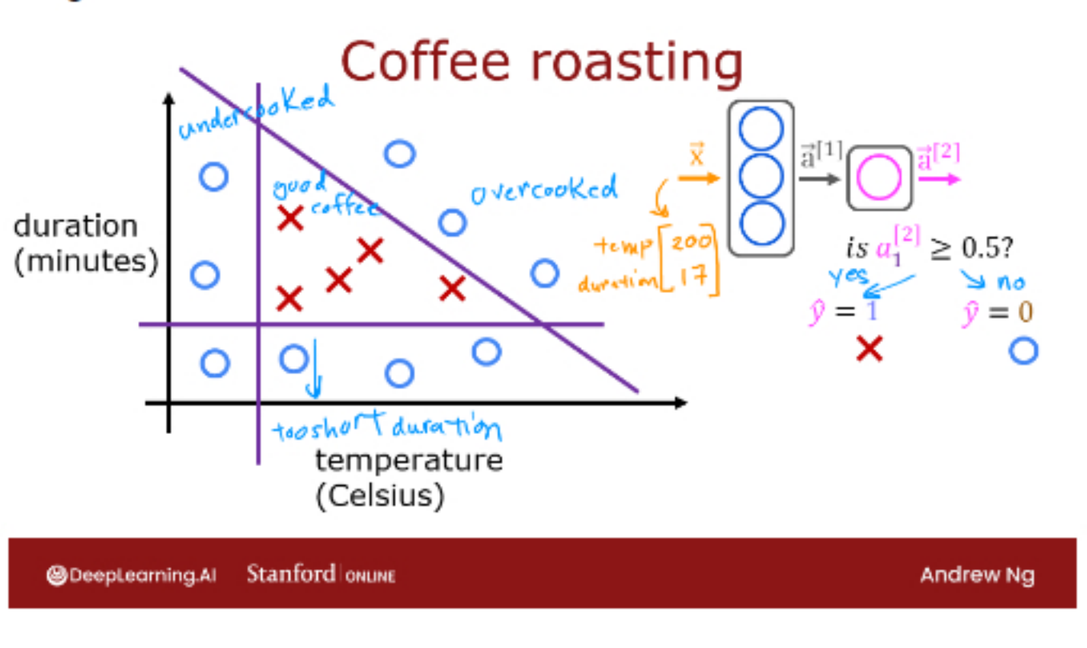
Optional Lab - Simple Neural Network

In this lab we will build a small neural network using Tensorflow.



