1. Implement a 3 layer multilayer perceptron neural network with 2-4-1 architecture and solve the EX OR classification problem using backpropagation algorithm. Note: Don't consider bias at any neuron. Use Sigmoid activation function at every neuron. Train for 100 epochs. Plot the convergence graph.

In [51]:
```
import numpy as np
import matplotlib.pyplot as plt
X=np.array(([0,0],[0,1],[1,0],[1,1]), dtype=float)
y=np.array(([0],[1],[1],[0]), dtype=float)
```

In [52]:
```
def sigmoid(t):
    '''This will return the sigmoid value of the function'''
    return 1/(1+np.exp(-t))
```

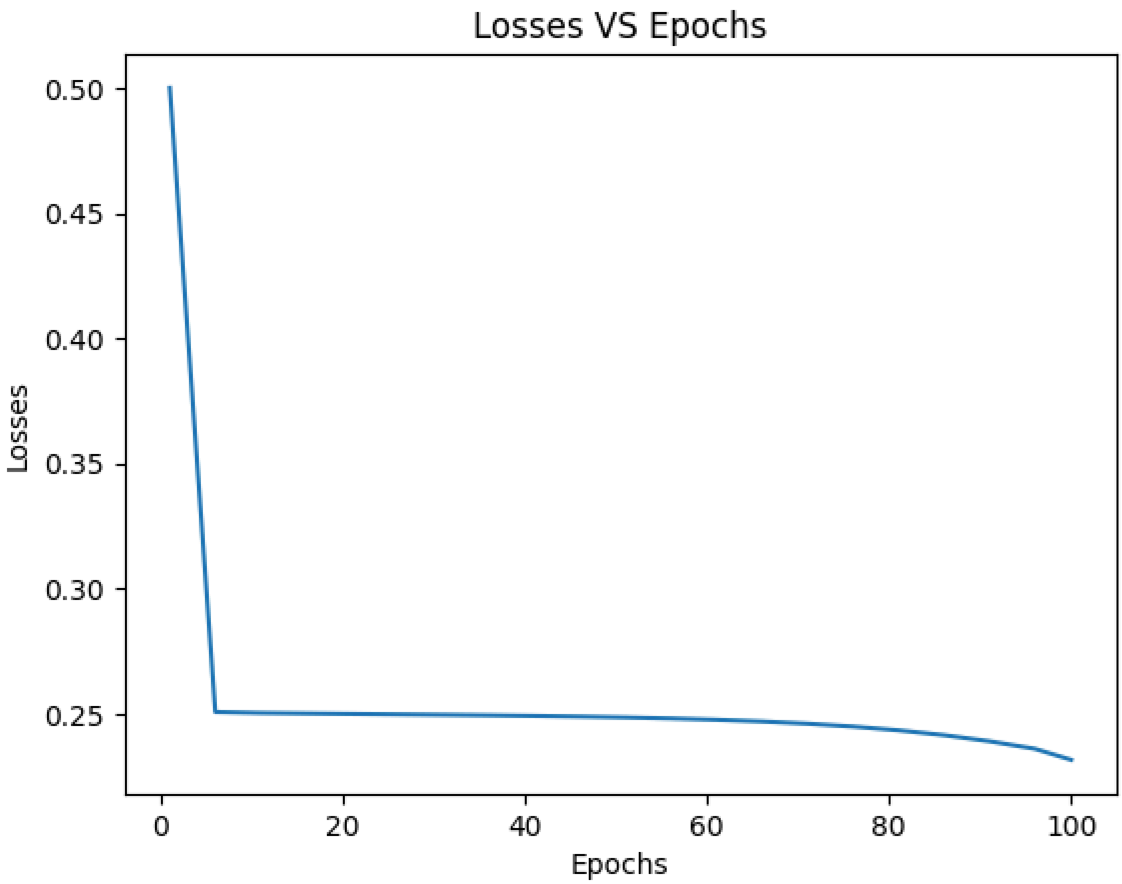
In [53]:
```
def sigmoid_derivative(d):
    return d * (1 - d)
```

In [54]:
```
class NeuralNetwork:
    def __init__(self, x,y):
        self.input = x
        self.weights1= np.random.rand(self.input.shape[1],4)
        self.weights2 = np.random.rand(4,1)
        self.y = y
        self.output = np. zeros(y.shape)

    def feedforward(self):
        '''This will perform the forward propagation for the next 2 layers'''
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.layer2 = sigmoid(np.dot(self.layer1, self.weights2))
        return self.layer2

    def backprop(self):
        '''Back propagation of the final hidden layers to initial layers'''
        derv_weights2 = np.dot(self.layer1.T, 2*(self.y -self.output)*sigmoid_derivativ
        derv_weights1 = np.dot(self.input.T, np.dot(2*(self.y -self.output)*sigmoid_der

        self.weights1 += derv_weights1
        self.weights2 += derv_weights2

    def train(self, X, y):
        self.output = self.feedforward()
        self.backprop()
```

