CS 422/622 Project 3

Due 11/12 11:59pm

Logistics: You must implement everything stated in this project description that is marked with an **implement** tag. Whenever you see the **write-up** tag, that is something that must be addressed in the README for the project. **You may only use numpy, matplotlib and sklearn.**

Deliverables: Each student should submit a single ZIP file, containing your project code (∗.py files) and your writeup (PDF). Your zip file should be named lastname1_firstname_project3.zip. For example, a zip file for Sara Smith would look like smith_sara_project3.zip. Your code should run without errors on the ECC linux machines. If your code does not run for a particular problem, you will lose 50% on that problem. You should submit only one py file, named accordingly.

1 Neural Network (100 points)

File name: **neural_network.py**

Implement: You will implement three functions listed here and detailed below.

```
def calculate_loss(model, X, y):
def predict(model, x):
def build_model(X, y, nn_hdim, num_passes=20000, print_loss=False):
```

You will create a 2-layer neural network. Your network will take 2D data as input, that is x_1 and x_2. The output of the network will be 2D as well, that is $\hat{y}$ is a 2D vector. If x is the 2-dimensional input to our network then we calculate our prediction $\hat{y}$ (also two-dimensional) as follows:
$a = xW_1 + b_1$
$h = tanh(a)$
$z = hW_2 + b_2$
$\hat{y} = softmax(z)$

Just think about a and z as helper variables. W_1, b_1, W_2, b_2 are parameters of our network, which we need to learn from our training data. Looking at the matrix multiplications above we can figure out the dimensionality of these matrices. If we use 500 nodes for our hidden layer then $W_1 \in \mathbb{R}^{2\times 500}$, $b_1 \in \mathbb{R}^{500}$, $W_2 \in \mathbb{R}^{500\times 2}$, $b_2 \in \mathbb{R}^2$.

The softmax function is as follows:

$$\hat{y}_j = \frac{e^{z_j}}{\Sigma_{k=1}^{K} e^{z_k}}$$

Learning the parameters for our network means finding parameters (W_1, b_1, W_2, b_2) that minimize the error on our training data. A common choice with the softmax output is the categorical cross-entropy loss (also known as negative log likelihood). If we have N training examples and C classes then the loss for our prediction $\hat{y}$ with respect to the true labels y is given by:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n\in N} \sum_{i\in C} y_{n,i} \log \hat{y}_{n,i}$$

The formula looks complicated, but all it really does is sum over our training examples and add to the loss if we predicted the incorrect class. We can use gradient descent to find the minimum.

As an input, gradient descent needs the gradients (vector of derivatives) of the loss function with respect to our parameters: $\frac{\partial L}{\partial W_1}$, $\frac{\partial L}{\partial b_1}$, $\frac{\partial L}{\partial W_2}$, $\frac{\partial L}{\partial b_2}$. To calculate these gradients we use the backpropagation algorithm, which is a way to efficiently calculate the gradients starting from the output.

Applying the backpropagation formula we find the following (trust me on this):

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$$
$$\frac{\partial L}{\partial a} = (1 - \tanh^2 a) \circ \frac{\partial L}{\partial \hat{y}} W_2^T$$
$$\frac{\partial L}{\partial W_2} = h^T \frac{\partial L}{\partial \hat{y}}$$
$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}}$$
$$\frac{\partial L}{\partial W_1} = x^T \frac{\partial L}{\partial a}$$
$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial a}$$

First you will implement the loss function we defined above. We use this to evaluate how well our model is doing.

```
# Helper function to evaluate the total loss on the dataset
# model is the current version of the model {'W1':W1,'b1':b1,'W2':W2,'b2':b2'} It's a dictionary.
# X is all the training data
# y is the training labels
def calculate_loss(model, X, y):
```

You will also implement a helper function to calculate the output of the network. It does forward propagation as defined above and returns the class with the highest probability.

```
# Helper function to predict an output (0 or 1)
# model is the current version of the model {'W1':W1,'b1':b1,'W2':W2,'b2':b2'} It's a dictionary.
# x is one sample (without the label)
def predict(model, x):
```

