

# **An introduction to the back-propagation algorithm**

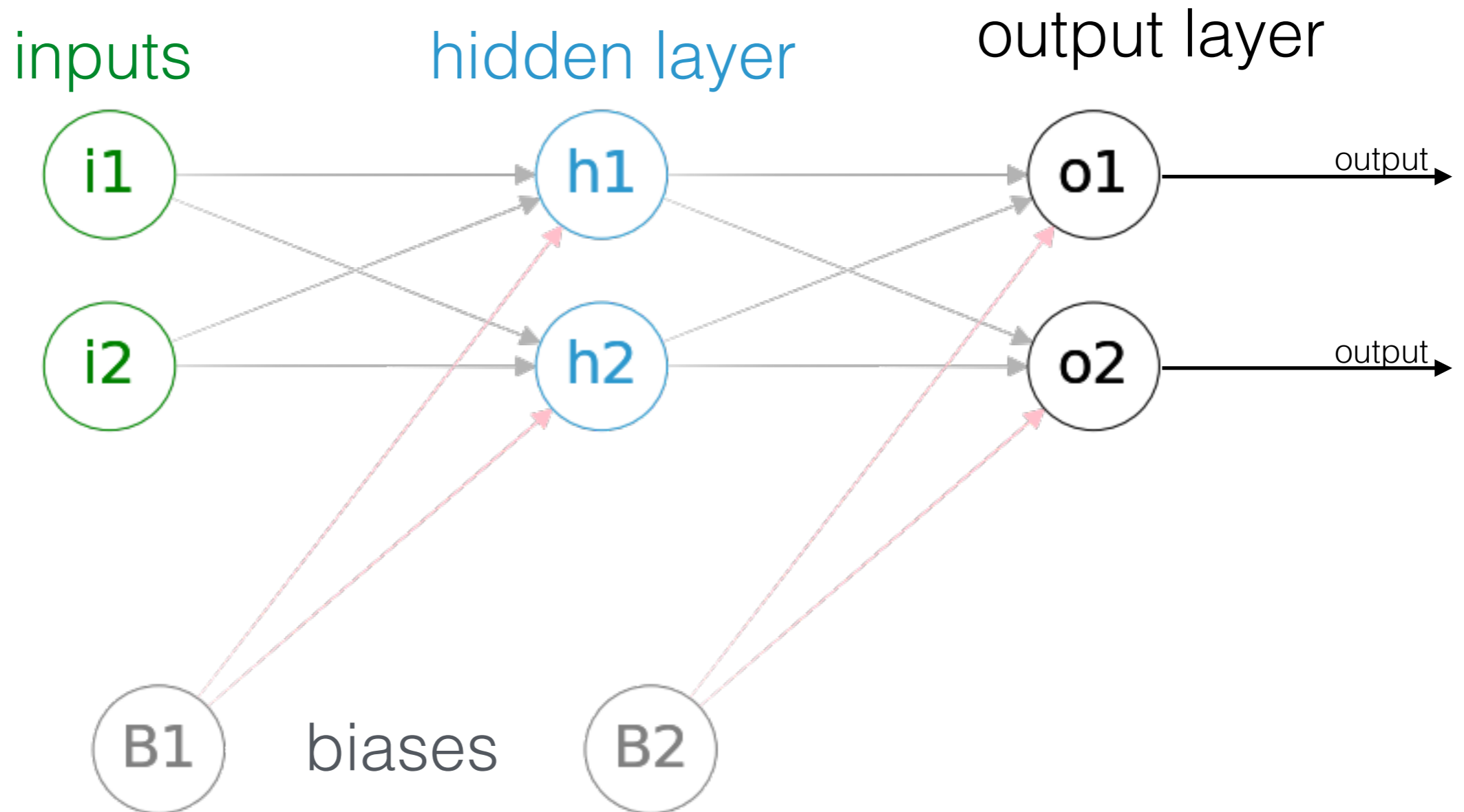
by Dominic Waithe

# Why neural networks

- Conventional algorithm: a computer follows a set of instructions in order to solve a problem. Fine if you know what to do.....
- A neural network learns to solve a problem by example.
  - Provides a mapping from one space to another.
  - The input space could be images, text, genome sequence, sound.
  - The output is often a classification (dog, cats, guinea pigs).
- In many challenging examples a neural network can learn how to recognise and classify things better than a custom designed conventional algorithm.

# A basic **Feedforward** neural network

- Two input nodes (2D data), one hidden layer (with 2 nodes) and two output nodes (= 2 classes).



# A basic **Feedforward** neural network

- A network transforms the inputs to the outputs, which in this case are both numbers.

input  $i1 = 0.8, i2 = 0.5$  outputs  $i1 = 0.0, i2 = 1.0$

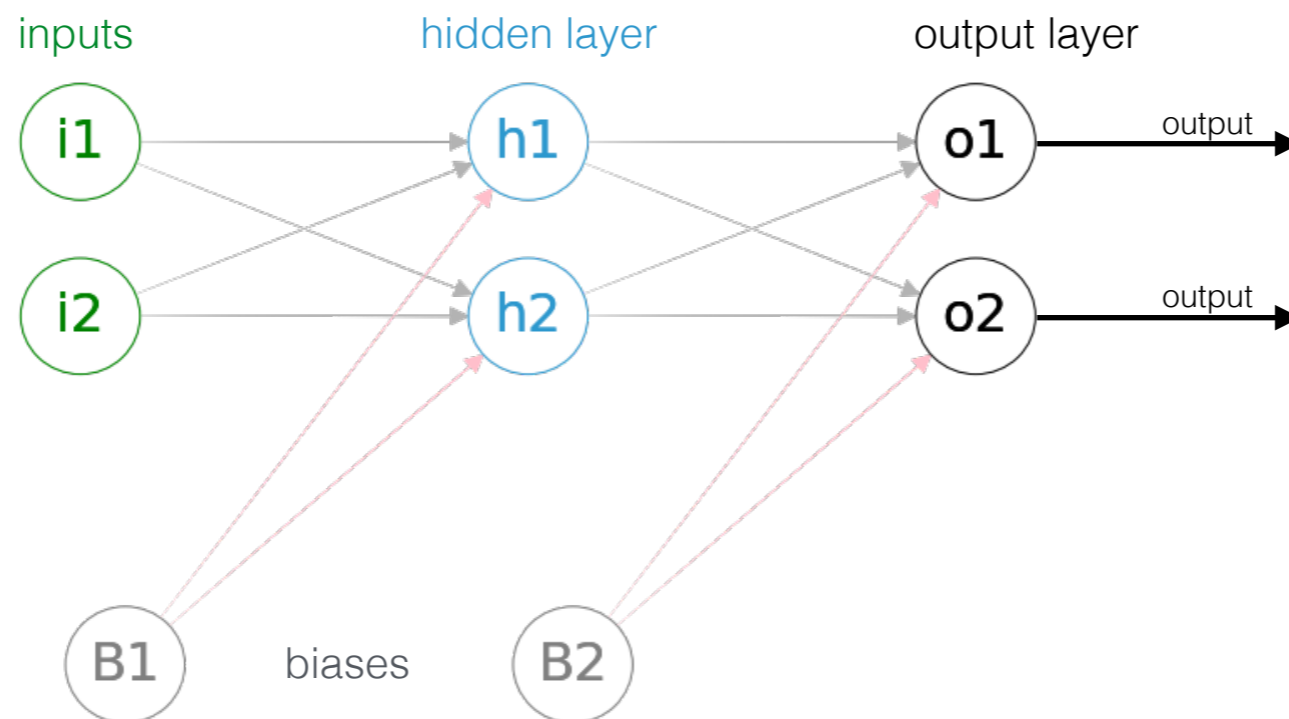
input  $i1 = 0.5, i2 = 0.2$  outputs  $i1 = 1.0, i2 = 0.0$

input  $i1 = 0.5, i2 = 0.9$  outputs  $i1 = 1.0, i2 = 0.0$

input  $i1 = 0.2, i2 = 0.5$  outputs  $i1 = 1.0, i2 = 0.0$

etc  
etc

input  $i1 = 0.2, i2 = 0.5$  outputs  $i1 = 1.0, i2 = 0.0$   
input  $i1 = 0.2, i2 = 0.5$  outputs  $i1 = 1.0, i2 = 0.0$   
input  $i1 = 0.2, i2 = 0.5$  outputs  $i1 = 1.0, i2 = 0.0$



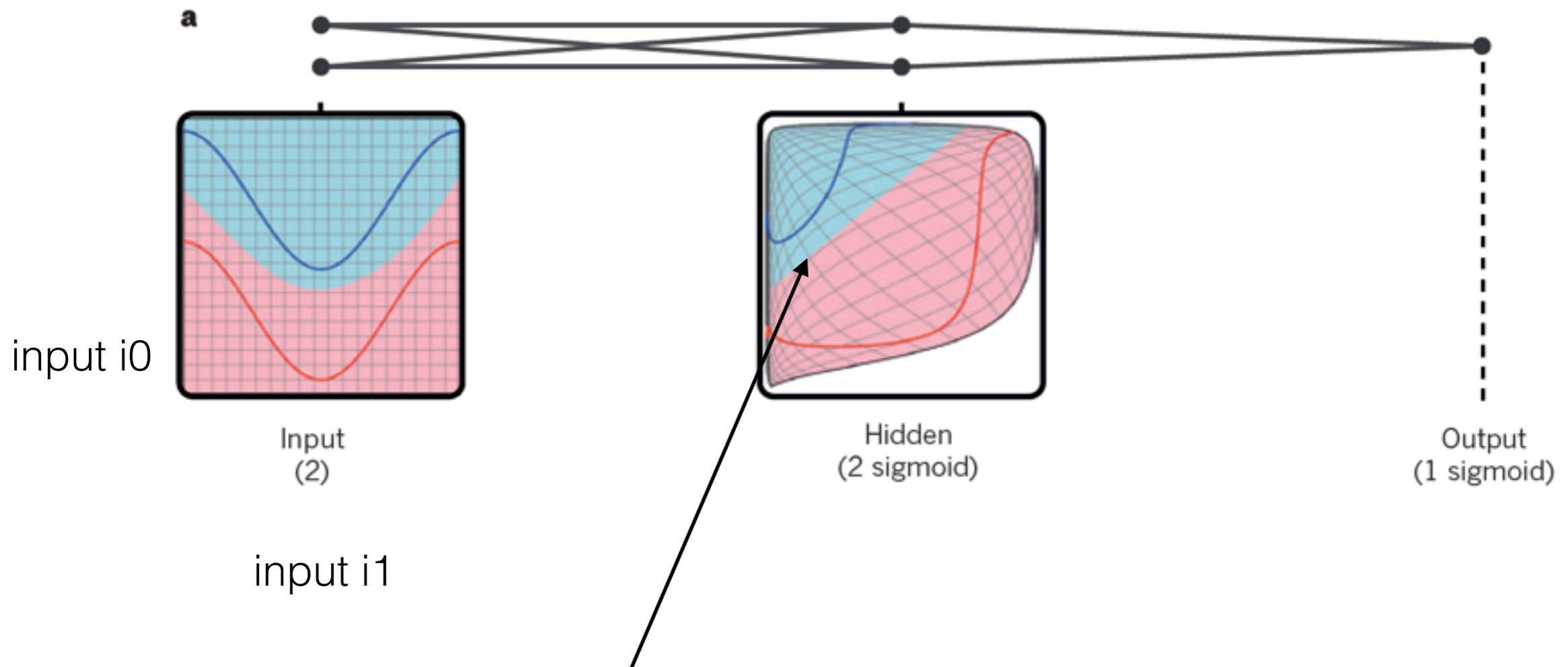
# How do we train the network

- How do we make the inputs generate the outputs we want?
- Answer: By transforming the data through a series of non-linear transformations at least in the case of neural networks.
- What does that look like?

# A basic **Feedforward** neural network

input training data A  
input training data B

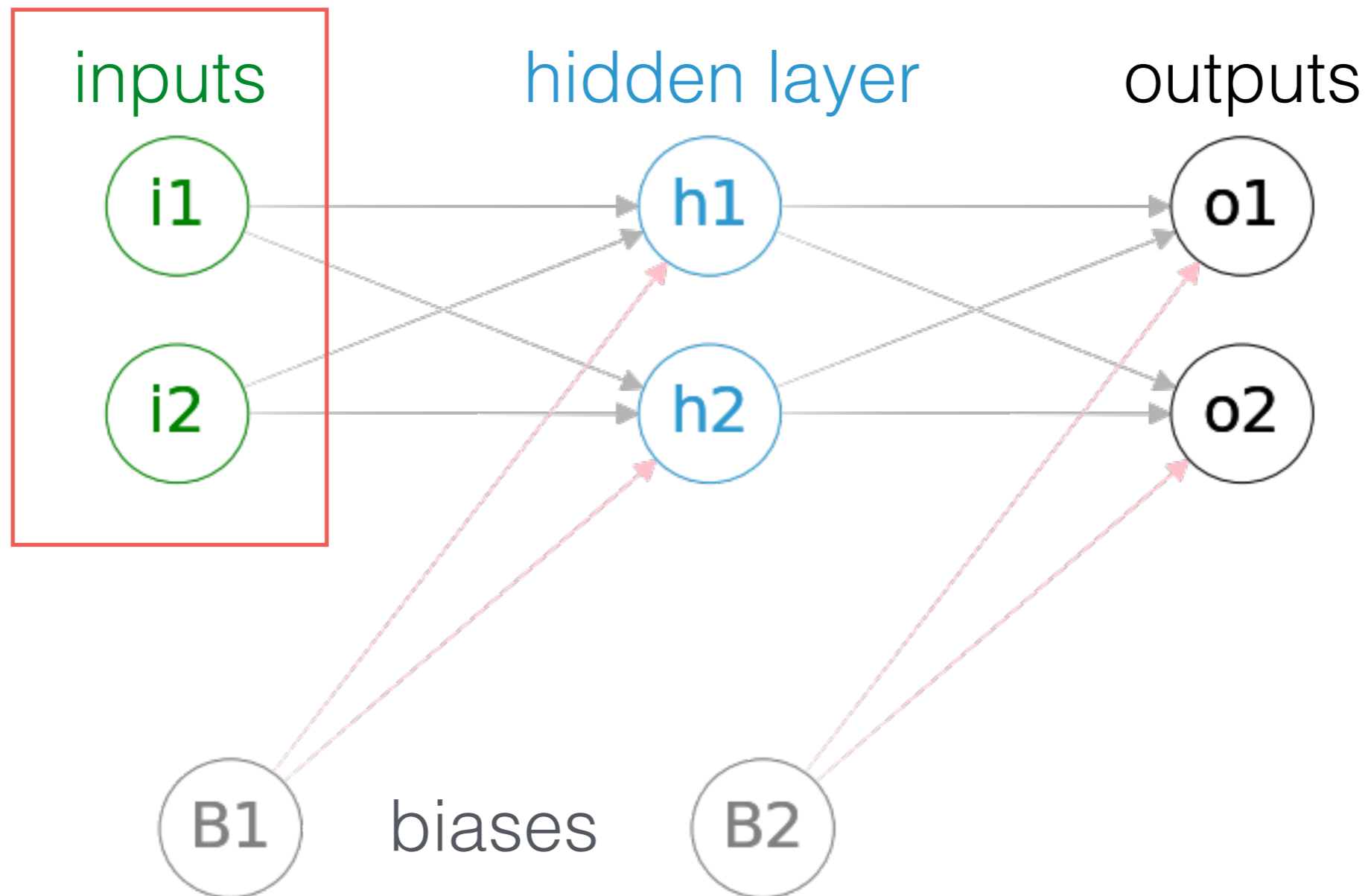
classification area for A  
classification area for B



Through our non-linear transformation we are able to bisect the data with a straight-line

