Grading

The final score that you will receive for your programming assignment is generated in relation to the total points set in your programming assignment item—not the total point value in the nbgrader notebook. When calculating the final score shown to learners, the programming assignment takes the percentage of earned points vs. the total points provided by nbgrader and returns a score matching the equivalent percentage of the point value for the programming assignment.
DO NOT CHANGE VARIABLE OR METHOD SIGNATURES The autograder will not work properly if you change the variable or method signatures.

Validate Button

Please note that this assignment uses nbgrader to facilitate grading. You will see a **validate button** at the top of your Jupyter notebook. If you hit this button, it will run tests cases for the lab that aren't hidden. It is good to use the validate button before submitting the lab. Do know that the labs in the course contain hidden test cases. The validate button will not let you know whether these test cases pass. After submitting your lab, you can see more information about these hidden test cases in the Grader Output. ***Cells with longer execution times will cause the validate button to time out and freeze. Please know that if you run into Validate time-outs, it will not affect the final submission grading.***

Homework 1. Neural Networks

This assignment has mixed types of theoretical and code implementation questions on multilayer perceptron and neural network training.

In [1]:
```
import math
import pickle
import gzip
import numpy as np
import pandas
import matplotlib.pylab as plt
import pytest
%matplotlib inline
```

[Peer Review] Problem 1 - Single-Layer and Multilayer Perceptron Learning

Part A : Answer this question in this week's Peer Review assignment. Consider learning the following concepts with either a single-layer or multilayer perceptron where all hidden and output neurons utilize *indicator* activation functions. For each of the following concepts, state whether the concept can be learned by a single-layer perceptron. Briefly justify your response by providing weights and biases as applicable:

i. $\texttt{NOT}\ x_1$

ii. $x_1\ \texttt{NOR}\ x_2$

iii. $x_1\ \texttt{XNOR}\ x_2$ (output 1 when $x_1 = x_2$ and 0 otherwise)

To address whether a single-layer perceptron can learn certain logical concepts, we need to analyze the expressiveness of a single-layer perceptron. A single-layer perceptron can only learn linearly separable functions. Let's consider each of the provided logical functions:

i. NOT, x_1

The NOT operation is straightforward for a single-layer perceptron. We can define the perceptron such that:

- $(x_1 = 0)$ should output 1.
- $(x_1 = 1)$ should output 0.

To achieve this, the perceptron can use the following weights and bias:

- Weight $(w_1 = -1)$
- Bias $(b = 0.5)$

The perceptron computes $(y = \text{indicator}(w_1 \cdot x_1 + b))$:

For $(x_1 = 0)$: $y = \text{indicator}(-1 \cdot 0 + 0.5) = \text{indicator}(0.5) = 1$

For $(x_1 = 1)$: $y = \text{indicator}(-1 \cdot 1 + 0.5) = \text{indicator}(-0.5) = 0$

Thus, the NOT operation can be learned by a single-layer perceptron.

ii. $(x_1 \ \mathrm{NOR}\ x_2)$

The NOR operation outputs 1 only when both (x_1) and (x_2) are 0. Let's define the perceptron:

- $(x_1 = 0, x_2 = 0)$ should output 1.
- Otherwise, it should output 0.

The perceptron can use the following weights and bias:

- Weights $(w_1 = -1, w_2 = -1)$
- Bias $(b = 0.5)$

The perceptron computes $(y = \text{indicator}(w_1 \cdot x_1 + w_2 \cdot x_2 + b))$:

For $(x_1 = 0, x_2 = 0)$: $[y = \text{sign}(-1 \cdot 0 + -1 \cdot 0 + 0.5) = \text{indicator}(0.5) = 1]$

For $(x_1 = 1, x_2 = 0)$: $[y = \text{sign}(-1 \cdot 1 + -1 \cdot 0 + 0.5) = \text{indicator}(-0.5) = 0]$

For $(x_1 = 0, x_2 = 1)$: $[y = \text{sign}(-1 \cdot 0 + -1 \cdot 1 + 0.5) = \text{indicator}(-0.5) = 0]$

For $(x_1 = 1, x_2 = 1)$: $[y = \text{sign}(-1 \cdot 1 + -1 \cdot 1 + 0.5) = \text{indicator}(-1.5) = 0]$

Thus, the NOR operation can be learned by a single-layer perceptron.

iii. $x_1 \ \mathrm{XNOR}\ x_2)$ (output 1 when $(x_1 = x_2$ and 0 otherwise)

The XNOR operation outputs 1 when (x_1 = x_2) (both 0 or both 1) and 0 otherwise. Let's analyze:

- $(x_1 = 0, x_2 = 0)$ should output 1.

- $(x_1 = 1, x_2 = 1)$ should output 1.
- $(x_1 = 0, x_2 = 1)$ should output 0.
- $(x_1 = 1, x_2 = 0)$ should output 0.