Finally, here comes the function to train our Neural Network. It implements batch gradient descent using the backpropagation derivates we found above.

```
# This function learns parameters for the neural network and returns the model.
# - X is the training data
# - y is the training labels
# - nn_hdim: Number of nodes in the hidden layer
# - num_passes: Number of passes through the training data for gradient descent
# - print_loss: If True, print the loss every 1000 iterations
def build_model(X, y, nn_hdim, num_passes=20000, print_loss=False):
```

You will use the following code to display your decision boundary:

```
def plot_decision_boundary(pred_func, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
```

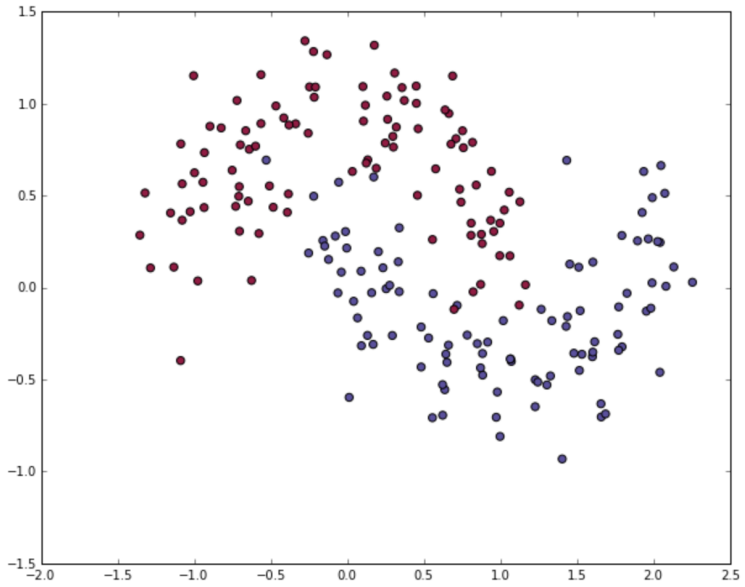

Figure 1: Dataset

```
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral)
```

This is the dataset you will be using to test your network. It is shown in figure 1.

```
# Generate a dataset and plot it
import numpy as np
from sklearn.datasets import make_moons
np.random.seed(0)
X, y = make_moons(200, noise=0.20)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)
```

Sample outputs are shown in figure 2. You will generate these outputs with the following code:

```
plt.figure(figsize=(16, 32))
hidden_layer_dimensions = [1, 2, 3, 4]
for i, nn_hdim in enumerate(hidden_layer_dimensions):
    plt.subplot(5, 2, i+1)
    plt.title('HiddenLayerSize%d' % nn_hdim)
    model = build_model(X, y, nn_hdim)
    plot_decision_boundary(lambda x: predict(model, x), X, y)
plt.show()
```

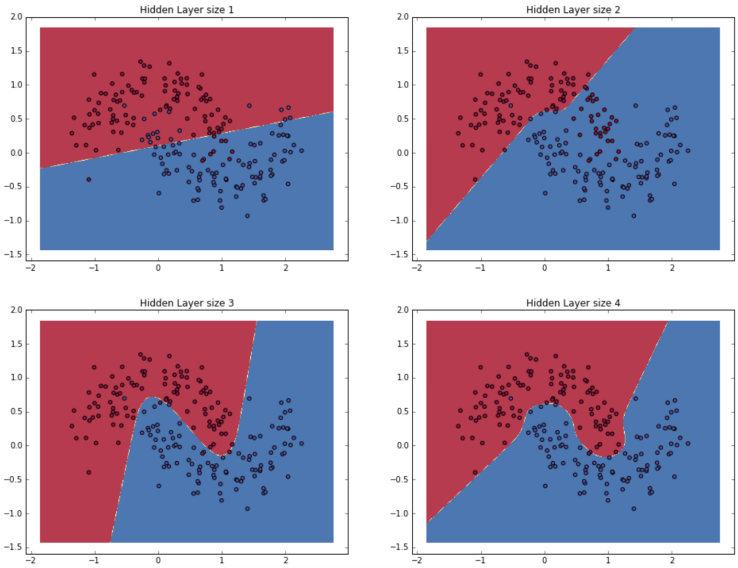

Figure 2: Example Outputs