XOR Network

In [1]:

```
import numpy as np
from numpy.random import randn, randint
import matplotlib.pyplot as plt
```

In [2]:

```
def sigmoid(x):
    '''Sigmoid function given values of x.'''
    return 1 / (1 + np.exp(-x))
```

In [3]:

```
def sigmoid_der(x):
    '''A function to compute the derivative of the
       sigmoid function given value x.'''
    return sigmoid(x) * (1. - sigmoid(x))
```

In [4]:

```
# input the data
data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
print(data)
```

```
[[0 0]
 [0 1]
 [1 0]
 [1 1]]
```

In [5]:

```
# make the corresponding labels
labels = np.array([0, 1, 1, 0])
print(labels)
```

```
[0 1 1 0]
```

In [6]:

```
# define some useful variables
n_inputs = 2  # number of input nodes
n_hidden = 5  # number of hidden nodes
n_outputs = 1  # number of output nodes
n_epochs = 6000 # number of times going through the dataset
learning_rate = .1 # learning rate
```

In [7]:

```
# make the weight matrix between input layer and hidden layer
w1 = randn(n_inputs + 1, n_hidden) * .01
print(w1.shape)
```

```
(3, 5)
```

In [8]:

```
# make the weight matrix between hidden layer and output node
w2 = randn(n_hidden+1, n_outputs) * .01
print(w2.shape)
```

```
(6, 1)
```

In [9]:

```
# add a bias column to the dataset
biases = np.ones([data.shape[0], 1])  # create column
data = np.concatenate((biases, data), axis=1)  # add to data as first column
print(data.shape)
```

```
(4, 3)
```

In [10]:

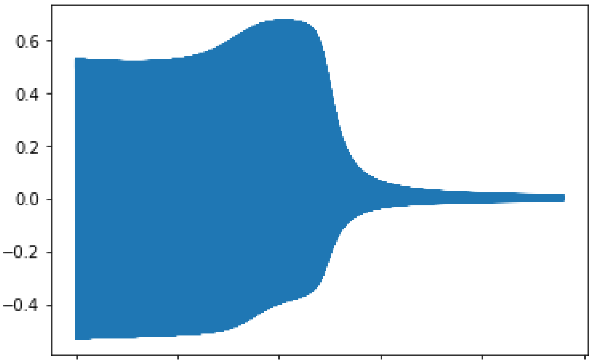
```
# create a list for the error each iteration
errors = []

# create an array for the outputs for each input over training
outputs = np.zeros([n_epochs, 4])

# create an empty array for the error of each input during training
class_errors = np.zeros([n_epochs, 4])
```

In [11]:

```
for epoch in range(n_epochs):
    for i in range(data.shape[0]):
        # forward pass
        x = data[i, :]  # take one example from data
        x = x[None, :]  # add a dimension to do matrix ops later
        y = labels[i]  # take the corresponding label

        h_linear = np.matmul(x, w1) # compute hidden values --- 1 x 5
        h_act = sigmoid(h_linear)  # activation function over hidden nodes --- 1 x 5
        h_act = np.concatenate((np.ones([1, 1]), h_act), 1) # add bias to hidden activations --- 1
x 6

        y_hat = sigmoid(np.matmul(h_act, w2))[0, 0] # compute output
        outputs[epoch, i] = y_hat  # add the output to the outputs array

        # backward pass
        d3 = y_hat - y # output error
        class_errors[epoch, i] = d3  # add the error for this output to class errors
        errors.append(d3)  # add error to list of errors from previous iterations to plot later

        d2 = d3 * w2[1:, :] * sigmoid_der(h_linear.T)  # get error of hidden layer nodes

        dw2 = d3 * h_act  # gradient w2
        dw1 = np.matmul(d2, x)  # gradient w1

        w1 -= (learning_rate * dw1.T)  # weight update for w1
        w2 -= (learning_rate * dw2.T)  # weight update for w2
```

In [12]:

```
# plot the output error over training
plt.plot(errors)
plt.show()
```



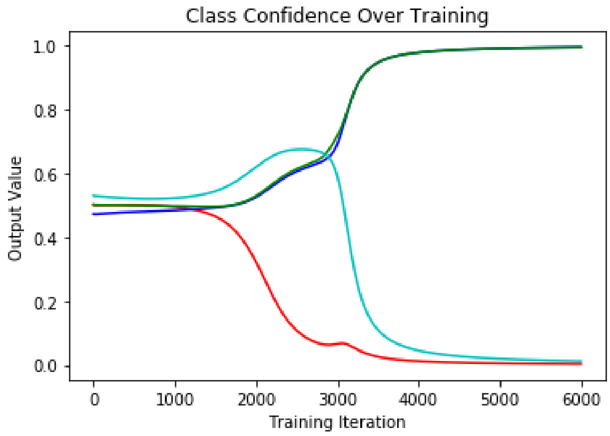
In [13]:

```python
# print the weights
print(w1)   # print the first weight matrix
print(w2)   # print the second weight matrix
```

```
[[ 7.552344    2.09695354 -0.3679197  -0.61184716  1.29167885]
 [-5.1723043  -5.92943069 -3.0493228  -2.88834858 -4.60314343]
 [-5.17627813 -5.94683665 -3.08289312 -2.92425765 -4.62220422]]
[[-5.03340061]
 [11.53972275]
 [-7.54658585]
 [-2.20951717]
 [-1.93103363]
 [-4.89156736]]
```

