Homework 8

Due: **November 19, 5pm** (late submission until November 22, 5pm -- no submission possible afterwards)

Written assigment: 20 points

Coding assignment: 25 points

Project report: 10 points

Name: Rohitha Ravindra Myla

Link to the github repo: https://github.com/rohitharavindra08/DATA2060

Written Assignment

Neural Networks

Here, we consider a 2-layer neural network that takes inputs of dimension d, has a hidden layer of size m, and produces scalar outputs.

The network's parameters are W, b_1, v, and b_2. W is a $m \times d$ matrix, b_1 is an m-dimensional vector, v is an m-dimensional vector, and b_2 is a scalar. For an input x, the output of the first layer of the network is:

$$h = \sigma(Wx + b_1)$$

and the output of the second layer is:

$$z = v \cdot h + b_2,$$

where σ is an activation function. For this question, let σ be the sigmoid activation function σ_{sigmoid} (in the formula below, we apply it element-wise):

$$\sigma_{\text{sigmoid}}(a) = \frac{1}{1+e^{-a}}$$

We will be using the following loss function:

$$L(z) = (z - y)^2,$$

where y is a real-valued label and z is the network's output. In this problem, you will calculate the partial derivative of $L(z)$ with respect to each of the network's parameters. Let w_{ij} be the entry at the i^{th} row and j^{th} column of W. Let v_i be the i^{th} component of v. Let b_{1i} be the ith component of b_1. (Note that $1 \le i \le m$ and $1 \le j \le d$.) (Hint: For each part of the problem, apply the chain rule.)

Question 1

Calculate $\frac{\partial L(z)}{\partial b_2}$. Show your work.

Solution:

The loss function is :

$$L(z) = (z - y)^2$$

The output of the second layer is:

$$z = v \cdot h + b_2$$

Chain rule:

$$\frac{\partial L(z)}{\partial b_2} = \frac{\partial L(z)}{\partial z} \cdot \frac{\partial z}{\partial b_2}$$

We have,

$$\frac{\partial L(z)}{\partial z} = 2(z - y)$$

$$\frac{\partial z}{\partial b_2} = 1$$

Therefore,

$$\frac{\partial L(z)}{\partial b_2} = 2(z - y)$$

Question 2

Calculate $\frac{\partial L(z)}{\partial v_i}$. Show your work.

Solution:

The output of the second layer is:

$$z = \sum_{i=1}^{m} v_i h_i + b_2$$

Chain rule:

$$\frac{\partial L(z)}{\partial v_i} = \frac{\partial L(z)}{\partial z} \cdot \frac{\partial z}{\partial v_i}$$

We know that,

$$\frac{\partial L(z)}{\partial z} = 2(z - y)$$

So,

$$\frac{\partial z}{\partial v_i} = h_i$$

Therefore,

$$\frac{\partial L(z)}{\partial v_i} = 2(z - y) h_i$$

Question 3

Calculate $\frac{\partial L(z)}{\partial b_{1i}}$. Show your work.

Solution:

The first layer output is given by:

$$h = \sigma(W x + b_1)$$

(where σ is the sigmoid activation function)

We need to find:

$$\frac{\partial L(z)}{\partial b_{1i}}$$

Chain rule:

$$\frac{\partial L(z)}{\partial b_{1i}} = \frac{\partial L(z)}{\partial z} \cdot \frac{\partial z}{\partial h_i} \cdot \frac{\partial h_i}{\partial b_{1i}}$$

We already have,

$$\frac{\partial L(z)}{\partial z} = 2(z - y)$$

So,

$$\frac{\partial z}{\partial h_i} = v_i$$

Now,

$$\frac{\partial h_i}{\partial b_{1i}} = h_i(1 - h_i)$$

Therefore,

$$\frac{\partial L(z)}{\partial b_{1i}} = 2(z - y) v_i h_i (1 - h_i)$$

Question 4

Calculate $\frac{\partial L(z)}{\partial w_{ij}}$. Show your work.

Solution:

We need to find:

$$\frac{\partial L(z)}{\partial w_{ij}}$$

Chain rule:

$$\frac{\partial L(z)}{\partial w_{ij}} = \frac{\partial L(z)}{\partial z} \cdot \frac{\partial z}{\partial h_i} \cdot \frac{\partial h_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_{ij}}$$

We already have,

$$\frac{\partial L(z)}{\partial z} = 2(z - y)$$

$$\frac{\partial z}{\partial h_i} = v_i$$

$$\frac{\partial h_i}{\partial a_i} = h_i(1 - h_i)$$

Now,

$$\frac{\partial a_i}{\partial w_{ij}} = x_j$$

Therefore,

$$\frac{\partial L(z)}{\partial w_{ij}} = 2(z - y) v_i h_i (1 - h_i) x_j$$

Programming Assignment

Introduction

In this assignment, you will be implementing feed forward neural networks using stochastic gradient descent. You will implement two neural networks: a single layer neural network and a two-layer neural network. You will compare the performance of both models on the UCI Wine Dataset, which you previously used in HW2. The task is to predict the quality of a wine (scored out of 10) given various attributes of the wine (for example, acidity, alcohol content). The book section relevant to this assignment is 20.1.

