In [22]:

```
# Run this to ensure that you have the relevant packages:
import types
def imports():
    for name, val in globals().items():
        if isinstance(val, types.ModuleType):
            yield val.__name__
list(imports())
```

Out[22]:

```
['builtins', 'builtins', 'numpy', 'pandas', 'matplotlib.pyplot', 'type
s']
```

In [1]:

```
import numpy as np
import pandas as pd
# I'll use pandas to manipulate data, but not to write the algo...

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"


# loading the reacy, binarized data:
Xtraining = pd.read_csv('/Users/a8407352/Desktop/deepLearn/datasets/titanic/tra
Xtraining = Xtraining.drop(Xtraining.columns[0],axis=1)
```

In [2]:

```
age_avg=Xtraining['0'].mean()
age_std=Xtraining['0'].std()
age_null_count =Xtraining['0'].isnull().sum()
age_null_random_list = np.random.randint(age_avg - age_std, age_avg + age_std, s
Xtraining['0'][np.isnan(Xtraining['0'])] = age_null_random_list
Xtraining['0'] = Xtraining['0'].astype(int)

Xtraining['CategoricalAge'] = pd.cut(Xtraining['0'], 5)

print (Xtraining[['CategoricalAge', '16']].groupby(['CategoricalAge'], as_index=

Xtraining.loc[ Xtraining['0'] <= 16, '0']                            = 0
Xtraining.loc[(Xtraining['0'] > 16) & (Xtraining['0'] <= 32), '0'] = 1
Xtraining.loc[(Xtraining['0'] > 32) & (Xtraining['0'] <= 48), '0'] = 2
Xtraining.loc[(Xtraining['0'] > 48) & (Xtraining['0'] <= 64), '0'] = 3
Xtraining.loc[ Xtraining['0'] > 64, '0']                             = 4
```

```
  CategoricalAge        16
0  (-0.08, 16.0]  0.201923
1   (16.0, 32.0]  0.229381
2   (32.0, 48.0]  0.172414
3   (48.0, 64.0]  0.246377
4   (64.0, 80.0]  0.181818
```

In [3]:

```
# Xtraining=Xtraining.drop(Xtraining['CategoricalAge'], inplace=True)
Xtraining = Xtraining.iloc[:,0:25]
Xtraining.head(10)
```

Out[3]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| **1** | 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| **2** | 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **3** | 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| **4** | 2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| **5** | 2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| **6** | 3 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| **7** | 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **8** | 1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| **9** | 0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | ... | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |

10 rows × 25 columns

In [4]:

```
yTrain = pd.read_csv('/Users/a8407352/Desktop/deepLearn/datasets/titanic/trainLa
yTrain = yTrain.drop(yTrain.columns[0],axis=1)
yTrain.head()
```

Out[4]:

| | Survived |
|---|---|
| **0** | 0 |
| **1** | 1 |
| **2** | 1 |
| **3** | 1 |
| **4** | 0 |

In [5]:

```
def sigmoid(x, deriv=False):
    if deriv:
        return x*(1-x)
    return 1/(1+np.exp(-x))


weights1 = np.random.random((Xtraining.shape[1],1)) #bias can be added here, li
weights2 = np.random.random((1,1))  #L1.shape[0]
```

In [6]:

```
errors = []
index = []
for batch in range(10000):
    L0 = Xtraining
    L1 = sigmoid(Xtraining.dot(weights1)) #891X1
    L2 = sigmoid(L1.dot(weights2))
#     more layers here...possibly.
#         error:
    er = yTrain.values-L2        # 891X1
    index.append(batch)
    errors.append(np.mean(np.abs(er))) #since we take the mean of a batch of er
    #within the results, or smoothen results.
    if batch%100==0:
        print ('error:' +str(np.mean(np.abs(er))))



#         learning / backprop:
#     propogate:
#      regular vector multuplication to find the delta:
    delta1 = er*sigmoid(L2, deriv=True) # 2X1
    er2 =  delta1.dot(weights2.T)  # 2X2
    delta2 = er2 * sigmoid(L1, deriv=True) # 2X2
#     update weights with gradient descent:
    weights2 += L2.T.dot(delta1)
    weights1 += L1.T.dot(delta2)

```

```
error:0    0.503945
dtype: float64
error:0    0.503955
dtype: float64

error:0    0.503965
dtype: float64
error:0    0.503974
dtype: float64
error:0    0.503983
dtype: float64
error:0    0.503992
dtype: float64
error:0    0.504001
dtype: float64
error:0    0.50401
dtype: float64
error:0    0.504018
dtype: float64
error:0    0.504026
```

In [7]:

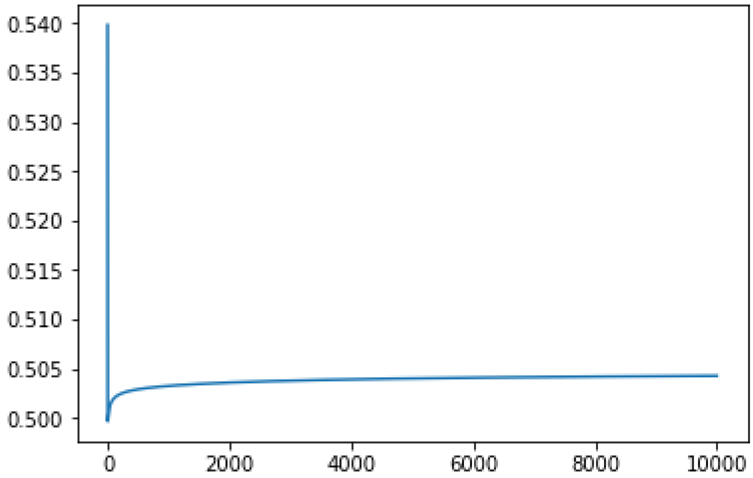
```
# plotting:
import matplotlib.pyplot as plt
# print(errors)
```

In [8]:

```
plt.plot(index,errors)
```

Out[8]:

```
[<matplotlib.lines.Line2D at 0x116c0f2b0>]
```




In [9]:

```
weights1 = np.random.random((Xtraining.shape[1],1)) #bias can be added here, li
weights2 = np.random.random((1,1))  #L1.shape[0]
```

In [10]:

```
def Relu(x, deriv=False):
    if deriv:
        x[x<=0] = 0
        x[x>0] = 1
        return x #notice that the derivative is not defined for x=0, but we'll 
    else:
        return np.maximum(0,x)
```

In [11]:

```
errors = []
index = []
for batchNum in range(100):
    L0 = Xtraining
    dotProduct = Xtraining.dot(weights1)
    L1 = Relu(dotProduct) #891X1
    L2 = Relu(L1.dot(weights2))

# obviously, we see by now that ReLu is not adequate for the task,
# i.e. for binary classification we need to squeeze  the output into a 0 to 1 
# or in other words, probability.
```

In [12]:

```
# Let's change the loss function - perhaps a log loss would be better.
def cross_entropy(predictions, targets):
    N = predictions.shape[0]
    ce = -np.sum(targets*np.log(predictions))/N
    return ce
```

In [13]:

```
errors = []
index = []
for batch in range(10000):
    L0 = Xtraining
    L1 = sigmoid(Xtraining.dot(weights1)) #891X1
    L2 = sigmoid(L1.dot(weights2))
#     more layers here...possibly.
#         error:
    er = cross_entropy(L2,yTrain.values)        # 891X1
    index.append(batch)
    errors.append(np.mean(np.abs(er)))
    if batch%100==0:
        print ('error:' +str(np.mean(np.abs(er))))



#         learning / backprop:
#     propogate:
#      regular vector multuplication to find the delta:
    delta1 = er*sigmoid(L2, deriv=True) # 2X1
    er2 =  delta1.dot(weights2.T)  # 2X2
    delta2 = er2 * sigmoid(L1, deriv=True) # 2X2
#     update weights with gradient descent:
    weights2 += L2.T.dot(delta1)
    weights1 += L1.T.dot(delta2)

```

```
error:0.21551942851278608
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
```

```
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
```

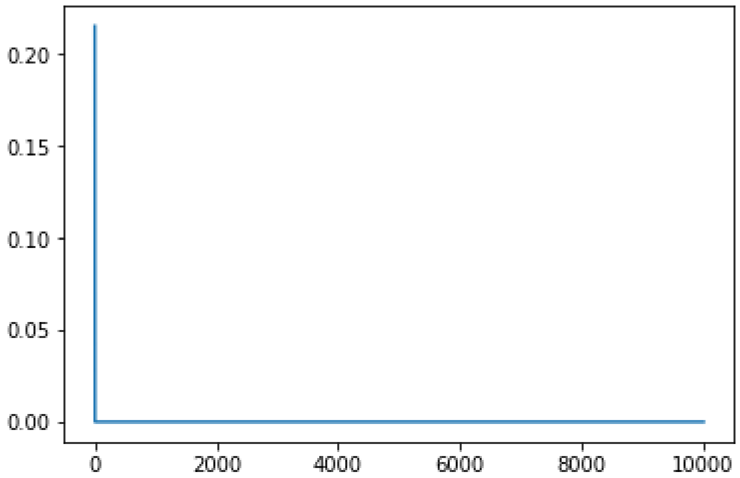
```
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
error:7.889626101711672e-13
```

In [14]:

```
plt.plot(index,errors)
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x114ac99b0>]
```




In [15]:

```
# Not much different (the error is calculated differently, but the learning rat
# not getting much better). I tend to think that this is more because of the ar
# which is quite arbitrary. The network takes the form of 891X891  two layers.
# I will plot a similar one with 5X5 teo layers:
```