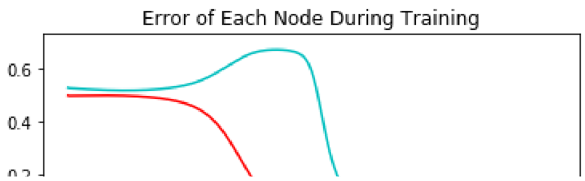
In [14]:

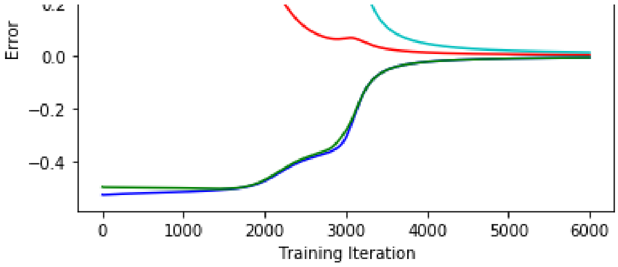
```python
# plot confidence of each class over training
fig = plt.figure()
subplot1 = fig.add_subplot(111)

subplot1.set_title('Class Confidence Over Training')
subplot1.set_ylabel('Output Value')
subplot1.set_xlabel('Training Iteration')

subplot1.plot(outputs[:, 0], c='r') # output given [0, 0] --- red
subplot1.plot(outputs[:, 1], c='b') # output given [1, 0] --- blue
subplot1.plot(outputs[:, 2], c='g') # output given [0, 1] --- green
subplot1.plot(outputs[:, 3], c='c') # output given [1, 1] --- cyan
plt.show()
```



In [15]:

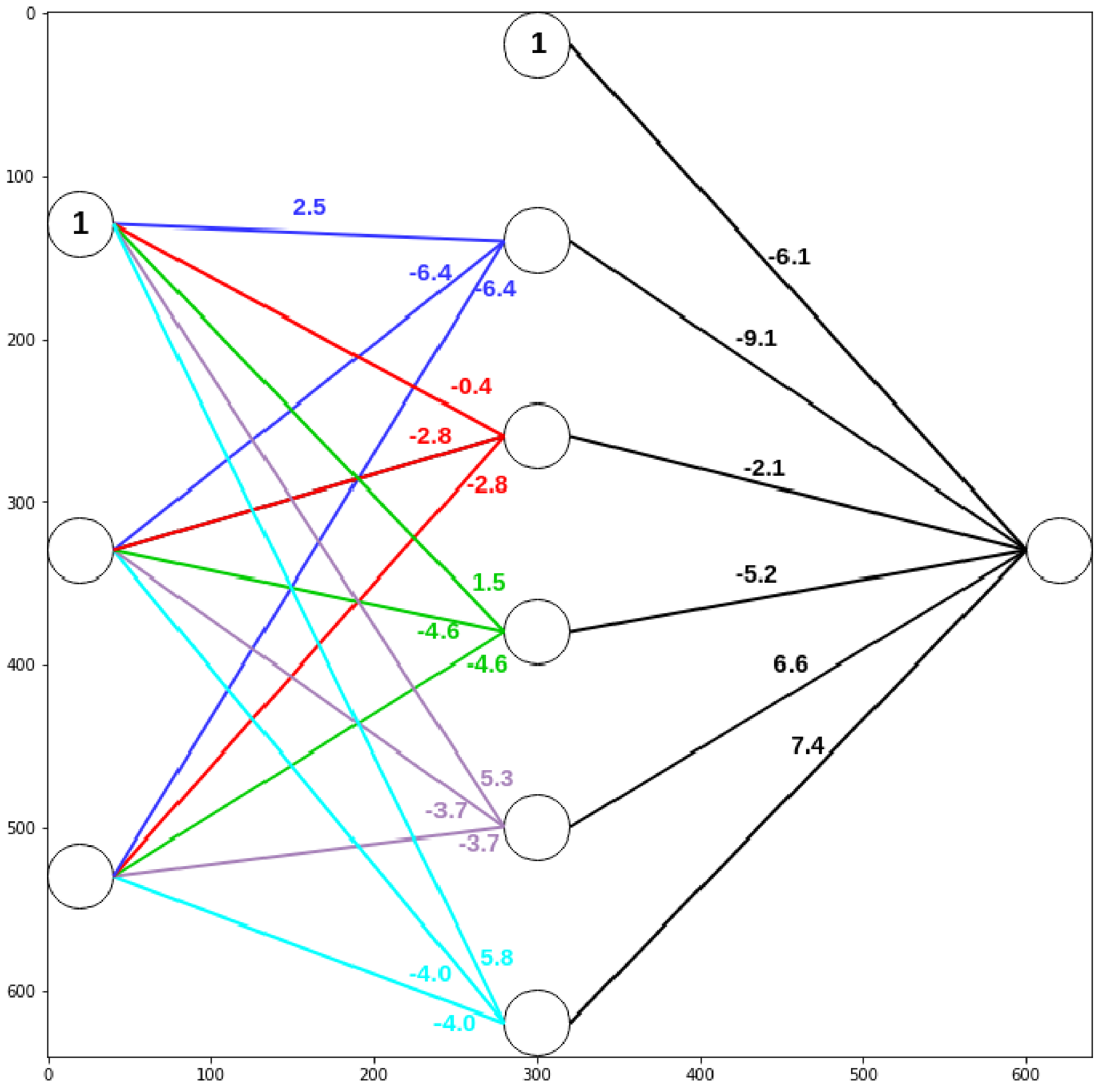
```python
# plot class errors
fig = plt.figure()
subplot1 = fig.add_subplot(111)

subplot1.set_xlabel('Training Iteration')
subplot1.set_ylabel('Error')
subplot1.set_title('Error of Each Node During Training')

subplot1.plot(class_errors[:, 0], c='r') # output given [0, 0] --- red
subplot1.plot(class_errors[:, 1], c='b') # output given [1, 0] --- blue
subplot1.plot(class_errors[:, 2], c='g') # output given [0, 1] --- green
subplot1.plot(class_errors[:, 3], c='c') # output given [1, 1] --- cyan
plt.show()
```





In [16]:

```
net_diagram = plt.imread('XOR_network.png') # read in the network figure

# show the figure
fig = plt.figure(figsize=(12, 12))
subplot1 = fig.add_subplot(111)
subplot1.imshow(net_diagram)
subplot1.grid(False)
plt.show()
```



IRIS Network

In [27]:

```
# read in the data as X and labels as Y
X = np.genfromtxt('iris_data.csv', delimiter=',')
Y = np.genfromtxt('iris_classes.csv', delimiter=',')
print(X.shape, Y.shape)  # print the shape of each
```

```
(150, 4) (150,)
```

In [28]:

```
# define useful variables
n_inputs = X.shape[1]
n_hidden = 100
n_out = 3
learning_rate = .1
n_epochs = 10
```

In [29]:

```
# initialize the weight matrices with small random numbers
w1 = randn(n_hidden, n_inputs + 1) * 0.01
w2 = randn(n_hidden, n_hidden + 1) * 0.01
w3 = randn(n_out, n_hidden + 1) * 0.01
print(w1.shape, w2.shape, w3.shape)
```

```
(100, 5) (100, 101) (3, 101)
```

In [30]:

```
# create a list for the error each iteration
errors = []

# create an empty array for error of each class
class_errors = np.zeros([n_epochs*X.shape[0], data.shape[1]])
```

In [31]:

```
# add bias column to data
X = np.concatenate((np.ones([X.shape[0], 1]), X), 1)
print(X.shape)
```

```
(150, 5)
```

In [32]:

```
# turn labels into one-hot vectors
y = np.zeros([Y.shape[0], 3])
y[range(Y.shape[0]), np.int32(Y - 1.)] = 1.
Y = y
print(Y)
```

```
[[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
```

```
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
```

```
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]]
```

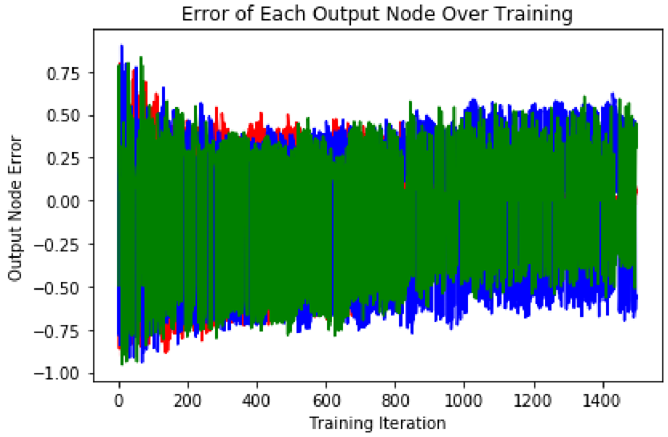
In [33]:

```python
for epoch in range(n_epochs):
    # shuffle dataset each epoch
    samples = np.random.permutation(X.shape[0])
    count = 0  # used for indexing later

    # go through each sample and do forward and backward pass
    for sample in samples:
        # forward pass
        x = X[sample, :]  # take one example from data
        x = x[:, None]  # put the first dimension back in for matrix ops
        y = Y[sample, :]  # take the corresponding label
        y = y[:, None] # put the first dimension back in for matrix ops

        h_linear = np.matmul(w1, x) # compute hidden layer 1 vals
        h_act = sigmoid(h_linear)  # activation function over hidden layer 1 nodes
        h_act = np.concatenate((np.ones([1, 1]), h_act), 0) # add bias to hidden 1 activations

        h2_linear = np.matmul(w2, h_act)  # hidden layer 2 vals
        h2_act = sigmoid(h2_linear)  # hidden layer 2 activations
        h2_act = np.concatenate((np.ones([1, 1]), h2_act), 0) # hidden layer 2 biases

        y_hat = sigmoid(np.matmul(w3, h2_act)) # compute output

        # backward pass
        d4 = y_hat - y # output error
        class_errors[epoch * X.shape[0] + count, :] = d4[:, 0]  # add error of each node for
plotting
```

```
        errors.append(np.mean(np.absolute(d4)))   # add the mean error for plotting later

        # compute errors of hidden layer 2
        d3 = np.matmul(w3[:, 1:].T, d4) * sigmoid_der(h2_linear)

        # compute errors of hidden layer 1
        d2 = np.matmul(w2[:, 1:].T, d3) * sigmoid_der(h_linear)

        dw3 = np.matmul(d4, h2_act.T) # gradient w3
        dw2 = np.matmul(d3, h_act.T)  # gradient w2
        dw1 = np.matmul(d2, x.T)  # gradient w1

        w1 -= (learning_rate * dw1)   # weight update for w1
        w2 -= (learning_rate * dw2)   # weight update for w2
        w3 -= (learning_rate * dw3) # weight update for w3

        count += 1  # increase count by 1
```

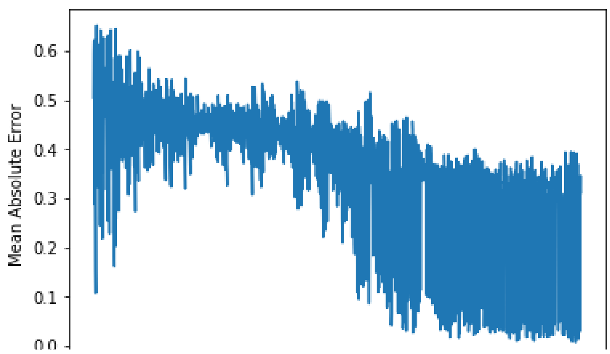
In [34]:

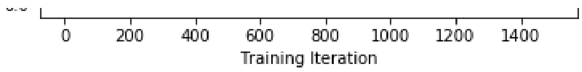
```
fig = plt.figure()
subplot1 = fig.add_subplot(111)

subplot1.set_title('Error of Each Output Node Over Training')
subplot1.set_xlabel('Training Iteration')
subplot1.set_ylabel('Output Node Error')

subplot1.plot(class_errors[:, 0], c='r') # class 1 is red
subplot1.plot(class_errors[:, 1], c='b') # class 2 is blue
subplot1.plot(class_errors[:, 2], c='g') # class 3 is green

plt.tight_layout()
plt.show()
```



In [35]:

```
fig = plt.figure()
subplot1 = fig.add_subplot(111)
subplot1.set_ylabel('Mean Absolute Error')
subplot1.set_xlabel('Training Iteration')

subplot1.plot(errors)
plt.show()
```





In [38]:

```
# calculate the training accuracy
h_linear = np.matmul(w1, X.T)
h_act = sigmoid(h_linear)
h_act = np.concatenate((np.ones([1, 150]), h_act), 0)

h2_linear = np.matmul(w2, h_act)
h2_act = sigmoid(h2_linear)
h2_act = np.concatenate((np.ones([1, 150]), h2_act), 0)

y_hat = sigmoid(np.matmul(w3, h2_act))
print(y_hat.shape)

# mean hamming distance between outputs and labels
acc = (np.mean(np.argmax(y_hat, 0) == np.argmax(Y, 1)))
print('The training accuracy is {}'.format(np.round(acc, 2)))
```

```
(3, 150)
The training accuracy is 0.95
```

In [ ]:

```

```