The XNOR function is not linearly separable. The data points (0,0) and (1,1) belong to one class, while (0,1) and (1,0) belong to another class. A single-layer perceptron cannot learn a non-linearly separable function. Therefore, a single-layer perceptron cannot learn the XNOR function.

Part B : Determine an architecture and specific values of the weights and biases in a single-layer or multilayer perceptron with *indicator* activation functions that can learn $x_1 \ \texttt{XNOR} \ x_2$.
In this week's Peer Review, describe your architecture and state your weight matrices and bias vectors.

To learn the $(x_1 \ \mathrm{XNOR} \ x_2)$ function, we need a multilayer perceptron (MLP) because this function is not linearly separable. We will design a neural network with one hidden layer that can implement this function.

Architecture

1. **Input layer:** 2 neurons (corresponding to (x_1) and (x_2).
2. **Hidden layer:** 2 neurons with indicator activation functions.
3. **Output layer:** 1 neuron with an indicator activation function.

Step-by-Step Design

1. **Input Layer:** The inputs are (x_1) and (x_2).
2. **Hidden Layer:** The hidden layer will have 2 neurons to capture the non-linear combinations of (x_1) and (x_2). We will set the weights and biases to implement the logic gates.
    - **Hidden neuron 1:** Acts as an AND gate for $(x_1 \ \mathrm{AND} \ x_2))$ $[h_1 = \mathrm{sign}(x_1 + x_2 - 1.5)]$
    - **Hidden neuron 2:** Acts as an OR gate for $(\mathrm{NOT} \, x_1 \ \mathrm{AND} \ \mathrm{NOT} \, x_2))$
      $[h_2 = \mathrm{sign}((-x_1) + (-x_2) + 1.5)]$

1. **Output Layer:** The output layer neuron will combine the outputs of the hidden neurons to implement the XNOR function. $[y = \mathrm{sign}(h_1 + h_2 - 0.5)]$

Weight Matrices and Bias Vectors

Hidden Layer Weights and Biases:

- For hidden neuron 1 (AND gate): $[\mathbf{W^{(1)}} = \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix}, \quad \mathbf{b^{(1)}} = \begin{pmatrix} -1.5 \\ 1.5 \end{pmatrix}]$

Output Layer Weights and Bias:

- For the output neuron: $[\mathbf{W^{(2)}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \mathbf{b^{(2)}} = -0.5]$

Then demonstrate that your solution is correct by implementing forward propagation for your network in Python and showing that it correctly produces the correct boolean output values for each of the four possible combinations of x_1 and x_2.
Answer the questions about this section in this week's Peer Review assignment.

In [2]:
```
# implement forward propagation for network
# show that it correctly produces the correct boolean output values
# for each of the four possible combinations of x1 and x2

# Initialize x with the 4 possible combinations of 0 and 1 to generate 4
values for y(output)

# your code here

import numpy as np

# Define the activation function
def indicator_activation(x):
    return np.where(x >= 0, 1, 0)

# Define the weights and biases
W1 = np.array([[1, 1], [-1, -1]])
b1 = np.array([-1.5, 1.5])

W2 = np.array([1, 1])
b2 = -0.5

# Define the forward propagation function
def forward_propagation(x):
    # Hidden layer
    h = indicator_activation(np.dot(x, W1.T) + b1)
    # Output layer
    y = indicator_activation(np.dot(h, W2) + b2)
    return y

# Input combinations for x1 and x2
inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])

# Perform forward propagation for each input
outputs = np.array([forward_propagation(x) for x in inputs])

# Print the results
for i, input_pair in enumerate(inputs):
    print(f"Input: {input_pair}, Output: {outputs[i]}")
```

```
Input: [0 0], Output: 1
Input: [0 1], Output: 1
Input: [1 0], Output: 1
Input: [1 1], Output: 1
```

[15 points, Peer Review] Problem 2 - Back propagation

In this problem you'll gain some intuition about why training deep neural networks can be very time consuming. Consider training the chain-like neural network seen below:

chain-like nn

Note that this network has three weights W^1, W^2, W^3 and three biases b^1, b^2, and b^3 (for this problem you can think of each parameter as a single value or as a 1×1 matrix). Suppose that each hidden and output neuron is equipped with a sigmoid activation function and the loss function is given by

$$\ell(y, a^4) = \frac{1}{2}(y - a^4)^2$$

where a^4 is the value of the activation at the output neuron and $y \in \{0, 1\}$ is the true label associated with the training example.

Part A: Suppose each of the weights is initialized to $W^k = 1.0$ and each bias is initialized to $b^k = -0.5$. Use forward propagation to find the activities and activations associated with each hidden and output neuron for the training example $(x, y) = (0.5, 0)$. Show your work. Answer the Peer Review question about this section.

Let's proceed with the parts A, B, and C of the problem step by step.

