Olimov Bekhzod (올리모브 벡조드)

Kyungpook National University

Task 0. Generate randomly 2-dimensional training/test data

In [1]:

```
# Importing necessary libraries
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
seed = 6
```

In [2]:

```
def generate_data(mean_1, cov_1, mean_2, cov_2, num_classes, num_samples):
    """
    Generates datasets

    Inputs:

    mean_1 -- a numpy array with the mean values of the first class
    cov_1 -- a numpy array containing the covariance matrix of the first class
    mean_2 -- a numpy array with the mean values of the second class
    cov_2 -- a numpy array containing the covariance matrix of the second class
    num_classes -- total number of classes in the dataset
    num_samples -- total number of samples in the given class

    Outputs:

    training_data -- a numpy array containing training data
    test_data -- a numpy array containing test data
    """

    ######################################## TRAINING DATA START
########################################
    np.random.seed(seed)
    # features and one-hot encoded targets for class 1
    class_1_features_train = np.random.multivariate_normal(mean=mean_1, cov=cov_1, size=num_samples
)
    class_1_targets_mask = np.zeros(num_samples).astype(int).reshape(-1)
    class_1_targets_train = np.eye(num_classes)[class_1_targets_mask]
    print("~~~~~~~~~~~~~~~~~~~~~~ TRAINING DATA STATS ~~~~~~~~~~~~~~~~~~~~~\n")
    print("Class 1 mean:\n {}".format(class_1_features_train.mean(axis=0)))
    print("Class 1 cov:\n {}".format(np.cov(class_1_features_train.T)))

    # features and one-hot encoded targets for class 2
    class_2_features_train = np.random.multivariate_normal(mean=mean_2, cov=cov_2, size=num_samples
)
    class_2_targets_mask = np.ones(num_samples).astype(int).reshape(-1)
    class_2_targets_train = np.eye(num_classes)[class_2_targets_mask]
    print("------------------------------------------------------------")
    print("Class 2 mean:\n {}".format(class_2_features_train.mean(axis=0)))
    print("Class 2 cov:\n {}".format(np.cov(class_2_features_train.T)))

    # combined features and targets
    X_train = np.concatenate([class_1_features_train, class_2_features_train], axis=0)
    y_train = np.concatenate([class_1_targets_train, class_2_targets_train], axis=0)


    # data for training
    training_data = np.concatenate([X_train, y_train], axis=1)
    np.random.shuffle(training_data)

    ######################################### TRAINING DATA END
#########################################
```

```
    ########################################## TEST DATA START
##########################################

    # features and one-hot encoded targets for class 1
    class_1_features_test = np.random.multivariate_normal(mean=class_1_mean, cov=class_1_cov, size=
num_samples)
    class_1_targets_mask = np.zeros(num_samples).astype(int).reshape(-1)
    class_1_targets_test = np.eye(num_classes)[class_1_targets_mask]
    print("\n~~~~~~~~~~~~~~~~~~~~~~~~ TEST DATA STATS ~~~~~~~~~~~~~~~~~~~~~~~~\n")
    print("Class 1 mean:\n {}".format(class_1_features_test.mean(axis=0)))
    print("Class 1 cov:\n {}".format(np.cov(class_1_features_test.T)))

    # features and one-hot encoded targets for class 2
    class_2_features_test = np.random.multivariate_normal(mean=class_2_mean, cov=class_2_cov, size=
num_samples)
    class_2_targets_mask = np.ones(num_samples).astype(int).reshape(-1)
    class_2_targets_test = np.eye(num_classes)[class_2_targets_mask]
    print("---------------------------------------------------------------")
    print("Class 2 mean:\n {}".format(class_2_features_test.mean(axis=0)))
    print("Class 2 cov:\n {}".format(np.cov(class_2_features_test.T)))

    # combined features and targets
    X_test = np.concatenate([class_1_features_test, class_2_features_test], axis=0)
    y_test = np.concatenate([class_1_targets_test, class_2_targets_test], axis=0)

    # data for testing
    test_data = np.concatenate([X_test, y_test], axis=1)
    np.random.shuffle(test_data)

    ########################################## TEST DATA END
##########################################

    return training_data, test_data
```