In [56]:
```
model=NeuralNetwork(X,y)
iterations = 100
losses = []
ep = []
for i in range(iterations):
    if i % 5 == 0:
        losses.append(np.mean(np.square(y - model.output)))
        ep.append(i+1)
```

```
    model.train(X, y)

print("For iteration #", iterations)
print ("Input : \n" + str(X))
print ("Actual Output: \n" + str(y))
print ("Predicted Output: \n" + str(model.feedforward()))
loss = np.mean(np.square(y - model.feedforward()))
print ("Loss: \n" + str(loss))
losses.append(loss)
ep.append(100)
print ("\n")

plt.plot(ep, losses)
plt.title('Losses VS Epochs')
plt.xlabel('Epochs')
plt.ylabel('Losses')
plt.show()
```

```
For iteration # 100
Input :
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]
Actual Output:
[[0.]
 [1.]
 [1.]
 [0.]]
Predicted Output:
[[0.45078146]
 [0.51598121]
 [0.54783161]
 [0.53342321]]
Loss:
0.23161867180459328
```



1. Implement a 3 layer multilayer perceptron neural network with 2-6-1 architecture and solve the EX OR classification problem using backpropagation algorithm. Note: Don't consider bias at any neuron. Use Sigmoid activation function at every neuron. Train for 100 epochs. Plot the convergence graph.

In [57]:
```
import numpy as np
X = np.array(([0,0], [0,1],[1,0], [1,1], [0,0], [1,0]), dtype = float)
y = np.array(([0], [1], [1], [0], [0], [1]), dtype = float)
```

In [58]:
```
def sigmoid(t):
    '''This will return the sigmoid value of the function'''
    return 1/(1+np.exp(-t))
```

In [59]:
```
def sigmoid_derivative(d):
    return d * (1 - d)
```

In [60]:
```
class NeuralNetwork:
    def __init__(self, x,y):
        self.input = x
        self.weights1= np.random.rand(self.input.shape[1],6)
        self.weights2 = np.random.rand(6,1)
        self.y = y
        self.output = np. zeros(y.shape)
```

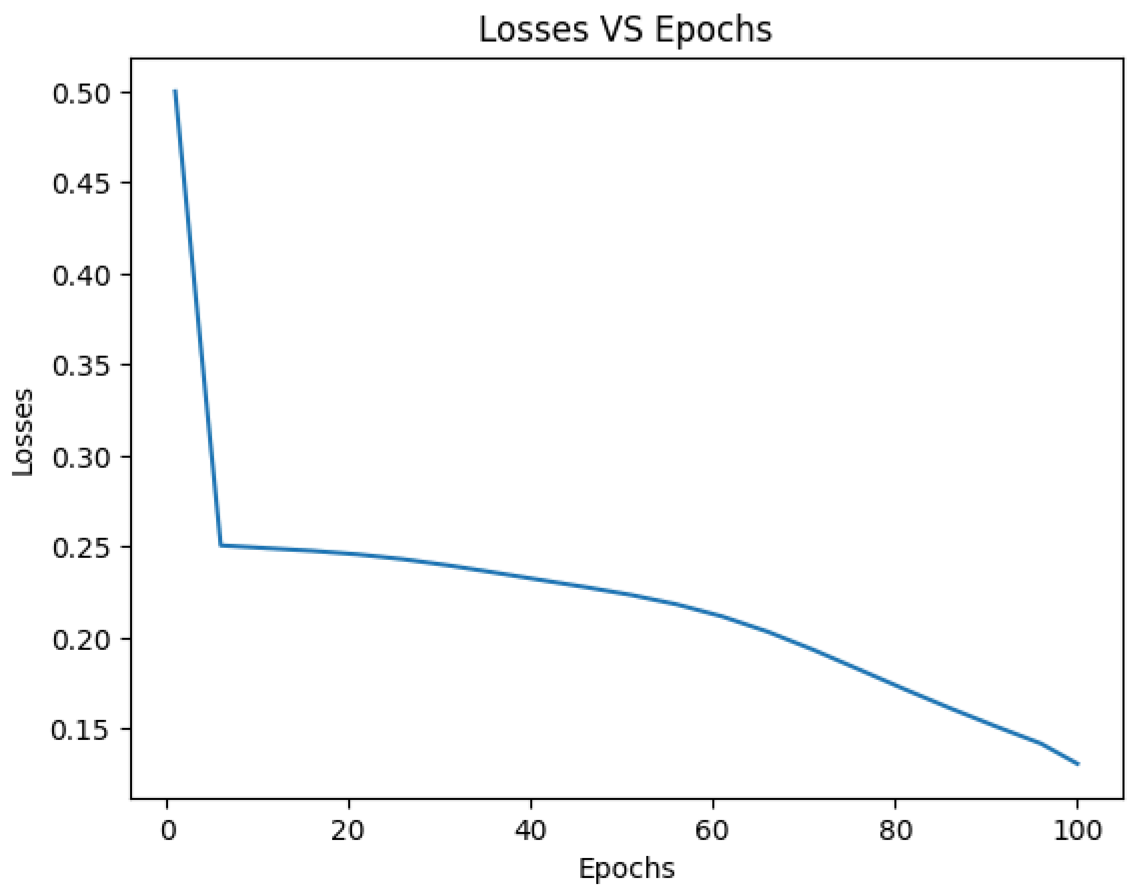
```
    def feedforward(self):
        '''This will perform the forward propagation for the next 2 layers'''
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.layer2 = sigmoid(np.dot(self.layer1, self.weights2))
        return self.layer2

    def backprop(self):
        '''Back propagation of the final hidden layers to initial layers'''
        derv_weights2 = np.dot(self.layer1.T, 2*(self.y -self.output)*sigmoid_derivativ
        derv_weights1 = np.dot(self.input.T, np.dot(2*(self.y -self.output)*sigmoid_der

        self.weights1 += derv_weights1
        self.weights2 += derv_weights2

    def train(self, X, y):
        self.output = self.feedforward()
        self.backprop()
```

