Homework 7: Computation Graphs, Backpropagation, and Neural Networks

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

1 Introduction

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. The reason so many people have success is that, thanks to GitHub, they can copy the exact settings that others have used to achieve success. 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 and suggestions 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 is implemented. With this assignment, you'll have an opportunity to linger on these low-level implementation details, which one usually rushes through in more specialized neural network classes. Every major neural network type (RNNs, CNNs, Resnets, etc) can be implemented using the basic building blocks we'll develop in this assignment.

To help things along, we[1] have designed a minimal 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.

2 Computation Graph Framework

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.) In some sense, computation graphs have nothing

[1] Philipp Meerkamp, Pierre Garapon, and David Rosenberg

to do with machine learning or neural networks. They are just a way to represent a function that facilitates efficient computation of the function value and it's gradients with respect to inputs. The tutorial takes this perspective, and there is very little 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 test_ComputationGraphFunction` are a bit intricate so they can work with any node or ComputationGraphFunction, but it's using the exact same gradient_checker logic we saw in the first homework assignment.

Once you've understood how linear regression works in our framework, you're ready to start implementing your own algorithms...

3 Ridge Regression

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. As we discussed in class, 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. While we don't have this sharing in the multilayer perceptron, we do have it in RNNs and CNNs, the two neural network architectures that have had the most impact. So being able to handle ridge regression is a necessary condition (and possibly sufficient) for being able to represent the these important architectures.

We've provided some skeleton code in ridge_regression.py, and some test code in ridge_regression.t.py, that you should eventually be able to pass.

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 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 = 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`.

```
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.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.

```
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.l2_reg = l2_reg
        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
    )

        # TODO
        self.prediction = nodes.VectorScalarAffineNode(x=self.x, w=self.w, b=self.b,
                                                       node_name="prediction")
        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")
        # TODO
        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)
```

Setting 1:
Epoch 1950 : Ave objective= 0.3081544879368522 Ave training loss: 0.199031856598395
setting 2:
Epoch 450 : Ave objective= 0.04005683874474115 Ave training loss: 0.0276461661108703

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 Multilayer Perceptron

In this problem, we'll be implement a multilayer perceptron, with a single hidden layer, for the regression setting. 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.

4.1 The standard non-linear layer

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

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

where $x \in \mathbf{R}^d$, $W \in \mathbf{R}^{m \times d}$, and $b \in \mathbf{R}^m$, 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 $L(x) = W_1 x + b_1$, and the other implementing the nonlinear function $h(L) = \tanh(L)$. In this problem we'll work out how to implement the backward method for each of these nodes.

4.1.1 The Affine Transformation

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 $f : \mathbf{R}^m \to \mathbf{R}$, for which $J = f(Wx + b)$. Our goal is to find the partial derivative of J with respect to each element of W, namely $\partial J / \partial W_{ij}$. 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 intermediate variable $y = (y_1, \ldots, y_m)^T$, namely $\frac{\partial J}{\partial y_i}$ for $i = 1, \ldots, m$. Then by the chain rule, we have

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

1. 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, $\partial_{x_j} \left(\sum_{i=1}^{n} x_i^2 \right) = 2x_i \delta_{ij} = 2x_j$.]
   For $\frac{\partial J}{\partial W_{ij}} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial W_{ij}}$, only the item with $r = i$ is non-zero. $\implies \frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} \frac{\partial y_i}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j$

2. Now let's vectorize this. Let's write $\frac{\partial J}{\partial y} \in \mathbf{R}^{m \times 1}$ for the column vector whose ith entry is $\frac{\partial J}{\partial y_i}$. Let's also define the matrix $\frac{\partial J}{\partial W} \in \mathbf{R}^{m \times d}$, 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.]
   Based on the rules given in question, item in row i and column j should have expression $\frac{\partial J}{\partial y_i} x_j$.

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} x_j \implies \frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y} \otimes x$$

3. In the usual way, define $\frac{\partial J}{\partial x} \in \mathbf{R}^d$, 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.]

$$\frac{\partial J}{\partial x_i} = \sum_{k=1}^{m} \frac{\partial J}{\partial y_k} \frac{\partial y_k}{\partial x_i} = \sum_{k=1}^{m} \frac{\partial J}{\partial y_k} W_{ki}$$
$$\implies \frac{\partial J}{\partial x} = \frac{\partial J}{\partial f(x)} \frac{\partial f(x)}{\partial x} = W^T \left(\frac{\partial J}{\partial y} \right)$$

4. Show that $\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}$, where $\frac{\partial J}{\partial b}$ is defined in the usual way.

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

4.1.2 Element-wise Transformers

Our nonlinear activation function nodes take an array (e.g. a vector, matrix, higher-order tensor, etc), and apply the same nonlinear transformation $\sigma : \mathbf{R} \to \mathbf{R}$ 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 $x \in \mathbf{R}$, 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 $f : S \mapsto \mathbf{R}$, 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 $\sum_i A_i$.

1. 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.

$$\frac{\partial J}{\partial A_i} = \frac{\partial J}{\partial S_i} \frac{d\sigma(A_i)}{dA_i}$$
$$\implies \frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \frac{\partial S}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A)$$

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.

```
    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.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]
```

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

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

```
        num_hidden_units = self.num_hidden_units
        s = self.init_param_scale
        ## TODO: Initialize parameters (small random numbers -- not all 0, to break
    symmetry )
        parameter_vals = {"W1": s*np.random.standard_normal((num_hidden_units,
    num_ftrs)),
                          "b1": s*np.random.standard_normal((num_hidden_units)),
                          "w2": s*np.random.standard_normal((num_hidden_units)),
                          "b2": s*np.array(np.random.randn()) }

        self.graph.set_parameters(parameter_vals)

        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)
```

Setting 1:
Epoch 4950 : Ave objective= 0.21758927284741764 Ave training loss: 0.20883813853643404
Setting 2:
Epoch 450 : Ave objective= 0.03750042435029375 Ave training loss: 0.024624794808589402

4. [Optional] See if you can get a fit on the training set with an MLP that uses just the scalar input that is about as good as the fit using the featurized inputs. You can do that by tweaking model parameters (e.g. the number of hidden nodes or layers) and/or the parameters of optimization. You **may use** any neural network framework (PyTorch, TensorFlow, etc), which can help by providing more advanced optimization techniques (e.g. Adam), variable initialization methods, and/or various normalization approaches (batch norm, etc).

4.3 [OPTIONAL}

1. [Optional] Implement a Softmax node.
2. [Optional] Implement a negative log-likelihood loss node for multiclass classification.
3. [Optional] Use the classes above to apply an MLP to the simple multiclass classification dataset we had on a previous assignment.