t.py
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is 4.413758046406103e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is 2.1862879532491196e-09.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is 5.838672005255799e-10.
.DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is 2.230540396328285e-09.
.DEBUG: (Parameter W1) Max rel error for partial deriv 1.5295129068763488e-05.
DEBUG: (Parameter b1) Max rel error for partial deriv 1.7459806546186938e-06.
DEBUG: (Parameter w2) Max rel error for partial deriv 9.211319418451656e-10.
DEBUG: (Parameter b2) Max rel error for partial deriv 5.93888528116735e-10.
.
----------------------------------------------------------------------
Ran 3 tests in 0.013s

OK
```



```
Epoch  4950 : Ave objective= 0.2679233803442227  Ave training loss:  0.26361604621661144

Epoch  450 : Ave objective= 0.048600025031827074  Ave training loss:  0.04055385451506695
```

(Optional) We consider a generic classification problem with K classes over inputs x of dimension d. Using a MLP we will compute a K-dimensional vector z representing

scores,

$$z = W_2 \tanh(W_1 x + b_1) + b_2,$$

with $W_1 \in \mathbb{R}^{m \times d}$, $b_1 \in \mathbb{R}^m$, $W_2 \in \mathbb{R}^{K \times m}$ and $b_1 \in \mathbb{R}^K$. Our model assumes that x belongs to class k with probability

$$e^{z_k} / \sum_{k=1}^{K} e^{z_k},$$

which corresponds to applying a Softmax to the scores. Given this probabilistic model we can train the model by minimizing the negative log-likelihood.

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.**

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.**

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.**