In [61]:

```
model=NeuralNetwork(X,y)
iterations = 100
losses = []
ep = []
for i in range(iterations):
    if i % 5 == 0:
        losses.append(np.mean(np.square(y - model.output)))
        ep.append(i+1)
    model.train(X, y)

print("For iteration #", iterations)
print ("Input : \n" + str(X))
print ("Actual Output: \n" + str(y))
print ("Predicted Output: \n" + str(model.feedforward()))
loss = np.mean(np.square(y - model.feedforward()))
print ("Loss: \n" + str(loss))
losses.append(loss)
ep.append(100)
print ("\n")

plt.plot(ep, losses)
plt.title('Losses VS Epochs')
plt.xlabel('Epochs')
plt.ylabel('Losses')
plt.show()
```

```
For iteration # 100
Input :
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]
 [0. 0.]
 [1. 0.]]
Actual Output:
[[0.]
 [1.]
 [1.]
 [0.]
 [0.]
 [1.]]
```

```
Predicted Output:
[[0.2829594 ]
 [0.47307237]
 [0.75870183]
 [0.47972837]
 [0.2829594 ]
 [0.75870183]]
Loss:
0.1307289468139013
```



1. Implement a 3 layer multilayer perceptron neural network with 2-6-1 architecture and solve the EX OR classification problem using backpropagation algorithm. Note: Consider bias at every neuron. Use Sigmoid activation function at every neuron. Train for 100 epochs. Plot the convergence graph.

In [69]:

```
X=np.array(([0,0],[0,1],[1,0],[1,1], [0,0],[1,0]), dtype=float)
y=np.array(([0],[1],[1],[0],[0],[1]), dtype=float)
```

In [71]:

```
class NeuralNetwork:
    def __init__(self, x,y):
        self.input = x
        self.weights1= np.random.rand(2,6)
        self.weights2 = np.random.rand(6,1)
        self.bias1 = np.random.rand(1,6)
        self.bias2 = np.random.rand(1,1)
        self.y = y
```

