DSC 190 - "Super Homework"

Due: Wednesday, June 14

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 PM.

Problem 1.

As a data scientist you will have the opportunity to work on problems that are of great importance to society. This is not one of those problems.

The *menu-match* dataset consists of 646 images of food from three different restaurants: an Asian restaurant, an Italian restaurant, and a soup restaurant. The data set was constructed by employees of Microsoft Research[1].

The data set is available at the following link:

`https://f000.backblazeb2.com/file/dsc-data/menu-match.npz`

The file is in compressed numpy format; it can be loaded with 'np.load'. Once loaded, it behaves like a dictionary with four keys: `X_train`, `X_test`, `y_train`, and `y_test`, corresponding to the training data, test data, training labels, and test labels, respectively. For example, to get the training data:

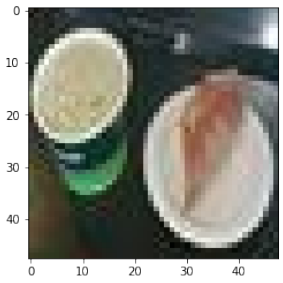
```
>>> data = np.load("menu-match.npz")
>>> data['X_train']
```

The training data is a `480 x 48 x 48 x 3` tensor, since it consists of 480 images, each `48 x 48` pixels with 3 colors.

Note that the given labels are strings: "a" for Asian restaurant, "i" for Italian, and "s" for soup.

You may display an image with `matplotlib` using `plt.imshow`. For example:

```
>>> # plot test example #30
>>> plt.imshow(data['X_test'][30])
```



Using tensorflow, train a convolutional neural network to predict whether a given image is from the Asian restaurant or not (thus turning the problem into a binary classification problem). You will need to determine the network architecture, but the example code given in lecture is a good starting point. Report the test accuracy – your model should be able to get above 70% of the test set correct (preferably higher!). Show your code.

Solution:

[1] `http://neelj.com/projects/menumatch/`

Problem 2.

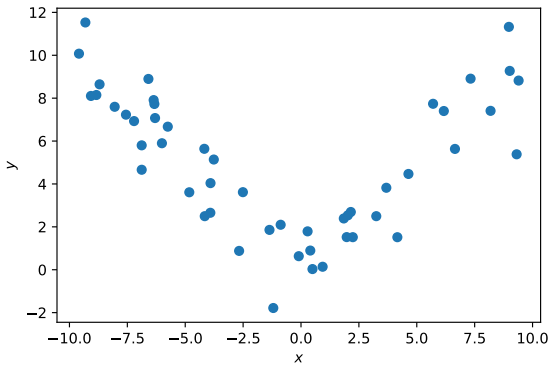
In this problem, you will train a simple deep neural network "from scratch" using only `numpy`.

This problem will make use of the following data set for regression:

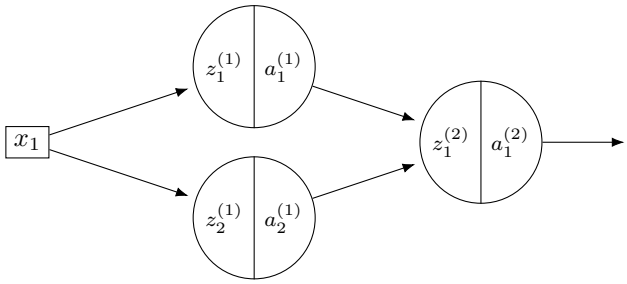
`https://f000.backblazeb2.com/file/jeldridge-data/009-noisy_abs/data.csv`

Each line in that file contains a training point; the first number being the feature, x, and the second being the target, y.

When plotted, the data looks like a noisy absolute value function:



The task is to train a neural network $H(x)$ to approximate this function. The architecture that our network will use is very simple:



We will assume that the hidden nodes use the ReLU, while the output node uses a linear activation, and that all nodes include a bias. This means that the network has seven parameters in total.