Stencil Code & Data

We have provided the following stencil code:

- `Models` contains the `OneLayerNN` model and the `TwoLayerNN` model which you will be implementing.
- `Check Model` contains a series of tests to ensure you are coding your model properly.
- `Main` is the entry point of program which will read in the dataset, run the models and print the results.

You should not need to add any code to `Main`. If you do for debugging or other purposes, please make sure all of your additions are commented out in the final handin. All the functions you need to fill in reside in `Models`, marked by `TODO`s.

We have provided unit tests for the functions that compute gradients for the 2-layer neural network. These are `_layer1_weights_gradient`, `_layer1_bias_gradient`, `_layer2_weights_gradient`, and `_layer2_bias_gradient`. To enable these unit tests, uncomment `test_gradients` in the `main` function of `main.py`.

Your program assumes the data is formatted as follows: The first column of data in each file is the dependent variable (the observations y) and all other columns are the independent input variables ($x_1, x_2, \ldots, x_n$). We have taken care of all data preprocessing, as usual.

If you're curious and would like to read about the dataset, you can find more information here, but it is strongly recommended that you use the versions that we've provided in the course directory to maintain consistent formatting.

The Assignment

Neural Networks

For this assignment, we will be evaluating each model using total squared loss (or L2 loss). Recall that the L2 loss function is defined as:

$L(h) = \sum\limits_{i=1}^m \ell(h({\bf x}_i), y_i) = \sum\limits_{i=1}^m(y_{i}-h({\bf x}_{i}))^{2}$

where y_i is the target value of i^{th} sample and $h({\bf x}_{i})$ is the predicted value given the learned model weights. Each of the two models will use stochastic gradient descent to minimize this loss function.

For this assignment, you will be implementing two models:

- `OneLayerNN`: The one-layer neural network is an equivalent model to Linear Regression. It also learns linear functions of the inputs:

$$h({\bf x}) = \langle {\bf w}, {\bf x} \rangle + b.$$

  Therefore, when using squared loss, the ERM hypothesis has weights

$${\bf w} = \text{argmin}_{{\bf w}} \sum_{i = 1}^{m}(y_{i} - h({\bf x}_i))^{2} .$$

  To find the optimal set of weights, you should use Stochastic Gradient Descent. *Hint:* Compute the derivative of the loss with respect to ${\bf w}$. Then, use the SGD algorithm to minimize the loss.

- `TwoLayerNN`: For this model, you will be implement a neural network with a fully connected hidden layer.

  For an input ${\bf x}$, the output of the first layer of the network is

$${\bf v} = \sigma(W_1 {\bf x}+{\bf b}_1)$$

  and the output of the second layer is

$$h = \langle {\bf w}_2, {\bf v} \rangle + b_2 ~.$$

  σ is an activation function. In your implementation, you will take in the activation function $\sigma(a)$ as a parameter to TwoLayerNN. Additionally, you will need to pass in the derivative of the activation function $\sigma'(a)$ for training. Doing so will allow you to easily swap out the sigmoid activation function with other activation functions, such as ReLU. (You can explore other activation functions for extra credit.)

  To complete this assignment, however, you only need to train the network with the sigmoid activation function. Recall that the sigmoid activation function is (in the above formula, we apply it element-wise),

$$\sigma_{\text{sigmoid}}(a) = \frac{1}{1+e^{-a}} .$$

Training Neural Networks

The primary objective of training a neural network is to find a set of weights and biases that minimize the loss of our network, which in this case, is L2 loss. If these weights are all initialized to the same constant value, then they will all learn the same features. To avoid this, be sure that your implementation randomly initializes the weights. Numpy functions such as `np.random.normal` or `np.random.uniform` may be useful.

When training the two layer neural network, first calculate the gradients of the weights and biases for both layers before updating them. In the stencil, we have given you four methods for computing gradients: `_layer1_weights_gradient`, `_layer1_bias_gradient`, `_layer2_weights_gradient`, and `_layer2_bias_gradient`. Each of these methods should be called before performing gradient descent, i.e. before updating all of the gradients.

We also expect that you implement backpropagation as outlined in lecture i.e. computing all the outputs in the forward pass and saving them for use in the backward pass so that backpropagation achieves O(E) complexity, where E represents the number of edges in the network.

Computing Gradients

Please refer to Lecture 17, slide 34 (Backpropagation) for the definition of the gradient computations.

Remember that σ_1 and σ_2 are the activation functions at each layer, and not necessarily the same! Keep in mind that in your implementation for this assignment there should be no activation applied to the output layer of the network.

Finally, note that the initial input matrix is comprised of rows, but the input to each gradient function is a vector. This is due to contradicting conventions about how to represent training data and neural network inputs. To resolve this, you may choose to reshape or transpose the input matrix somewhere in the train method.