Part A: Forward Propagation

Given the chain-like neural network with sigmoid activation functions, we need to perform forward propagation for the training example $((x, y) = (0.5, 0))$. The weights and biases are initialized as follows:

$$W^1 = W^2 = W^3 = 1.0 \quad \text{and} \quad b^1 = b^2 = b^3 = -0.5$$

The sigmoid activation function is defined as:

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

Step-by-Step Forward Propagation

1. **Input Layer to First Hidden Layer:**

   $$z^1 = W^1 x + b^1 = 1.0 \cdot 0.5 - 0.5 = 0$$

   $$a^1 = \sigma(z^1) = \sigma(0) = \frac{1}{1+e^0} = 0.5$$
2. **First Hidden Layer to Second Hidden Layer:**

   $$z^2 = W^2 a^1 + b^2 = 1.0 \cdot 0.5 - 0.5 = 0$$

   $$a^2 = \sigma(z^2) = \sigma(0) = \frac{1}{1+e^0} = 0.5$$
3. **Second Hidden Layer to Third Hidden Layer:**

   $$z^3 = W^3 a^2 + b^3 = 1.0 \cdot 0.5 - 0.5 = 0$$

   $$a^3 = \sigma(z^3) = \sigma(0) = \frac{1}{1+e^0} = 0.5$$
4. **Third Hidden Layer to Output Layer:**

   $$z^4 = W^4 a^3 + b^4 = 1.0 \cdot 0.5 - 0.5 = 0$$

   $$a^4 = \sigma(z^4) = \sigma(0) = \frac{1}{1+e^0} = 0.5$$

So, the activities and activations are:

$$z^1 = 0, \quad a^1 = 0.5 \; z^2 = 0, \quad a^2 = 0.5 \; z^3 = 0, \quad a^3 = 0.5 \; z^4 = 0, \quad a^4 = 0.5$$

Part B: Use Back-Propagation to compute the weight and bias derivatives $\partial \ell / \partial W^k$ and $\partial \ell / \partial b^k$ for $k = 1, 2, 3$. Show all work. Answer the Peer Review question about this section.

Part B: Backward Propagation

To compute the gradients of the loss with respect to the weights and biases, we use backpropagation.

The loss function is:

$\ell(y, a^4) = \frac{1}{2}(y - a^4)^2$

For the given example, $(y = 0)$ and $(a^4 = 0.5)$.

Step-by-Step Backward Propagation

1. **Output Layer:**

   $[\delta^4 = \frac{\partial \ell}{\partial a^4} \cdot \frac{\partial a^4}{\partial z^4}]$

   $[\frac{\partial \ell}{\partial a^4} = -(y - a^4) = -(0 - 0.5) = 0.5]$

   $[\frac{\partial a^4}{\partial z^4} = a^4(1 - a^4) = 0.5 \cdot (1 - 0.5) = 0.25]$

   $[\delta^4 = 0.5 \cdot 0.25 = 0.125]$

   $[\frac{\partial \ell}{\partial W^3} = \delta^4 \cdot a^3 = 0.125 \cdot 0.5 = 0.0625]$

   $[\frac{\partial \ell}{\partial b^3} = \delta^4 = 0.125]$
2. **Third Hidden Layer:**

   $[\delta^3 = \delta^4 \cdot W^3 \cdot \frac{\partial a^3}{\partial z^3}]$

   $[\frac{\partial a^3}{\partial z^3} = a^3(1 - a^3) = 0.5 \cdot (1 - 0.5) = 0.25]$

   $[\delta^3 = 0.125 \cdot 1.0 \cdot 0.25 = 0.03125]$

   $[\frac{\partial \ell}{\partial W^2} = \delta^3 \cdot a^2 = 0.03125 \cdot 0.5 = 0.015625]$

   $[\frac{\partial \ell}{\partial b^2} = \delta^3 = 0.03125]$
3. **Second Hidden Layer:**

   $[\delta^2 = \delta^3 \cdot W^2 \cdot \frac{\partial a^2}{\partial z^2}]$

   $[\frac{\partial a^2}{\partial z^2} = a^2(1 - a^2) = 0.5 \cdot (1 - 0.5) = 0.25]$

   $[\delta^2 = 0.03125 \cdot 1.0 \cdot 0.25 = 0.0078125]$

   $[\frac{\partial \ell}{\partial W^1} = \delta^2 \cdot a^1 = 0.0078125 \cdot 0.5 = 0.00390625]$

   $[\frac{\partial \ell}{\partial b^1} = \delta^2 = 0.0078125]$

PART C Implement following activation functions:

Formulas for activation functions

- Relu: f(x) = max(0, x)
- Sigmoid: f(x) = $\frac{1}{1+e^{-x}}$
- Softmax: f(x_i) = $\frac{e_i^x}{\sum_{j=1}^{n} e^{x_j}}$