In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('./deeplearning.mplstyle')
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from lab_utils_common import dlc
from lab_coffee_utils import load_coffee_data, plt_roast, plt_prob, plt_layer, plt_network, plt_output_unit
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
tf.autograph.set_verbosity(0)
```

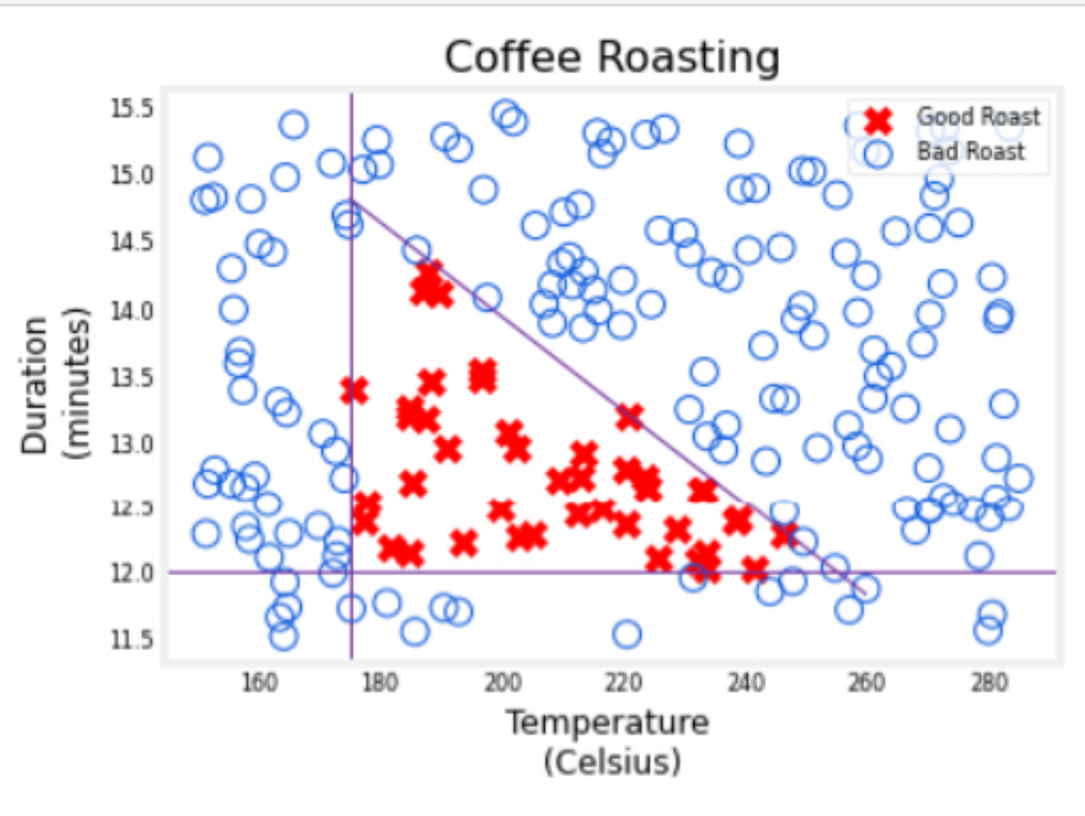
DataSet

In [2]:
```python
X,Y = load_coffee_data();
print(X.shape, Y.shape)
```

```
(200, 2) (200, 1)
```

Let's plot the coffee roasting data below. The two features are Temperature in Celsius and Duration in minutes. Coffee Roasting at Home suggests that the duration is best kept between 12 and 15 minutes while the temp should be between 175 and 260 degrees Celsius. Of course, as temperature rises, the duration should shrink.

In [3]:
```python
plt_roast(X,Y)
```



Normalize Data

Fitting the weights to the data (back-propagation, covered in next week's lectures) will proceed more quickly if the data is normalized. This is the same procedure you used in Course 1 where features in the data are each normalized to have a similar range. The procedure below uses a Keras normalization layer. It has the following steps:

- create a "Normalization Layer". Note, as applied here, this is not a layer in your model.
- 'adapt' the data. This learns the mean and variance of the data set and saves the values internally.
- normalize the data.
  It is important to apply normalization to any future data that utilizes the learned model.

In [4]:
```python
print(f"Temperature Max, Min pre normalization: {np.max(X[:,0]):0.2f}, {np.min(X[:,0]):0.2f}")
print(f"Duration    Max, Min pre normalization: {np.max(X[:,1]):0.2f}, {np.min(X[:,1]):0.2f}")
norm_l = tf.keras.layers.Normalization(axis=-1)
norm_l.adapt(X)  # learns mean, variance
Xn = norm_l(X)
print(f"Temperature Max, Min post normalization: {np.max(Xn[:,0]):0.2f}, {np.min(Xn[:,0]):0.2f}")
print(f"Duration    Max, Min post normalization: {np.max(Xn[:,1]):0.2f}, {np.min(Xn[:,1]):0.2f}")
```

```
Temperature Max, Min pre normalization: 284.99, 151.32
Duration    Max, Min pre normalization: 15.45, 11.51
Temperature Max, Min post normalization: 1.66, -1.69
Duration    Max, Min post normalization: 1.79, -1.70
```

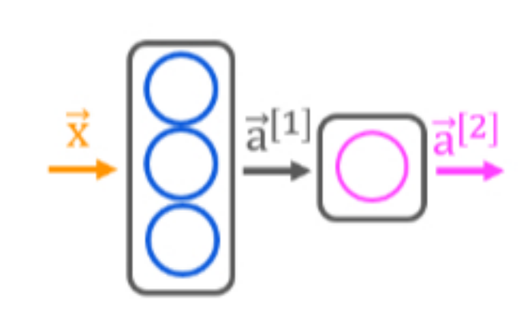
Tile/copy our data to increase the training set size and reduce the number of training epochs.

In [5]:
```python
Xt = np.tile(Xn,(1000,1))
Yt= np.tile(Y,(1000,1))
print(Xt.shape, Yt.shape)
```