Important Note: External libraries that make the implementation trivial are prohibited. Specifically, `numpy.linalg.lstsqnp` (and similar functions) cannot be used in your implementation. Additionally, you **cannot** use Tensorflow or other neural network libraries. You should implement the neural networks using only Python and Numpy.

Model

Run the evironment test below, make sure you get all green checks. If not, you will lose 2 points for each red or missing sign.

```
from __future__ import print_function
from packaging.version import parse as Version
from platform import python_version

OK = '\x1b[42m[ OK ]\x1b[0m'
FAIL = "\x1b[41m[FAIL]\x1b[0m"

try:
    import importlib
except ImportError:
    print(FAIL, "Python version 3.12.5 is required,"
                " but %s is installed." % sys.version)

def import_version(pkg, min_ver, fail_msg=""):
```

```python
    mod = None
    try:
        mod = importlib.import_module(pkg)
        if pkg in {'PIL'}:
            ver = mod.VERSION
        else:
            ver = mod.__version__
        if Version(ver) == Version(min_ver):
            print(OK, "%s version %s is installed."
                  % (lib, min_ver))
        else:
            print(FAIL, "%s version %s is required, but %s installed."
                  % (lib, min_ver, ver))
    except ImportError:
        print(FAIL, '%s not installed. %s' % (pkg, fail_msg))
    return mod


# first check the python version
pyversion = Version(python_version())

if pyversion >= Version("3.12.5"):
    print(OK, "Python version is %s" % pyversion)
elif pyversion < Version("3.12.5"):
    print(FAIL, "Python version 3.12.5 is required,"
                " but %s is installed." % pyversion)
else:
    print(FAIL, "Unknown Python version: %s" % pyversion)


print()
requirements = {'matplotlib': "3.9.1", 'numpy': "2.0.1",'sklearn':
"1.5.1",
                'pandas': "2.2.2"}

# now the dependencies
for lib, required_version in list(requirements.items()):
    import_version(lib, required_version)
```

```
[ OK ] Python version is 3.12.7

[ OK ] matplotlib version 3.9.1 is installed.
[ OK ] numpy version 2.0.1 is installed.
[ OK ] sklearn version 1.5.1 is installed.
[ OK ] pandas version 2.2.2 is installed.
```

```python
import numpy as np
import random

random.seed(0)
```

```
np.random.seed(42)

def l2_loss(predictions,Y):
    '''
        Computes L2 loss (sum squared loss) between true values, Y,
and predictions.
        :param Y: A 1D Numpy array with real values (float64)
        :param predictions: A 1D Numpy array of the same size of Y
        :return: L2 loss using predictions for Y.
    '''
    # TODO
    return np.sum((predictions - Y) ** 2)

def sigmoid(x):
    '''
        Sigmoid function f(x) =  1/(1 + exp(-x))
        :param x: A scalar or Numpy array
        :return: Sigmoid function evaluated at x (applied element-wise
if it is an array)
    '''
    return np.where(x > 0, 1 / (1 + np.exp(-x)), np.exp(x) /
(np.exp(x) + np.exp(0)))

def sigmoid_derivative(x):
    '''
        First derivative of the sigmoid function with respect to x.
        :param x: A scalar or Numpy array
        :return: Derivative of sigmoid evaluated at x (applied
element-wise if it is an array)
    '''
    # TODO
    sig = sigmoid(x)
    return sig * (1 - sig)

class OneLayerNN:
    '''
        One layer neural network trained with Stocastic Gradient
Descent (SGD)
    '''
    def __init__(self):
        '''
        @attrs:
            weights: The weights of the neural network model.
            batch_size: The number of examples in each batch
            learning_rate: The learning rate to use for SGD
            epochs: The number of times to pass through the dataset
            v: The resulting predictions computed during the forward
pass
        '''
        # initialize self.weights in train()
```

```
        self.weights = None
        self.learning_rate = 0.001
        self.epochs = 25
        self.batch_size = 1

        # initialize self.v in forward_pass()
        self.v = None

    def train(self, X, Y, print_loss=True):
        '''
        Trains the OneLayerNN model using SGD.
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :param print_loss: If True, print the loss after each epoch.
        :return: None
        '''
        # TODO: initialize weights

        # TODO: Train network for certain number of epochs

        # TODO: Shuffle the examples (X) and labels (Y)

        # TODO: We need to iterate over each data point for each epoch
        # iterate through the examples in batch size increments

        # TODO: Perform the forward and backward pass on the current
batch

        # Print the loss after every epoch

        self.weights = np.random.randn(X.shape[1])

        for epoch in range(self.epochs):

            indices = np.arange(X.shape[0])
            np.random.shuffle(indices)
            X_shuffled = X[indices]
            Y_shuffled = Y[indices]

            for i in range(X.shape[0]):
                x_i = X_shuffled[i]
                y_i = Y_shuffled[i]

                self.forward_pass(x_i)

                self.backward_pass(x_i, y_i)

            if print_loss:
                print('Epoch: {} | Loss: {}'.format(epoch,
```

```
self.loss(X, Y)))

    def forward_pass(self, X):
        '''
        Computes the predictions for a single layer given examples X
and
        stores them in self.v
        :param X: 2D Numpy array where each row contains an example.
        :return: None
        '''
        # TODO:
        self.v = np.dot(X, self.weights.T)

    def backward_pass(self, X, Y):
        '''
        Computes the weights gradient and updates self.weights
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: None
        '''
        # TODO: Compute the gradients for the model's weights using