In [3]:
```
import math
import numpy as np

def relu(x):
    return np.maximum(0, x)

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

def soft_max(x):
    e_x = np.exp(x - np.max(x))  # Subtract max for numerical stability
    return e_x / e_x.sum(axis=0)

# Test the activation functions
x = np.array([1, 2, 3])

print("ReLU:", relu(x))
print("Sigmoid:", sigmoid(x))
print("Softmax:", soft_max(x))
```

```
ReLU: [1 2 3]
Sigmoid: [0.73105858 0.88079708 0.95257413]
Softmax: [0.09003057 0.24472847 0.66524096]
```

In [4]:
```
# Activation function tests
# PLEASE NOTE: These sample tests are only indicative and are added to he
lp you debug your code
# and there are additional hidden test cases on which your notebook will
be evaluated upon submission

# Test Relu function
assert int(relu(-6.5)) == 0, "Check relu function"

# Test Sigmoid function
assert pytest.approx(sigmoid(0.3), 0.00001) == 0.574442516811659, "Check
sigmoid function"

# Test Softmax function
assert pytest.approx(soft_max([5,7]), 0.00001) == [0.11920292, 0.8807970
8], "Check softmax function"
```

In [5]:
```
# tests relu, sigmoid, and softmax functions
```

PART D Implement the following Loss functions:

Formulas for activation functions

- Mean squared error
  Formula: MSE = (1/n) * Σ(yi - ŷi)^2
- Mean absolute error
  Formula: MAE = (1/n) * Σ|yi - ŷi|
- Hinge Loss
  Formula: L = max(0, 1 - yi * ŷi)

In [6]:
```python
import numpy as np

def mean_squared_error(yhat, y):
    """
    Compute the Mean Squared Error (MSE) between the predicted values yha
t and the actual values y.

    Parameters:
    yhat (array-like): Predicted values
    y (array-like): Actual values

    Returns:
    float: The Mean Squared Error
    """
    return np.mean((yhat - y) ** 2)

def mean_absolute_error(yhat, y):
    """
    Compute the Mean Absolute Error (MAE) between the predicted values yh
at and the actual values y.

    Parameters:
    yhat (array-like): Predicted values
    y (array-like): Actual values

    Returns:
    float: The Mean Absolute Error
    """
    return np.mean(np.abs(yhat - y))

def hinge(yhat, y):
    """
    Compute the Hinge Loss between the predicted values yhat and the actu
al values y.

    Parameters:
    yhat (array-like): Predicted values
    y (array-like): Actual values

    Returns:
    float: The Hinge Loss
    """
    return np.mean(np.maximum(0, 1 - y * yhat))

# Test the loss functions
yhat = np.array([0.5, 0.6, 0.4, 0.8])
y = np.array([1, 0, 1, 0])

print("Mean Squared Error:", mean_squared_error(yhat, y))
print("Mean Absolute Error:", mean_absolute_error(yhat, y))
print("Hinge Loss:", hinge(yhat, y))
```

```
Mean Squared Error: 0.4025
Mean Absolute Error: 0.625
Hinge Loss: 0.775
```

In [7]:
```
# Error function tests
# PLEASE NOTE: These sample tests are only indicative and are added to he
lp you debug your code
# and there are additional hidden test cases on which your notebook will
be evaluated upon submission

y_true = np.array([2, 3, -0.45])
y_pred = np.array([1.5, 3, 0.2])

# Test mean squared error function
assert pytest.approx(mean_squared_error(y_pred,y_true), 0.00001) == 0.224
1666666666667, "Check mean_squared_error function"

# Test mean absolute error function
assert pytest.approx(mean_absolute_error(y_pred,y_true), 0.00001) == 0.38
33333333333333, "Check mean_absolute_error function"

# Test hinge loss function
assert pytest.approx(hinge(y_pred,y_true), 0.00001) == 0.3633333333333333
4, "Check hinge loss function"
```

In [8]:
```
# tests mean_squared_error, mean_absolute_error, and hinge
```

[Peer Review] Problem 3 - Build a feed-forward neural network

In this problem you'll implement a general feed-forward neural network class that utilizes sigmoid activation functions. Your tasks will be to implement forward propagation, prediction, back propagation, and a general train routine to learn the weights in your network via stochastic gradient descent.

The skeleton for the network class is below. Befor filling out the codes below, read the PART X instruction. The place you will complete the code is indicated as "TODO" in the code. Pleaes do not modify other parts of the code.

