CS69000DPL - Deep Learning

Spring 2020

Instructor and Copyright: Bruno Ribeiro (https://www.cs.purdue.edu/homes/ribeirob/)

Homework 1

Due date: 1/28 (Tuesday) 6:00am

Homework: MLP Classifier

Q0 (REQUIRED): Mark the appropriate answers with an X

(0pts correct answer, -1,000pts incorrect answer: (0,-1,000) pts): A correct answer to the following questions is worth 0pts. An incorrect answer is worth -1,000pts, which carries over to other homeworks and exams, and can result in an F grade in the course.

(1) Student interaction with other students / individuals:

- (a) I have copied part of my homework from another student or another person (plagiarism).
- (b) Yes, I discussed the homework with another person but came up with my own answers. Their name(s) is (are)
- (c) No, I did not discuss the homework with anyone

(2) On using online resources:

- (a) I have copied one of my answers directly from a website (plagiarism).
- (b) I have used online resources to help me answer this question, but I came up with my own answers (you are allowed to use online resources as long as the answer is your own). Here is a list of the websites I have used in this homework:
- (c) I have not used any online resources except the ones provided in the course website.

Q1: Task (10 pts)

In this homework you will have to modify this Jupyter notebook to

1. Add one extra hidden layer
2. Change the ReLU activations to sigmoid activations at all hidden layers
3. Not change anything else (i.e., keep the softmax activation, etc.)
4. Change the textual description of the neural network, its foward and backward passes

Details:

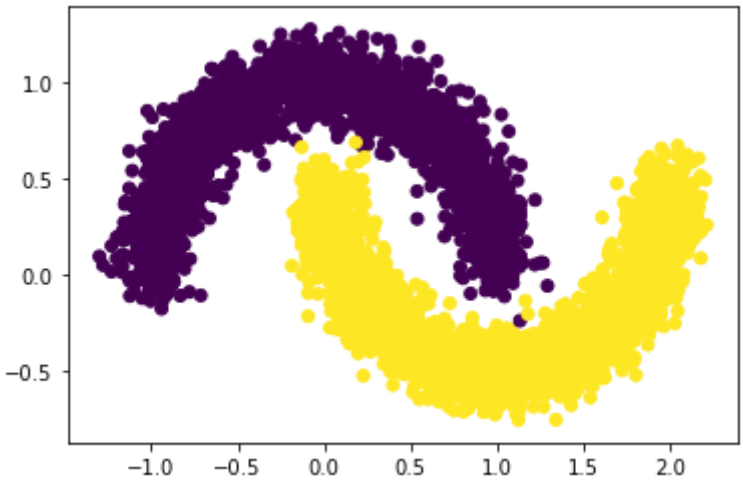
- You should change the textual mathematical description of the backpropagation procedure to accomodate the change from ReLU to sigmoid and the addition of an extra hidden layer. Finding where you need to change the description is considered part of the homework.
- The extra hidden layer should have the same number of neurons.
- You must change the `forward` and `backward` functions in the code to accomodate the change from ReLU to sigmoid and the addition of one hidden layer
- Run the python notebook and submit the resulting .ipynb file with your modifications and the resulting test accuracy and visual test evaluation
- Make sure you have made all changes and saved them before submitting your homework
- Make sure you show your derivations to get partial credit.
- Comment your changes in the code to help us give partial credit.

Code and description

```
%matplotlib inline
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
import pylab
import numpy as np

X, y = make_moons(n_samples=5000, random_state=42, noise=0.1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42,test_size=0.3)

pylab.scatter(X[:,0], X[:,1], c=y)
pylab.show()
```



Create the Neural Network Weights and Initialize Them

We first define:

- the input layer (two coordinates)
- number of hidden layers (we will use two)
- the number of output neurons (number of classes, two in our case)

```
# There are only two features in the data X[:,0] and X[:,1]
n_feature = 2
# There are only two classes: 0 (purple) and 1 (yellow)
n_class = 2

def init_weights(n_hidden=100):
    # Initialize weights with Standard Normal random variables
    model = dict(
        W1=np.random.randn(n_feature + 1, n_hidden),
        W2=np.random.randn(n_hidden + 1, n_hidden),
        W3=np.random.randn(n_hidden + 1, n_class)
    )

    return model
```

Define the nonlinear activation function (will be used in the last layer)

We will use the softmax function.