backprop

        # TODO: Update the weights using gradient descent

        predictions = self.v
        gradient = -2 * np.dot((Y - predictions), X)
        self.weights -= self.learning_rate * gradient


    def backprop(self, X, Y):
        '''
        Returns the average weights gradient for the given batch
        :param X: 2D Numpy array where each row contains an example.
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: A 1D Numpy array representing the weights gradient
        '''
        # TODO: Compute the average weights gradient
        # Refer to the SGD algorithm in slide 12 in Lecture 17:
Backpropagation

        grad_w = np.zeros_like(self.weights)

        for x_i, y_i in zip(X, Y):
            self.forward_pass(x_i)

            error = self.v - y_i
            grad_w += error * x_i
```

```
        grad_w /= X.shape[0]

        return grad_w


    def gradient_descent(self, grad_W):
        '''
        Updates the weights using the given gradient
        :param grad_W: A 1D Numpy array representing the weights
gradient
        :return: None
        '''
        # TODO: Update the weights using the given gradient and the
learning rate
        # Refer to the SGD algorithm in slide 12 in Lecture 17:
Backpropagation\

        self.weights -= self.learning_rate * grad_W

    def loss(self, X, Y):
        '''
        Returns the total squared error on some dataset (X, Y).
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: A float which is the squared error of the model on
the dataset
        '''
        # Perform the forward pass and compute the l2 loss
        self.forward_pass(X)
        return l2_loss(self.v, Y)


    def average_loss(self, X, Y):
        '''
        Returns the mean squared error on some dataset (X, Y).
        MSE = Total squared error/# of examples
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: A float which is the mean squared error of the model
on the dataset
        '''
        return self.loss(X, Y) / X.shape[0]



class TwoLayerNN:
```

```
    def __init__(self, hidden_size, activation=sigmoid,
activation_derivative=sigmoid_derivative):
        '''
        @attrs:
            activation: the activation function applied after the
first layer
            activation_derivative: the derivative of the activation
function. Used for training.
            hidden_size: The hidden size of the network (an integer)
            batch_size: The number of examples in each batch
            learning_rate: The learning rate to use for SGD
            epochs: The number of times to pass through the dataset
            wh: The first (hidden) layer weights of the neural network
model.
            bh: The first (hidden) layer bias of the neural network
model.
            wout: The second (output) layer weights of the neural
network model.
            bout: The second (output) layer bias of the neural network
model.
            a1: The output of the first layer computed during the
forward pass
            v1: The activated output of the first layer computed
during the forward pass
            a2: The output of the second layer computed during the
forward pass
            v2: The resulting predictions computed during the forward
pass (layer 2 has the identity activation function)
            output_neurons: The number of outputs of the network
        '''
        self.activation = activation
        self.activation_derivative = activation_derivative
        self.hidden_size = hidden_size
        self.learning_rate = 0.01
        self.epochs = 25
        self.batch_size = 1

        # initialize the following weights and biases in the train()
method
        self.wh = None
        self.bh = None
        self.wout = None
        self.bout = None

        # initialize the following values in the forward_pass() method
        # these values will be stored and used for the backward_pass()
        # note that you may not need to use them all in
backward_pass()
        self.a1 = None
```

```
        self.v1 = None
        self.a2 = None
        self.v2 = None


        # In this assignment, we will only use output_neurons = 1.
        self.output_neurons = 1

    def _get_layer2_bias_gradient(self, x, y):
        '''
        Computes the gradient of the loss with respect to the output
bias, bout.
        :param x: Numpy array for a single training example with
dimension: input_size by 1
        :param y: Label for the training example
        :return: the partial derivates dL/dbout, a numpy array of
dimension: output_neurons by 1
        '''
        # TODO:
        return 2 * (self.v2 - y)

    def _get_layer2_weights_gradient(self, x, y):
        '''
        Computes the gradient of the loss with respect to the output
weights, wout.
        :param x: Numpy array for a single training example with
dimension: input_size by 1
        :param y: Label for the training example
        :return: the partial derivates dL/dwout, a numpy array of
dimension: output_neurons by hidden_size
        '''
        # TODO:
        return np.dot(2 * (self.v2 - y), self.v1.T)

    def _get_layer1_bias_gradient(self, x, y):
        '''
        Computes the gradient of the loss with respect to the hidden
bias, bh.
        :param x: Numpy array for a single training example with
dimension: input_size by 1
        :param y: Label for the training example
        :return: the partial derivates dL/dbh, a numpy array of
dimension: hidden_size by 1
        '''
        # TODO:

        delta = 2 * (self.v2 - y) * self.wout.T *
self.activation_derivative(self.a1)
        return delta
```

```
    def _get_layer1_weights_gradient(self, x, y):
        '''
        Computes the gradient of the loss with respect to the hidden
weights, wh.
        :param x: Numpy array for a single training example with
dimension: input_size by 1