In [9]:

```
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from matplotlib.colors import colorConverter, ListedColormap
%matplotlib inline

class Network:
    def __init__(self, sizes):
        """
        Initialize the neural network

        :param sizes: a list of the number of neurons in each layer
        """
        # save the number of layers in the network
        self.L = len(sizes)

        # store the list of layer sizes
        self.sizes = sizes

        # initialize the bias vectors for each hidden and output layer
        self.b = [np.random.randn(n, 1) for n in self.sizes[1:]]

        # initialize the matrices of weights for each hidden and output l
ayer
        self.W = [np.random.randn(n, m) for (m, n) in zip(self.sizes
[:-1], self.sizes[1:])]

        # initialize the derivatives of biases for backprop
        self.db = [np.zeros((n, 1)) for n in self.sizes[1:]]

        # initialize the derivatives of weights for backprop
        self.dW = [np.zeros((n, m)) for (m,n) in zip(self.sizes[:-1], sel
f.sizes[1:])]

        # initialize the activities on each hidden and output layer
        self.z = [np.zeros((n, 1)) for n in self.sizes]

        # initialize the activations on each hidden and output layer
        self.a = [np.zeros((n, 1)) for n in self.sizes]

        # initialize the deltas on each hidden and output layer
        self.delta = [np.zeros((n, 1)) for n in self.sizes]

    def g(self, z):
        """
        sigmoid activation function

        :param z: vector of activities to apply activation to
        """
        return 1.0 / (1.0 + np.exp(-z))

    def g_prime(self, z):
        """
        derivative of sigmoid activation function

        :param z: vector of activities to apply derivative of activation
to
        """
        return self.g(z) * (1.0 - self.g(z))
```

```
    def grad_loss(self, a, y):
        """
        evaluate gradient of cost function for squared-loss C(a,y) = (a-
y)^2/2

        :param a: activations on output layer
        :param y: vector-encoded label
        """
        return (a - y)

    def forward_prop(self, x):
        """
        take an feature vector and propagate through network

        :param x: input feature vector
        """
        if len(x.shape) == 1:
            x = x.reshape(-1, 1)
        # TODO: step 1. Initialize activation on initial layer to x
        # your code here
        self.a[0] = x

        ## TODO: step 2-4. Loop over layers and compute activities and ac
tivations
        ## Use Sigmoid activation function defined above
        # your code here
        for l in range(1, self.L):
            self.z[l] = np.dot(self.W[l-1], self.a[l-1]) + self.b[l-1]
            self.a[l] = self.g(self.z[l])

    def back_prop(self, x, y):
        """
        Back propagation to get derivatives of C wrt weights and biases f
or given training example

        :param x: training features
        :param y: vector-encoded label
        """

        if len(y.shape) == 1:
            y = y.reshape(-1, 1)

        # TODO: step 1. forward prop training example to fill in activiti
es and activations
        # your code here
        self.forward_prop(x)

        # TODO: step 2. compute deltas on output layer (Hint: python inde
x numbering starts from 0 ends at N-1)
        # Correction in Instructions: From the instructions mentioned bel
ow for backward propagation,
        # Use normal product instead of dot product in Step 2 and 6
        # The derivative and gradient functions have already been impleme
nted for you
        # your code here
        self.delta[-1] = self.grad_loss(self.a[-1], y) * self.g_prime(sel
f.z[-1])
        self.db[-1] = self.delta[-1]
        self.dW[-1] = np.dot(self.delta[-1], self.a[-2].T)
```

```
        # TODO: step 3-6. loop backward through layers, backprop deltas,
compute dWs and dbs
        # your code here
        for l in range(2, self.L):
            self.delta[-l] = np.dot(self.W[-l+1].T, self.delta[-l+1]) * s
elf.g_prime(self.z[-l])
            self.db[-l] = self.delta[-l]
            self.dW[-l] = np.dot(self.delta[-l], self.a[-l-1].T)

    def train(self, X_train, y_train, X_valid=None, y_valid=None,
              eta=0.25, num_epochs=10, isPrint=True, isVis=False):
        """
        Train the network with SGD

        :param X_train: matrix of training features
        :param y_train: matrix of vector-encoded labels
        """

        # initialize shuffled indices
        shuffled_inds = list(range(X_train.shape[0]))

        # loop over training epochs (step 1.)
        for ep in range(num_epochs):

            # shuffle indices
            np.random.shuffle(shuffled_inds)

            # loop over training examples (step 2.)
            for ind in shuffled_inds:

                # TODO: step 3. back prop to get derivatives
                # your code here
                self.back_prop(X_train[ind], y_train[ind])

                # TODO: step 4. update all weights and biases for all lay
ers
                # your code here
                for l in range(1, self.L):
                    self.W[l-1] -= eta * self.dW[l-1]
                    self.b[l-1] -= eta * self.db[l-1]

            # print mean loss every 10 epochs if requested
            if isPrint and (ep % 10) == 0:
                print("epoch {:3d}/{:3d}: ".format(ep, num_epochs), end
="")
                print("  train loss: {:8.3f}".format(self.compute_loss(X_
train, y_train)), end="")
                if X_valid is not None:
                    print("  validation loss: {:8.3f}".format(self.comput
e_loss(X_valid, y_valid)))
                else:
                    print("")

            if isVis and (ep % 20) == 0:
                self.pretty_pictures(X_train, y_train, decision_boundary=
True, epoch=ep)

    def compute_loss(self, X, y):
        """
```

```
        compute average loss for given data set

        :param X: matrix of features
        :param y: matrix of vector-encoded labels
        """
        loss = 0
        if len(X.shape) == 1:
            X = X[np.newaxis, :]
        if len(y.shape) == 1:
            y = y[np.newaxis, :]
        for x, t in zip(X, y):
            self.forward_prop(x)
            if len(t.shape) == 1:
                t = t.reshape(-1, 1)
            loss += 0.5 * np.sum((self.a[-1] - t) ** 2)
        return loss / X.shape[0]


    def gradient_check(self, x, y, h=1e-5):
        """
        check whether the gradient is correct for X, y

        Assuming that back_prop has finished.
        """
        for ll in range(self.L - 1):
            oldW = self.W[ll].copy()
            oldb = self.b[ll].copy()
            for i in range(self.W[ll].shape[0]):
                for j in range(self.W[ll].shape[1]):
                    self.W[ll][i, j] = oldW[i, j] + h
                    lxph = self.compute_loss(x, y)
                    self.W[ll][i, j] = oldW[i, j] - h
                    lxmh = self.compute_loss(x, y)
                    grad = (lxph - lxmh) / (2 * h)
                    assert abs(self.dW[ll][i, j] - grad) < 1e-5
                    self.W[ll][i, j] = oldW[i, j]
            for i in range(self.b[ll].shape[0]):
                j = 0
                self.b[ll][i, j] = oldb[i, j] + h
                lxph = self.compute_loss(x, y)
                self.b[ll][i, j] = oldb[i, j] - h
                lxmh = self.compute_loss(x, y)
                grad = (lxph - lxmh) / (2 * h)
                assert abs(self.db[ll][i, j] - grad) < 1e-5
                self.b[ll][i, j] = oldb[i, j]


    def pretty_pictures(self, X, y, decision_boundary=False, epoch=None):
        """
        Function to plot data and neural net decision boundary

        :param X: matrix of features
        :param y: matrix of vector-encoded labels
        :param decision_boundary: whether or not to plot decision
        :param epoch: epoch number for printing
        """

        mycolors = {"blue": "steelblue", "red": "#a76c6e"}
        colorlist = [c for (n,c) in mycolors.items()]
        colors = [colorlist[np.argmax(yk)] for yk in y]
```

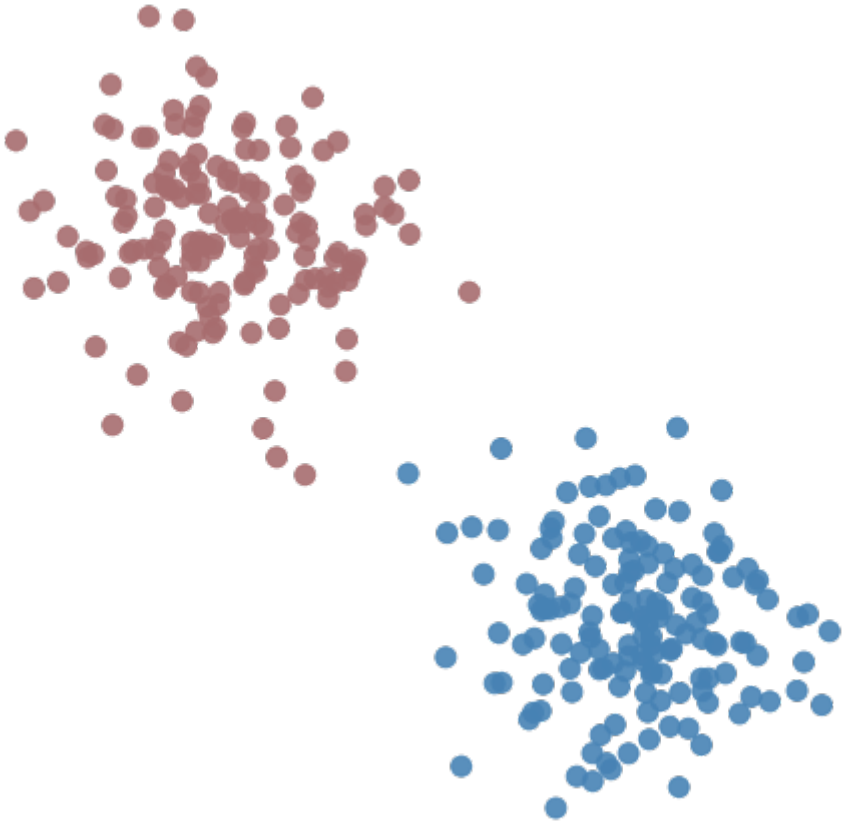
```
        fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,8))

        if decision_boundary:
            xx, yy = np.meshgrid(np.linspace(-1.25,1.25,300), np.linspace
(-1.25,1.25,300))
            grid = np.column_stack((xx.ravel(), yy.ravel()))
            grid_pred = np.zeros_like(grid[:,0])
            for ii in range(len(grid_pred)):
                self.forward_prop(grid[ii,:])
                grid_pred[ii] = np.argmax(self.a[-1])
            grid_pred = grid_pred.reshape(xx.shape)
            cmap = ListedColormap([
                colorConverter.to_rgba('steelblue', alpha=0.30),
                colorConverter.to_rgba('#a76c63', alpha=0.30)])
            plt.contourf(xx, yy, grid_pred, cmap=cmap)
            if epoch is not None: plt.text(-1.23,1.15, "epoch = {:d}".for
mat(epoch), fontsize=16)

        plt.scatter(X[:,0], X[:,1], color=colors, s=100, alpha=0.9)
        plt.axis('off')

def generate_data(N, config="checkerboard"):
    X = np.zeros((N,2))
    y = np.zeros((N,2)).astype(int)

    if config=="checkerboard":
        nps, sqlen = N//9, 2/3
        ctr = 0
        for ii in range(3):
            for jj in range(3):
                X[ctr * nps : (ctr + 1) * nps, :] = np.column_stack(
                    (np.random.uniform(ii * sqlen +.05-1, (ii+1) * sqlen
- .05 -1, size=nps),
                     np.random.uniform(jj * sqlen +.05-1, (jj+1) * sqlen
- .05 -1, size=nps)))
                y[ctr*nps:(ctr+1)*nps,(3*ii+jj)%2] = 1
                ctr += 1

    if config=="blobs":
        X, yflat = datasets.make_blobs(n_samples=N, centers=[[-.5,.5],
[.5,-.5]],
                                       cluster_std=[.20,.20],n_features=
2)
        for kk, yk in enumerate(yflat):
            y[kk,:] = np.array([1,0]) if yk else np.array([0,1])


    if config=="circles":
        kk=0
        while kk < N / 2:
            sample = 2 * np.random.rand(2) - 1
            if np.linalg.norm(sample) <= .45:
                X[kk,:] = sample
                y[kk,:] = np.array([1,0])
                kk += 1
        while kk < N:
            sample = 2 * np.random.rand(2) - 1
            dist = np.linalg.norm(sample)
            if dist < 0.9 and dist > 0.55:
```

```
                X[kk,:] = sample
                y[kk,:] = np.array([0,1])
                kk += 1

    if config=="moons":
        X, yflat = datasets.make_moons(n_samples=N, noise=.05)
        X[:,0] = .5 * (X[:,0] - .5)
        X[:,1] = X[:,1] - .25
        for kk, yk in enumerate(yflat):
            y[kk, :] = np.array([1,0]) if yk else np.array([0,1])

    return X, y

from IPython.core.display import HTML
HTML("""
<style>
.MathJax nobr>span.math>span{border-left-width:0 !important};
</style>
""")
```