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

If there are only two classes, the softmax is equivalent to the logistic function (do the math to convince yourself).

Python + Numpy tricks

Numpy is *very* handy. If we give a vector $\mathbf{z}$, np.exp($\mathbf{z}$) returns a vector with all elements of z exponentiated.

If $\mathbf{z}$ is a vector, $\mathbf{z}$.sum() returns the sum of the elements in $\mathbf{z}$.

```
# Defines the softmax function. For two classes, this is equivalent to the logistic regression
def softmax(x):
    return np.exp(x) / np.exp(x).sum()
```

Define the forward pass

Here, we define how the neural network get an input x and use the model parameters (weights) $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ and $\mathbf{W}^{(3)}$ and biases $\mathbf{b}^{(1)}$ and $\mathbf{b}^{(2)}$ and $\mathbf{b}^{(3)}$ to predict the class labels of the input.

All our vectors are column vectors, the number of neurons in each layer is denoted by $m_0, m_1, \ldots, m_K$, where m_0 is the dimension of the input. The weight matrix $\mathbf{W}^{(k)}$, thus has dimensions $m_k \times m_{k-1}$.

Hidden layer activation

From the input vector $\mathbf{x}$ to the first hidden layer neurons $h^{(1)}$, we need to get the intermediate value

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

and pass it through an activation function.

Our activation function is the Sigmoid:

$$\sigma(z) = \frac{1}{1+e^{-z}},$$

in which all the operations are performed elementwise.

Thus,

$$\mathbf{h}^{(1)} = \sigma(\mathbf{z}^{(1)})$$

Once we get the hidden layer values $\mathbf{h}^{(1)}$, we do

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}$$

and pass it through the sigmoid activation function.

Thus,

$$\mathbf{h}^{(2)} = \sigma(\mathbf{z}^{(2)})$$

Once we get the hidden layer values $\mathbf{h}^{(2)}$, we do

$$\mathbf{z}^{(3)} = \mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}$$

and pass it through the activation of the last layer

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{(3)}).$$

```
def sigmoid(z):
    return 1/(1+np.exp(-z))

# For a single example $x$
def forward(x, model):
    x = np.append(x, 1)

    # Input times first layer matrix
    z_1 = x @ model['W1']
    # Sigmoid activation goes to first hidden layer
    h_1 = sigmoid(z_1)

    # First hidden layer times second layer matrix
    h_1 = np.append(h_1, 1)
    z_2 = h_1 @ model['W2']
    # Sigmoid activation goes to second hidden layer
    h_2 = sigmoid(z_2)

    # Second hidden layer values to output
    h_2 = np.append(h_2, 1)
    hat_y = softmax(h_2 @ model['W3'])

    return h_1, h_2, hat_y
```

Define backpropagation

Now, we need to backpropagate the derivatives of each example

The backpropagation function gets:

- all input data $\{\mathbf{x}(i)\}_i$
- corresponding hidden values of *all* training examples, for *all* layers: $\{\mathbf{h}^{(1)}(i), \mathbf{h}^{(2)}(i)\}_i$
- errors of each output neuron for *all* training examples: $\{\mathbf{y}(i) - \hat{\mathbf{y}}(i)\}_i$ (this is a subtraction of two vectors)

Our score function of training example i will be, which is the log likelihood of a one-hot enconding vector of the classes,

$$L(i) = \sum_j y_j(i) \log \hat{y}_j(i),$$

where $y_i(j)$ is the element j of the vector representing the one-hot enconding of the class of training example i and $\hat{y}_j(i)$ is the output of the j-th output neuron for training example i,

$$\hat{y}_j(i) = \frac{\exp(z_j^{(2)})}{\sum_m \exp(z_m^{(2)})}.$$

The derivative of the loss with respect to $\mathbf{z}^{(3)}$ will be a vector (that we will represent using the variables without the index j)

$$\frac{\partial L(i)}{\partial \mathbf{z}^{(3)}} = (y_1(i) - \hat{y}_1(i), \dots, y_{\text{n\_class}}(i) - \hat{y}_{\text{n\_class}}(i)) = \mathbf{y}(i) - \hat{\mathbf{y}}(i).$$