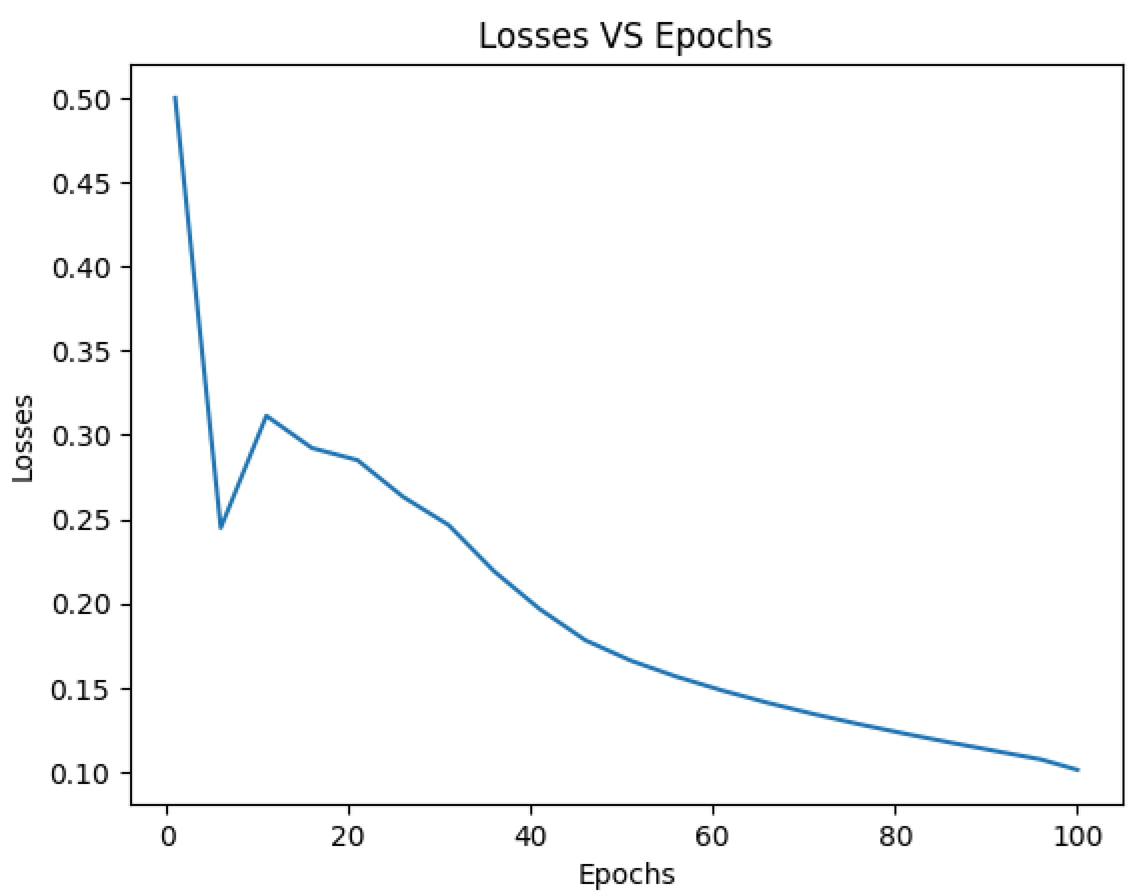
```python
        self.output = np. zeros(y.shape)

    def feedforward(self):
        '''This will perform the forward propagation for the next 2 layers'''
        self.layer1 = sigmoid(np.dot(self.input, self.weights1) + self.bias1)
        self.layer2 = sigmoid(np.dot(self.layer1, self.weights2) + self.bias2)
        return self.layer2

    def backprop(self):
        '''Backpropagation of the final hidden layers to initial layers'''
        error = self.y - self.output

        d_weights2 = np.dot(self.layer1.T, 2 * error * sigmoid_derivative(self.output))
        d_bias2 = np.sum(2 * error * sigmoid_derivative(self.output), axis=0, keepdims=

        error_hidden_layer = np.dot(2 * error * sigmoid_derivative(self.output), self.w

        d_weights1 = np.dot(self.input.T, error_hidden_layer * sigmoid_derivative(self.
        d_bias1 = np.sum(error_hidden_layer * sigmoid_derivative(self.layer1), axis=0)

        self.weights1 += d_weights1
        self.weights2 += d_weights2
        self.bias1 += d_bias1
        self.bias2 += d_bias2


    def train(self, X, y):
        self.output = self.feedforward()
        self.backprop()
```

In [72]:

```python
model=NeuralNetwork(X,y)
iterations = 100
losses = []
ep = []
for i in range(iterations):
    if i % 5 == 0:
        losses.append(np.mean(np.square(y - model.output)))
        ep.append(i+1)
    model.train(X, y)

print("For iteration #", iterations)
print ("Input : \n" + str(X))
print ("Actual Output: \n" + str(y))
print ("Predicted Output: \n" + str(model.feedforward()))
loss = np.mean(np.square(y - model.feedforward()))
print ("Loss: \n" + str(loss))
losses.append(loss)
ep.append(iterations)
print ("\n")

plt.plot(ep, losses)
plt.title('Losses VS Epochs')
plt.xlabel('Epochs')
plt.ylabel('Losses')
plt.show()
```

```
For iteration # 100
Input :
[[0. 0.]
```

```
 [0. 1.]
 [1. 0.]
 [1. 1.]
 [0. 0.]
 [1. 0.]]
Actual Output:
[[0.]
 [1.]
 [1.]
 [0.]
 [0.]
 [1.]]
Predicted Output:
[[0.1136025 ]
 [0.56074772]
 [0.807017  ]
 [0.56099046]
 [0.1136025 ]
 [0.807017  ]]
Loss:
0.10132479990887612
```



1. Implement a 3 layer multilayer perceptron neural network with 2-6-1 architecture and solve the EX OR classification problem using backpropagation algorithm. Note: Consider bias at every neuron. Use ReLU activation function at hidden layer neurons and Sigmoid activation function at output layer neuron Train for 100 epochs. Plot the convergence graph.

In [73]:

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

def relu_derivative(x):
    x[x<=0] = 0
    x[x>0] = 1
    return x
```

In [74]:

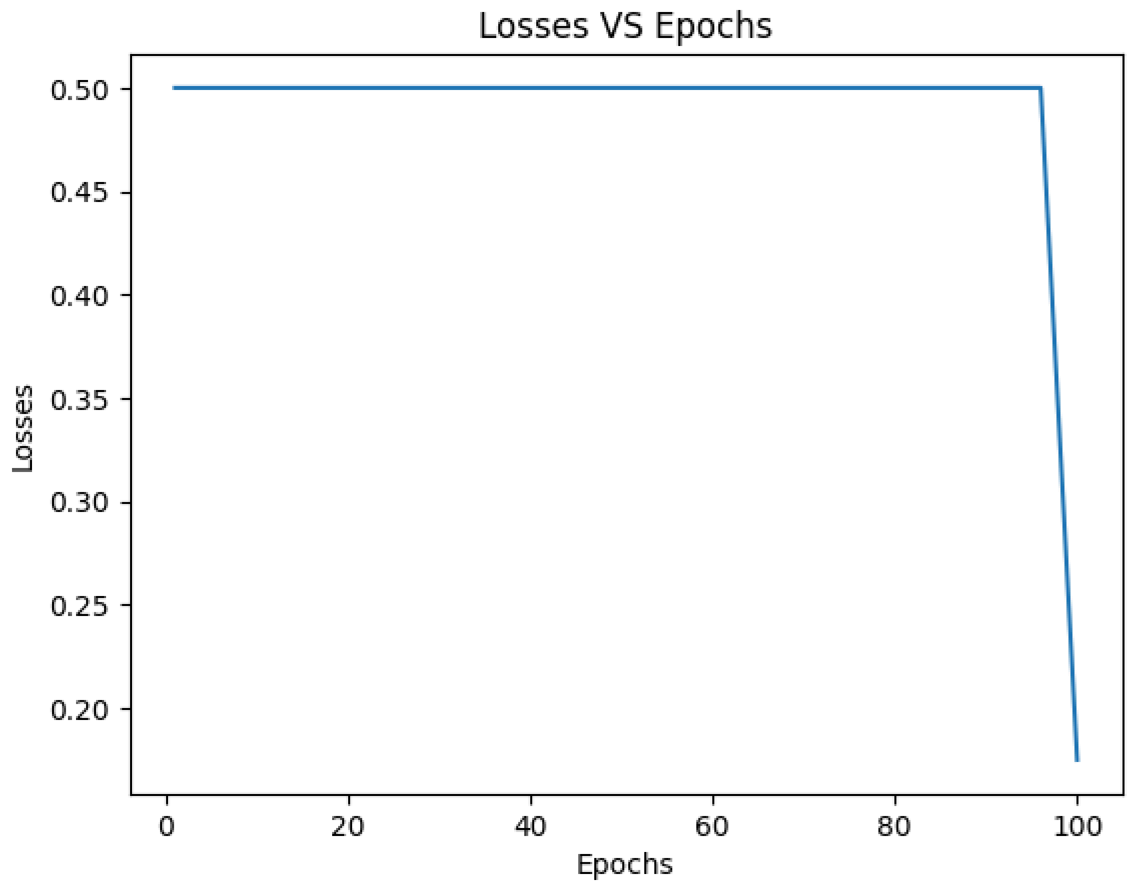
```
class NeuralNetwork:
    def __init__(self, x,y):
        self.input = x
        self.weights1= np.random.rand(2,6)
        self.weights2 = np.random.rand(6,1)
        self.bias1 = np.random.rand(1,6)
        self.bias2 = np.random.rand(1,1)
        self.y = y
        self.output = np. zeros(y.shape)

