CSC 578 HW 5: Backprop Hyper-Parameters

Fall 2020

(Version 0.01, with much credit again to Prof. Tomuro)
Graded out of 10 points.
Do all questions
Here is a zip file with all of the files for this homework

Overview

Similar to HW#3, you make required modifications to the NNDL book code and write a small application code. The objective of the assignment is to enhance you understanding of some of the hyper-parameters of neural networks.

The original NNDL code "network2.py" is hard-coding several things, including the sigmoid activation function, L2 regularization and the input data format (for MNIST). In this assignment, we make the code general so that it (implements and) accepts various hyper-parameters as (literally) parameters of the network.

For the application code, you do some systematic experiments that test various combinations of the hyper-parameter values.

Note: This homework is not as tightly specified as HW#3. You may need to **do your own research** to find out how best to implement a few things.

---

1. Network Code (in NN578_network2.py)

Intro

For this part, you will need to make modifications to the Network Code, adding some hyper-parameters and modifying some functions.

- (A) Hyper-parameters:
  - Cost function
    - QuadraticCost, CrossEntropy, LogLikelihood
  - Activation function
    - Sigmoid, Tanh, ReLU, Softmax
  - Regularization
    - L1 and L2
  - Dropout rate

- (B) Functions to modify (minimally):
  - set_parameters()
  - feedforward()
  - backprop()
  - update_mini_batch()
  - total_cost()

  Note that you may need to modify other functions additionally to implement Dropout.

(A) Hyper-parameters

Hyper-parameters are passed in through keyword arguments in the constructor/init (**cost, act_hidden, act_output, regularization**, and **dropoutpercent**). The values are stored in the (additional) instance variables as shown below:

```
## Additional keyword arguments for hyper-parameters
def __init__(self, sizes, cost=CrossEntropyCost, act_hidden=Sigmoid,
             act_output=None, regularization=None, lmbda=0.0,
             dropoutpercent=0.0):
    """The list ``sizes`` contains the number of neurons in the respective
    layers of the network.  For example, if the list was [2, 3, 1]
    then it would be a three-layer network, with the first layer
    containing 2 neurons, the second layer 3 neurons, and the
    third layer 1 neuron.  The biases and weights for the network
    are initialized randomly, using
    ``self.default_weight_initializer`` (see docstring for that method).
    """
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.default_weight_initializer()

    self.set_parameters(cost, act_hidden, act_output, regularization, lmbda,
                        dropoutpercent)

## nt: THIS NEEDS CHANGE.
## nt: convenience function for setting network hyperparameters
def set_parameters(self, cost=QuadraticCost, act_hidden=Sigmoid,
                   act_output=None, regularization=None, lmbda=0.0,
                   dropoutpercent=0.0):
    self.cost=cost
    self.act_hidden = act_hidden
    if act_output == None:
        self.act_output = self.act_hidden
    else:
        self.act_output = act_output
    self.regularization = regularization
```

```
self.lmbda = lmbda
self.dropoutpercent = dropoutpercent
```

1. **cost**

   - This hyper-parameter argument specifies the cost function.
   - Options are 'QuadraticCost', 'CrossEntropy', 'LogLikelihood'.
   - Each one must be implemented as a class. The scheme for class function is explained later in this document (see Class Function notes below).
   - The class should have two static functions: **fn()** executes the definition of the function (to compute the cost in during evaluation), and **derivative()** executes the function's derivative (to compute the error during learning). No other function such as delta() should be defined in the class because they are not necessary for this assignment.