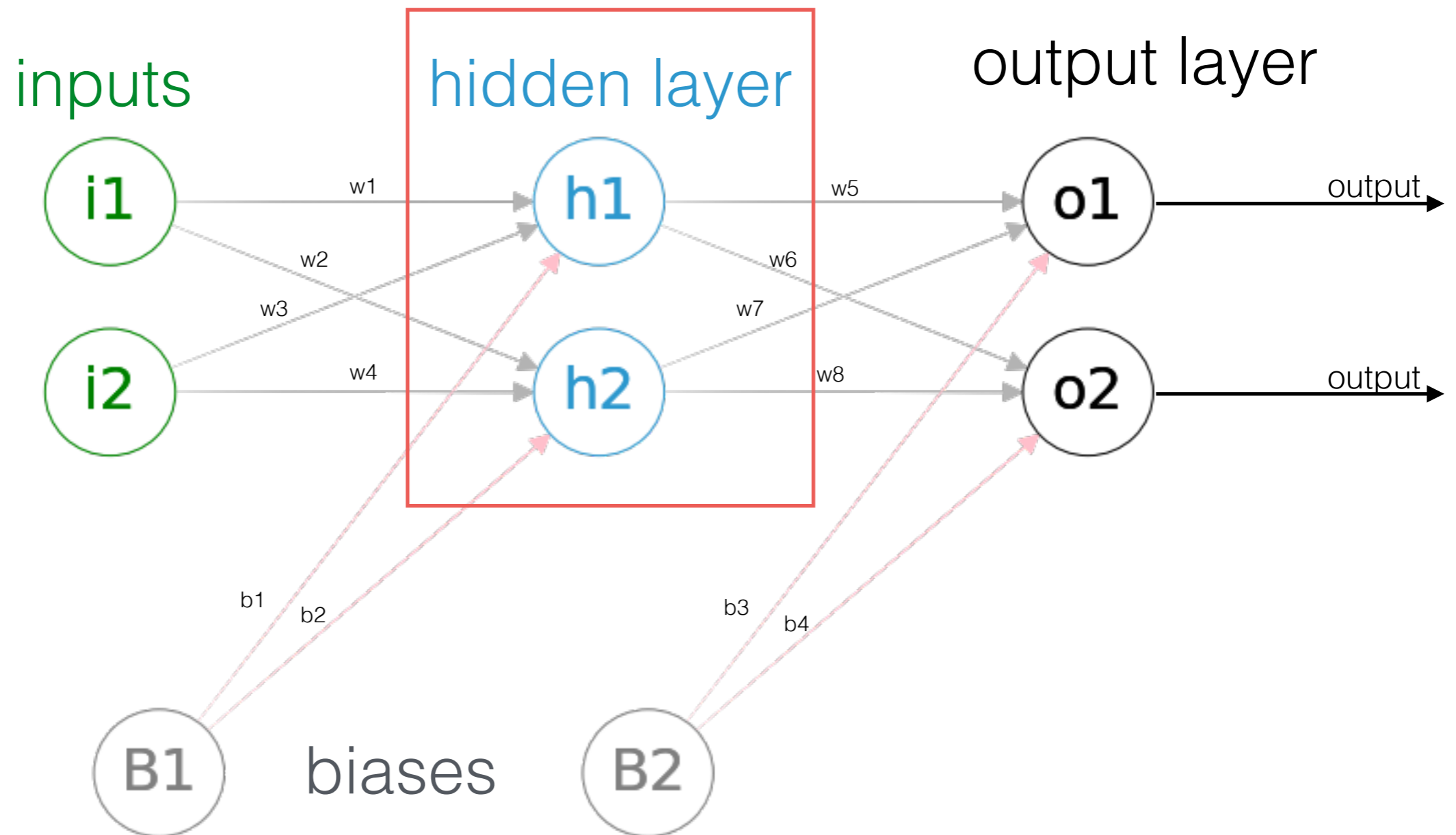
# A basic **Feedforward** neural network

- Two inputs (2D data), one hidden layer (with 2 nodes) and two outputs (= 2 classes).



# A basic **Feedforward** neural network

- Hidden layers are layers which are not inputs or outputs.

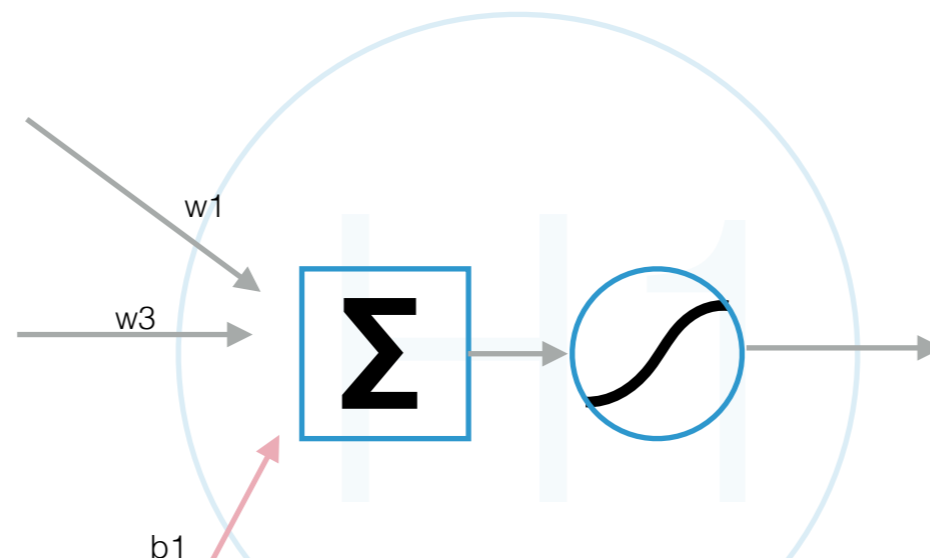
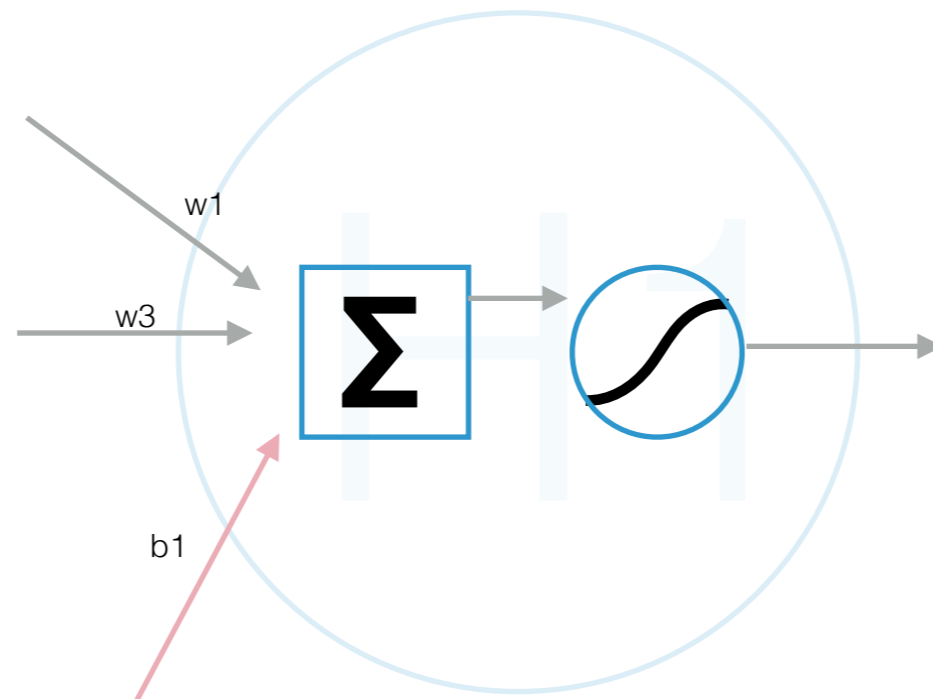




# A basic **Feedforward** neural network

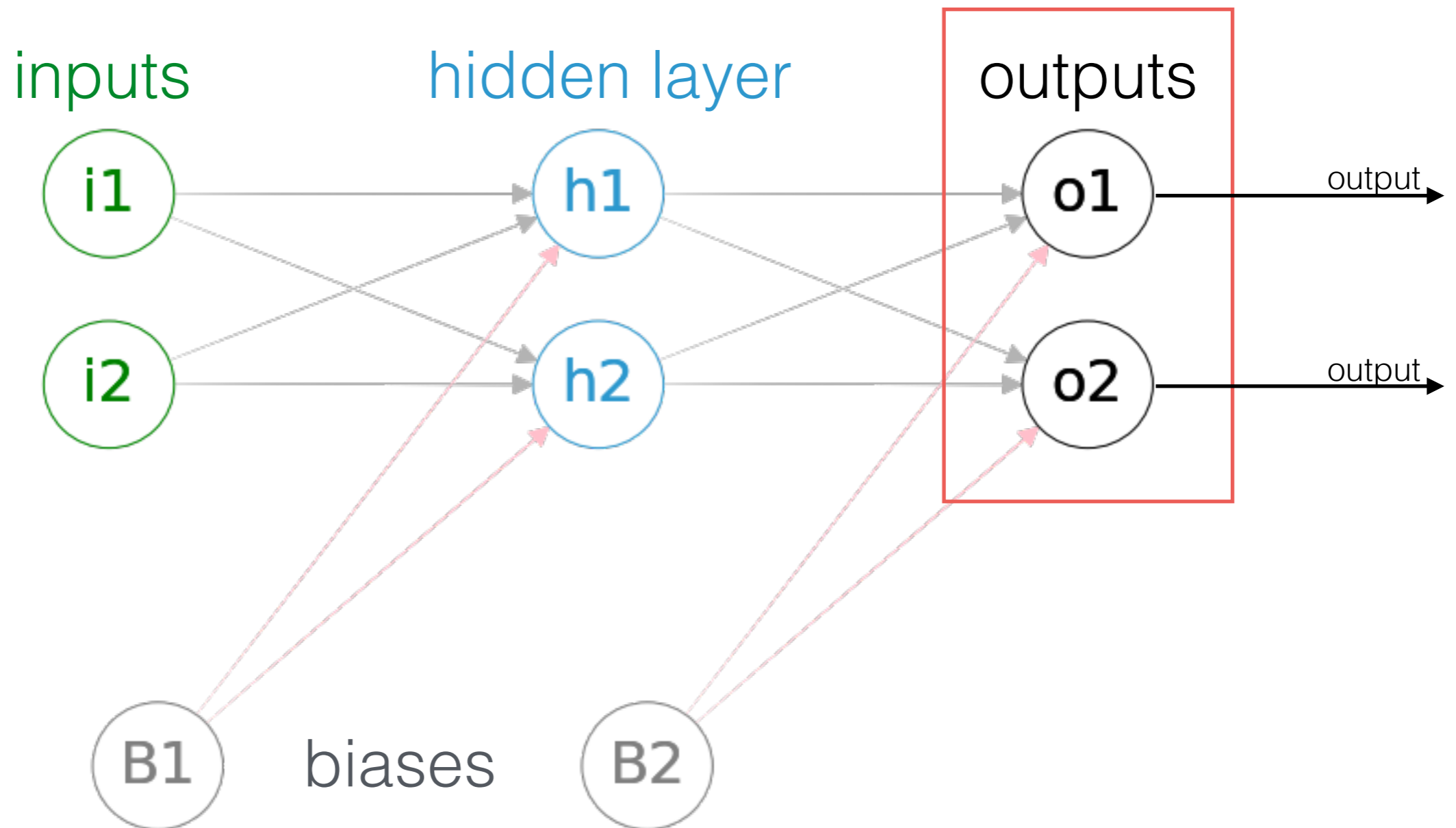
- Each node comprises a summation and an activation function.  
The activation is the non-linear component of network.

hidden layer



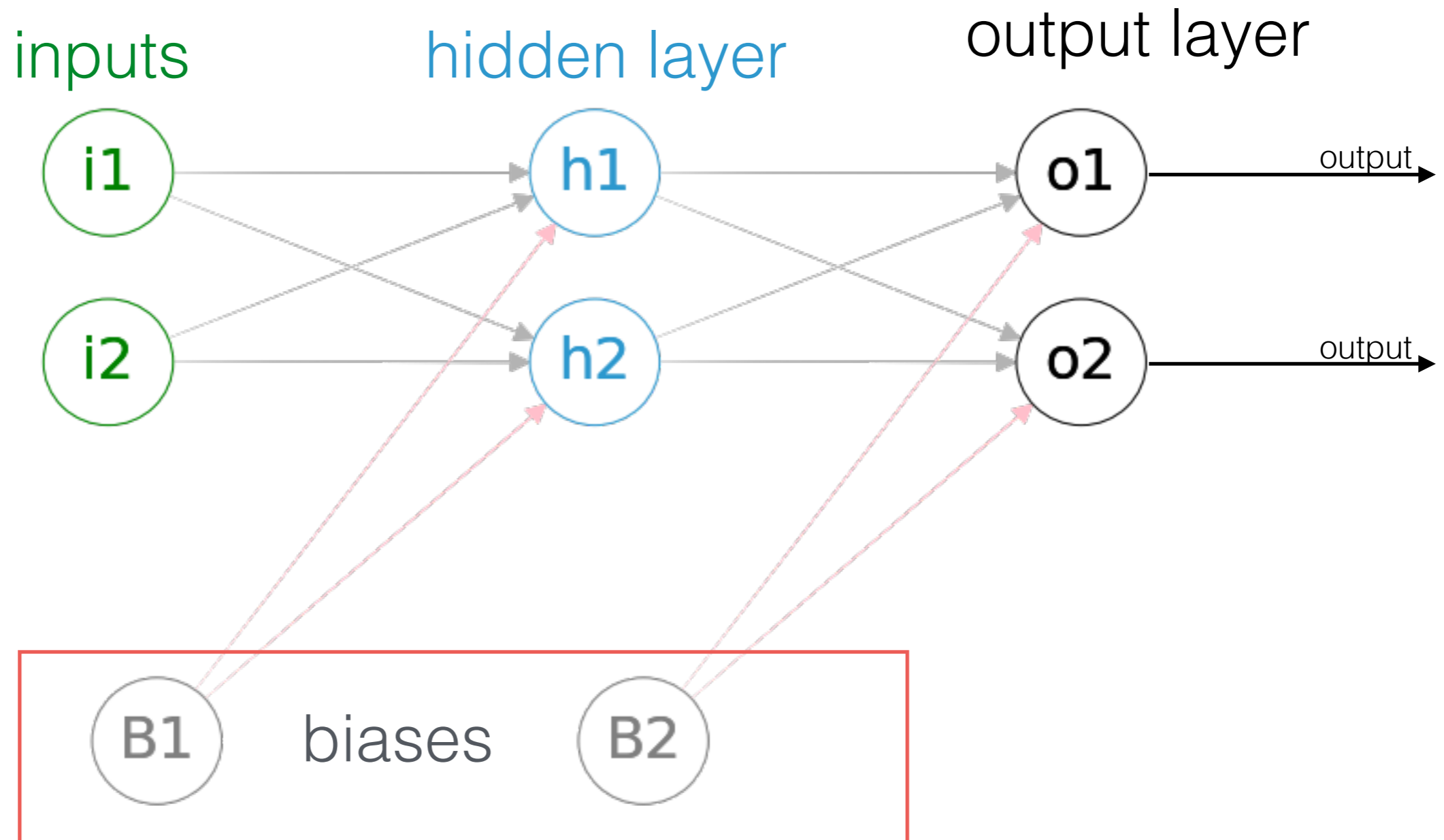
# A basic **Feedforward** neural network

- Output layers are the final layer and otherwise are indistinguishable from hidden layer.