```
(200000, 2) (200000, 1)
```

Tensorflow Model

Model



Let's build the "Coffee Roasting Network" described in lecture. There are two layers with sigmoid activations as shown below:

In [6]:
```python
tf.random.set_seed(1234)  # applied to achieve consistent results
model = Sequential(
    [
        tf.keras.Input(shape=(2,)),
        Dense(3, activation='sigmoid', name = 'layer1'),
        Dense(1, activation='sigmoid', name = 'layer2')
     ]
)
```

> **Note 1:** The `tf.keras.Input(shape=(2,)),` specifies the expected shape of the input. This allows Tensorflow to size the weights and bias parameters at this point. This is useful when exploring Tensorflow models. This statement can be omitted in practice and Tensorflow will size the network parameters when the input data is specified in the `model.fit` statement.
> **Note 2:** Including the sigmoid activation in the final layer is not considered best practice. It would instead be accounted for in the loss which improves numerical stability. This will be described in more detail in a later lab.

The `model.summary()` provides a description of the network:

In [7]:
```python
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 layer1 (Dense)              (None, 3)                 9

 layer2 (Dense)              (None, 1)                 4

=================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
_________________________________________________________________
```

The parameter counts shown in the summary correspond to the number of elements in the weight and bias arrays as shown below.

In [8]:
```python
L1_num_params = 2 * 3 + 3   # W1 parameters  + b1 parameters
L2_num_params = 3 * 1 + 1   # W2 parameters  + b2 parameters
print("L1 params = ", L1_num_params, ", L2 params = ", L2_num_params  )
```

```
L1 params =  9 , L2 params =  4
```

Let's examine the weights and biases Tensorflow has instantiated. The weights W should be of size (number of features in input, number of units in the layer) while the bias b size should match the number of units in the layer:

- In the first layer with 3 units, we expect W to have a size of (2,3) and b should have 3 elements.
- In the second layer with 1 unit, we expect W to have a size of (3,1) and b should have 1 element.

In [9]:
```python
W1, b1 = model.get_layer("layer1").get_weights()
W2, b2 = model.get_layer("layer2").get_weights()
print(f"W1{W1.shape}:\n", W1, f"\nb1{b1.shape}:", b1)
print(f"W2{W2.shape}:\n", W2, f"\nb2{b2.shape}:", b2)
```

```
W1(2, 3):
 [[ 0.08 -0.3   0.18]
 [-0.56 -0.15  0.89]]
b1(3,): [0. 0. 0.]
W2(3, 1):
 [[-0.43]
 [-0.88]
 [ 0.36]]
b2(1,): [0.]
```

The following statements will be described in detail in Week2. For now:

- The `model.compile` statement defines a loss function and specifies a compile optimization.
- The `model.fit` statement runs gradient descent and fits the weights to the data.

In [10]:
```python
model.compile(
    loss = tf.keras.losses.BinaryCrossentropy(),
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.01),
)

model.fit(
    Xt,Yt,
    epochs=10,
)
```

```
Epoch 2/10
6250/6250 [==============================] - 5s 780us/step - loss: 0.1165
Epoch 3/10
6250/6250 [==============================] - 5s 767us/step - loss: 0.0426
Epoch 4/10
6250/6250 [==============================] - 5s 801us/step - loss: 0.0160
Epoch 5/10
6250/6250 [==============================] - 5s 803us/step - loss: 0.0104
Epoch 6/10
6250/6250 [==============================] - 5s 779us/step - loss: 0.0073
Epoch 7/10
6250/6250 [==============================] - 5s 764us/step - loss: 0.0052
Epoch 8/10
6250/6250 [==============================] - 5s 830us/step - loss: 0.0037
Epoch 9/10
6250/6250 [==============================] - 5s 858us/step - loss: 0.0027
Epoch 10/10
6250/6250 [==============================] - 5s 791us/step - loss: 0.0020
```

Out[10]: `<keras.callbacks.History at 0x7fc45c4d63d0>`

Epochs and batches

In the `compile` statement above, the number of `epochs` was set to 10. This specifies that the entire data set should be applied during training 10 times. During training, you see output describing the progress of training that looks like this:

```
Epoch 1/10
6250/6250 [==============================] - 6s 910us/step - loss: 0.1782
```

The first line, `Epoch 1/10`, describes which epoch the model is currently running. For efficiency, the training data set is broken into 'batches'. The default size of a batch in Tensorflow is 32. There are 200000 examples in our expanded data set or 6250 batches. The notation on the 2nd line `6250/6250 [====` is describing which batch has been executed.

Updated Weights

After fitting, the weights have been updated:

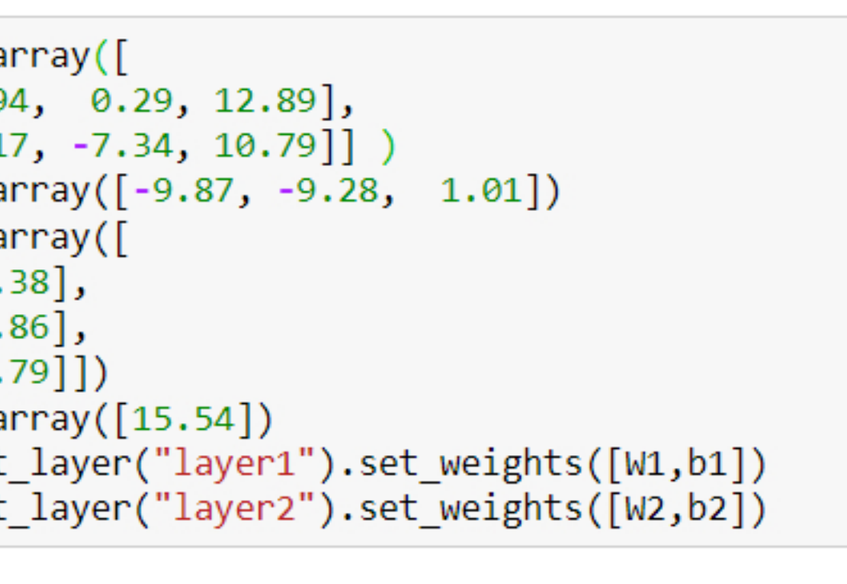
In [11]:
```python
W1, b1 = model.get_layer("layer1").get_weights()
W2, b2 = model.get_layer("layer2").get_weights()
print("W1:\n", W1, "\nb1:", b1)
print("W2:\n", W2, "\nb2:", b2)
```

```
W1:
 [[ -0.13  14.3  -11.1 ]
 [ -8.92  11.85  -0.25]]
b1: [-11.16   1.76 -12.1 ]
W2:
 [[-45.71]
 [-42.95]
 [-50.19]]
b2: [26.14]
```

Next, we will load some saved weights from a previous training run. This is so that this notebook remains robust to changes in Tensorflow over time. Different training runs can produce somewhat different results and the discussion below applies to a particular solution. Feel free to re-run the notebook with this cell commented out to see the difference.

In [12]:
```python
W1 = np.array([
    [-8.94,  0.29, 12.89],
    [-0.17, -7.34, 10.79]] )
b1 = np.array([-9.87, -9.28,  1.01])
W2 = np.array([
    [-31.38],
    [-27.86],
    [-32.79]])
b2 = np.array([15.54])
model.get_layer("layer1").set_weights([W1,b1])
model.get_layer("layer2").set_weights([W2,b2])
```

Predictions



Once you have a trained model, you can then use it to make predictions. Recall that the output of our model is a probability. In this case, the probability of a good roast. To make a decision, one must apply the probability to a threshold. In this case, we will use 0.5

Let's start by creating input data. The model is expecting one or more examples where examples are in the rows of matrix. In this case, we have two features so the matrix will be (m,2) where m is the number of examples. Recall, we have normalized the input features so we must normalize our test data as well. To make a prediction, you apply the `predict` method.

In [13]:
```python
X_test = np.array([
    [200,13.9],  # postive example
    [200,17]])   # negative example
X_testn = norm_l(X_test)
predictions = model.predict(X_testn)
print("predictions = \n", predictions)
```