    def feedforward(self):
        '''This will perform the forward propagation for the next 2 layers'''
        self.layer1 = relu(np.dot(self.input, self.weights1) + self.bias1)
        self.layer2 = sigmoid(np.dot(self.layer1, self.weights2) + self.bias2)
        return self.layer2

    def backprop(self):
        '''Backpropagation of the final hidden layers to initial layers'''
        error = self.y - self.output

        d_weights2 = np.dot(self.layer1.T, 2 * error * relu_derivative(self.output))
        d_bias2 = np.sum(2 * error * relu_derivative(self.output), axis=0, keepdims=Tru

        error_hidden_layer = np.dot(2 * error * sigmoid_derivative(self.output), self.w

        d_weights1 = np.dot(self.input.T, error_hidden_layer * sigmoid_derivative(self.
        d_bias1 = np.sum(error_hidden_layer * sigmoid_derivative(self.layer1), axis=0)

        self.weights1 += d_weights1
        self.weights2 += d_weights2
        self.bias1 += d_bias1
        self.bias2 += d_bias2

    def train(self, X, y):
        self.output = self.feedforward()
        self.backprop()
```

In [75]:

```
model=NeuralNetwork(X,y)
iterations = 100
losses = []
ep = []
for i in range(iterations):
    if i % 5 == 0:
        losses.append(np.mean(np.square(y - model.output)))
        ep.append(i+1)
    model.train(X, y)

print("For iteration #", iterations)
```

```
print ("Input : \n" + str(X))
print ("Actual Output: \n" + str(y))
print ("Predicted Output: \n" + str(model.feedforward()))
loss = np.mean(np.square(y - model.feedforward()))
print ("Loss: \n" + str(loss))
losses.append(loss)
ep.append(iterations)
print ("\n")

plt.plot(ep, losses)
plt.title('Losses VS Epochs')
plt.xlabel('Epochs')
plt.ylabel('Losses')
plt.show()
```

```
For iteration # 100
Input :
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]
 [0. 0.]
 [1. 0.]]
Actual Output:
[[0.]
 [1.]
 [1.]
 [0.]
 [0.]
 [1.]]
Predicted Output:
[[0.15586705]
 [0.99999834]
 [1.        ]
 [1.        ]
 [0.15586705]
 [1.        ]]
Loss:
0.17476484606961853
```



1. Implement a 3 layer multilayer perceptron neural network with 2-6-1 architecture and solve the EX OR classification problem using backpropagation algorithm. Note: Consider bias at every neuron. Use Sigmoid activation function at hidden layer neurons and ReLU activation function at output layer neuron Train for 100 epochs. Plot the convergence graph.

In [76]:

```
class NeuralNetwork:
    def __init__(self, x,y):
        self.input = x
        self.weights1= np.random.rand(2,6)
        self.weights2 = np.random.rand(6,1)
        self.bias1 = np.random.rand(1,6)
        self.bias2 = np.random.rand(1,1)
        self.y = y
        self.output = np. zeros(y.shape)

    def feedforward(self):
        '''This will perform the forward propagation for the next 2 layers'''
        self.layer1 = sigmoid(np.dot(self.input, self.weights1) + self.bias1)
        self.layer2 = relu(np.dot(self.layer1, self.weights2) + self.bias2)
        print(self.layer2)
        return self.layer2

    def backprop(self):
        '''Backpropagation of the final hidden layers to initial layers'''
        error = self.y - self.output

        d_weights2 = np.dot(self.layer1.T, 2 * error * sigmoid_derivative(self.output))
        d_bias2 = np.sum(2 * error * sigmoid_derivative(self.output), axis=0, keepdims=
```