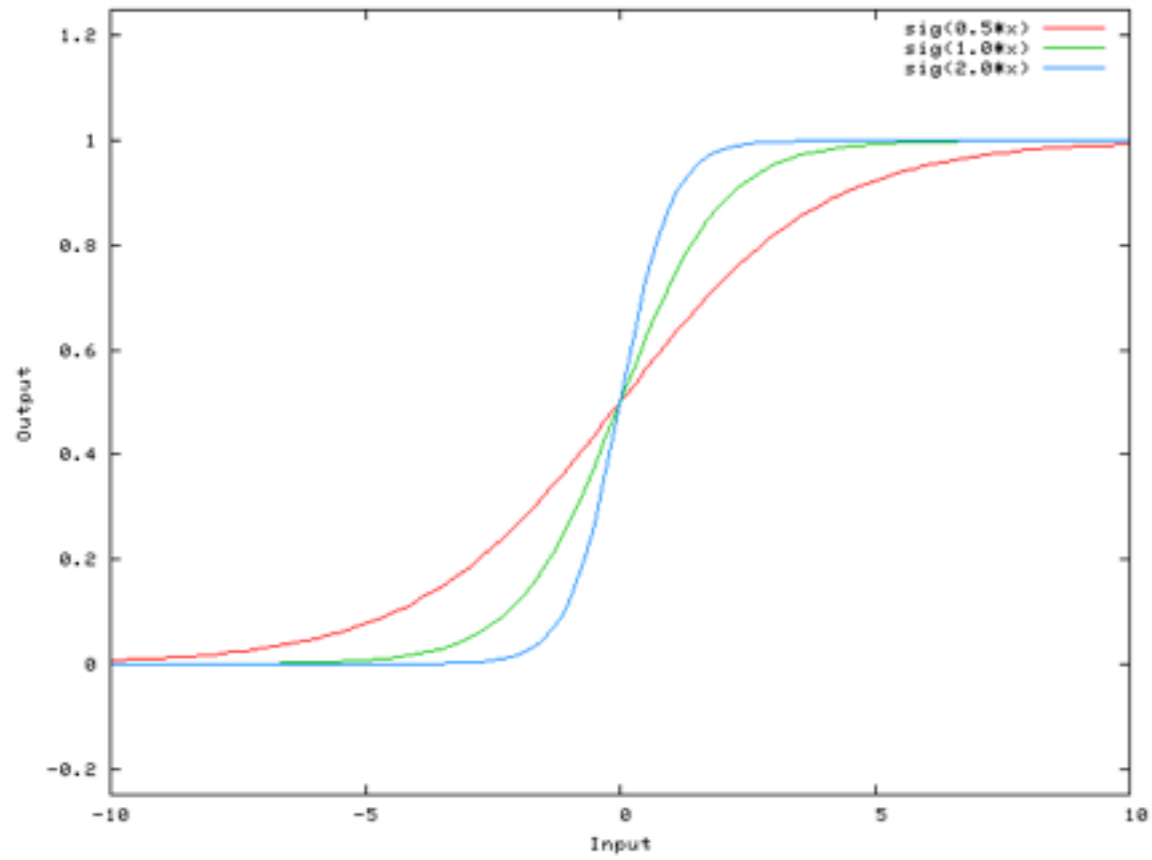


# A basic **Feedforward** neural network

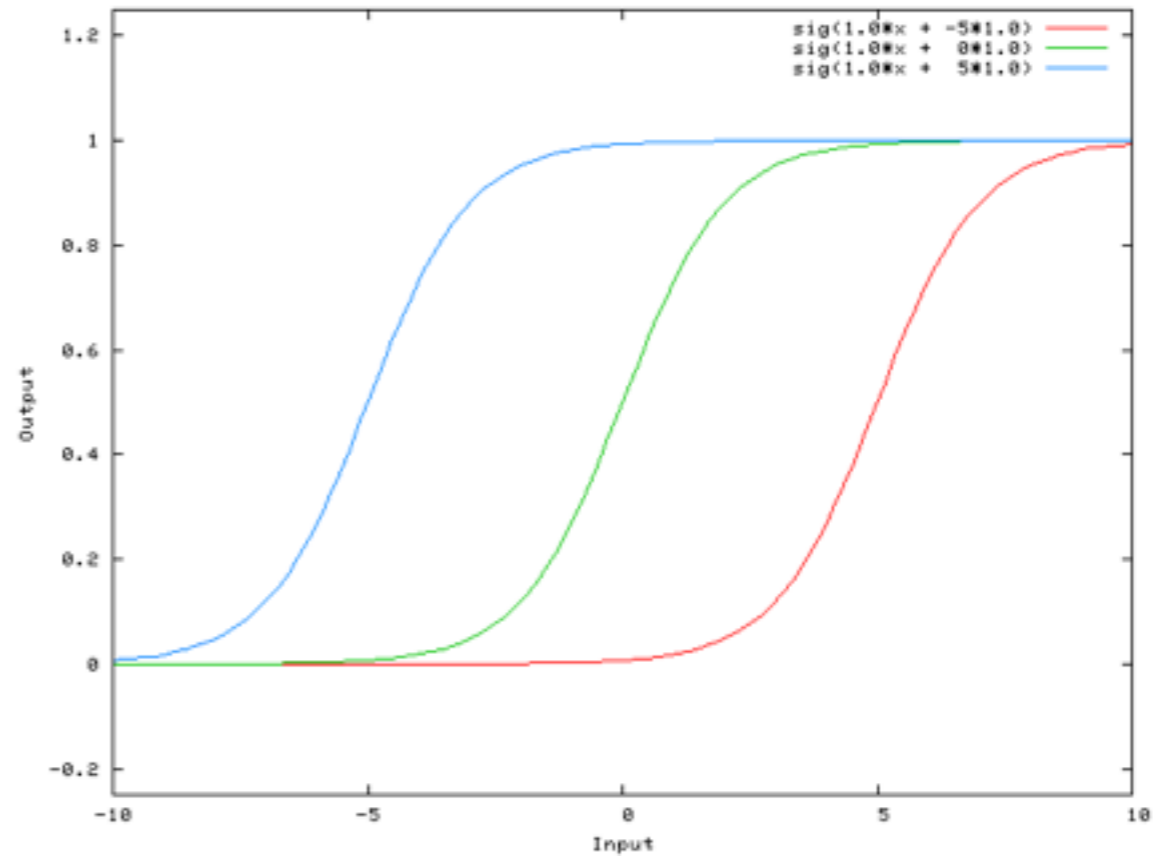
- The bias ensures that any input value can be mapped to any feasible output value. For the hidden layers and output layers.



# Extra slide bias



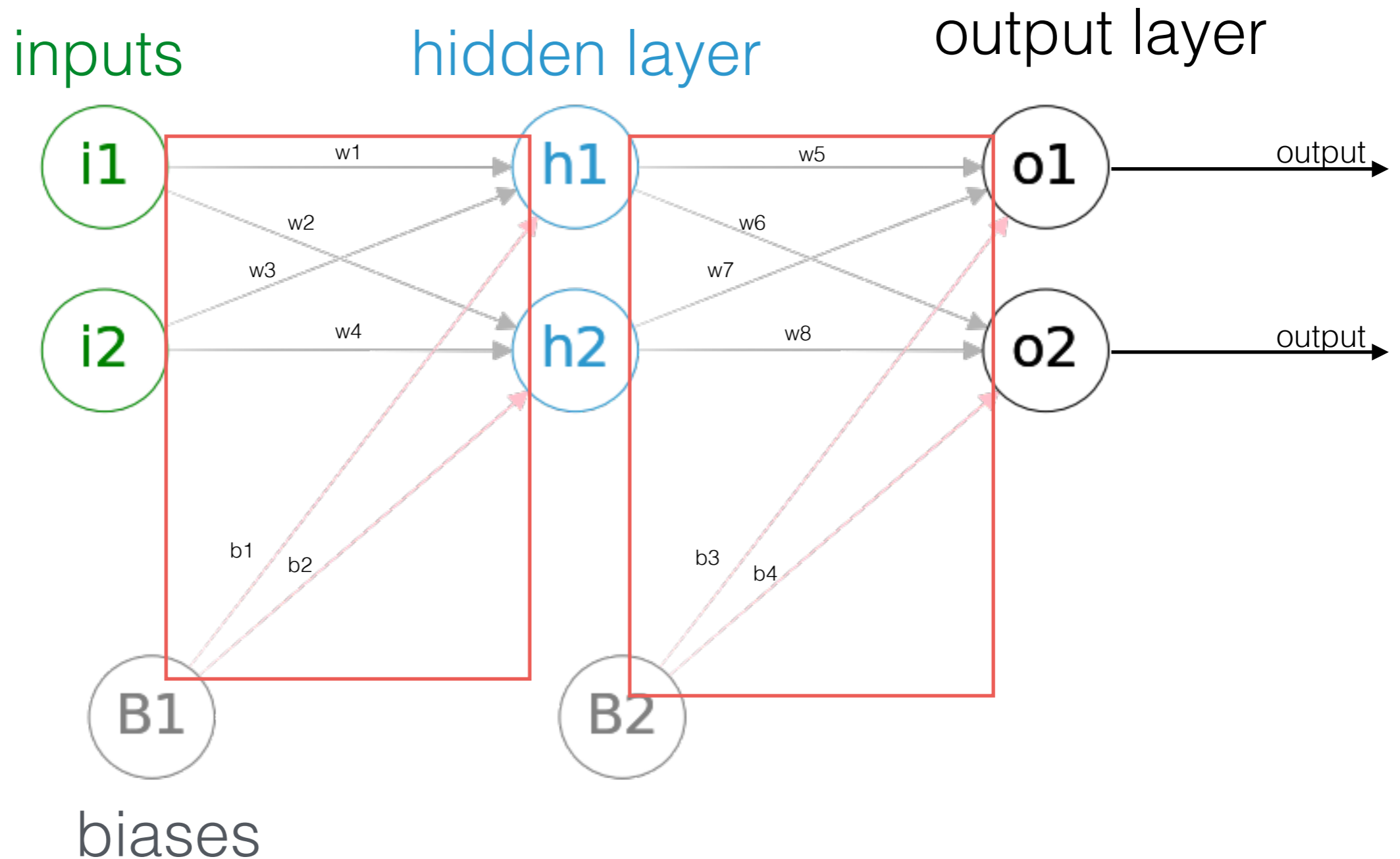
Without bias



With bias

# A basic **Feedforward** neural network

- The edges connect the nodes. Each edge connecting a node has a weight  $w_n$  e.g ( $w_6 = 0.5$ ), the bias also has a weight denoted  $b_1$  or  $b_2$ .

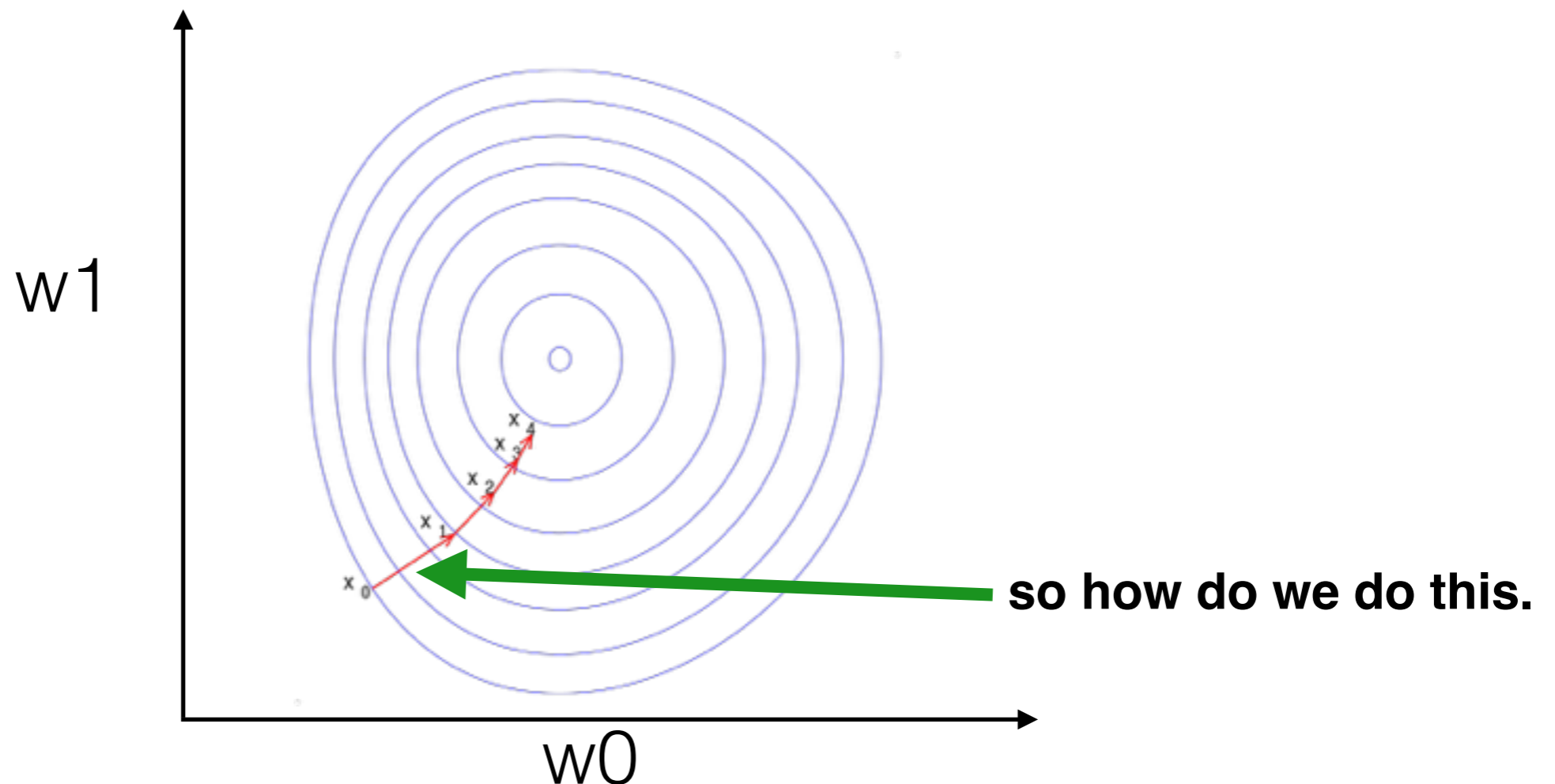


# Training the network

- Trained through **Gradient Descent**
  - **Forward pass**
  - **Backward pass**

# The Backward pass

- **Gradient decent.** A classical approach to optimisation is to calculate the gradient of our error function in response to changes in parameters (e.g. weights). We then move down the gradient to find the optimum solution.