In [3]:

```
class_1_mean, class_1_cov = np.array([1., 1.]), np.array([[1, 0.5], [0.5, 1]])
class_2_mean, class_2_cov = np.array([-1., 1.]), np.array([[1, 0.5], [0.5, 1]])
num_classes, num_samples = 2, 50

training_data, test_data = generate_data(class_1_mean, class_1_cov, class_2_mean, class_2_cov, num_
classes, num_samples)
```

```
~~~~~~~~~~~~~~~~~~~~~~ TRAINING DATA STATS ~~~~~~~~~~~~~~~~~~~~~

Class 1 mean:
 [0.83778094 0.91243052]
Class 1 cov:
 [[0.9960769  0.58018966]
 [0.58018966 1.09758139]]
---------------------------------------------------------------
Class 2 mean:
 [-0.8642191   1.02057469]
Class 2 cov:
 [[0.89307551 0.50002721]
 [0.50002721 0.9169554 ]]

~~~~~~~~~~~~~~~~~~~~~~~~ TEST DATA STATS ~~~~~~~~~~~~~~~~~~~~~~~

Class 1 mean:
 [0.94585142 1.05146269]
Class 1 cov:
 [[0.89329926 0.56557865]
 [0.56557865 0.78663658]]
---------------------------------------------------------------
Class 2 mean:
 [-1.06402925  1.06205427]
Class 2 cov:
 [[0.92892377 0.40671998]
 [0.40671998 0.89173059]]
```

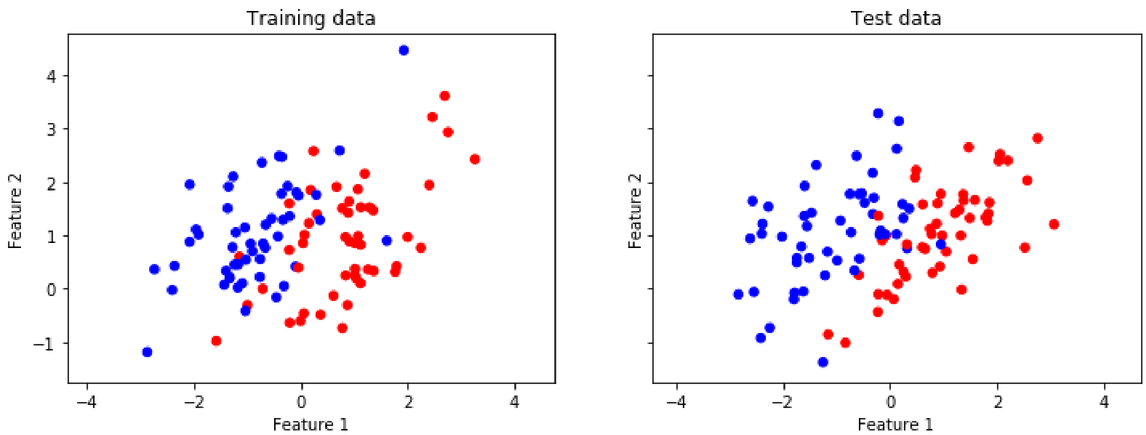
In [4]:

```
fix, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 4), sharey=True)
```

```
############################## TRAINING DATA ##############################

axes[0].scatter(training_data[:, 0], training_data[:, 1],
                c=training_data[:, 2], s=30, cmap=plt.cm.bwr)
axes[0].set_title('Training data')
axes[0].set_xlabel('Feature 1')
axes[0].set_ylabel('Feature 2')
axes[0].axis('equal')

############################## TEST DATA ##############################

axes[1].scatter(test_data[:, 0], test_data[:, 1],
                c=test_data[:, 2], s=30, cmap=plt.cm.bwr)
axes[1].set_title('Test data')
axes[1].set_xlabel('Feature 1')
axes[1].set_ylabel('Feature 2')
axes[1].axis('equal')
```

Out[4]:

```
(-3.1377960865227332,
 3.3813078245028647,
 -1.749048139117585,
 4.771753488790376)
```



In [5]:

```
# Extracting training data features and targets as well as test features and targets
X_train, y_train = training_data[:, :2], training_data[:, 2].reshape(-1, 1)
X_test, y_test = test_data[:, :2], test_data[:, 2].reshape(-1, 1)

print("~~~~~~~~~~~~~~~~~~~~~~~~ TRAINING DATA ~~~~~~~~~~~~~~~~~~~~~~~\n")
print ('The shape of X_train is: ' + str(X_train.shape))
print ('The shape of y_train is: ' + str(y_train.shape))
print('The feature values of the first five examples:\n{}'.format(X_train[:5]))
print('The labels of the first five examples:\n{}'.format(y_train[:5].T))

print("\n~~~~~~~~~~~~~~~~~~~~~~~~ TEST DATA ~~~~~~~~~~~~~~~~~~~~~~~\n")
print ('The shape of X_test is: ' + str(X_test.shape))
print ('The shape of y_test is: ' + str(y_test.shape))
print('The feature values of the first five examples:\n{}'.format(X_test[:5]))
print('The labels of the first five examples:\n{}'.format(y_test[:5].T))
```

```
~~~~~~~~~~~~~~~~~~~~~~~~ TRAINING DATA ~~~~~~~~~~~~~~~~~~~~~~~

The shape of X_train is: (100, 2)
The shape of y_train is: (100, 1)
The feature values of the first five examples:
[[ 0.15245457  1.22613591]
 [ 0.30001788  1.39463829]
 [-1.21376577  1.05650082]
 [ 0.67201309  1.90975093]
 [ 0.37351553 -0.48398591]]
The labels of the first five examples:
[[1. 1. 0. 1. 1.]]

~~~~~~~~~~~~~~~~~~~~~~~~ TEST DATA ~~~~~~~~~~~~~~~~~~~~~~~
```

```
---------------------------- TEST DATA ----------------------------

The shape of X_test is: (100, 2)
The shape of y_test is: (100, 1)
The feature values of the first five examples:
[[-2.37999418  1.2142028 ]
 [-1.61933161 -0.04900294]
 [ 1.84615784  1.4075603 ]
 [ 0.3200881   0.76031681]
 [ 0.98011156  0.99856559]]
The labels of the first five examples:
[[0. 0. 1. 0. 1.]]
```

Task 1. Build the fully connected model.

In [6]:

```
def parameter_initialization(X, y, num_hidden=20, std=0.01):
    """
    Initializes trainable parameters

    Inputs:

    X -- input data with the shape of (m, l1_ns)
    y -- target values with the shape of (m, l5_ns)
    num_hidden -- number of neurons in the hidden layer
    std -- standard deviation
    l1_ns -- input layer neurons
    l2_ns -- the first hidden layer neurons
    l3_ns -- the second hidden layer neurons
    l4_ns -- the third hidden layer neurons
    l5_ns -- output layer neurons

    Output:
    params -- a dictionary with trainable parameters
    """
    np.random.seed(seed)

    l1_ns = X.shape[1]                                          # input layer neurons
    l2_ns, l3_ns, l4_ns = num_hidden, num_hidden, num_hidden    # the first, the second, the third
hidden layer neurons
    l5_ns = y.shape[1]                                          # output layer neurons

    W1 = np.random.randn(l2_ns, l1_ns) * std
    b1 = np.zeros((l2_ns, 1))
    W2 = np.random.randn(l3_ns, l2_ns) * std
    b2 = np.zeros((l3_ns, 1))
    W3 = np.random.randn(l4_ns, l3_ns) * std
    b3 = np.zeros((l4_ns, 1))
    W4 = np.random.randn(l5_ns, l4_ns) * std
    b4 = np.zeros((l5_ns, 1))

    params = {"W1": W1, "b1": b1,
              "W2": W2, "b2": b2,
              "W3": W3, "b3": b3,
              "W4": W4, "b4": b4}

    return params
```

In [7]:

```
params = parameter_initialization(X_train, y_train)
print(f"W1 = {params['W1'].shape}")
print(f"b1 = {params['b1'].shape}")
print(f"W2 = {params['W2'].shape}")
print(f"b2 = {params['b2'].shape}")
print(f"W3 = {params['W3'].shape}")
print(f"b3 = {params['b3'].shape}")
print(f"W4 = {params['W4'].shape}")
print(f"b4 = {params['b4'].shape}")
```

```
W1 = (20, 2)
b1 = (20, 1)
W2 = (20, 20)
```

```
W2 = (20, 20)
b2 = (20, 1)
W3 = (20, 20)
b3 = (20, 1)
W4 = (1, 20)
b4 = (1, 1)
```

Activation functions and their derivatives

In [8]:

```
# Sigmoid activation function
def sigmoid(inp): return 1 / (1 + np.exp(-inp))

# Derivative of sigmoid activation function
def d_sigmoid(inp): return inp * (1 - inp)

# Hyperbolic tanh activation function
def tanh(inp): return ((np.exp(inp) - np.exp(-inp)) / (np.exp(inp) + np.exp(-inp)))

# Derivative of hyperbolic tanh activation function
def d_tanh(inp): return (1 - inp ** 2)

# ReLU activation function
def relu(inp): return np.maximum(0, inp)

# Derivative of the ReLU activation function
def d_relu(inp):

    inp[inp > 0] = 1
    inp[inp <= 0] = 0

    return inp
```

In [9]:

```
a = [1,2,-3,-4,-100, 10000, -211231231]
print(relu(a))
b = np.random.uniform(-5, 5, (5,5))
print(d_relu(b))
```

```
[    1     2     0     0     0 10000     0]
[[0. 1. 0. 1. 1.]
 [1. 1. 1. 0. 1.]
 [1. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 1. 0. 1. 1.]]
```

Task 2. Implement Forward Pass of the Network

In [10]:

```
def forward_pass(X, params, out_act='tanh'):

    """
    Performs forward propagation of the model

    Inputs:

    X -- input data with the shape of (m, l1_ns)
    params -- dictionary with trainable parameters (output of parameter initialization function)
    out_act -- activation function of the output layer, hyperbolic tanh is default

    Outputs:

    out4 -- prediction of the model
    for_backprop -- a dictionary with necessary values for backpropagation
    """

    # Obtaining trainable parameters
    W1 = params['W1']              # shape = (l2_ns, l1_ns)
```

```
    b1 = params['b1']                 # shape = (l2_ns, 1)
    W2 = params['W2']                 # shape = (l3_ns, l2_ns)
    b2 = params['b2']                 # shape = (l3_ns, 1)
    W3 = params['W3']                 # shape = (l4_ns, l3_ns)
    b3 = params['b3']                 # shape = (l4_ns, 1)
    W4 = params['W4']                 # shape = (l5_ns, l4_ns)
    b4 = params['b4']                 # shape = (l5_ns, 1)

    ########################### FORWARD PROPAGATION START ###########################

    act1 = W1.dot(X.T) + b1          # shape = (l2_ns, m) -> (l2_ns, l1_ns) . (l1_ns, m) + (l2_ns, 1)
    out1 = tanh(act1)                 # shape = (l2_ns, m)
    act2 = W2.dot(out1) + b2          # shape = (l3_ns, m) -> (l3_ns, l2_ns) . (l2_ns, m) + (l3_ns, 1)
    out2 = tanh(act2)                 # shape = (l3_ns, m)
    act3 = W3.dot(out2) + b3          # shape = (l3_ns, m) -> (l4_ns, l3_ns) . (l3_ns, m) + (l4_ns, 1)
    out3 = tanh(act3)                 # shape = (l3_ns, m)
    act4 = W4.dot(out3) + b4          # shape = (l3_ns, m) -> (l5_ns, l4_ns) . (l4_ns, m) + (l5_ns, 1)

    if out_act == 'tanh':
        out4 = tanh(act4)             # shape = (l5_ns, m)
    elif out_act == 'sigmoid':
        out4 = sigmoid(act4)          # shape = (l5_ns, m)
    elif out_act == 'relu':
        out4 = relu(act4)             # shape = (l5_ns, m)
    elif out_act == 'no':
        out4 = act4                   # shape = (l5_ns, m)

    ########################### FORWARD PROPAGATION END ###########################

    # Storing values for backpropagation
    for_backprop = {"out1": out1, "out2": out2,
                    "out3": out3, "out4": out4}

    return out4.T, for_backprop
```

In [11]:

```
out4, for_backprop = forward_pass(X_train, params)
print(out4.shape)
```

```
(100, 1)
```

In [12]:

```
def prediction(X, params):
    """
    Makes predictions using learned parameters

    Inputs:

    X -- input data with the shape of (m, l1_ns)
    params -- a dictionary with the optimal parameters obtained after finishing the training

    Output:

    predictions -- a vector of predicted values. The threshold of 0.5 is used to distinguish the c
lasses.
    """

    out4, for_backprop = forward_pass(X, params)
    predictions = (out4 > 0.5)

    return predictions
```

In [13]:

```
preds_train = prediction(X_train, params)
print(preds_train.shape)
```

```
(100, 1)
```

In [14]:

```
def accuracy(preds, targs):

    """
    Computes the accuracy of the model

    Inputs:

    preds -- prediction values of the model with shape of (m, l5_ns)
    targs -- target labels with the shape of (m, l5_ns)

    Output:

    acc -- accuracy percentage
    """

    m = len(preds) # total number of examples
    correct = 0    # number of correctly predicted samples

    # Computing number of correctly predicted examples
    for i in range(m):
        if preds[i] == targs[i]:
            correct += 1

    acc = correct / m * 100

    return acc
```

In [15]:

```
accuracy_train_before = accuracy(preds_train, y_train)
print(f"Accuracy of the model on the test data before training is: {accuracy_train_before}%")
```

```
Accuracy of the model on the test data before training is: 50.0%
```

Task 3. Implement Backward Pass of the Network

In [16]:

```
def mse(preds, targs):
    """
    Computes Mean Squared Error (MSE)

    Inputs:

    preds -- prediction values of the model with shape of (m, l5_ns)
    targs -- target labels with the shape of (m, l5_ns)

    Output:

    cost -- cost value of the model
    """

    m = targs.shape[0] # total number of examples

    sum_squared_differences = np.sum((preds - targs) ** 2)     # float
    cost = (1 / (2 * m)) * sum_squared_differences             # float

    return cost
```

In [17]:

```
cost = mse(out4, y_train)
print("Cost of the training set with randomly initialized parameters is: {:.4f}".format(cost))
```

```
Cost of the training set with randomly initialized parameters is: 0.2500
```

In [18]:

```
def backpropagation(X, y, params, for_backprop, out_act='tanh'):
```

```
    """
    Computes gradients of the trainable parameters.

    Inputs:

    X -- input data with the shape of (m, l1_ns)
    y -- target values with the shape of (m, l5_ns)
    params -- a dictionary with the trainable parameters
    for_backprop -- a dictionary with necessary values for backpropagation
    out_act -- activation function of the output layer, hyperbolic tanh is default

    Output:

    grads -- a dictionary with the gradients with respect to trainable parameters
    """

    m = X.shape[0]                          # total number of examples

    # Obtaining weight parameters
    W1 = params['W1']                       # shape = (l2_ns, l1_ns)
    W2 = params['W2']                       # shape = (l3_ns, l2_ns)
    W3 = params['W3']                       # shape = (l4_ns, l3_ns)
    W4 = params['W4']                       # shape = (l5_ns, l4_ns)

    # Obtaining output values of each layer from forward propagation
    out1 = for_backprop['out1']             # shape = (l2_ns, m)
    out2 = for_backprop['out2']             # shape = (l3_ns, m)
    out3 = for_backprop['out3']             # shape = (l4_ns, m)
    out4 = for_backprop['out4']             # shape = (l5_ns, m)

    ################################ BACKPROPAGATION START ################################

    if out_act == 'tanh':
        d_act4 = ((out4.T - y) * d_tanh(out4.T)).T          # shape = (l5_ns, m) -> (l5_ns, m) * (l
_ns, m)
    elif out_act == 'sigmoid':
        d_act4 = ((out4.T - y) * d_sigmoid(out4.T)).T       # shape = (l5_ns, m) -> (l5_ns, m) *
(l5_ns, m)
    elif out_act == 'relu':
        d_act4 = ((out4.T - y) * d_relu(out4.T)).T          # shape = (l5_ns, m) -> (l5_ns, m) * (l
_ns, m)
    elif out_act == 'no':
        d_act4 = (out4.T - y).T                             # shape = (l5_ns, m) -> (l5_ns, m) *
(l5_ns, m)

    d_W4 = (1 / m) * d_act4.dot(out3.T)                     # shape = (l5_ns, l4_ns) -> (l5_ns, m)
(m, l4_ns)
    d_b4 = (1 / m) * np.sum(d_act4, axis=1, keepdims=True)  # shape = (l5_ns, 1)

    d_act3 = W4.T.dot(d_act4) * d_tanh(out3)                # shape = (l4_ns, m) -> (l4_ns, m) * (l
_ns, m)
    d_W3 = (1 / m) * (d_act3.dot(out2.T))                   # shape = (l4_ns, l3_ns) -> (l4_ns, m)
(m, l3_ns)
    d_b3 = (1 / m) * np.sum(d_act3, axis=1, keepdims=True)  # shape = (l4_ns, 1)

    d_act2 = W3.T.dot(d_act3) * d_tanh(out2)                # shape = (l3_ns, m) -> (l3_ns, l4_ns)
(l4_ns, m) * (l3_ns, m)
    d_W2 = (1 / m) * (d_act2.dot(out1.T))                   # shape = (l3_ns, l2_ns) -> (l3_ns, m)
(m, l2_ns)
    d_b2 = (1 / m) * np.sum(d_act2, axis=1, keepdims=True)  # shape = (l3_ns, 1)

    d_act1 = W2.T.dot(d_act2) * d_tanh(out1)                # shape = (l2_ns, m) -> (l2_ns, l3_ns)
(l3_ns, m) * (l2_ns, m)
    d_W1 = (1 / m) * (d_act1.dot(X))                        # shape = (l2_ns, l1_ns) -> (l2_ns, m)
(m, l1_ns)
    d_b1 = (1 / m) * np.sum(d_act1, axis=1, keepdims=True)  # shape = (l2_ns, 1)


    ################################# BACKPROPAGATION END #################################

    # Storing gradient values
    grads = {"d_W1": d_W1, "d_b1": d_b1,
             "d_W2": d_W2, "d_b2": d_b2,
             "d_W3": d_W3, "d_b3": d_b3,
             "d_W4": d_W4, "d_b4": d_b4}
```

```
    return grads
```