        :param y: Label for the training example
        :return: the partial derivates dL/dwh, a numpy array of
dimension: hidden_size by input_size
        '''
        # TODO:

        delta = 2 * (self.v2 - y) * self.wout.T *
self.activation_derivative(self.a1)
        return np.dot(delta, x.T.reshape(1, -1))

    def train(self, X, Y, print_loss=True):
        '''
        Trains the TwoLayerNN with SGD using Backpropagation.
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :param learning_rate: The learning rate to use for SGD
        :param epochs: The number of times to pass through the dataset
        :param print_loss: If True, print the loss after each epoch.
        :return: None
        '''
        # NOTE:
        # Use numpy arrays of the following dimensions for your
model's parameters.
        # layer 1 weights (wh): hidden_size x input_size
        # layer 1 bias (bh): hidden_size x 1
        # layer 2 weights (wout): output_neurons x hidden_size
        # layer 2 bias (bout): output_neurons x 1
        # HINT: for best performance initialize weights with
np.random.normal or np.random.uniform

        # TODO: Weight and bias initialization

        # TODO: Train network for certain number of epochs

        # TODO: Shuffle the examples (X) and labels (Y)

        # TODO: We need to iterate over each data point for each epoch
        # iterate through the examples in batch size increments

        # TODO: Perform the forward and backward pass on the current
batch

        # Print the loss after every epoch
```

```
        input_size = X.shape[1]
        self.wh = np.random.uniform(-0.1, 0.1, (self.hidden_size,
X.shape[1]))
        self.bh = np.random.uniform(-0.1, 0.1, (self.hidden_size, 1))
        self.wout = np.random.uniform(-0.1, 0.1, (self.output_neurons,
self.hidden_size))
        self.bout = np.random.uniform(-0.1, 0.1, (self.output_neurons,
1))
        for epoch in range(self.epochs):
            for i in range(X.shape[0]):
                x_i = X[i].reshape(-1, 1)
                y_i = np.array([[Y[i]]])
                self.forward_pass(x_i)
                self.backward_pass(x_i, y_i)
            if print_loss:
                print(f'Epoch: {epoch} | Loss: {self.average_loss(X,
Y)}')


    def forward_pass(self, X):
        '''
        Computes the predictions for a 2 layer NN given examples X and
        stores them in self.v2.
        Stores intermediate values before the prediction task in
self.v1 and
        self.a1
        :param X: 2D Numpy array where each row contains an example.
        :return: None
        '''
        # TODO:
        if X.ndim == 1:
            X = X.reshape(-1, 1)

        self.a1 = np.dot(self.wh, X) + self.bh
        self.v1 = self.activation(self.a1)
        self.a2 = np.dot(self.wout, self.v1) + self.bout
        self.v2 = self.a2

    def backward_pass(self, X, Y):
        '''
        Computes the weights gradient and updates all four weights and
bias gradients
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: None
        '''
        # TODO: Compute the gradients for the model's weights using
backprop
```

```
        # TODO: Update the weights using gradient descent
        X = X.reshape(-1, 1)
        grad_wh, grad_bh, grad_wout, grad_bout = self.backprop(X, Y)
        self.gradient_descent(grad_wh, grad_bh, grad_wout, grad_bout)

    def backprop(self, X, Y):
        '''
        Computes the average weights and biases gradients for the
given batch
        :param X: 2D Numpy array where each row contains an example.
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: 4 Numpy arrays representing the computed gradients
for each weight and bias
        '''
        # TODO: Call the "get gradient" methods

        if X.ndim == 1:
            X = X.reshape(1, -1)
        if np.isscalar(Y) or Y.ndim == 0:
            Y = np.array([Y])

        self.forward_pass(X)

        error = self.v2 - Y

        grad_wout = np.dot(error, self.v1.T)
        grad_bout = error

        delta1 = np.dot(self.wout.T, error) *
self.activation_derivative(self.a1)

        grad_wh = np.dot(delta1, X.T)
        grad_bh = delta1

        return grad_wh, grad_bh, grad_wout, grad_bout


    def gradient_descent(self, grad_wh, grad_bh, grad_wout,
grad_bout):
        '''
        Updates the weights using the given gradients
        :param grad_wh: Numpy array representing the hidden weights
gradient
        :param grad_bh: Numpy array representing the hidden bias
gradient
        :param grad_wout: Numpy array representing the output weights
gradient
        :param grad_bout: Numpy array representing the output bias
```

```
gradient
        :return: None
        '''
        # TODO: Update the weights using the given gradients and the
learning rate
        # Refer to the SGD algorithm in slide 12 in Lecture 17:
Backpropagation

        self.wh -= self.learning_rate * grad_wh
        self.bh -= self.learning_rate * grad_bh
        self.wout -= self.learning_rate * grad_wout
        self.bout -= self.learning_rate * grad_bout

    def loss(self, X, Y):
        '''
        Returns the total squared error on some dataset (X, Y).
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: A float which is the squared error of the model on
the dataset
        '''
        # Perform the forward pass and compute the l2 loss
        self.forward_pass(X.T)
        return l2_loss(self.v2.flatten(), Y)

    def average_loss(self, X, Y):
        '''
        Returns the mean squared error on some dataset (X, Y).
        MSE = Total squared error/# of examples
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values
for each example
        :return: A float which is the mean squared error of the model
on the dataset
        '''
        return self.loss(X, Y) / X.shape[0]
```