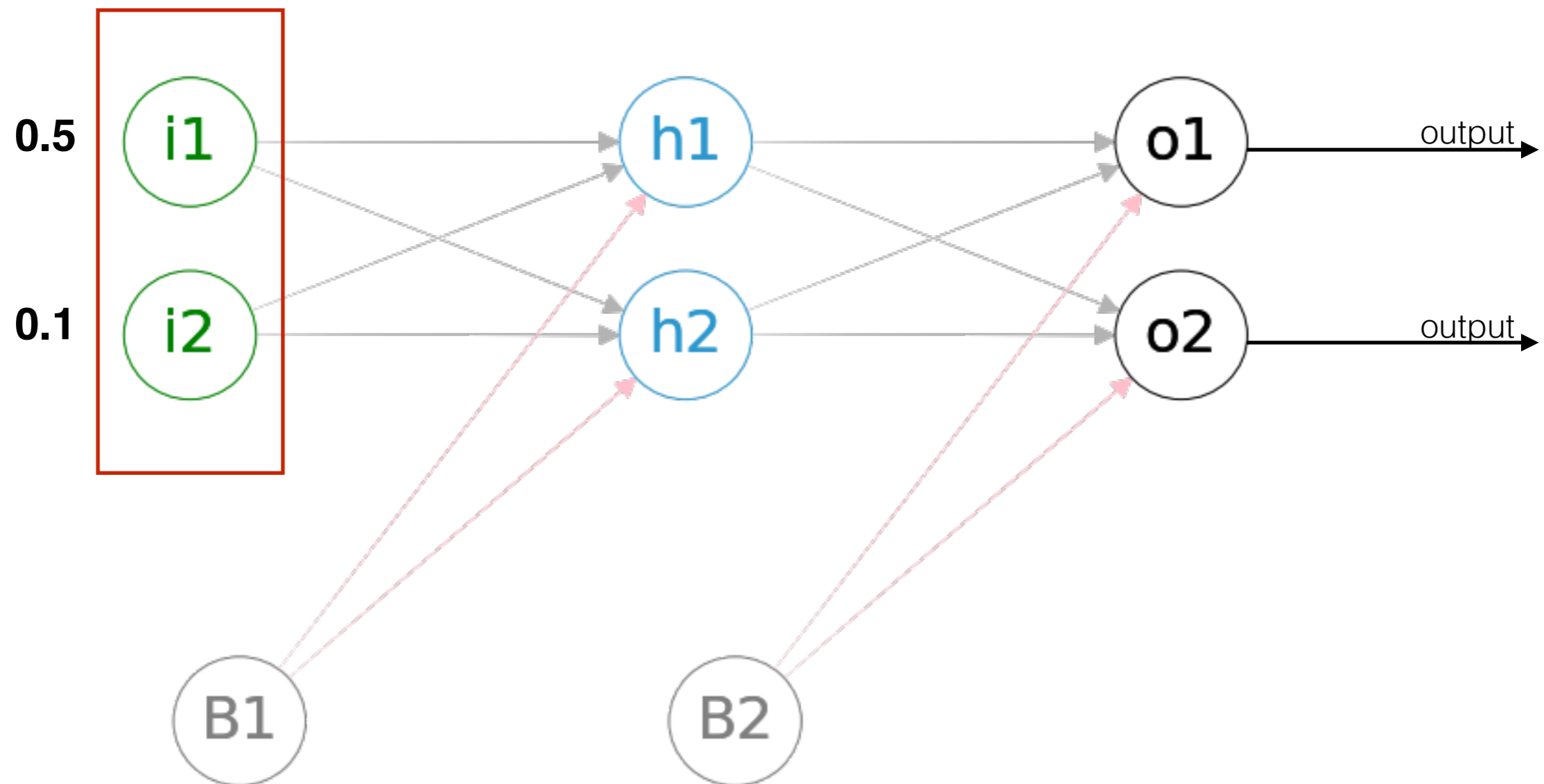


**Above)** How our optimisation would look if we had only two weights to optimise. Reality is hyper-dimensional.

source: [https://en.wikipedia.org/wiki/Gradient\\_descent](https://en.wikipedia.org/wiki/Gradient_descent)

# The Forward pass

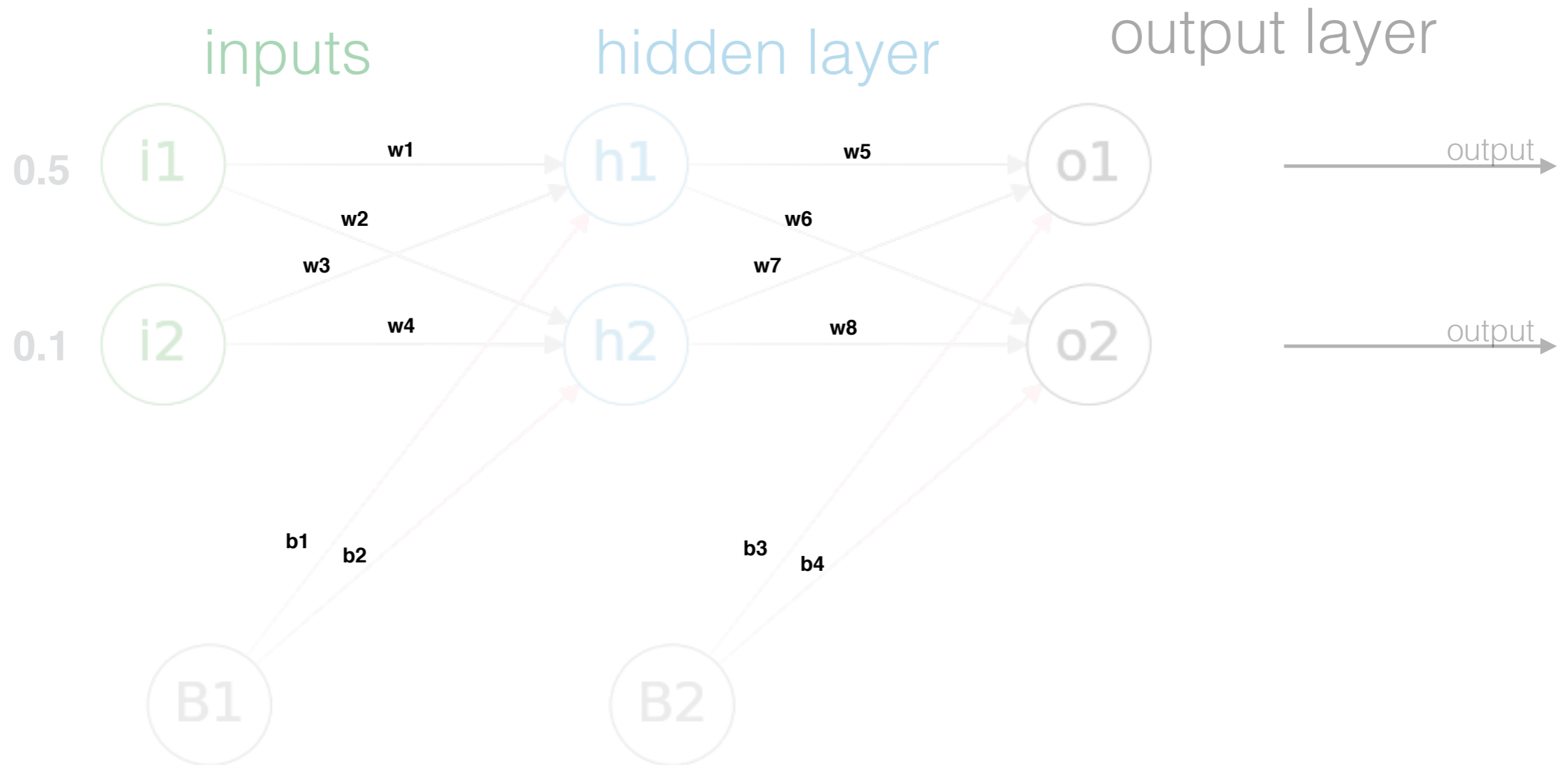
- First of all we generate the predictions of the network with our inputs by feeding them through the network.





# The Forward pass

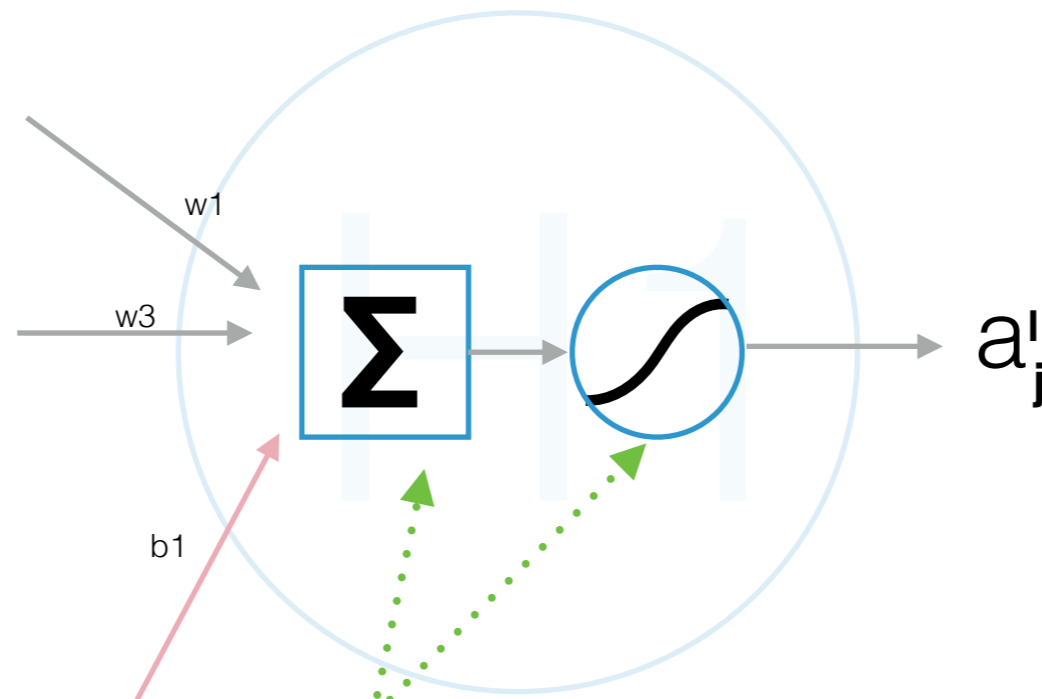
- The weights can be initialised in a number of ways. One way (easiest) is to choose random values.



# The Forward pass

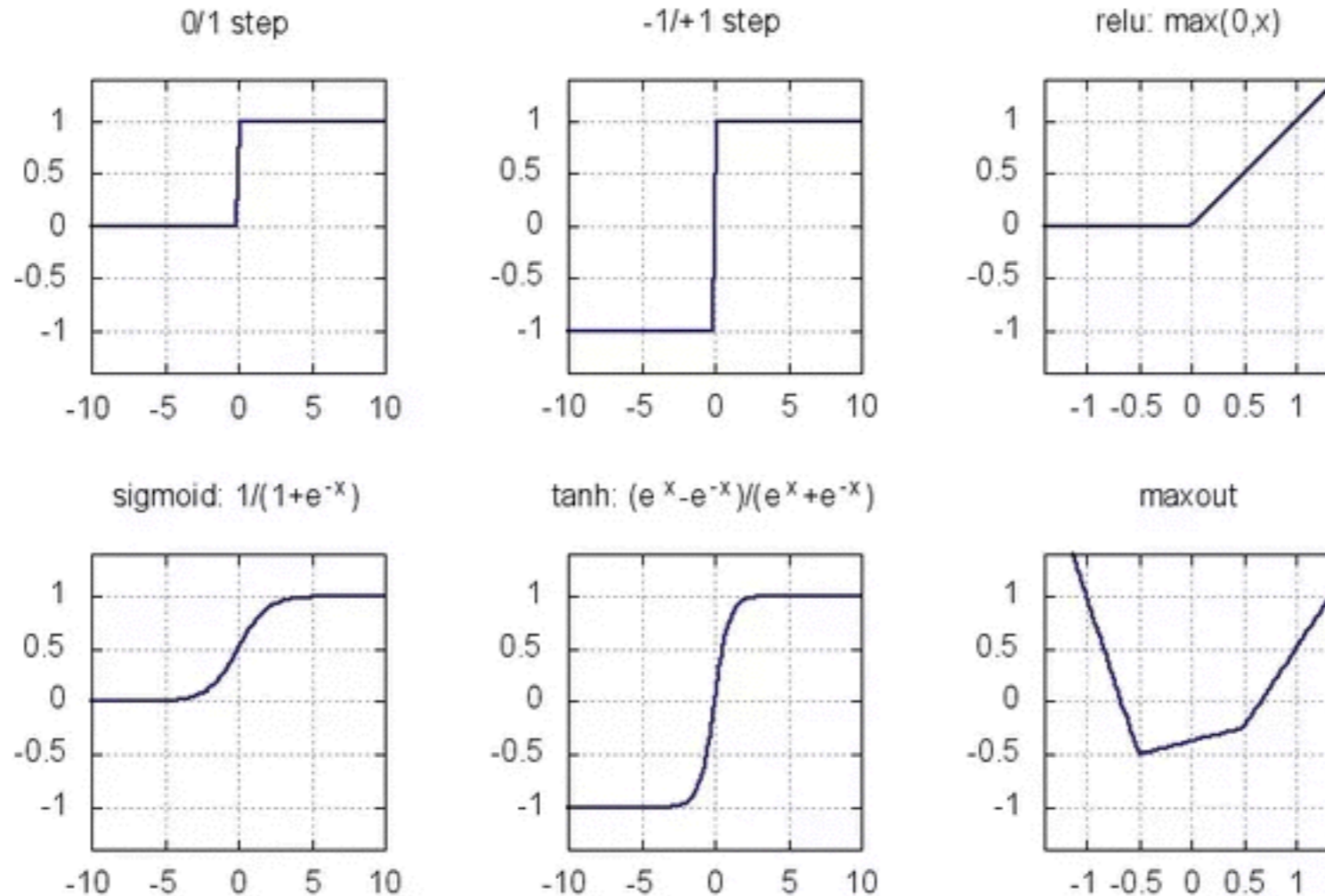
- The hidden layers comprise a summation and an activation function.

hidden layer



$$a_j^l = \sigma \left( \sum_k w_{jk}^l a_k^{l-1} + b_j^l \right)$$

# The Forward pass (Activation functions)



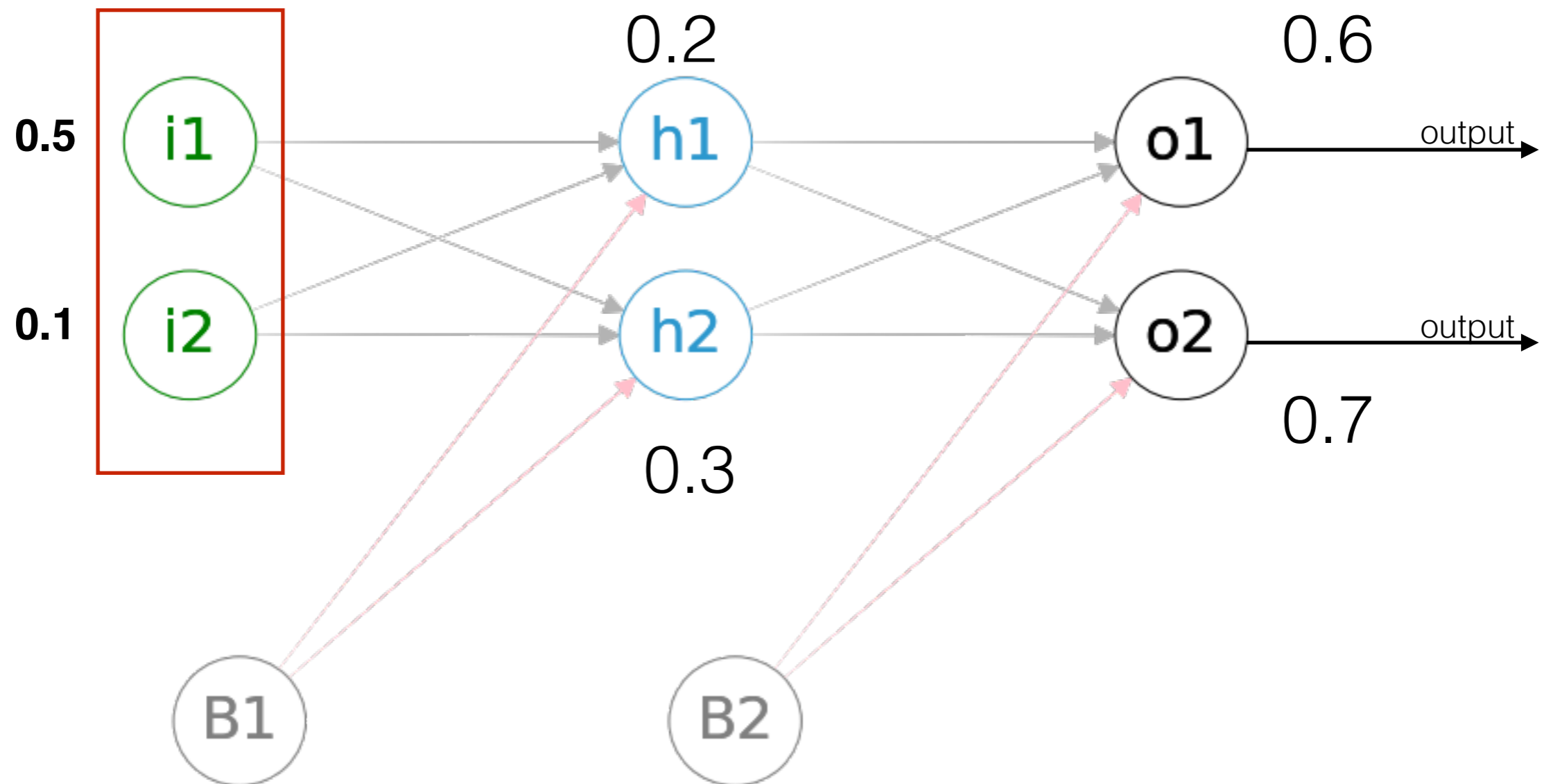
The non-linearity is what gives the network its power over other frameworks. Any can be used but must be differentiable (analytically).