In [19]:

```
grads = backpropagation(X_train, y_train, params, for_backprop)
print(f"d_W1 = {grads['d_W1'].shape}")
print(f"d_b1 = {grads['d_b1'].shape}")
print(f"d_W2 = {grads['d_W2'].shape}")
print(f"d_b2 = {grads['d_b2'].shape}")
print(f"d_W3 = {grads['d_W3'].shape}")
print(f"d_b3 = {grads['d_b3'].shape}")
print(f"d_W4 = {grads['d_W4'].shape}")
print(f"d_b4 = {grads['d_b4'].shape}")
```

```
d_W1 = (20, 2)
d_b1 = (20, 1)
d_W2 = (20, 20)
d_b2 = (20, 1)
d_W3 = (20, 20)
d_b3 = (20, 1)
d_W4 = (1, 20)
d_b4 = (1, 1)
```

In [20]:

```
def gradient_descent(params, grads, lr = 1.1):
    """
    Perform a single step of the gradient descent algorithm

    Inputs:

    params -- a dictionary with the trainable parameters
    grads -- a dictionary with the gradients with respect to trainable parameters

    Output:

    params -- a dictionary with the updated trainable parameters
    """

    # Obtaining trainable parameters to update
    W1 = params['W1']        # shape = (l2_ns, l1_ns)
    b1 = params['b1']        # shape = (l2_ns, 1)
    W2 = params['W2']        # shape = (l3_ns, l2_ns)
    b2 = params['b2']        # shape = (l3_ns, 1)
    W3 = params['W3']        # shape = (l4_ns, l3_ns)
    b3 = params['b3']        # shape = (l4_ns, 1)
    W4 = params['W4']        # shape = (l5_ns, l4_ns)
    b4 = params['b4']        # shape = (l5_ns, 1)

    # Obtaining gradient values
    d_W1 = grads['d_W1']     # shape = (l2_ns, l1_ns)
    d_b1 = grads['d_b1']     # shape = (l2_ns, 1)
    d_W2 = grads['d_W2']     # shape = (l3_ns, l2_ns)
    d_b2 = grads['d_b2']     # shape = (l3_ns, 1)
    d_W3 = grads['d_W3']     # shape = (l4_ns, l3_ns)
    d_b3 = grads['d_b3']     # shape = (l4_ns, 1)
    d_W4 = grads['d_W4']     # shape = (l5_ns, l4_ns)
    d_b4 = grads['d_b4']     # shape = (l5_ns, 1)


    ################ GRADIENT DESCENT START ################

    W1 -= lr * d_W1          # shape = (l2_ns, l1_ns)
    b1 -= lr * d_b1          # shape = (l2_ns, 1)
    W2 -= lr * d_W2          # shape = (l3_ns, l2_ns)
    b2 -= lr * d_b2          # shape = (l3_ns, 1)
    W3 -= lr * d_W3          # shape = (l4_ns, l3_ns)
    b3 -= lr * d_b3          # shape = (l4_ns, 1)
    W4 -= lr * d_W4          # shape = (l5_ns, l4_ns)
    b4 -= lr * d_b4          # shape = (l5_ns, 1)


    ################ GRADIENT DESCENT END ################
```

```
    params = {"W1": W1, "b1": b1,
              "W2": W2, "b2": b2,
              "W3": W3, "b3": b3,
              "W4": W4, "b4": b4}

    return params
```