Check Model

```
 import pytest
# Sets random seed for testing purposes
random.seed(0)
np.random.seed(0)


def test_OneLayerNN():
    '''
    Tests for OneLayerNN Model Weights Gradient
```

```
    '''

    test_model = OneLayerNN()

    # Creates Test Data
    x_bias = np.array([[0,4,1], [0,3,1], [5,0,1], [4,1,1], [0,5,1]])
    y = np.array([0,0,1,1,0])

    # Test Model Train
    test_model.train(x_bias, y, print_loss=False)

    act_weights = test_model.weights
    exp_weights = np.array([[ 0.17817953, -0.03543112,  0.34761945]])

    print('----Testing 1-Layer NN Gradients-----')

    print("\nTesting layer one weights gradient.")

    # Test layer 1 weights
    if not hasattr(act_weights, "shape"):
        print("Layer one weights gradient is not a numpy array. \n")
    elif act_weights.shape != (1, 3):
        print(
            f"Incorrect shape for layer one weights gradient.\
nExpected: {(1, 3)} \nActual: {act_weights.shape} \n")
    elif not act_weights == pytest.approx(exp_weights, .01):
        print(
            f"Incorrect values for layer one weights gradient.\
nExpected: {exp_weights} \nActual: {act_weights} \n")
    else:
        print("Layer one weights gradient is correct.\n")


def test_gradients_TwoLayerNN():
    '''
    Tests the gradient functions of TwoLayerNN
    '''
    model = TwoLayerNN(2)

    # Fake training example
    x = np.array([[15.0,-5.5]])
    y = np.array([2.0])

    input_neurons = 2
    model.wh = np.array(
        [[-0.17795209, -0.0759435],
        [-0.01952383, 0.13401861]]
    )
    model.bh = np.array(
        [[0.],
```

```
        [0.]]
    )
    model.wout = np.array([[-0.22206318, -0.17802104]])
    model.bout = np.array([[0.]])

    model.a1 = np.array(
        [[-2.25159215],
        [-1.02995984]]
    )
    model.v1 = np.array(
        [[0.09521222],
        [0.26309189]]
    )
    model.v2 = np.array([[-0.06797902]])

    # Expected gradients
    expected_layer1_weights = np.array([[1.18681586, -0.43516582],
[2.14121126, -0.7851108]])
    expected_layer1_bias = np.array([[0.07912106], [0.14274742]])
    expected_layer2_weights = np.array([[-0.39379374, -1.08813702]])
    expected_layer2_bias = np.array([[-4.13595804]])

    print('----Testing 2-Layer NN Gradients-----')

     # Test layer 1 weights
    print("\nTesting layer one weights gradient.")
    actual_layer1_weights = model._get_layer1_weights_gradient(x, y)
    if not hasattr(actual_layer1_weights, "shape"):
        print("Layer one weights gradient is not a numpy array.")
    elif actual_layer1_weights.shape != expected_layer1_weights.shape:
        print(
            "Incorrect shape for layer one weights gradient.\
nExpected: {0}\nActual: {1}".format(
            expected_layer1_weights.shape,
actual_layer1_weights.shape))
    elif not np.all(np.isclose(actual_layer1_weights,
expected_layer1_weights)):
        print(
            "Incorrect values for layer one weights gradient.\
nExpected: {0}\nActual: {1}".format(
            expected_layer1_weights, actual_layer1_weights))
    else:
        print("Layer one weights gradient is correct.")

    # Test layer 1 bias
    print("\nTesting layer one bias gradient.")
    actual_layer1_bias = model._get_layer1_bias_gradient(x, y)
    if not hasattr(actual_layer1_bias, "shape"):
        print("Layer one bias gradient is not a numpy array.")
    elif actual_layer1_bias.shape != expected_layer1_bias.shape:
```

```
        print(
            "Incorrect shape for layer one bias gradient.\nExpected:
{0}\nActual: {1}".format(
            expected_layer1_bias.shape, actual_layer1_bias.shape))
    elif not np.all(np.isclose(actual_layer1_bias,
expected_layer1_bias)):
        print(
            "Incorrect values for layer one bias gradient.\nExpected:
{0}\nActual: {1}".format(
            expected_layer1_bias, actual_layer1_bias))
    else:
        print("Layer one bias gradient is correct.")

    # Test layer 2 weights
    print("\nTesting layer two weights gradient.")
    actual_layer2_weights = model._get_layer2_weights_gradient(x, y)
    if not hasattr(actual_layer2_weights, "shape"):
        print("Layer two weights gradient is not a numpy array.")
    elif actual_layer2_weights.shape != expected_layer2_weights.shape:
        print(
            "Incorrect shape for layer two weights gradient.\
nExpected: {0}\nActual: {1}".format(
            expected_layer2_weights.shape,
actual_layer2_weights.shape))
    elif not np.all(np.isclose(actual_layer2_weights,
expected_layer2_weights)):
        print(
            "Incorrect values for layer two weights gradient.\
nExpected: {0}\nActual: {1}".format(
            expected_layer2_weights, actual_layer2_weights))
    else:
        print("Layer two weights gradient is correct.")

    # Test layer 2 bias
    print("\nTesting layer two bias gradient.")
    actual_layer2_bias = model._get_layer2_bias_gradient(x, y)
    if not hasattr(actual_layer2_bias, "shape"):
        print("Layer two bias gradient is not a numpy array.")
    elif actual_layer2_bias.shape != expected_layer2_bias.shape:
        print(
            "Incorrect shape for layer two bias gradient.\nExpected:
{0}\nActual: {1}\n".format(
            expected_layer2_bias.shape, actual_layer2_bias.shape))
    elif not np.all(np.isclose(actual_layer2_bias,
expected_layer2_bias)):
        print(
            "Incorrect values for layer two bias gradient.\nExpected:
{0}\nActual: {1}\n".format(
            expected_layer2_bias, actual_layer2_bias))
```

```
    else:
        print("Layer two bias gradient is correct.\n")

# Run to Test OneLayerNN
test_OneLayerNN()

# Run to Test TwoLayerNN
test_gradients_TwoLayerNN()
```

```
----Testing 1-Layer NN Gradients-----

Testing layer one weights gradient.
Incorrect shape for layer one weights gradient.
Expected: (1, 3)
Actual: (3,)

----Testing 2-Layer NN Gradients-----

Testing layer one weights gradient.
Layer one weights gradient is correct.

Testing layer one bias gradient.
Layer one bias gradient is correct.

Testing layer two weights gradient.
Layer two weights gradient is correct.

Testing layer two bias gradient.
Layer two bias gradient is correct.
```