Note: not all activation functions are differentiable in their entirety but we cheat and over-write these problematic areas.  
see [https://dwaithe.github.io/blog\\_20170508.html](https://dwaithe.github.io/blog_20170508.html) for more details

source: <https://www.quora.com/Do-people-use-power-functions-like-x-3-or-x-5-as-activation-functions-in-artificial-neural-networks>

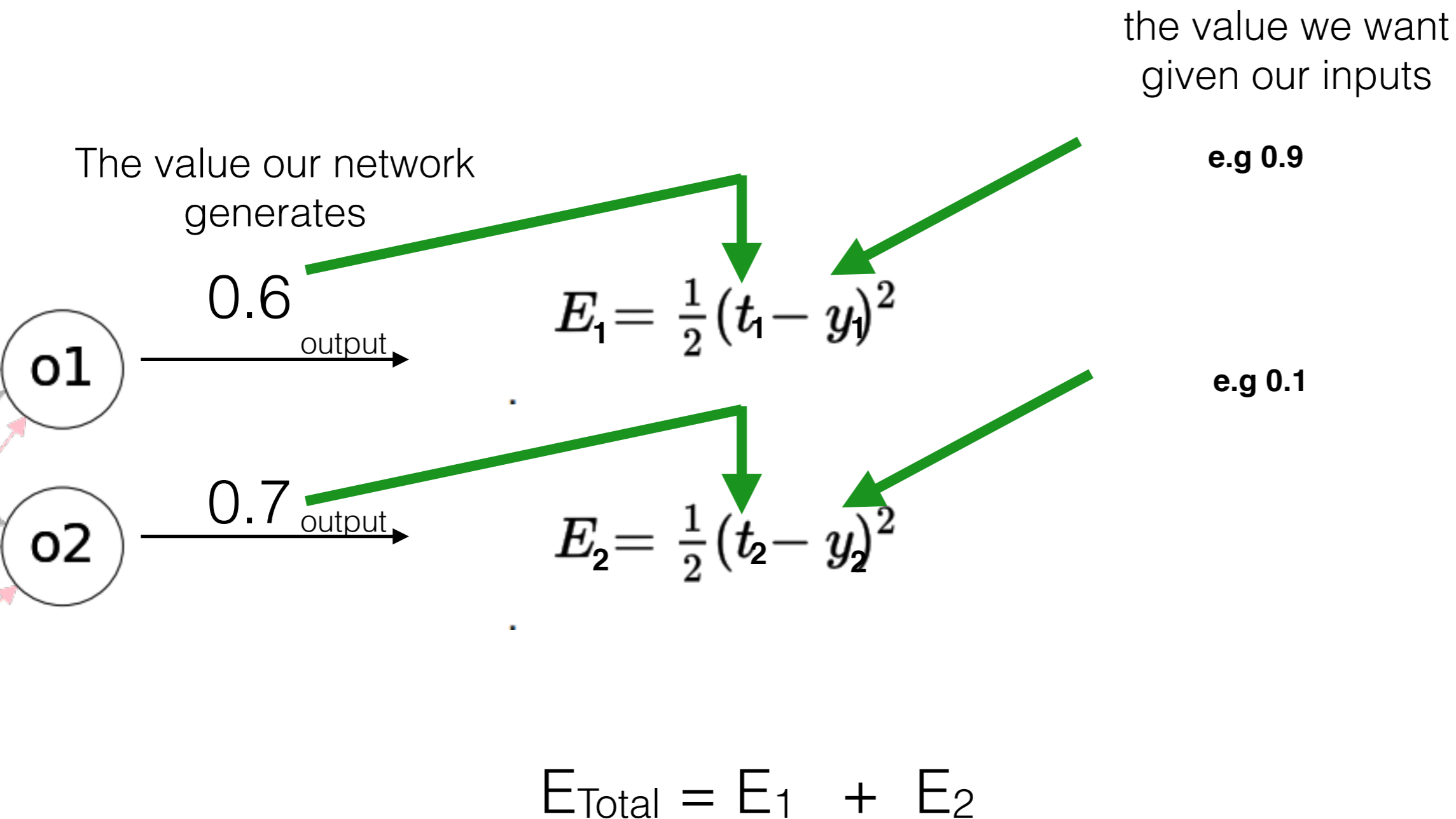
# The Forward pass

- First of all we generate the predictions of the network with our inputs by feeding them through the network.

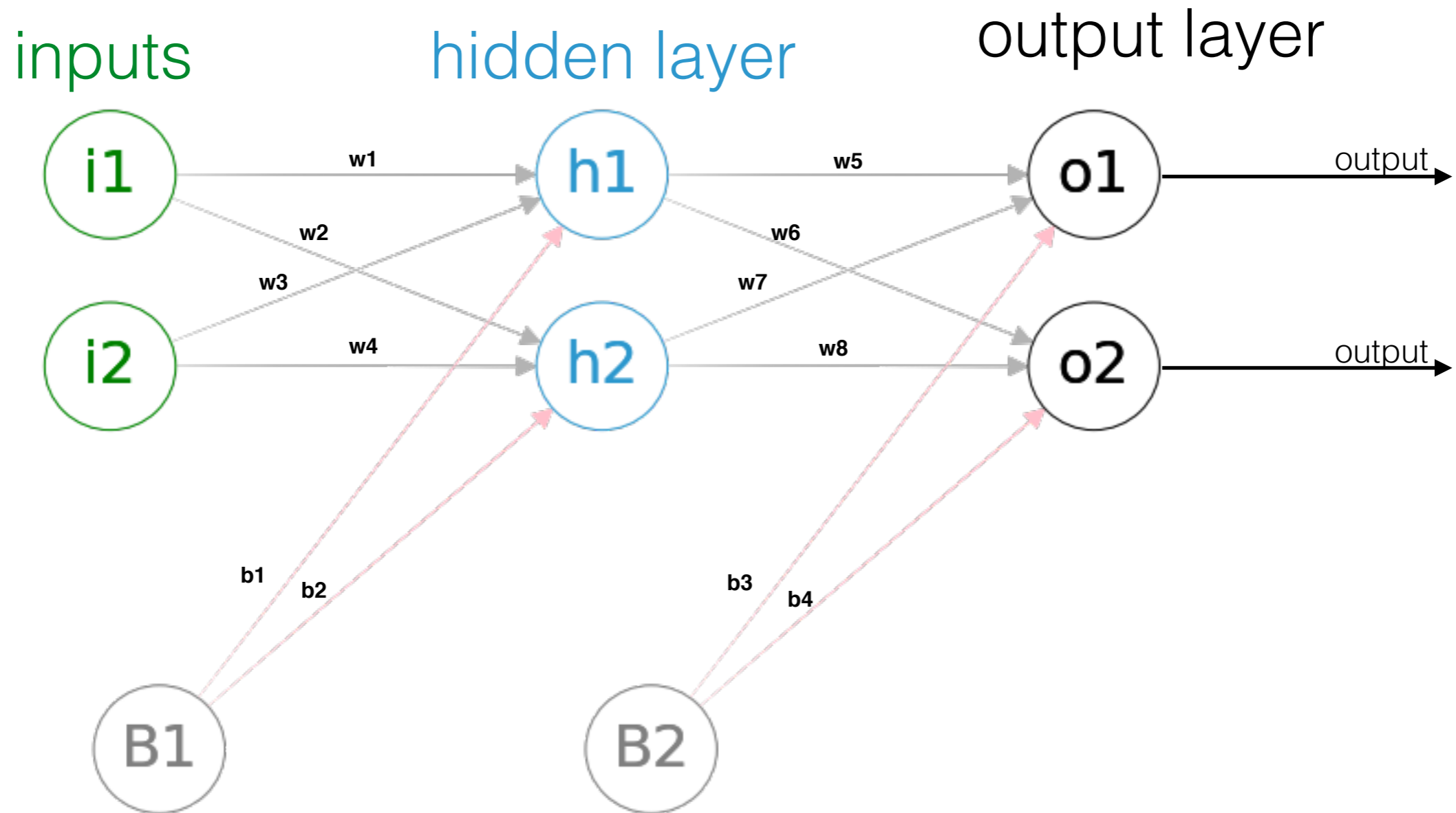


# The Forward pass

- Once outputs calculated. We calculated the error

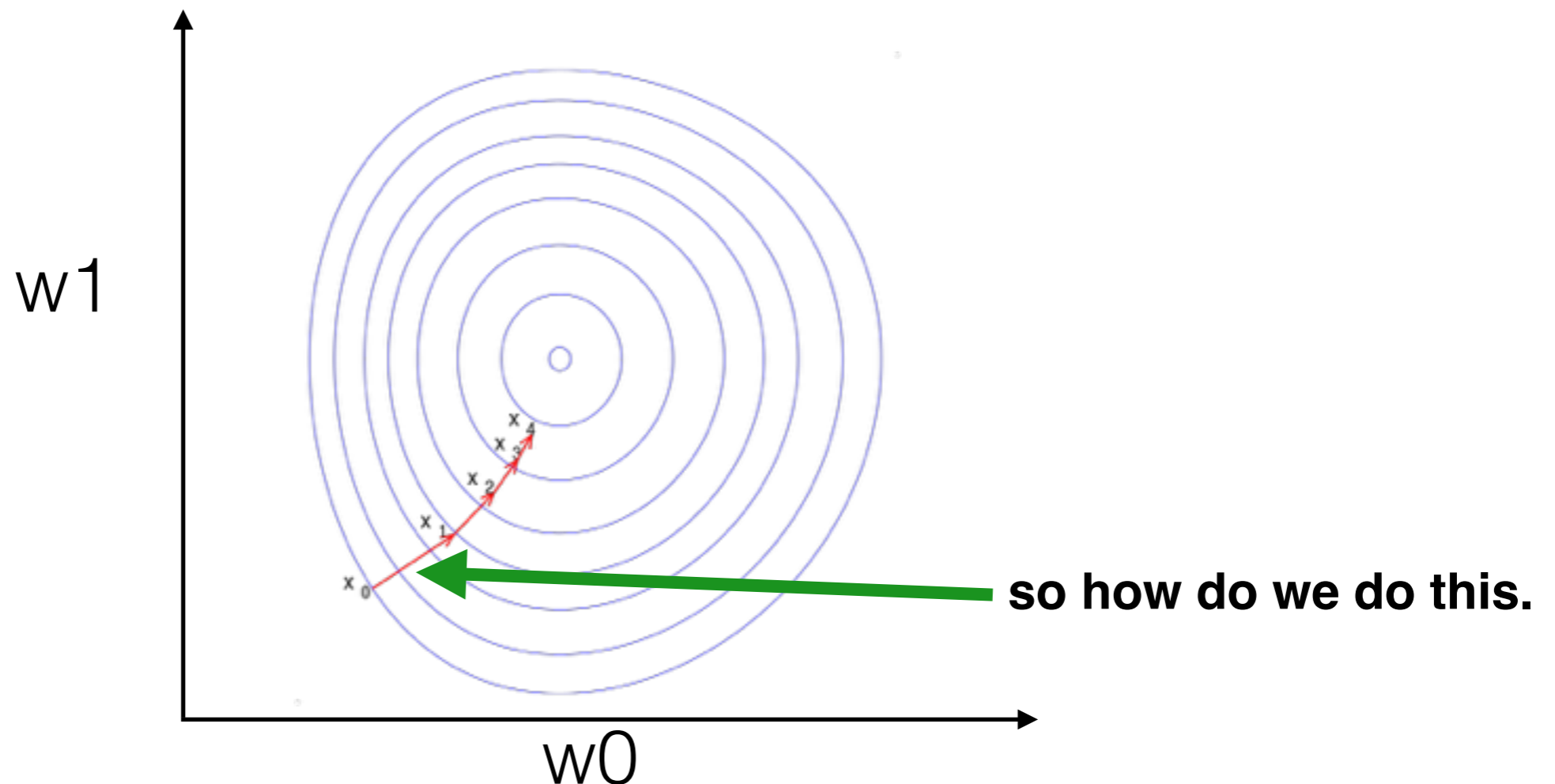


# The Backward pass



# The Backward pass

- **Gradient decent.** A classical approach to optimisation is to calculate the gradient of our error function in response to changes in parameters (e.g. weights). We then move down the gradient to find the optimum solution.

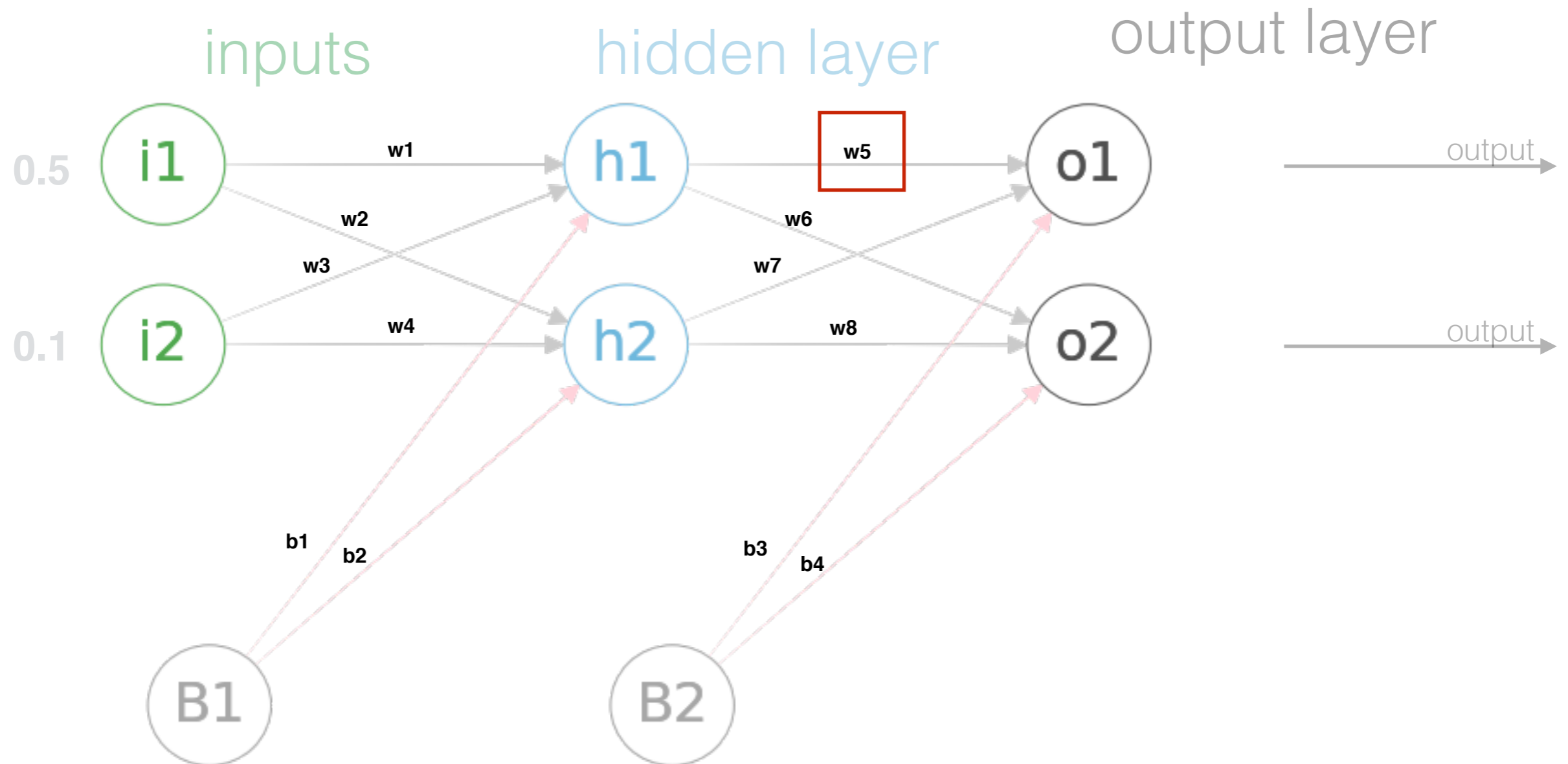


**Above)** How our optimisation would look if we had only two weights to optimise. Reality is hyper-dimensional.

source: [https://en.wikipedia.org/wiki/Gradient\\_descent](https://en.wikipedia.org/wiki/Gradient_descent)

# The Backward pass

- Now we work backward through all the weights and biases updating the weights so that the network will produce a result closer to what we want.