Note that we can get this derivative directly from the relative error `errs` **.**

Other derivatives

The derivative of a sigmoid function, which we will need, is given by:

$$\frac{\partial \sigma(\mathbf{z})}{\partial \mathbf{z}} = \sigma(\mathbf{z})\left(1 - \sigma(\mathbf{z})\right).$$

Next, we will compute the following derivatives for each training example:

- The derivative of the last layer parameters are $$ \frac{\partial L(i)}{\partial {\bf W}^{(3)}} = \frac{\partial L(i)}{\partial {\bf z}^{(3)}}

  ```
  \frac{\partial {\bf z}^{(3)}(i)}{\partial {\bf W}^{(3)}}
  ```

  = \left({\bf y}(i) - \hat{\bf y}(i)\right) {\bf h}^{(2)}(i)

  and

  \frac{\partial L(i)}{\partial {\bf b}^{(3)}} = \frac{\partial L(i)}{\partial {\bf z}^{(3)}}

  ```
  \frac{\partial {\bf z}^{(3)}(i)}{\partial {\bf b}^{(3)}}
  ```

  = \left({\bf y}(i) - \hat{\bf y}(i)\right),

  as

  \frac{\partial {\bf z}^{(3)}(i)}{\partial {\bf b}^{(3)}} = 1. $$
- The derivatives with respect to the hidden values of the previous layer are $$ \frac{\partial L(i)}{\partial {\bf h}^{(2)}} = \frac{\partial {\bf z}^{(3)}(i)}{\partial {\bf h}^{(2)}}

  ```
  \frac{\partial L(i)}{\partial {\bf z}^{(3)}}
  ```

  = {{\bf W}^{(3)}}^T \left({\bf y}(i) - \hat{\bf y}(i)\right) $$

We then backpropagate $\partial L(i)/\partial \mathbf{h}^{(2)}$ to the previous layer.

We compute first

$$\frac{\partial L(i)}{\partial \mathbf{z}^{(2)}} = \frac{\partial L(i)}{\partial \mathbf{h}^{(2)}} \frac{\partial \mathbf{h}(i)}{\partial \mathbf{z}^{(2)}} = \left({\mathbf{W}^{(3)}}^T \left(\mathbf{y}(i) - \hat{\mathbf{y}}(i) \right) \right) \odot \left(\mathbf{h}^{(2)}(i)(1 - \mathbf{h}^{(2)}(i)) \right),$$

where $\odot$ is the Hadamard (elementwise) product.

The derivatives for the next layer are:

$$\frac{\partial L(i)}{\partial \mathbf{W}^{(2)}} = \frac{\partial L(i)}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}(i)}{\partial \mathbf{W}^{(2)}} = \frac{\partial L(i)}{\partial \mathbf{z}^{(2)}} (\mathbf{h}^{(1)}(i))^T,$$

and

$$\frac{\partial L(i)}{\partial \mathbf{b}^{(2)}} = \frac{\partial L(i)}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}(i)}{\partial \mathbf{b}^{(2)}} = \frac{\partial L(i)}{\partial \mathbf{z}^{(2)}},$$

since

$$\frac{\partial \mathbf{z}^{(2)}(i)}{\partial \mathbf{W}^{(2)}} = (\mathbf{h}^{(1)}(i))^T,$$

and

$$\frac{\partial \mathbf{z}^{(2)}(i)}{\partial \mathbf{b}^{(2)}} = 1.$$

- The derivatives with respect to the first hidden values of the previous layer are

$$\frac{\partial L(i)}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{z}^{(2)}(i)}{\partial \mathbf{h}^{(1)}} \frac{\partial L(i)}{\partial \mathbf{z}^{(2)}} = {\mathbf{W}^{(2)}}^T \frac{\partial L(i)}{\partial \mathbf{z}^{(2)}}$$

We then backpropagate $\partial L(i)/\partial \mathbf{h}^{(1)}$ to the previous layer.