Main

```
import os
from sklearn.model_selection import train_test_split


def test_models(dataset, test_size=0.2):
    '''
        Tests LinearRegression, OneLayerNN, TwoLayerNN on a given
dataset.
        :param dataset The path to the dataset
        :return None
    '''

    # Check if the file exists
    if not os.path.exists(dataset):
        print('The file {} does not exist'.format(dataset))
        exit()
```

```
    # Load in the dataset
    data = np.loadtxt(dataset, skiprows = 1)
    X, Y = data[:, 1:], data[:, 0]

    # Normalize the features
    X = (X-np.mean(X, axis=0))/np.std(X, axis=0)

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=test_size)

    print('Running models on {} dataset'.format(dataset))

    # Add a bias
    X_train_b = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
    X_test_b = np.append(X_test, np.ones((len(X_test), 1)), axis=1)

    #### 1-Layer NN ######
    print('----- 1-Layer NN -----')
    nnmodel = OneLayerNN()
    nnmodel.train(X_train_b, Y_train, print_loss=False)
    print('Average Training Loss:', nnmodel.average_loss(X_train_b,
Y_train))
    print('Average Testing Loss:', nnmodel.average_loss(X_test_b,
Y_test))

    #### 2-Layer NN ######
    print('----- 2-Layer NN -----')
    model = TwoLayerNN(10)
    # Use X without a bias, since we learn a bias in the 2 layer NN.
    model.train(X_train, Y_train, print_loss=False)
    print('Average Training Loss:', model.average_loss(X_train,
Y_train))
    print('Average Testing Loss:', model.average_loss(X_test, Y_test))


# Set random seeds. DO NOT CHANGE THIS IN YOUR FINAL SUBMISSION.
random.seed(0)
np.random.seed(0)
# Uncomment to test gradient calculating functions for 2-layer NN
test_models('/Users/rohitharavindramyla/Desktop/DATA
2060/HW/wine.txt')
# test_gradients()
```

```
Running models on /Users/rohitharavindramyla/Desktop/DATA
2060/HW/wine.txt dataset
----- 1-Layer NN -----
Average Training Loss: 0.5471914144813309
Average Testing Loss: 0.6720387035747013
----- 2-Layer NN -----
```

```
Average Training Loss: 0.4980827476444488
Average Testing Loss: 0.601848698912215
```