# The Backward pass

- Classically there are two ways to calculate the gradient at .
  - 1) Change the weights small amount and see how the total error changes.
  - 2) If the function is differentiable you can calculate the derivative directly (analytical method).
- The second method is faster and more accurate, but in maths, not all functions are differentiable.
- Fortunately all the functions chosen for neural networks are **differentiable**.  
Note: not all activation functions are differentiable in their entirety but we cheat and over-write these problematic areas. see [https://dwaithe.github.io/blog\\_20170508.html](https://dwaithe.github.io/blog_20170508.html).

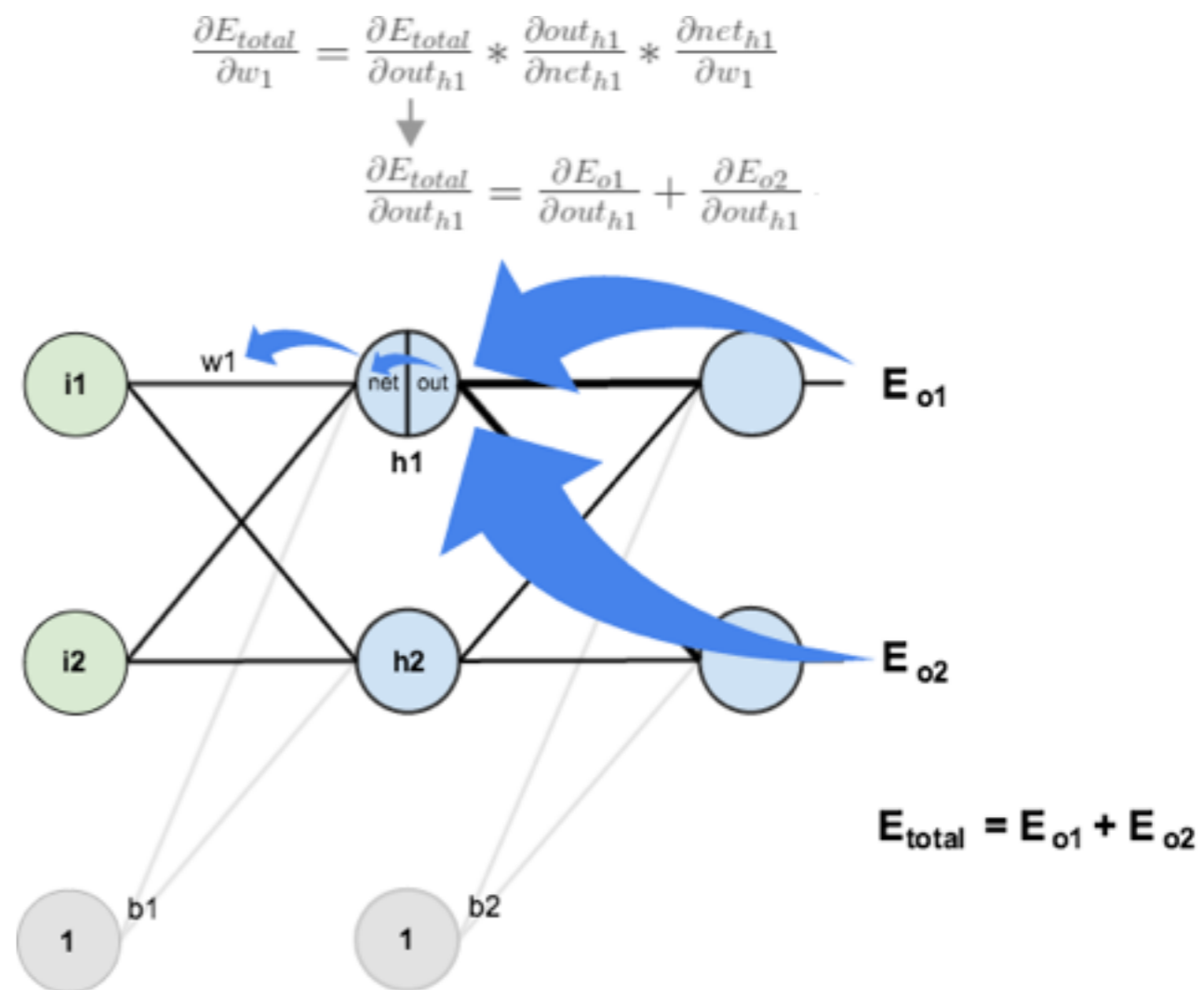
# The Backward pass

1.  $\frac{\partial E}{\partial w_{ij}} = \delta_j * x_i$  for all weights and biases
2.  $o_j' = o_j * (1 - o_j)$  for output layer nodes using softmax
3.  $\varphi_j' = (1 - \varphi_j) * (1 + \varphi_j)$  for hidden layer nodes using tanh
4.  $\varphi_j' = \varphi_j * (1 - \varphi_j)$  for hidden layer nodes using logistic sigmoid
5.  $e_j = (o_j - t_j)$  for hidden and output layer nodes
6.  $\delta_j = e_j * o_j'$  if  $j$  is an output node
7.  $\delta_j = (\sum \delta_j w_j) * \varphi_j'$  if  $j$  is a hidden node
8.  $\Delta w_{ij} = \alpha * \frac{\partial E}{\partial w_{ij}}$  delta for all weights and biases
9.  $w_{ij}' = w_{ij} + \Delta w_{ij}$  update for all weights and biases

# The Backward pass

- Weights can be quite distant from the output but influence everything down stream resulting in large composite derivatives at each node.

e.g.  $w_1$  has dependency on the outputs ( $E_{o1}$ ,  $E_{o2}$ ), activation and summation functions of  $h_1$ .




# The Backward pass

- The **chain-rule** is a calculus method for breaking down large equations containing multiple derivatives.

11

## Chain Rule



---

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}$$

Example

$$f(n) = \cos(n) \quad n = e^{2w} \quad f(n(w)) = \cos(e^{2w})$$
$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (-\sin(n))(2e^{2w}) = (-\sin(e^{2w}))(2e^{2w})$$

Application to Gradient Calculation

$$\frac{\partial \hat{F}^m}{\partial w_{i,j}^m} = \frac{\partial \hat{F}^m}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} \qquad \frac{\partial \hat{F}^m}{\partial b_i^m} = \frac{\partial \hat{F}^m}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m}$$

- It allows us to calculate the derivative of any weight in the network with respect to the output error as long as we systematically calculate all the intermediates

# The Backward pass

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

You might also see this written as:

$$\frac{\partial E_{total}}{\partial w_1} = \left( \sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}} \right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \left( \sum_o \delta_o * w_{ho} \right) * out_{h1} (1 - out_{h1}) * i_1$$

$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} i_1 \quad \text{This is the delta rule.}$$

# The Backward pass

The weights are updated as follows

$$W(t+1) = W(t) - \alpha \frac{\partial E_{total}}{\partial w_1} :$$

$\alpha$  is the learning-rate

$W(t+1)$  is the new weight

$W(t)$  is the current weight

$\frac{\partial E_{total}}{\partial w_1}$  : is the derivative with respect to the the total error.

# Repeat many times

- Forward pass
  - Backward pass
- until convergence, error tends to zero.
- 

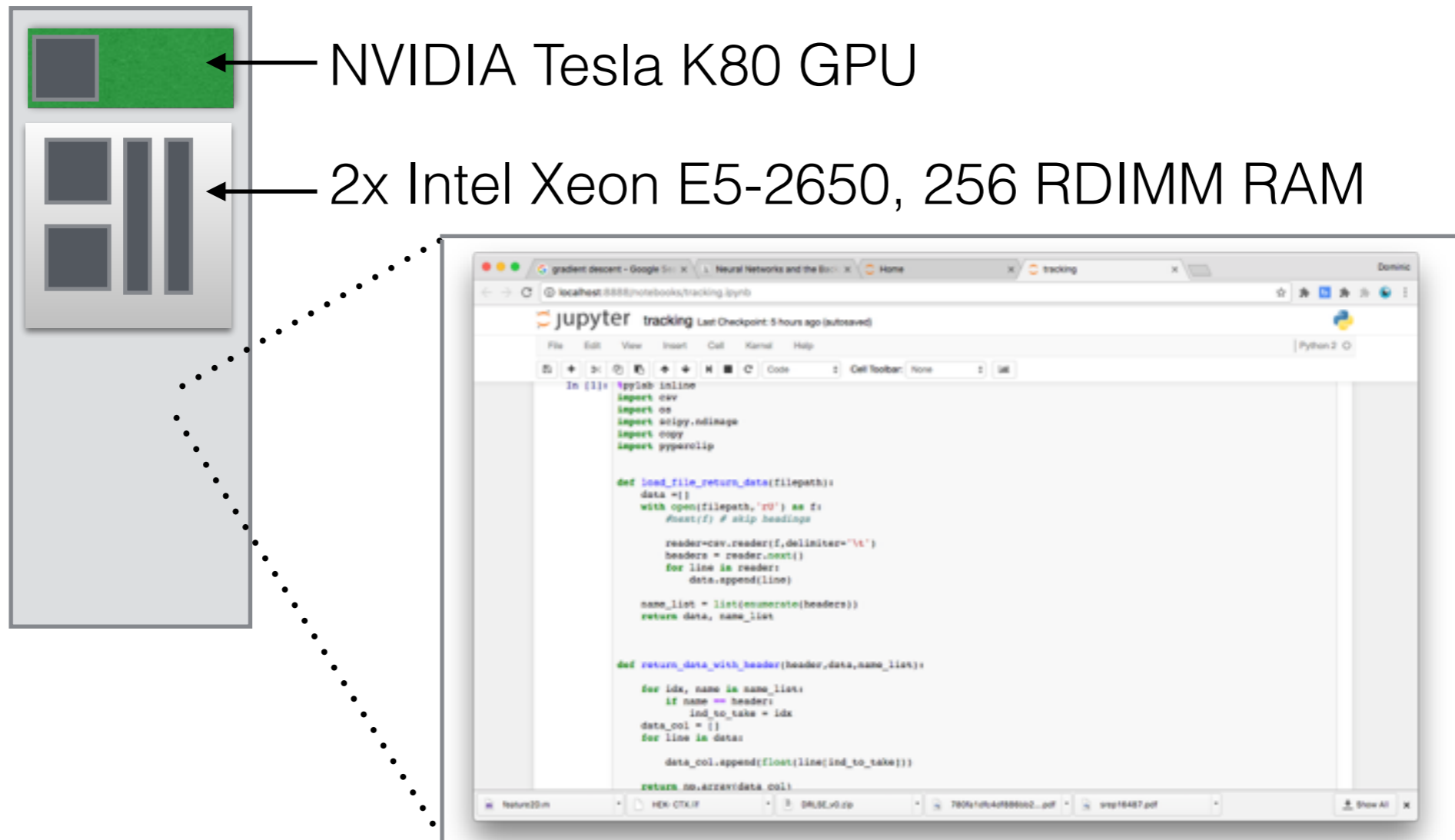
# **What I am doing**

by Dominic Waithe



# What I am using

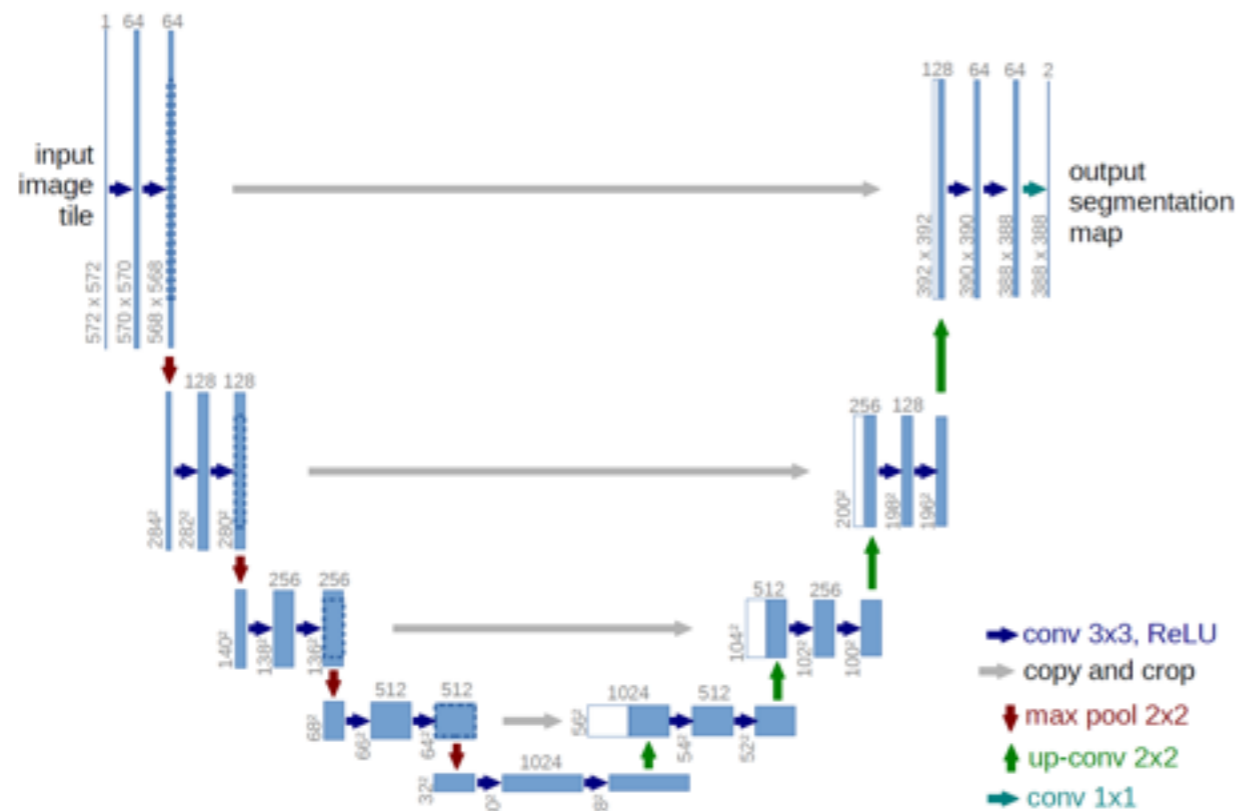
Harrier server - hosted in Begbroke, maintained by CBRG, paid for by Micron



I run Jupyter Notebooks hosted on the server and accessed on my local machine, using keras and tensorflow.

# What I have been working on

I have been using a network specialised for segmentation of biological images called U-net.