Again, we compute

$$\frac{\partial L(i)}{\partial \mathbf{z}^{(1)}} = \frac{\partial L(i)}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{z}^{(1)}} = \frac{\partial L(i)}{\partial \mathbf{h}^{(1)}} \odot \left(\mathbf{h}^{(1)}(1-\mathbf{h}^{(1)})\right),$$

Which can then be used to give us the final derivatives:

$$\frac{\partial L(i)}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathbf{z}^{(1)}(i)}{\partial \mathbf{W}^{(1)}} \frac{\partial L(i)}{\partial \mathbf{z}^{(1)}} = \frac{\partial L(i)}{\partial \mathbf{z}^{(1)}} (\mathbf{x}(i))^T,$$

and

$$\frac{\partial L(i)}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathbf{z}^{(1)}(i)}{\partial \mathbf{b}^{(1)}} \frac{\partial L(i)}{\partial \mathbf{z}^{(1)}} = \frac{\partial L(i)}{\partial \mathbf{z}^{(1)}},$$

since

$$\frac{\partial \mathbf{z}^{(1)}(i)}{\partial \mathbf{W}^{(1)}} = (\mathbf{x}(i))^T,$$

and

$$\frac{\partial \mathbf{z}^{(1)}(i)}{\partial \mathbf{b}^{(1)}} = 1.$$

The final output are the derivatives of the parameters.

In the above, we have described the backpropagation algorithm *per training example*. The following python code will, as described earlier, give all examples as inputs. Thus, the input is a matrix whose rows are the vectors of each training example.

This function outputs

$$\mathbf{dW}^{(1)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{W}^{(1)}},$$

$$\mathbf{db}^{(1)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{b}^{(1)}},$$

$$\mathbf{dW}^{(2)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{W}^{(2)}},$$

$$\mathbf{db}^{(2)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{b}^{(2)}}.$$

$$\mathbf{dW}^{(3)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{W}^{(3)}},$$

and

$$\mathbf{db}^{(3)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{b}^{(3)}}.$$

```
def backward(model, xs, h1s, h2s, errs):
    """xs, h1s, h2s, errs contain all information (input, hidden state, error) of all data in the minibatch"""
    # errs is the gradients of output layer for the minibatch
    dW3 = (h2s.T @ errs)/xs.shape[0]

    # Get gradient of second hidden layer
    dh2 = errs @ model['W3'].T
    dz2 = dh2 * h2s * (1 - h2s)

    # The bias "neuron" is the constant 1, we don't need to backpropagate its gradient
    # since it has no inputs, so we just remove its column from the gradient
    dz2 = dz2[:, :-1]

    # Get gradient of second weight matrix
    dW2 = (h1s.T @ dz2) / xs.shape[0]

    # Get gradient of first hidden layer
    dh1 = dz2 @ model['W2'].T
    dz1 = dh1 * (h1s * (1 - h1s))

    # The bias "neuron" is the constant 1, we don't need to backpropagate its gradient
    # since it has no inputs, so we just remove its column from the gradient
    dz1 = dz1[:, :-1]

    # Add the 1 to the data, to compute the gradient of W1
    xs = np.hstack([xs, np.ones((xs.shape[0], 1))])

    dW1 = (xs.T @ dz1) / xs.shape[0]

    return dict(W1=dW1, W2=dW2, W3=dW3)
```

Perform the forward and backward procedures to get gradients

For each input example i in the training data, perform a forward pass and:

- store all the hidden units of all the hidden layers associated with example i (we only have to store one vector of hidden values)
- store the gradient of the error of example i with respect to the prediction