Project Report

Question 1

Compare the average loss of the two models. Provide an explanation for what you observe.

Solution:

The average losses for the two models are,

1-Layer NN: Average Training Loss: 0.5471 Average Testing Loss: 0.672

2-Layer NN: Average Training Loss: 0.498 Average Testing Loss: 0.6018

The 2-layer neural network clearly performed better than the 1-layer network since it had lower training and testing losses. This makes sense because the 2-layer network has an additional hidden layer, which allows it to capture more complex patterns and relationships in the data.

The sigmoid activation function in the 2-layer network was a key part of this because it introduces non-linearity. This helps the model learn dependencies that a single-layer network (which is essentially just linear regression) wouldn't be able to pick up on.

One thing to note is the slight gap between training and testing losses in both models. It suggests a bit of overfitting, but it's not too bad—it's still within a reasonable range. Overall, the 2-layer network's extra capacity and non-linear learning capabilities gave it the edge.

Question 2

Comment on your parameter choices. These include the learning rate, the hidden layer size and the number of epochs for training.

Solution:

For the 2-layer NN, the parameters were chosen to strike a good balance between keeping the model complex enough to capture patterns and stable enough to train effectively. The hidden layer with 10 neurons worked well for learning non-linear relationships in the dataset without making the model too large or prone to overfitting. Using the sigmoid activation function added non-linearity, which is crucial for this kind of task, and the derivative helped ensure smooth updates during backpropagation. The learning rate of 0.01 was a good choice—it allowed the model to converge steadily without jumping around or taking forever to train. With 25 epochs, the model had enough time to learn what it needed to, and using a batch size of 1 (SGD) added just the right amount of randomness to help it generalize better.

For the 1-layer NN, the parameters reflected its straightforward design as a baseline model. Since it didn't have any hidden layers, it essentially performed linear regression. The learning rate was set to 0.001, which made sense because linear models converge faster, and a smaller rate ensures precise updates without overshooting. We trained it for 25 epochs, which gave it plenty of time to learn the linear relationships in the data. The batch size of 1 matched the setup of the 2-layer NN, keeping things consistent and ensuring a fair comparison between the two models.

Therfore, the parameters for both models made sense given their structures. The 2-layer NN was more complex, and its parameters allowed it to outperform the 1-layer NN by capturing non-linear patterns. On the other hand, the 1-layer NN worked well as a baseline, showing the limitations of linear modeling. Together, these setups gave us a clear comparison and highlighted how adding hidden layers and non-linear activations improves performance.

Question 3

Among machine learning techniques, neural networks have a reputation for being 'black boxes', where the logic of their decision making is difficult or impossible for humans to interpret.

1. If a 'black box' model gives an answer that disagrees with a human expert, which answer should be believed? Are there circumstances where we should believe one party more often than the other?
2. Companies often hide the logic behind their products by making their software closed-source. Are there differences between companies selling closed source software, where the logic is known but hidden, versus companies selling 'black box' artificial intelligence software, where the logic might be altogether unknown? Is using one type of software more justifiable than using the other?

Please credit any outside sources you use in your answer.

Solution:

1. Honestly, it depends on the situation and what both the model and the human bring to the table.

If the model is trained on a solid, representative dataset and has consistently performed well on similar tasks, then it probably deserves the benefit of the doubt. Models are great at handling large-scale patterns and processing tons of data quickly—things that might be hard for a human to do. For example, in fraud detection, where even subtle patterns matter, the model might have an edge.

But humans bring something that models don’t: intuition and the ability to handle unique situations or things that fall outside the model’s training. In rare cases or areas like medical diagnostics, a doctor might pick up on something the model just wouldn’t recognize because it hasn’t seen it before.

So, who to trust? it’s maybe about combining both strengths. Use the model for consistency and pattern recognition, but let the human step in while making ethical decisions or rare cases come up. If the stakes are high, a disagreement should always lead to a deeper review instead of blindly trusting one side.

2. Yes, I think there’s definitely a difference between the two.

With closed-source software, the logic is set and predictable, even if you can’t see the source code. It’s easier to test or audit, which makes it more reliable for things like healthcare or finance, where transparency is important. For example, even if you don't know how it's built, regulators can step in and check whether it's working as intended.

Where as, black-box AI software works differently because it learns its logic from data. This makes it super powerful for things like image recognition or language processing, where traditional programming doesn't work as well. But the downside is, even the developers might not fully understand how it's making decisions, which can be a problem in high-stakes situations.

So, which one's better, It depends. Closed-source software is definitely safer when you need accountability and transparency. But black-box AI can be justified in areas where performance and innovation matter more than explainability, like product recommendations or automation. Either way, if we are using black-box AI, it's really important for companies to validate it thoroughly, set clear boundaries for its use, and make sure users understand its limitations.