In [21]:

```
params = gradient_descent(params, grads)
print(f"W1 = {params['W1'].shape}")
print(f"b1 = {params['b1'].shape}")
print(f"W2 = {params['W2'].shape}")
print(f"b2 = {params['b2'].shape}")
print(f"W3 = {params['W3'].shape}")
print(f"b3 = {params['b3'].shape}")
print(f"W4 = {params['W4'].shape}")
print(f"b4 = {params['b4'].shape}")
```

```
W1 = (20, 2)
b1 = (20, 1)
W2 = (20, 20)
b2 = (20, 1)
W3 = (20, 20)
b3 = (20, 1)
W4 = (1, 20)
b4 = (1, 1)
```

Combine all functions into a single model

In [22]:

```
def model(X, y, num_iterations, verbose=False):
    """
    Creates a model with the trained parameters

    Inputs:

    X -- input data with the shape of (m, l1_ns)
    y -- target values with the shape of (m, l4_ns)
    num_iterations -- number of iterations through the data
    verbose -- information about the training process

    Outputs:

    params -- a dictionary with the optimal parameters obtained after finishing the training
    cost_history -- a list with the values of costs
    accuracy_history -- a list with the accuracy scores
    """

    # Lists to store the cost and accuracy values
    cost_history, accuracy_history = [], []

    # Parameter initialization
    params = parameter_initialization(X, y)

    # Updating parameters
    for iteration in range(0, num_iterations):

        # Forward propagation
        out4, for_backprop = forward_pass(X, params)

        # Cost calculation
        cost = mse(out4, y)
        cost_history.append(cost)

        # Backward propagation
        grads = backpropagation(X, y, params, for_backprop)

        # Gradient descent step
        params = gradient_descent(params, grads)

        # Accuracy score
```

```
        preds = prediction(X, params)
        accuracy_score = accuracy(preds, y)
        accuracy_history.append(accuracy_score)

        # Training process information
        if verbose:
            if iteration % 500 == 0 and iteration != 0:
                print ("Cost after iteration {} is: {:.5f}".format(iteration, cost))

    return params, cost_history, accuracy_history
```

Training the model and computing training accuracy

In [23]:

```
params, cost_history, accuracy_history  = model(X_train, y_train, 7000, verbose=True)
```

```
Cost after iteration 500 is: 0.12499
Cost after iteration 1000 is: 0.05033
Cost after iteration 1500 is: 0.04776
Cost after iteration 2000 is: 0.04764
Cost after iteration 2500 is: 0.04767
Cost after iteration 3000 is: 0.04766
Cost after iteration 3500 is: 0.04754
Cost after iteration 4000 is: 0.04731
Cost after iteration 4500 is: 0.04703
Cost after iteration 5000 is: 0.04648
Cost after iteration 5500 is: 0.04211
Cost after iteration 6000 is: 0.04101
Cost after iteration 6500 is: 0.04025
```

In [24]:

```
training_predictions = prediction(X_train, params)
training_accuracy = accuracy(training_predictions, y_train)
print(f"Accuracy of the model on the training data after training is: {training_accuracy}%")
```

```
Accuracy of the model on the training data after training is: 92.0%
```