**U-Net: Convolutional Networks for Biomedical Image Segmentation**  
Olaf Ronneberger, Philipp Fischer, Thomas Brox

There is a down-sampling and up-sampling component.

# How the model looks like when implemented in Keras

```
def get_unet(img_rows, img_cols):  
    """This sets up the U-NET network structure. The same as in  
    https://github.com/jocicmarko/ultrasound-nerve-segmentation  
    except that I use different activation function (not sigmoid) in the  
    last layer and also I use a different loss function (not dice_coef)."""  
    inputs = Input((1, img_rows, img_cols))  
    conv1 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(inputs)  
    conv1 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(conv1)  
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)  
  
    conv2 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(pool1)  
    conv2 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(conv2)  
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)  
  
    conv3 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(pool2)  
    conv3 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(conv3)  
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)  
  
    conv4 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(pool3)  
    conv4 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(conv4)  
    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)  
  
    conv5 = Convolution2D(512, 3, 3, activation='relu', border_mode='same')(pool4)  
    conv5 = Convolution2D(512, 3, 3, activation='relu', border_mode='same')(conv5)  
  
    up6 = merge([UpSampling2D(size=(2, 2))(conv5), conv4], mode='concat', concat_axis=1)  
    conv6 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(up6)  
    conv6 = Convolution2D(256, 3, 3, activation='relu', border_mode='same')(conv6)  
  
    up7 = merge([UpSampling2D(size=(2, 2))(conv6), conv3], mode='concat', concat_axis=1)  
    conv7 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(up7)  
    conv7 = Convolution2D(128, 3, 3, activation='relu', border_mode='same')(conv7)  
  
    up8 = merge([UpSampling2D(size=(2, 2))(conv7), conv2], mode='concat', concat_axis=1)  
    conv8 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(up8)  
    conv8 = Convolution2D(64, 3, 3, activation='relu', border_mode='same')(conv8)  
  
    up9 = merge([UpSampling2D(size=(2, 2))(conv8), conv1], mode='concat', concat_axis=1)  
    conv9 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(up9)  
    conv9 = Convolution2D(32, 3, 3, activation='relu', border_mode='same')(conv9)  
  
    conv10 = Convolution2D(1, 1, 1, activation=None)(conv9)  
  
    model = Model(input=inputs, output=conv10)  
  
    model.compile(optimizer=Adam(lr=1e-5), loss='mse', metrics=['accuracy', accuracy_custom])  
  
    return model
```

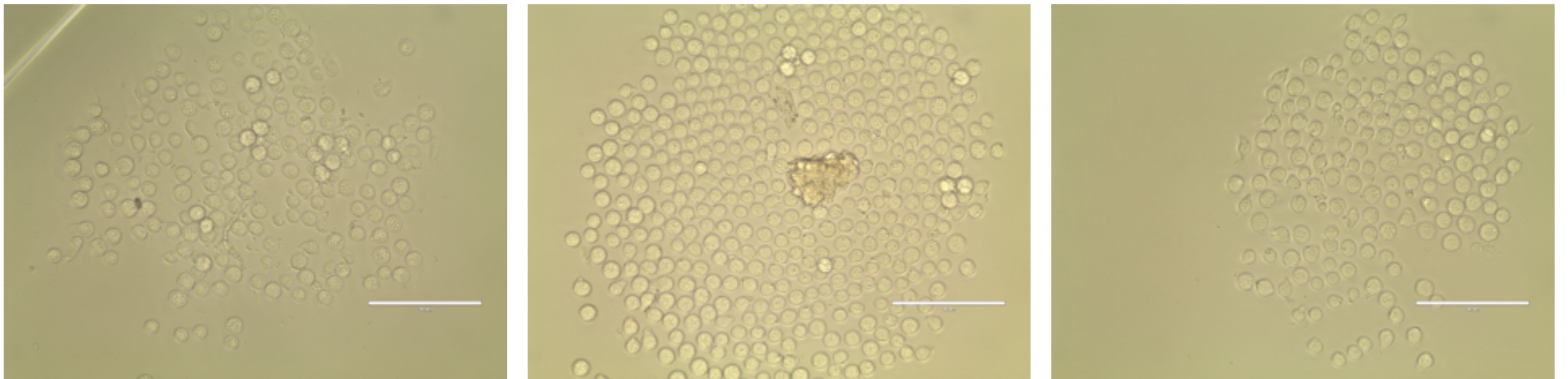
# What I have been working on

Microscopy Cell Counting with Fully Convolutional Regression Networks

Weidi Xie, J. Alison Noble, Andrew Zisserman

Department of Engineering Science, University of Oxford, UK

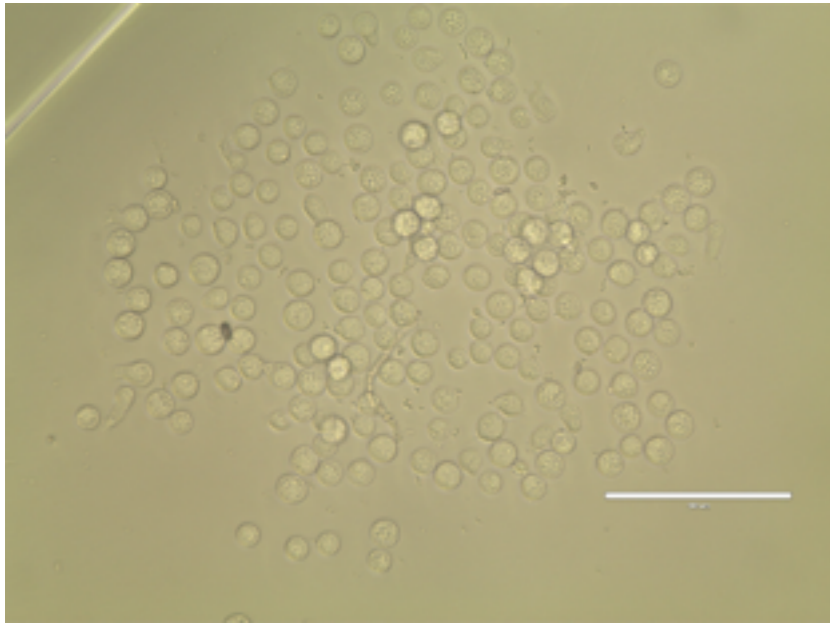
Weidi Xie gave me his latest code for using U-Net specifically for counting. Implemented in Keras.



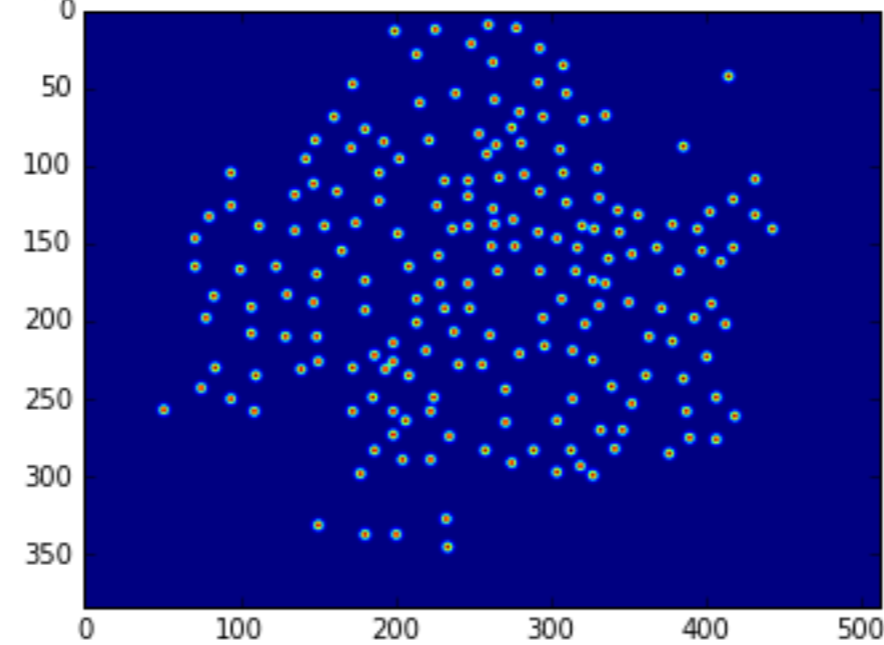
Want to apply to the goal of counting in phase contrast images. Images courtesy of Caroline Scott.

# U-Net variant for density estimation applied to phase contrast images

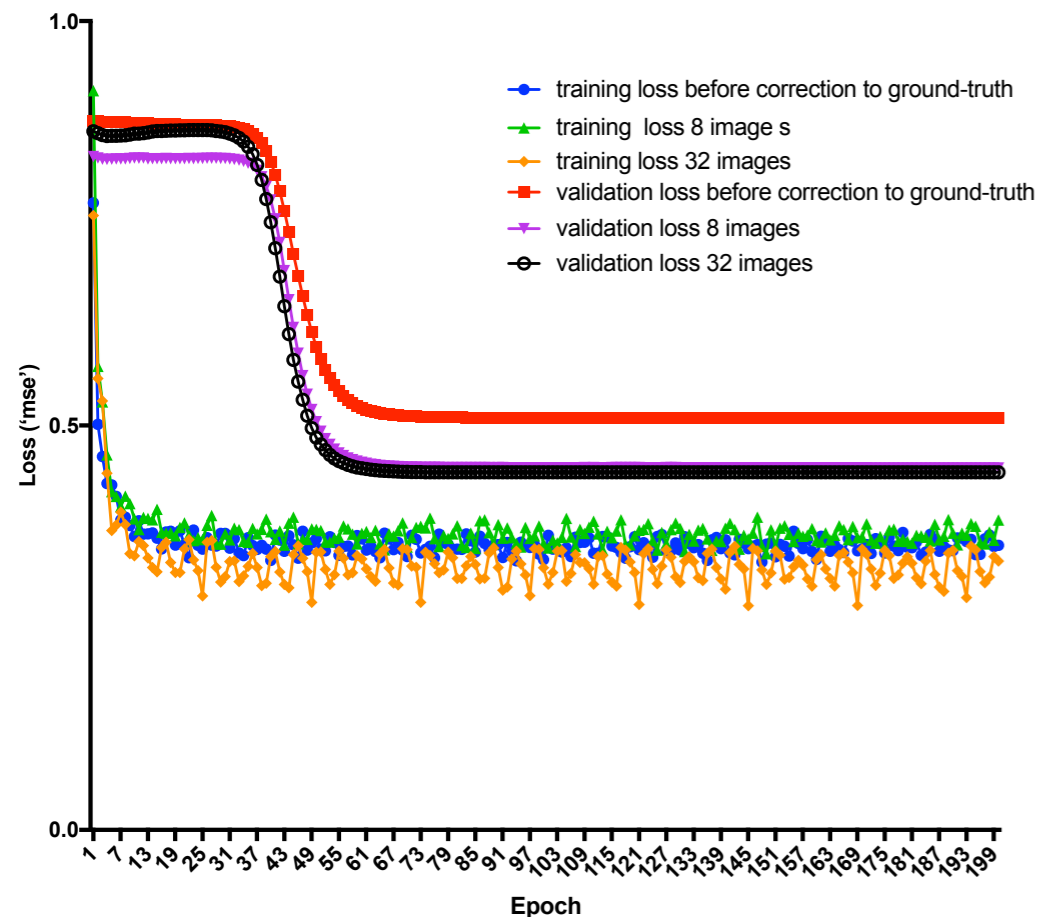
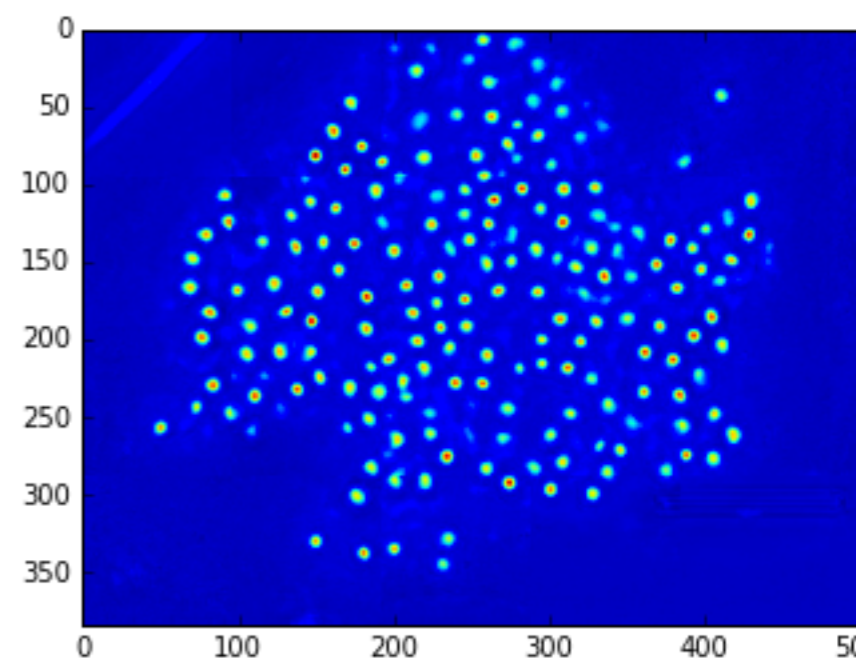
input image



ground-truth density



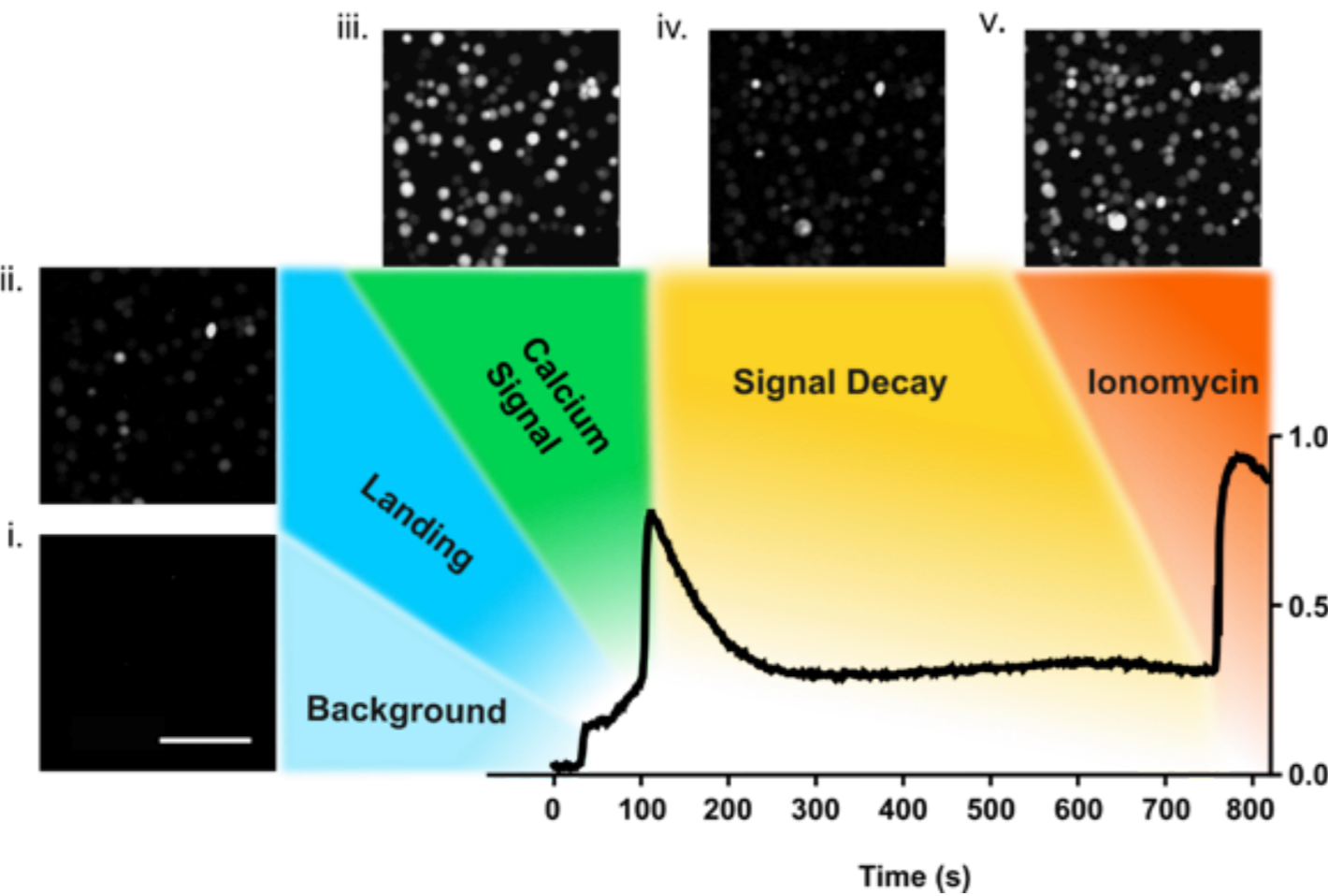
predicted density



So far achieved 85 %  
accuracy with  
32 training images.