In what follows, you will be writing code that, for convenience, accepts all seven parameters as a *parameter vector* $\vec{w}$ with seven entries:[2]

$$\vec{w} = (W_{00}^{(1)}, W_{01}^{(1)}, b_0^{(1)}, b_1^{(1)}, W_{00}^{(2)}, W_{10}^{(2)}, b_0^{(2)})^T$$

You might sometimes find it more convenient to work with the matrices $W^{(1)}$ and $W^{(2)}$, as well as the bias vectors $\vec{b}^{(1)}$ and $\vec{b}^{(2)}$. Therefore, we've provided a helper function, `weights_and_biases`, that will "convert" a parameter vector to this alternative form. It is available in the following GitHub Gist:

`https://gist.github.com/eldridgejm/4acce839c661f93412860f437f2a244e`

a) Write a function, `H(x, w)`, which takes in a number, `x`, and a parameter vector `w`, and returns the result of evaluating the neural network at x with the parameters w. Use your function to compute $H(5, (1, 2, -1, -2, 3, 4, -5)^T)$. Show your code.

[2]Note that the indices on W and b start counting from zero, since numpy arrays are zero-indexed.

Hint 1: `H` should return a single number, but you might want to write a helper function which returns all of the a_{ij}^{ℓ}'s and $z_{ij}^{(\ell)}$'s as well, as they will be useful in the next problem and you need to compute them to compute $H(x)$ anyways.

Hint 2: You can check your answer by evaluating the network by hand.

Hint 3: Your answer should be a two-digit integer whose digits add to 12.

Solution: $H(5,(1,2,-1,-2,3,4,-5)^T) = 39$.

```
def relu(z):
    return np.maximum(z, 0)

def feed_forward(x, W, b):
    Z = {}
    A = {}

    Z[1] = W[1].T * x + b[1]
    A[1] = relu(Z[1])

    Z[2] = W[2].T @ A[1] + b[2]
    A[2] = Z[2]

    return Z, A

def H(x, w):
    W, b = weights_and_biases(w)
    Z, A = feed_forward(x, W, b)
    return A[2][0,0]
```

b) Write a function `del_H(x, w)` that takes in a number, `x`, and a parameter vector `w`, performs back-propagation to compute $\nabla H(x,\vec{w})$, and returns the result as a numpy array with 7 entries. Using your function, compute $\nabla H(5,(1,2,-1,-2,3,4,-5)^T)$. Show your code.

Hint 1: Don't worry about implementing backprop in its most general form. Here, you know the architecture of the network, and you can do backprop by hand; just turn the calculations that you do by hand into code and return the result.

Hint 2: Before writing the code, try running backprop by hand on the input given in the problem. This way you can more easily debug your code by printing out the $\partial H/\partial z$'s and $\partial H/\partial a$'s and comparing them with what you got by hand.

Hint 3: When your code is written correctly, the first entry of your result should be 15, and the entries should sum to 55.

Solution: $\nabla H(5,(1,2,-1,-2,3,4,-5)^T) = (15,20,3,4,4,8,1)^T$

```
def relu_prime(z):
    return np.where(z < 0, 0, 1)

def del_H(x, w):
    W, b = weights_and_biases(w)

    dz = {1: {}, 2: {}}
    da = {1: {}, 2: {}}
    dw = {1: {}, 2: {}}
```

```
    db = {1: {}, 2: {}}

    Z, A = feed_forward(x, W, b)

    # Layer 2

    dz[2][0] = np.ones_like(Z[2][0])

    dw[2][0] = A[1][0] * dz[2][0]
    dw[2][1] = A[1][1] * dz[2][0]
    db[2][0] = dz[2][0]

    # Layer 1

    da[1][0] = W[2][0,0] * dz[2][0]
    da[1][1] = W[2][1,0] * dz[2][0]

    dz[1][0] = relu_prime(Z[1][0]) * da[1][0]
    dz[1][1] = relu_prime(Z[1][1]) * da[1][1]

    dw[1][0] = x * dz[1][0]
    dw[1][1] = x * dz[1][1]
    db[1][0] = dz[1][0]
    db[1][1] = dz[1][1]

    return np.array([dw[1][0], dw[1][1], db[1][0], db[1][1], dw[2][0], dw[2][1], db[2][0]]).flat
```

c) Let's train our network using the square loss. The empirical risk with respect to the square loss is:

$$R(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n}(H(x^{(i)};\vec{w}) - y_i)^2,$$

where the summation ranges over the training data.