```
        error_hidden_layer = np.dot(2 * error * relu_derivative(self.output), self.weig

        d_weights1 = np.dot(self.input.T, error_hidden_layer * relu_derivative(self.lay
        d_bias1 = np.sum(error_hidden_layer * relu_derivative(self.layer1), axis=0)

        self.weights1 += d_weights1
        self.weights2 += d_weights2
        self.bias1 += d_bias1
        self.bias2 += d_bias2

    def train(self, X, y):
        self.output = self.feedforward()
        self.backprop()
```

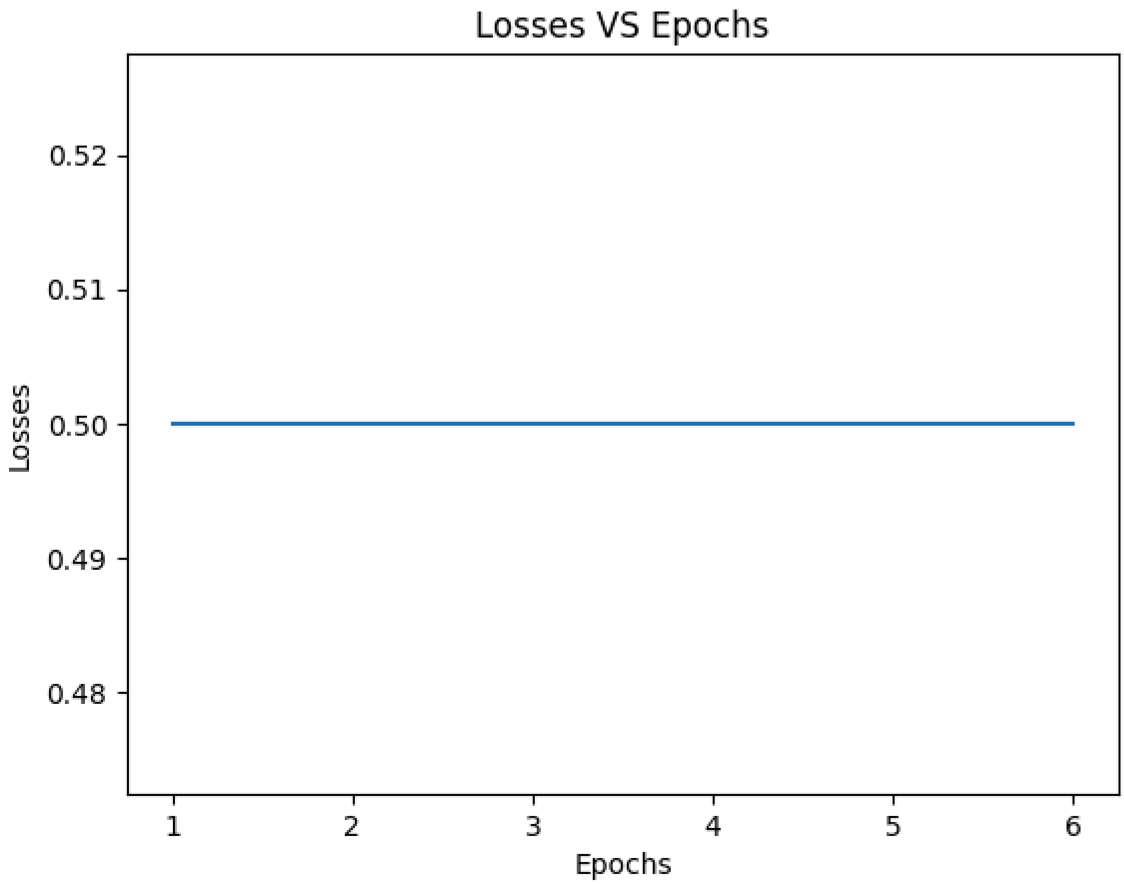
In [77]:

```
model=NeuralNetwork(X,y)
iterations = 7
losses = []
ep = []
for i in range(iterations):
    if i % 5 == 0:
        losses.append(np.mean(np.square(y - model.output)))
        ep.append(i+1)
    model.train(X, y)

print("For iteration #", iterations)
print ("Input : \n" + str(X))
print ("Actual Output: \n" + str(y))
print ("Predicted Output: \n" + str(model.feedforward()))
loss = np.mean(np.square(y - model.feedforward()))
print ("Loss: \n" + str(loss))
losses.append(loss)
ep.append(iterations)
print ("\n")

plt.plot(ep, losses)
plt.title('Losses VS Epochs')
plt.xlabel('Epochs')
plt.ylabel('Losses')
plt.show()
```

```
[[1.8954321 ]
 [2.13740723]
 [2.13029638]
 [2.31880229]
 [1.8954321 ]
 [2.13029638]]
[[47.82401529]
 [45.73744062]
 [46.53973967]
 [45.16199956]
 [47.82401529]
 [46.53973967]]
[[1177712.23326635]
 [1177712.23326635]
 [1177712.23326635]
 [1177712.23326635]
 [1177712.23326635]
 [1177712.23326635]]
```