Plot the decision boundary

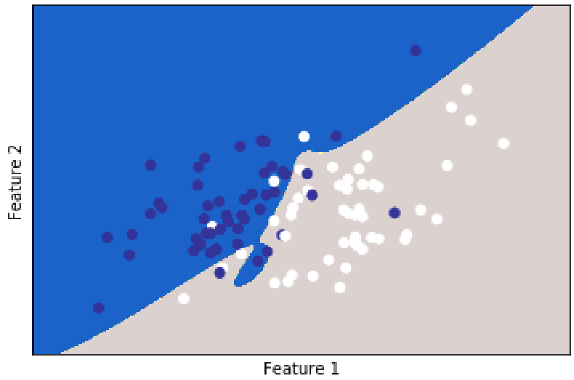
In [25]:

```
def decision_boundary(model, X, y):
    """
    Plots decision boundary of the model on the given data

    Inputs:

    model -- a network with trained parameters
    X -- input data with the shape of (m, l1_ns)
    y -- target values with the shape of (m, l4_ns)

    Output:

    Scatter plot
    """

    cm = plt.cm.terrain

    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    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 grid
    Z = model(np.c_[xx.ravel(), yy.ravel()])
```

```
    Z = Z.reshape(xx.shape)

    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=cm)
    plt.ylabel('Feature 2')
    plt.xlabel('Feature 1')
    plt.scatter(X[0, :], X[1, :], c=y, cmap=cm)
    plt.xticks(())
    plt.yticks(())
```

In [26]:

```
# Plot the decision boundary
y_train = y_train.reshape(y_train.shape[0], )
decision_boundary(lambda x: prediction(x, params), X_train.T, y_train.T)
```

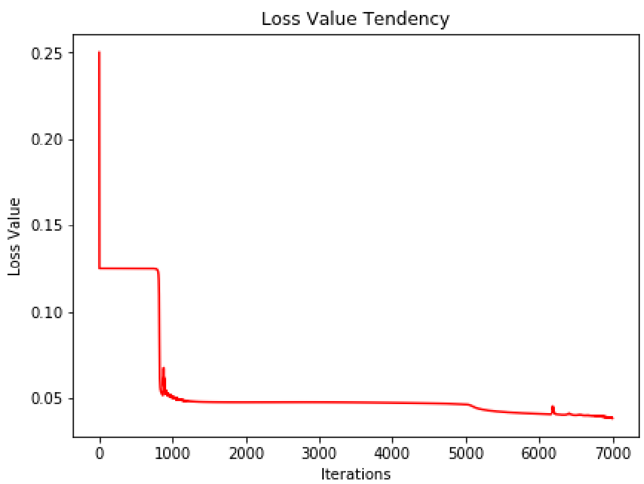


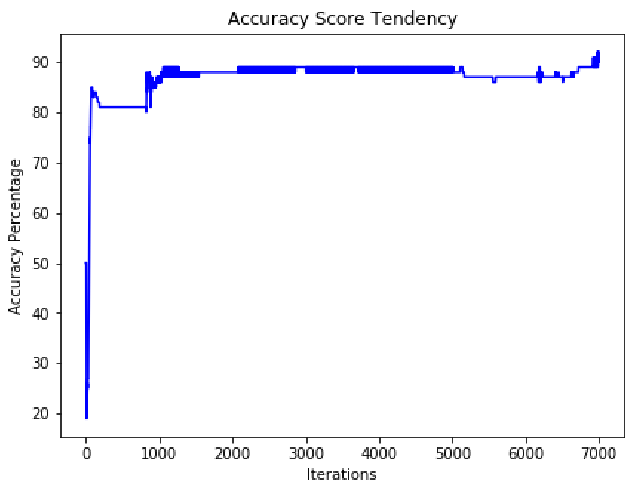
In [27]:

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))

iterations = len(cost_history)
axes[0].plot(range(iterations), cost_history, c='r')
axes[0].set_xlabel('Iterations')
axes[0].set_ylabel('Loss Value')
axes[0].set_title('Loss Value Tendency')


axes[1].plot(range(iterations), accuracy_history, c='b')
axes[1].set_xlabel('Iterations')
axes[1].set_ylabel('Accuracy Percentage')
axes[1].set_title('Accuracy Score Tendency')
```

Out[27]:

```
Text(0.5, 1.0, 'Accuracy Score Tendency')
```





Test accuracy after training process

In [28]:

```
test_predictions = prediction(X_test, params)
test_accuracy = accuracy(test_predictions, y_test)
print(f"Accuracy of the model on the test data after training is: {test_accuracy}%")
```

```
Accuracy of the model on the test data after training is: 89.0%
```

In [29]:

```
y_test = y_test.reshape(100, )
decision_boundary(lambda x: prediction(x, params), X_test.T, y_test.T)
```