     1. **QuadraticCost** is fully implemented already in the starter code, as shown below (and you do not need to modify it).
     2. **CrossEntropy** is partially written (by taking fn() from the original NNDL code "network2.py"). You must add **derivative()**.
     3. You will write a whole class **LogLikelihood**.

```
class QuadraticCost(object):

    @staticmethod
    def fn(a, y):
        """Return the cost associated with an output ``a``
        and desired output ``y``.
        """
        return 0.5*np.linalg.norm(y-a)**2

    ## nt: addition
    @staticmethod
    def derivative(a, y):
        """Return the first derivative of the function."""
        return -(y-a)

class CrossEntropyCost(object):

    @staticmethod
    def fn(a, y):
        """Return the cost associated with an output ``a``
        and desired output ``y``.
        """
        return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))

    @staticmethod
    def derivative(a, y):
```

```
        """Return the first derivative of the function."""
        ###
        ### YOU WRITE YOUR CODE HERE
        ###
```

- Both functions receive `a` (activation) and `y` (target output), which are from one data instance and represented by column vectors.
- `fn()` returns a **scalar**, while `derivative()` returns a **column vector** (containing the cost derivative for each node in the output layer; no multiplication by the derivative of the activation function).
- NOTES on **LogLikelihood**:
    1. This cost function should really be used when **'act_output'** (the activation function of the output layer) = **'Softmax'**. You can check for it if you like (and print warning or abort execution, for instance), but that's not a requirement for the homework.
    2. For its function and derivative formula, you can look at the week 3 video at about 2:11 in. For `derivative()`, you compute the derivative of the node for which the target output is 1. For other nodes, the value should be 0.

2. act_hidden

- This parameter specifies the activation function for nodes on **all** hidden layers, but EXCLUDING the output layer.
- Parameter options are 'Sigmoid', 'Tanh', 'ReLU', 'Softmax'.
- Each one must be implemented as a class (see Class Function notes below). The class should have two functions: a static method `fn()` executes the definition of the function (to compute the node activation value), and a class method `derivative()` executes the function's derivative (to compute the error during learning).
    1. `Sigmoid` is fully implemented already in the starter code, as shown below (and you do not need to modify it).
    2. `Softmax` is partially written. You must add `derivative()`. Note that, since its derivative (already written) returns a 2D matrix instead of a vector (reference), it is handled differently in computing the error/delta in `backprop()`. See Class Function notes below. But for the purpose of writing the class, you only fill in the definition of the function in `fn()`.
    3. For **Tanh**, a skeleton class is already written. You complete the class.
    4. You write a whole class for **ReLU**.

```
class Sigmoid(object):
    @staticmethod
    def fn(z):
        """The sigmoid function."""
        return 1.0/(1.0+np.exp(-z))
```

```
    @classmethod
    def derivative(cls,z):
        """Derivative of the sigmoid function."""
        return cls.fn(z)*(1-cls.fn(z))

class Softmax(object):
    @staticmethod
    # Parameter z is an array of shape (len(z), 1).
    def fn(z):
        """The softmax of vector z."""
        ###
        ### YOU WRITE YOUR CODE HERE
        ###

    @classmethod
    def derivative(cls,z):
        """Derivative of the softmax.
        IMPORTANT: The derivative is an N*N matrix.
        """
        a = cls.fn(z) # obtain the softmax vector
        return np.diagflat(a) - np.dot(a, a.T)
```

For those functions, you can look at the Recorded zoom session from Week 3 (about 1:10 in) and/or the Review slides from Week 4 for the definitions.

Note that, although Softmax is almost never used for hidden layers, we do not have to disallow it in this homework.

3. act_output

- This parameter specifies the activation function for nodes on the output layer.
- Parameter options are 'Sigmoid', 'Tanh' and 'Softmax'.

  NOTES on Tanh: If Tanh is selected as the activation function for the output layer:

  1. Because the output value will be between 1 and -1 (instead of 1 and 0), the only cost function that goes with Tanh is the quadratic cost. So if the cost function was set to anything besides 'QuadraticCost', change/overwrite the cost function to QuadraticCost and print a warning (to the user, e.g. "Tanh only accepts 'QuadraticCost' cost function. Changing to QuadraticCost.."). **You must add code to do it by yourself**, in `set_parameters()`.
  2. Note that, in the network startup code (NN578_network2.py), some code is added (in `SGD()`) that changes the dataset when the output layer's activation is Tanh, in particular to make the target y values to be 1 or -1 (instead of 1 or 0). It is already written for you, so you don't need to be concerned about the data.

4. regularization:

- This parameter specifies the regularization method.

- Parameter options are 'L2' and 'L1'.
- The selected method is applied to all hidden layers and the output layer.
- You can implement them in any way you like, for example as function classes or inline if-else conditionals. For definitions/formulas and explanation, see the Avoiding overfitting video from Week 3 or the review slides from Week 4.

  IMPORTANT: The start-up code has L2 hard-coded in (unchanged from the original NNDL code for this part). **You make necessary changes to the code by yourself** to incorporate the two methods.
- The regularization is relevant at two places in the backprop algorithm:

  1. During training, when weights are adjusted at the end of a mini-bath – the function `update_mini_batch()`.
  2. During evaluation, when the cost is computed – the function `total_cost()`.

  NOTE: Both of those functions have the parameter **lmbda**, passed in from the function `SGD()`. You utilize its value in implementing the regularizations.

5. dropoutpercent

- This parameter specifies the percentage of dropout.
- The value is between 0 and 1. For example, 0.4 means 40% of the nodes on a layer are dropped (or made inaccessible).
- Dropout consists in randomly setting a fraction rate of units in a layer to 0 at each update during training time, which helps prevent overfitting.
- Assume the same dropout percentage is applied to **all hidden layers**. *Dropout should not be applied to input or output layer.*
- You can implement the parameter in any way you like. **You make necessary changes to the code by yourself, wherever needed.**
- Many dropout schemes have been proposed in neural networks. For this assignment, you implement the following scheme.

  1. Dropout is applied during the **training phase only**. No dropout is applied during the testing/evaluation phrase.
  2. Use the same dropout nodes during one **mini-batch**. That means you have to store which nodes were used/dropped somewhere else. Think about it and implement in your way.
  3. **Scale** the output values of the layer. This scheme is explained at this site. In particular, the following code is very useful. The first line is generating a dropout mask (`u1`) and the second line is applying the mask to the activation of a hidden layer (`h1`) during the **forward** propagation phase in the backprop function.

     ```
     # Dropout training, notice the scaling of 1/p
     u1 = np.random.binomial(1, p, size=h1.shape) / p
     h1 *= u1
     ```

Then during the **backward** propagation phase, you apply the mask to the delta of a hidden layer (`dh1`). This is necessary because, since a dropout mask is applied as an additional multiplier (function) after the activation function, it essentially became a constant coefficient of the activation function (i.e., `c*a(z)`), and shows up in the derivative of that function – Let $f(z) = c * a(z)$, then $f'(z) = c * a'(z)$.

```
dh1 *= u1
```

IMPORTANT NOTES:

- The variable `p` above is the ratio of nodes to RETAIN, not to remove. So essentially, `p = 1 - self.dropoutpercent`.
- `np.random.binomial()` is **probabilistic**, so does not guarantee the p proportion of success. But for the purpose of the assignment, it's fine to use it (i.e., the code above). However, if you like to implement correctly, you can use `random.sample()` method in Python's standard library.
- During forward propagation, dropout should be applied to/after **activation**, NOT to the z/weighted sum (e.g. sigma(0) = 0.5, which is not right or convenient here).
- During backward propagation, dropout should be applied to the delta (the error at a given hidden layer).

Class Function notes

Static or class functions in a class can be called through the class name. When a class name is bound to an instance variable, you can invoke a specific static/class function in the class by prefixing the instance variable. For example, here is a line in the function `backprop()`:

```
a_prime = (self.act_output).derivative(zs[-1])
```

So whatever `self.act_output` is bound to (e.g. Sigmoid, Tanh, ReLU), the function `derivative()` defined inside the class is being invoked.

(B) Functions to modify.

In addition to `set_parameters()`, `feedforward()`, `backprop()`, `update_mini_batch()` and `total_cost()`, you need to modify other parts of the code to implement **dropout**. It's up to you to figure out and decide.

Whatever you did, you should ***describe and explain*** *it in the documentation.* You may not get full points if modifications you made were not explained sufficiently.

2. Application code (Start-up code: 578hw5.ipynb,html: 578hw5.html)

Write Jupyter Notebook code to do these experiments using the iris dataset. Ensure your code works for each experiment, and show the output in an Jupyter notebook.

Just like HW#3, use the iris dataset iris.csv and the iris-423.dat network file to create an initial network. You may play with the learning rate (eta / η) and mini_batch size on your own.

Do the following experiments. For each experiment, train the network using iris-train-1.csv for **15 epochs** (only) and test it using iris-test-1.csv.

Note the output results in the right-most column were generated on IBM CognitiveClassLab.

	act_hidden	act_output	cost	regularization	lmbda	dropout	OUTPUT
1	Sigmoid	Sigmoid	Quadratic	(default)	0.0	0.0	Result-1.txt
2	Sigmoid	Sigmoid	CrossEntropy	(default)	0.0	0.0	Result-2.txt
3	Sigmoid	Softmax	CrossEntropy	(default)	0.0	0.0	Result-3.txt
4	Sigmoid	Softmax	LogLikelihood	(default)	0.0	0.0	Result-4.txt
5	ReLU	Softmax	CrossEntropy	(default)	0.0	0.0	Result-5.txt
6	ReLU	Softmax	LogLikelihood	(default)	0.0	0.0	Result-6.txt
7	Tanh	Sigmoid	Quadratic	(default)	0.0	0.0	Result-7.txt
8	Tanh	Tanh	Quadratic	(default)	0.0	0.0	Result-8.txt
9	Sigmoid	Sigmoid	Quadratic	L2	3.0	0.0	Result-9.txt
10	Sigmoid	Sigmoid	Quadratic	L1	3.0	0.0	Result-10.txt

For the experiments below, use iris4-20-7-3.dat. Note that results may vary for these experiments because dropout nodes are randomly (but probabilistically) chosen.

	act_hidden	act_output	cost	regularization	lmbda	dropout	OUTPUT
11	Sigmoid	Sigmoid	Quadratic	(default)	0.0	0.1	Result-11.txt
12	Sigmoid	Sigmoid	Quadratic	(default)	0.0	0.5	Result-12.txt

Note that those are the absolute minimal experiments. Although not required for this homework, you should try other combinations of hyper-parameters to test your code more thoroughly and rigorously.

Submission

1. Completed "NN578_network2.py", and the application Notebook file and its html version. Be sure to add your **name, course/section number** and the **assignment name** at the top of all code files.

2. Documentation.

   - **In pdf only.**
   - A *minimum* of 2.0 pages (i.e., two pages filled, and some in the next page).
   - Write as much as you can. I consider terse answers insufficient, therefore won't give a full credit when I grade.
   - Create a presentable document. Don't make me work hard to find the information I asked for. (There are a lot of these to read.)
   - Content should include:
     - Your **name, course/section number** and the **assignment name** at the top of the file.
     - Then, in separate, labeled sections, include:

       Experiment results Whether or not your results for the first **10 experiments** in the 'Application code' section above matched with the given results. If your results were different, describe the discrepancies and what you speculated the discrepancies came from.

Implementation Explain how you implemented each of the requirements, and for each if they were **Complete**, meaning you did the code and verified that it worked; **Not attempted**, meaning you didn't get there; or **Partial**, meaning that you have some code but it did not completely work, and explain why. Give as detailed explanations as possible. In particular, be sure to explain everything you did to implement Dropout correctly.

Reflections Your reaction and reflection on this assignment overall (e.g. difficulty level, challenges you had).

- Write as much as you can. Try to make a **thorough and well-organized report**. It's not to impress me; it's for your exercise.
- Also some kind of graphs/plots are nice (to make a more professional presentation), although no extra credits are given this time.

DO NOT Zip the files – Submit files separately.