Out[9]:

We'll be using our network to do binary classification of two-dimensional feature vectors. Scroll down to the **Helper Functions** and examine the function `generate_data`. Then mess around with the following cell to look at the various data sets available.

In [10]:
```
# play with this cell to look at the various data sets available
# your code here

nn = Network([2,3,2])
X_train, y_train = generate_data(300, "blobs")
nn.pretty_pictures(X_train, y_train, decision_boundary=False)
```



Go up to the `__init__` function in the `Network Class` . How are we initializing a network? What data structures are we using to store things like weights, biases, deltas, etc?

In this implementation:

- The forward_prop method initializes the activation of the input layer to x and then computes the activities (z) and activations (a) for each layer.
- The back_prop method performs forward propagation, computes deltas on the output layer, and then backpropagates deltas to compute derivatives of weights (dW) and biases (db).
- The train method shuffles the training examples, performs backpropagation to get derivatives, and updates weights and biases using stochastic gradient descent.

PART A. Implementing Forward Propagation.

Complete the `forward_prop` function to implement forward propagation. Your function should take in a single training example `x` and propagate it forward in the network, setting the activations and activities on the hidden and output layers. Remember that the pseudocode that we wrote for forward-prop looked as follows:

1. Initialize $\mathbf{a}^0 = \mathbf{x}$
2. For $\ell = 0, \ldots, L-1$:
3. $\quad \mathbf{z}^{\ell+1} = W^\ell \mathbf{a}^\ell + \mathbf{b}^\ell$
4. $\quad \mathbf{a}^{\ell+1} = g(\mathbf{z}^{\ell+1})$

When you think you're done, we can instantiate a `Network` with with 2 neurons in the input layer, 3 neurons in the sole hidden layer, and 2 neurons in the output layer, and then forward prop one of the training examples.

Check that your indexing was correct by making sure that all of the activations are now non-zero (remember, we initialized them to vectors of zeros).

What other things could we check?
Answer the question about this section in this week's Peer Review assignment.