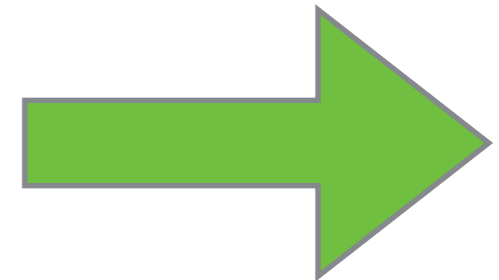
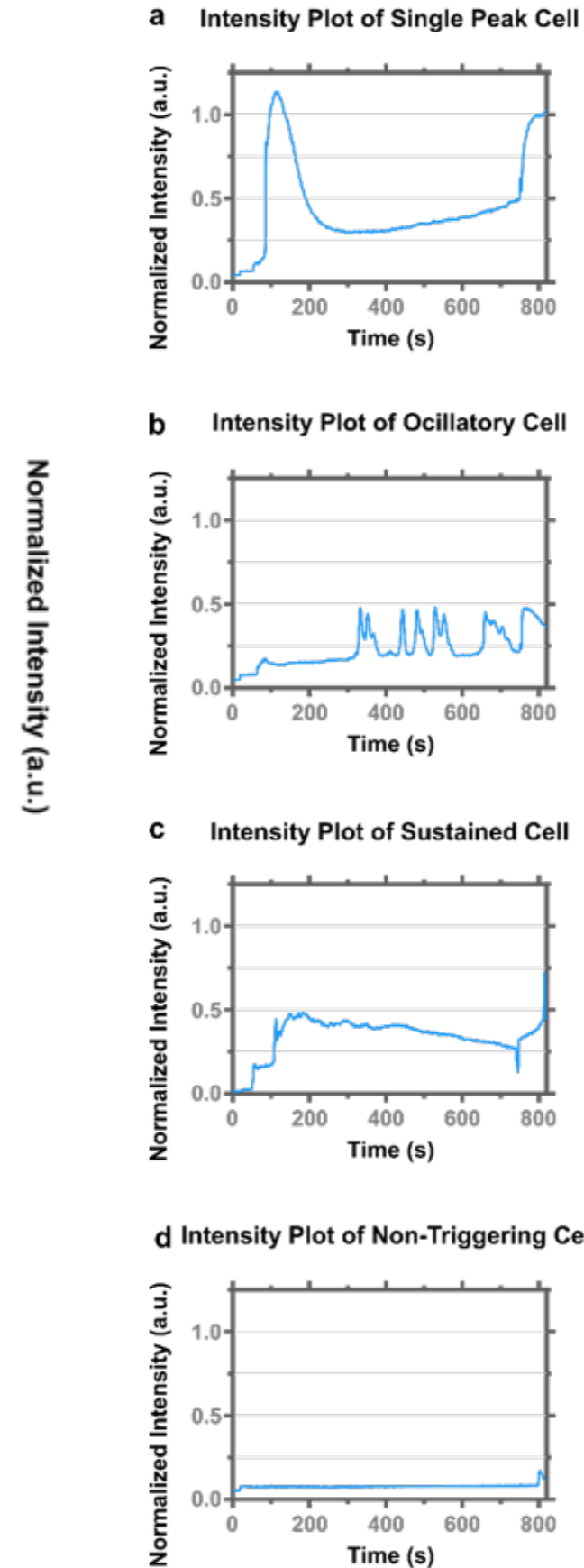
Still some work to be done

# Project with Liliana Barber



T-cells trigger intracellular calcium release when they touch surface.

Courtesy of Angela Lee and Marco Fritzsche



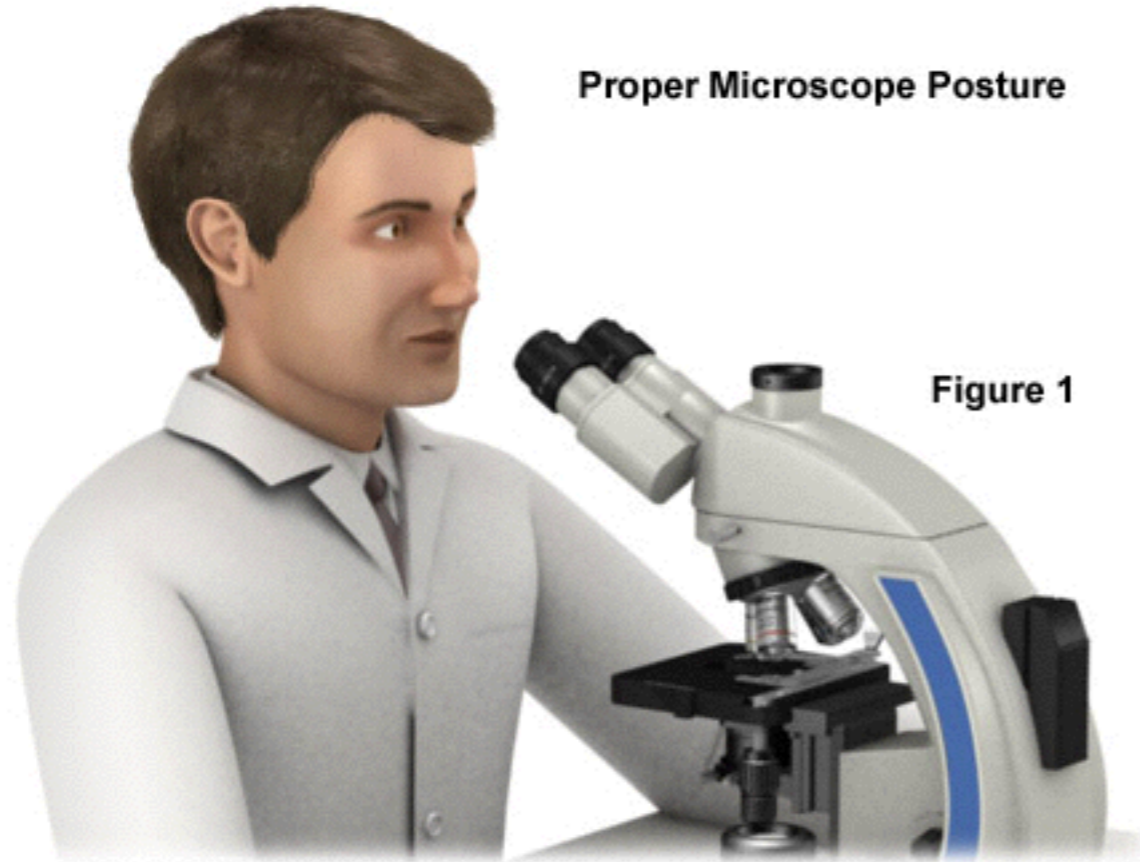
**build neural network classifier to decide automatically what type of curve is present**

# **What I will be doing**

by Dominic Waithe

**BBSRC TRDF Grant, starting in July**

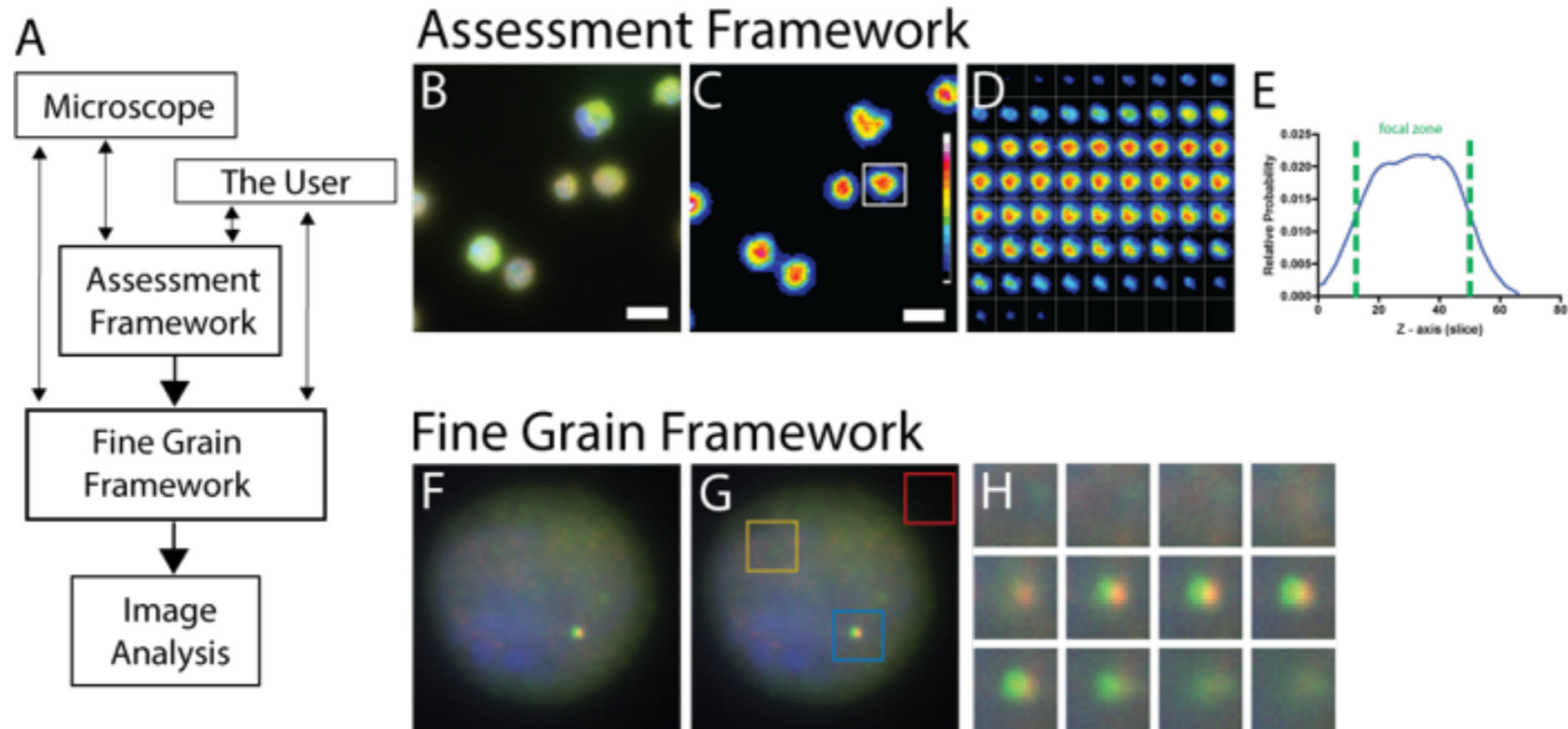
# Microscopy can be very tedious and time consuming.



Hard to automate assays which are used often in bioscience research. Too ad-hoc or short-term.

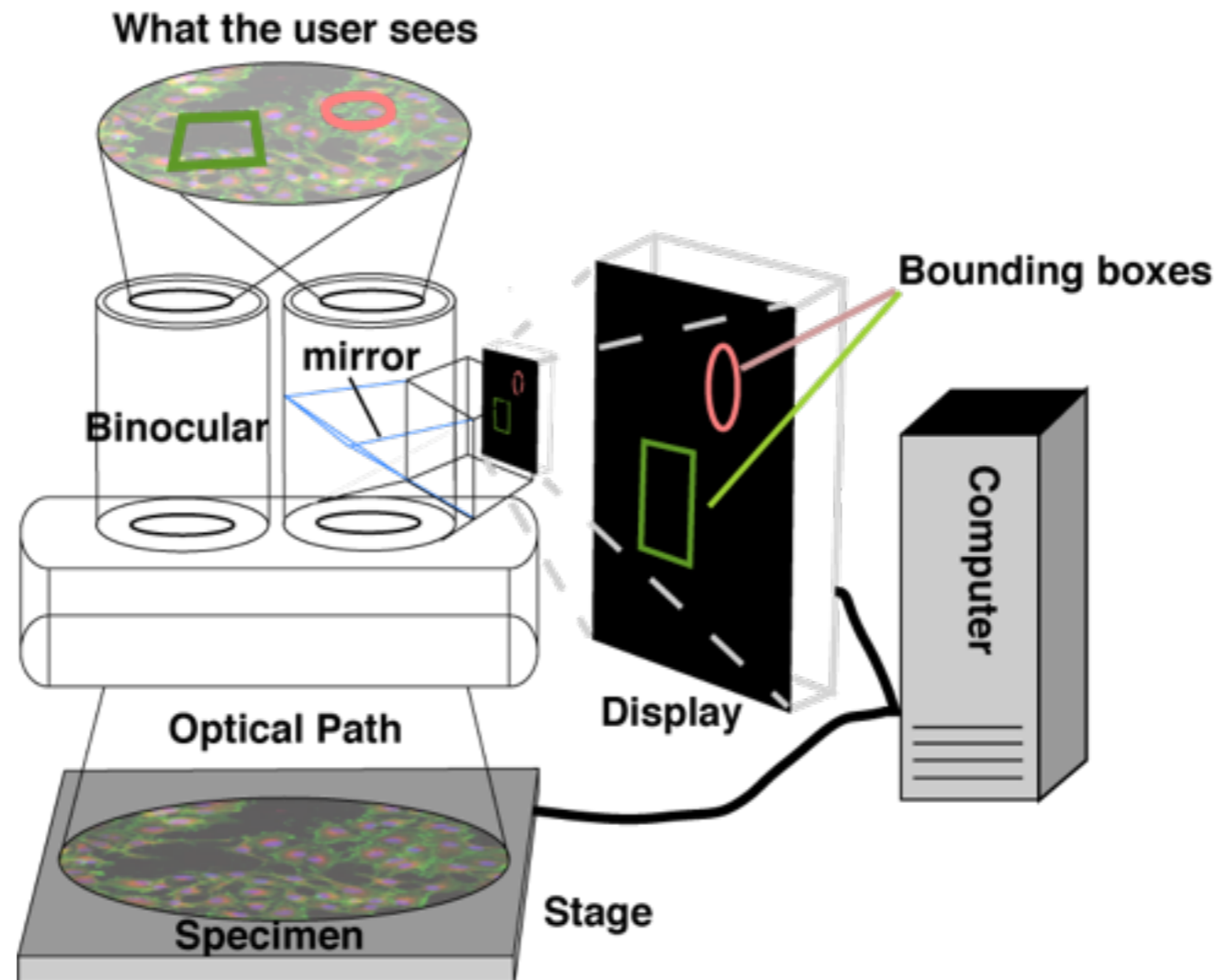


# Online application of neural networks to automate microscopy acquisition



Will use recurrent neural network and networks which model attention to find and classify cells based on user decisions made during training.

# Augmented reality microscope



Users will be updated with classifications using an augmented reality display which will overlay graphics when looking down eye-piece.

**wish me luck.**

**The End**

**Thank you for your attention**