We will train the network using gradient descent, meaning that we need the gradient of the risk, $\nabla R(\vec{w})$. In symbols, this is:

$$\nabla R(\vec{w}) = \frac{2}{n}\sum_{i=1}^{n}(H(x^{(i)};\vec{w}) - y_i)\nabla H(x^{(i)},\vec{w}).$$

Write a function, `del_R(w)` which takes in a parameter vector `w` and returns the gradient at `w`, $\nabla R(\vec{w})$, as a numpy array with 7 entries. Use your function to compute $\nabla R((1, 2, -1, -2, 3, 4, -5)^T)$. Show your code.

Hint 1: Notice that, to compute the gradient, you'll need to sum over the 50 points in the training data provided above, but the training data is not an argument to `del_R`. That's OK: either store the training data in a global variable, or, if you'd prefer to write nicer code, write a higher-order function `make_del_R(data)` which accepts the training data and returns `del_R` as a closure.

Hint 2: When implemented correctly, your function should return an array whose elements sum to around 2135. If you get something around 2148 instead, make sure you're not using the ReLU on the output node.

Solution:

$$\nabla R((1, 2, -1, -2, 3, 4, -5)^T) = \begin{pmatrix} 301.82 \\ 402.42 \\ 40.85 \\ 54.46 \\ 86.98 \\ 173.97 \\ 13.61 \end{pmatrix}$$

```
def del_R(w):
    terms = np.vstack([(H(x, w) - y) * del_H(x, w) for (x, y) in data])
    # n = 50 in this case
    return 2 / 50 * np.sum(terms, axis=0)
```

d) Run gradient descent to train your neural network on the data. Plot your trained $H(x)$ in the interval $[-10, 10]$, on top of a scatter plot of the training data; it should fit the data reasonably well.

Hint 1: Code for gradient descent was provided in lecture. Don't worry about using *stochastic* gradient descent (though you can if you'd like…).

Hint 2: To run gradient descent, you will need to choose an initial parameter vector, a stopping criterion (threshold), and a learning rate. Try setting the initial parameter vector randomly using `np.random.uniform(0, 1, 7)`. The learning rate should be pretty small. Modify the gradient descent code to print `np.linalg.norm(x - x_new)` on every iteration; if it doesn't look like this is converging to zero, you picked too big of a step size. If gradient descent takes more than a minute or two to finish, you either set the learning rate incorrectly, or your threshold is too high.

Hint 3: Your neural network will likely not fit the data well on the first try; use your plot to determine whether or not this is the case. Remember, the neural network objective function is very non-convex, and gradient descent will likely get stuck in a local minimum. You will probabaly need to try a few random initializations of the parameter vector.

Hint 4: It can be annoying to find an initialization of the parameter vector that works just to lose it by re-running the cell on accident. You avoid this by using a random "seed". For example:

```
np.random.seed(42)
w_init = np.random.uniform(0, 1, 7)
```

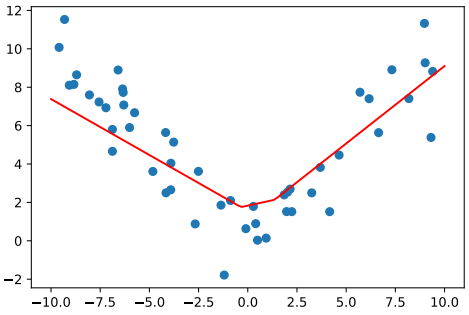
Here, `42` is the "seed". Using a seed will guarantee that `np.random.uniform` returns the same sequence of "random" numbers. Changing the seed to, say, `42`, changes the sequence.

Suggestion 1: Once you're done, take a moment to appreciate how much easier it is to train, for example, a Gaussian RBF network, or how easy `tensorflow` makes this process.

Solution:

```
np.random.seed(47)
w_init = np.random.uniform(0, 1, 7)
w_opt = gradient_descent(del_R, w_init, threshold=.00005, learning_rate=.00001)
```

This gives the following plot:


12
10
8
6
4
2
0
−2
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0