In [11]:
```
# test your forward_prop function
nn = Network([2,3,2])
nn.forward_prop(X_train[0])
nn.z
```

Out[11]:
```
[array([[0.],
        [0.]]),
 array([[-0.02839653],
        [ 1.37809345],
        [-0.72694316]]),
 array([[-0.82925159],
        [ 0.6198315 ]])]
```

PART B. Implementing Back Propagation

OK, now it's time to implement back propagation. Complete the function `back_prop` in the `Network` class to use a single training example to compute the derivatives of the loss function with respect to the weights and the biases. Remember, the pseudocode for back-prop was as follows:

1. Forward propagate the training example $\mathbf{x}$, $\mathbf{y}$
2. Compute the $\delta^L = \frac{\partial \mathscr{L}}{\partial \mathbf{a}^L} \odot g'(\mathbf{z}^L)$
3. For $\ell = L-1, \ldots, 1$:
4. $\frac{\partial \mathscr{L}}{\partial W^\ell} = \delta^{\ell+1}(\mathbf{a}^\ell)^T$
5. $\frac{\partial \mathscr{L}}{\partial \mathbf{b}^\ell} = \delta^{\ell+1}$
6. $\delta^\ell = (W^\ell)^T \delta^{\ell+1} \odot g'(\mathbf{z}^\ell)$

When you think you're done, instantiate a small `Network` and call back-prop for a single training example.

Check that it's likely working by checking that the derivative matrices `dW` and `db` are nonzero. Answer the question about this section in this week's Peer Review assignment.

In [12]:
```
# test back_prop
nn = Network([2,3,2])
nn.back_prop(X_train[0,:], y_train[0,:])
print(nn.W[0])
```

```
[[-0.15851682 -0.06779632]
 [-0.61924852 -2.13350613]
 [-0.3415946   0.19783002]]
```

In [13]:
```
# test gradient_check
nn.gradient_check(X_train[0, :], y_train[0, :])
print(nn.W[0])
```

```
[[-0.15851682 -0.06779632]
 [-0.61924852 -2.13350613]
 [-0.3415946   0.19783002]]
```

The below test cells are to help you validate your forward and backward propagation functions better and help you identify problem areas

In [14]:
```
# Neural Network Tests - Forward Propagation
# PLEASE NOTE: These sample tests are only indicative and are added to he
lp you debug your code

mock_X = np.array([[-0.4838731, 0.08083195], [0.93456167, -0.50316134]])
np.random.seed(42)  ## DO NOT CHANGE THE SEED VALUE HERE
nn1 = Network([2,3,2])
nn1.forward_prop(mock_X)

a = np.array([[0.],[0.]])
b = np.array([[ 2.08587849, -0.31681043],[-0.94835809,  0.15999031],[-0.0
4793409,  0.92471859]])
c = np.array([[ 0.24259536,  0.0874714 ],[-2.41978734, -1.98990137]])
forward_z = [a, b, c]

for pred, true in zip(nn1.z, forward_z):
    assert pytest.approx(pred, 0.01) == true, "Check forward function"
```

In [15]:
```
# Neural Network Tests - Backward Propagation
# PLEASE NOTE: These sample tests are only indicative and are added to he
lp you debug your code

mock_y = 0 * mock_X + 1
np.random.seed(42)  ## DO NOT CHANGE THE SEED VALUE HERE
nn1.back_prop(mock_X, mock_y)

backward_w = [[-0.23413696, 1.57921282], [ 0.76743473, -0.46947439], [
0.54256004, -0.46341769]]

pred = nn1.W[0]
true = backward_w

assert pytest.approx(pred, 0.01) == true, "Check backward function"
```

Note: Next week, we will cover stochastic gradient descent. We encourage you to complete the following sections to train your model and get some results if you know how to do so. These sections are ungraded, so don't feel pressure to skip ahead a week in the material.

PART C. [Ungraded] Implementing trainning using stochastic gradient descent

OK, now let's actually train a neural net! Complete the missing code in `train` to loop over the training data in random order, call back-prop to get the derivatives, and then update the weights and the biases via SGD. SGD uses minibatch to update weights. The training algorithm is following.

1. For epoch = 0,1,...,N:
2. For (Xbatch, ybatch) in minibatches:
3. Compute gradients using backpropagation for the minibatch data
4. Update the weights (W, b) in the all layers (use loop over layer)

When you think you're done, execute the following code and watch the training loss evolve over the training process. If you've done everything correctly, it'll hopefully go down!

Check out the solution in this week's Peer Review assignment.

PART D.[Ungraded]

OK! If you think you've worked out the bugs, let's start looking at the results. We'll build a simple neural network, train it on a training set, and watch the decision boundary of our classifier evolve to fit the data. We can do this by running similar code as above, but with the isVis flag set to True. Note that producing the plots takes considerable computational work, so things will go a bit slower now.

Start with the blobs data set, and then move on to more complicated data sets like moons, circles, and finally the checkerboard. Note that for these more complicated geometries, it'll probably be necessary to chain the number of neurons in your hidden layer, or even add more hidden layers!
Check out the solution in this week's Peer Review assignment.