In [16]:

```
import matplotlib.pyplot as plt

def draw_neural_net(ax, left, right, bottom, top, layer_sizes):
    '''
    Draw a neural network cartoon using matplotilb.

    :usage:
        >>> fig = plt.figure(figsize=(12, 12))
        >>> draw_neural_net(fig.gca(), .1, .9, .1, .9, [4, 7, 2])

    :parameters:
        - ax : matplotlib.axes.AxesSubplot
            The axes on which to plot the cartoon (get e.g. by plt.gca())
        - left : float
            The center of the leftmost node(s) will be placed here
        - right : float
            The center of the rightmost node(s) will be placed here
        - bottom : float
            The center of the bottommost node(s) will be placed here
        - top : float
            The center of the topmost node(s) will be placed here
        - layer_sizes : list of int
            List of layer sizes, including input and output dimensionality
    '''
    n_layers = len(layer_sizes)
    v_spacing = (top - bottom)/float(max(layer_sizes))
    h_spacing = (right - left)/float(len(layer_sizes) - 1)
    # Nodes
    for n, layer_size in enumerate(layer_sizes):
        layer_top = v_spacing*(layer_size - 1)/2. + (top + bottom)/2.
        for m in range(layer_size):
            circle = plt.Circle((n*h_spacing + left, layer_top - m*v_spacing), v
                                color='w', ec='k', zorder=4)
            ax.add_artist(circle)
    # Edges
    for n, (layer_size_a, layer_size_b) in enumerate(zip(layer_sizes[:-1], layer
        layer_top_a = v_spacing*(layer_size_a - 1)/2. + (top + bottom)/2.
        layer_top_b = v_spacing*(layer_size_b - 1)/2. + (top + bottom)/2.
        for m in range(layer_size_a):
            for o in range(layer_size_b):
                line = plt.Line2D([n*h_spacing + left, (n + 1)*h_spacing + left
                                  [layer_top_a - m*v_spacing, layer_top_b - o*v_
                ax.add_artist(line)

```

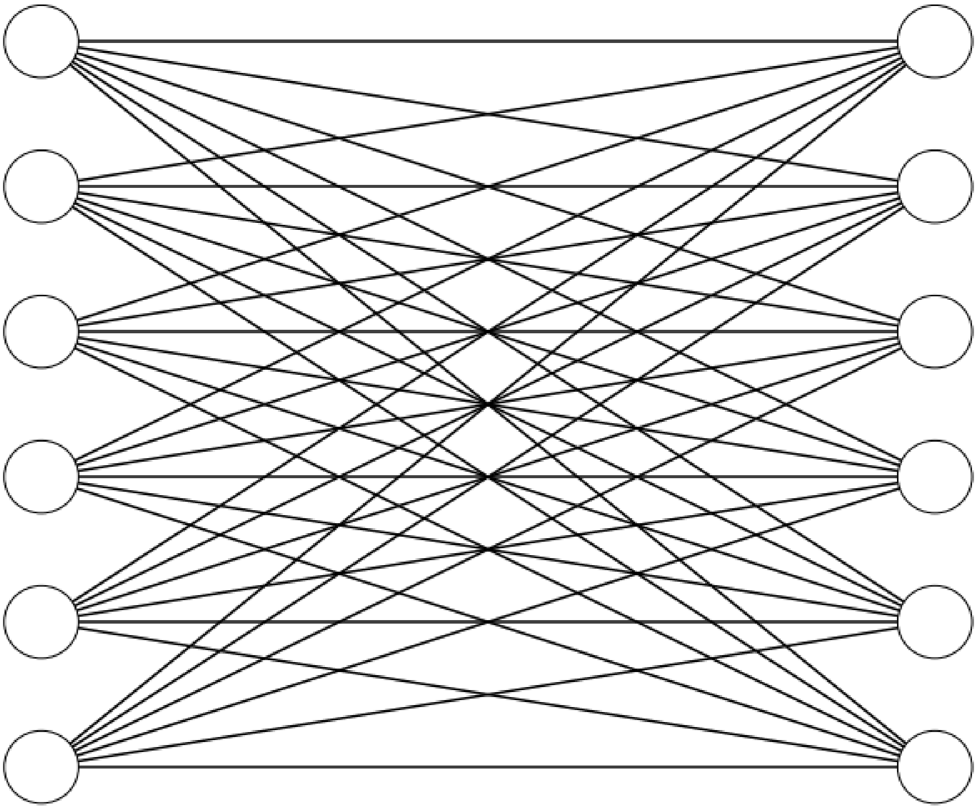
In [17]:

```
fig = plt.figure(figsize=(12, 12))
ax = fig.gca()
ax.axis('off')
draw_neural_net(ax, .1, .9, .1, .9, [6, 6])
fig.savefig('nn.png')

```

Out[17]:

```
(0.0, 1.0, 0.0, 1.0)
```



In [18]:

```
# As can be seen, this doesn't really make a whole lot sense and further experi
```

In [20]:

```
# References:
# I used numerous tutorials and stackoverflow samples whose code I adopted, put
# to have what I tried to achieve as a playground for the most simple ANNs.
```