```
[[1.96019037e+19]
 [1.96019037e+19]
 [1.96019037e+19]
 [1.96019037e+19]
 [1.96019037e+19]
 [1.96019037e+19]]
[[9.03807625e+58]
 [9.03807625e+58]
 [9.03807625e+58]
 [9.03807625e+58]
 [9.03807625e+58]
 [9.03807625e+58]]
[[8.85950074e+177]
 [8.85950074e+177]
 [8.85950074e+177]
 [8.85950074e+177]
 [8.85950074e+177]
 [8.85950074e+177]]
[[nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]]
For iteration # 7
Input :
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]
 [0. 0.]
 [1. 0.]]
Actual Output:
[[0.]
 [1.]
 [1.]
 [0.]
 [0.]
 [1.]]
[[nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]]
Predicted Output:
[[nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]]
[[nan]
 [nan]
 [nan]
 [nan]
 [nan]
 [nan]]
Loss:
nan
```

```
C:\Users\ADMIN\AppData\Local\Temp\ipykernel_25556\2418103403.py:3: RuntimeWarning: overf
low encountered in exp
  return 1/(1+np.exp(-t))
C:\Users\ADMIN\AppData\Local\Temp\ipykernel_25556\1397129429.py:2: RuntimeWarning: overf
low encountered in multiply
  return d * (1 - d)
```



1. Implement a 3 layer multilayer perceptron neural network with 2-6-1 architecture and solve the EX OR classification problem using backpropagation algorithm. Note: Consider bias at every neuron. Use ReLU activation function at every neuron Train for 100 epochs. Plot the convergence graph.

In [78]:
```
class NeuralNetwork:
    def __init__(self, x,y):
        self.input = x
        self.weights1= np.random.rand(2,6)
        self.weights2 = np.random.rand(6,1)
        self.bias1 = np.random.rand(1,6)
        self.bias2 = np.random.rand(1,1)
        self.y = y
        self.output = np. zeros(y.shape)

    def feedforward(self):
        '''This will perform the forward propagation for the next 2 layers'''
        self.layer1 = relu(np.dot(self.input, self.weights1) + self.bias1)
        self.layer2 = relu(np.dot(self.layer1, self.weights2) + self.bias2)
        print(self.layer2)
```

```
        return self.layer2

    def backprop(self):
        '''Backpropagation of the final hidden layers to initial layers'''
        error = self.y - self.output

        d_weights2 = np.dot(self.layer1.T, 2 * error * relu_derivative(self.output))
        d_bias2 = np.sum(2 * error * relu_derivative(self.output), axis=0, keepdims=Tru

        error_hidden_layer = np.dot(2 * error * relu_derivative(self.output), self.weig

        d_weights1 = np.dot(self.input.T, error_hidden_layer * relu_derivative(self.lay
        d_bias1 = np.sum(error_hidden_layer * relu_derivative(self.layer1), axis=0)

        self.weights1 += d_weights1
        self.weights2 += d_weights2
        self.bias1 += d_bias1
        self.bias2 += d_bias2

    def train(self, X, y):
        self.output = self.feedforward()
        self.backprop()
```

In [79]:

```
model=NeuralNetwork(X,y)
iterations = 40
losses = []
ep = []
for i in range(iterations):
    if i % 5 == 0:
        losses.append(np.mean(np.square(y - model.output)))
        ep.append(i+1)
    model.train(X, y)

print("For iteration #", iterations)
print ("Input : \n" + str(X))
print ("Actual Output: \n" + str(y))
print ("Predicted Output: \n" + str(model.feedforward()))
loss = np.mean(np.square(y - model.feedforward()))
print ("Loss: \n" + str(loss))
losses.append(loss)
ep.append(iterations)
print ("\n")

plt.plot(ep, losses)
plt.title('Losses VS Epochs')
plt.xlabel('Epochs')
plt.ylabel('Losses')
plt.show()
```

```
[[2.63889729]
 [4.69845804]
 [3.72705056]
 [5.7866113 ]
 [2.63889729]
 [3.72705056]]
[[0.]
 [0.]
 [0.]
 [0.]
```

```
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
```

```
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]
   [0.]]
  [[0.]
   [0.]
   [0.]
   [0.]
```

```
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
  [0.]
  [0.]]
 [[0.]
  [0.]
  [0.]
  [0.]
```

```
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
For iteration # 40
Input :
[[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]
 [0. 0.]
 [1. 0.]]
Actual Output:
[[0.]
```

```
 [1.]
 [1.]
 [0.]
 [0.]
 [1.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Predicted Output:
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Loss:
0.5
```