Once we have store all hidden layer values and all the derivatives of all examples, we will do the backard pass and return the derivatives of the error with respect to the paramters $\mathbf{W}^{(1)}$, $\mathbf{b}^{(1)}$, $\mathbf{W}^{(2)}$, $\mathbf{b}^{(2)}$, $\mathbf{W}^{(3)}$, and $\mathbf{b}^{(3)}$.

```
def get_gradient(model, X_train, y_train):
    xs, h1s, h2s, errs = [], [], [], []

    for x, cls_idx in zip(X_train, y_train):
        h_1, h_2, y_pred = forward(x, model)

        # Create one-hot coding of true label
        y_true = np.zeros(n_class)
        y_true[int(cls_idx)] = 1.

        # Compute the gradient of output layer
        err = y_true - y_pred

        # Accumulate the informations of the examples
        # x: input
        # h1: hidden state 1
        # h2: hidden state 2
        # err: gradient of output layer
        xs.append(x)
        h1s.append(h_1)
        h2s.append(h_2)
        errs.append(err)

    # Backprop using the informations we get from the current minibatch
    return backward(model, np.array(xs), np.array(h1s), np.array(h2s), np.array(errs))
```

Define one gradient ascent step

We now perform a single gradient ascent step.

Get the gradients and perform the following updates for N training examples:

$$\mathbf{W}^{(1)} = \mathbf{W}^{(1)} + \epsilon \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{W}^{(1)}},$$

$$\mathbf{b}^{(1)} = \mathbf{b}^{(1)} + \epsilon \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{b}^{(1)}},$$

$$\mathbf{W}^{(2)} = \mathbf{W}^{(2)} + \epsilon \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{W}^{(2)}},$$

$$\mathbf{b}^{(2)} = \mathbf{b}^{(2)} + \epsilon \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{b}^{(2)}},$$

$$\mathbf{W}^{(3)} = \mathbf{W}^{(3)} + \epsilon \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{W}^{(3)}},$$

and

$$\mathbf{b}^{(3)} = \mathbf{b}^{(3)} + \epsilon \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L(i)}{\partial \mathbf{b}^{(3)}},$$

where $\epsilon = 10^{-1}$ in our example.

```
def gradient_step(model, X_train, y_train, learning_rate=1e-1):
    grad = get_gradient(model, X_train, y_train)
    model = model.copy()

    # Update every parameters in our networks (W1 and W2) using their gradients
    for layer in grad:
        # Learning rate: 1e-1
        model[layer] += learning_rate * grad[layer]

    return model
```

Repeat gradient ascent a few more times...

```
def gradient_ascent(model, X_train, y_train, no_iter=10, learning_rate=1e-1):
    for iter in range(no_iter):
        print('Iteration {}'.format(iter))

        model = gradient_step(model, X_train, y_train, learning_rate)

    return model
```

Train the model

We now train the model.

```
no_iter = 50

# Reset model
model = init_weights()

# Train the model
model = gradient_ascent(model, X_train, y_train, no_iter=no_iter)
```

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Iteration 17
Iteration 18
Iteration 19
Iteration 20
Iteration 21
Iteration 22
Iteration 23
Iteration 24
Iteration 25
Iteration 26
Iteration 27
Iteration 28
Iteration 29
Iteration 30
Iteration 31
Iteration 32
Iteration 33
Iteration 34
Iteration 35
Iteration 36
Iteration 37
Iteration 38
Iteration 39
Iteration 40
Iteration 41
Iteration 42
Iteration 43
Iteration 44
Iteration 45
Iteration 46
Iteration 47
Iteration 48
Iteration 49
```

Let's evaluate the model on test data

Note that the output has two neurons for the two classes. Our prediction will chose the class corresponding to the largest output neuron value.

```
y_pred = np.zeros_like(y_test)

accuracy = 0

for i, x in enumerate(X_test):
    # Predict the distribution of label
    _, _, prob = forward(x, model)
    # Get label by picking the most probable one
    y = np.argmax(prob)
    y_pred[i] = y

# Accuracy of predictions with the true labels and take the percentage
# Because our dataset is balanced, measuring just the accuracy is OK
accuracy = (y_pred == y_test).sum() / y_test.size
print('Test accuracy after {} gradient steps: {}'.format(no_iter,accuracy))
```

```
Test accuracy after 50 gradient steps: 0.9393333333333334
```

Let's visually evaluate the model on the test data

```
pylab.scatter(X_test[:,0], X_test[:,1], c=y_pred)
pylab.show()
```