```
predictions =
 [[9.63e-01]
 [3.03e-08]]
```

To convert the probabilities to a decision, we apply a threshold:

In [14]:
```python
yhat = np.zeros_like(predictions)
for i in range(len(predictions)):
    if predictions[i] >= 0.5:
        yhat[i] = 1
    else:
        yhat[i] = 0
print(f"decisions = \n{yhat}")
```

```
decisions =
[[1.]
 [0.]]
```

This can be accomplished more succinctly:

In [15]:
```python
yhat = (predictions >= 0.5).astype(int)
print(f"decisions = \n{yhat}")
```

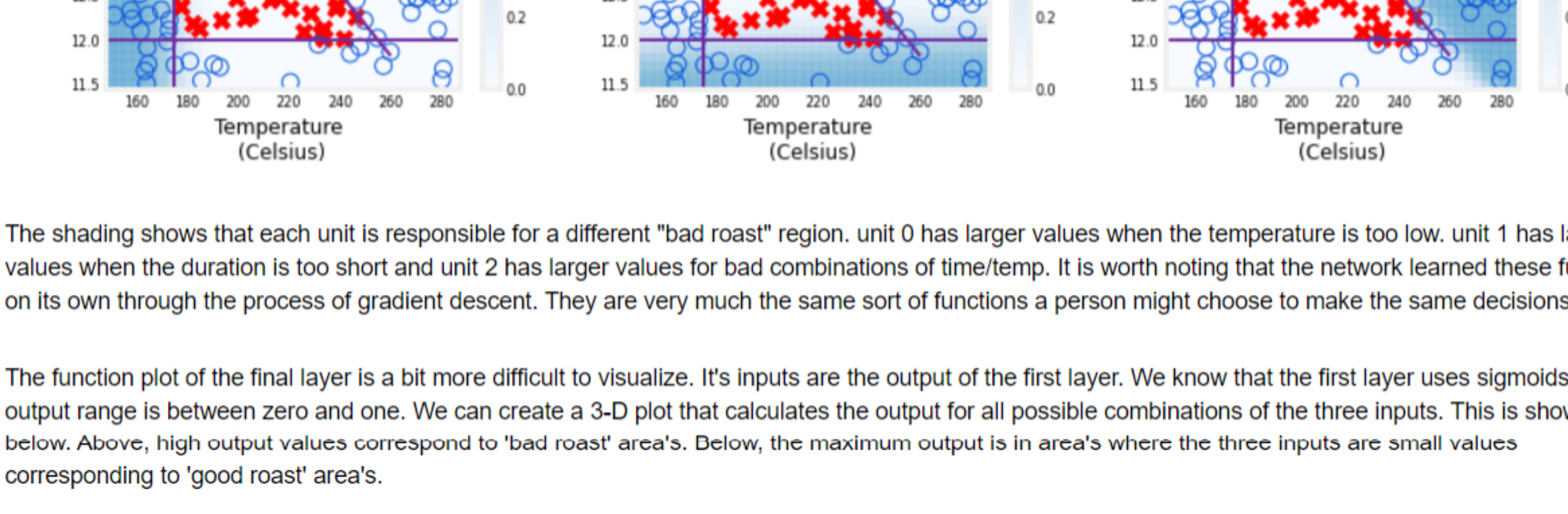
```
decisions =
[[1]
 [0]]
```

Layer Functions

Let's examine the functions of the units to determine their role in the coffee roasting decision. We will plot the output of each node for all values of the inputs (duration,temp). Each unit is a logistic function whose output can range from zero to one. The shading in the graph represents the output value.

> Note: In labs we typically number things starting at zero while the lectures may start with 1.

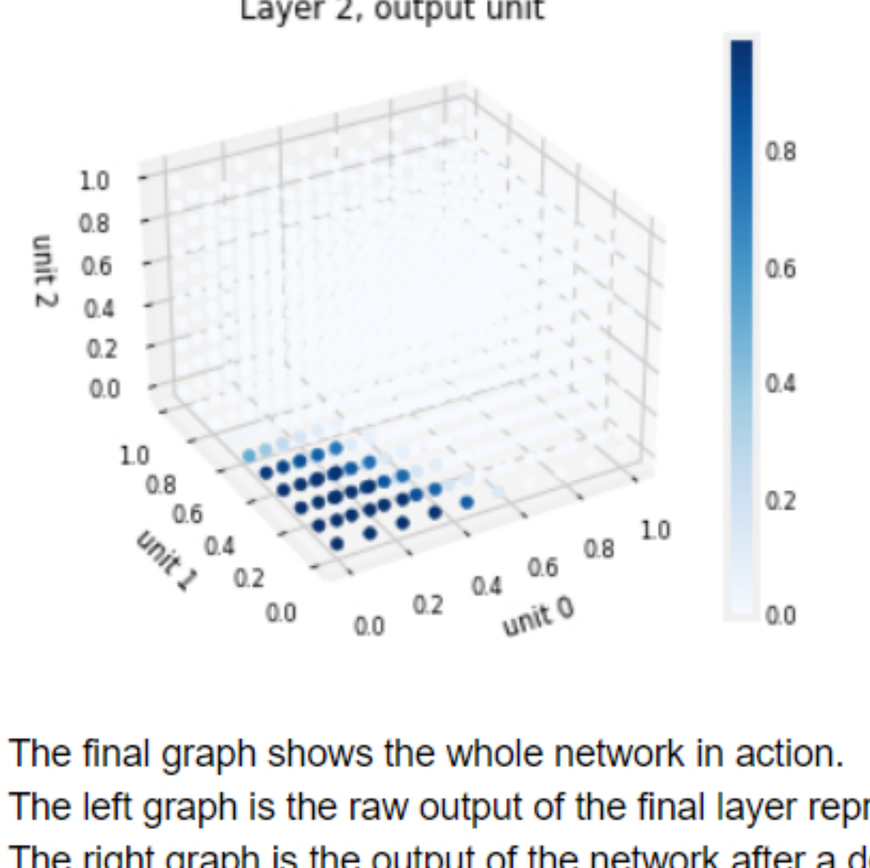
In [16]:
```python
plt_layer(X,Y.reshape(-1,),W1,b1,norm_l)
```



The shading shows that each unit is responsible for a different "bad roast" region. unit 0 has larger values when the temperature is too low. unit 1 has larger values when the duration is too short and unit 2 has larger values for bad combinations of time/temp. It is worth noting that the network learned these functions on its own through the process of gradient descent. They are very much the same sort of functions a person might choose to make the same decisions.

The function plot of the final layer is a bit more difficult to visualize. It's inputs are the output of the first layer. We know that the first layer uses sigmoids so their output range is between zero and one. We can create a 3-D plot that calculates the output for all possible combinations of the three inputs. This is shown below. Above, high output values correspond to 'bad roast' area's. Below, the maximum output is in area's where the three inputs are small values corresponding to 'good roast' area's.

In [17]:
```python
plt_output_unit(W2,b2)
```



The final graph shows the whole network in action.
The left graph is the raw output of the final layer represented by the blue shading. This is overlaid on the training data represented by the X's and O's.
The right graph is the output of the network after a decision threshold. The X's and O's here correspond to decisions made by the network.
The following takes a moment to run

In [*]:
```python
netf= lambda x : model.predict(norm_l(x))
plt_network(X,Y,netf)
```

Congratulations!

You have built a small neural network in Tensorflow. The network demonstrated the ability of neural networks to handle complex decisions by dividing the decisions between multiple units.

In [ ]: