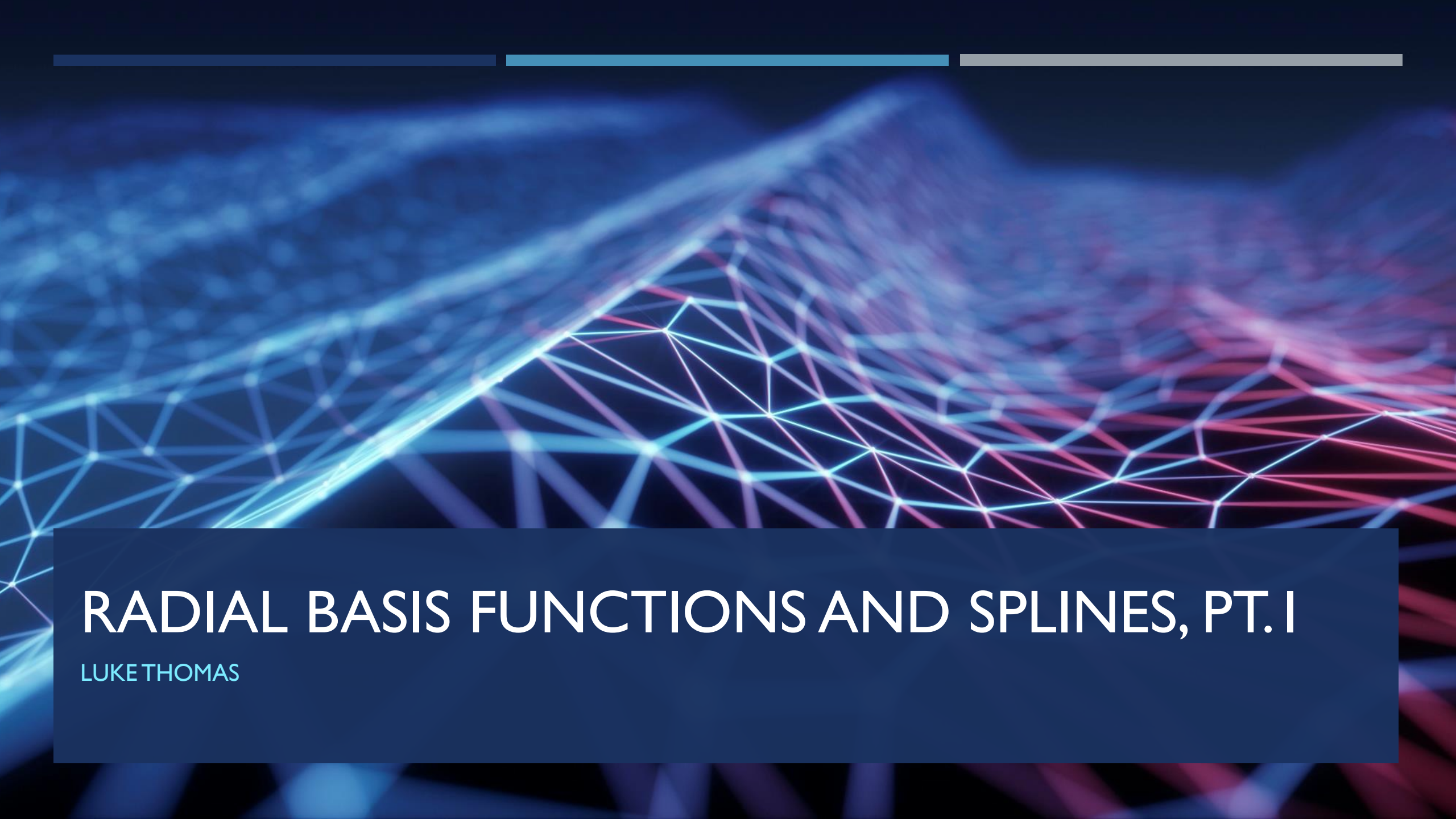


RADIAL BASIS FUNCTIONS AND SPLINES, PT. I

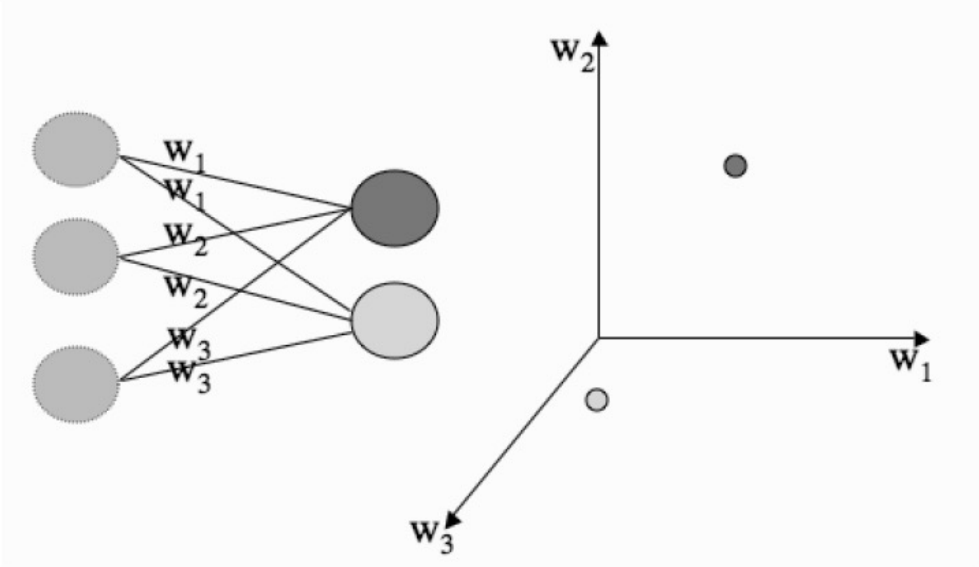
LUKE THOMAS

INTRODUCTION

- Previously in MLP's, we learnt that hidden nodes would activate when the sum of their inputs multiplied by their weights was greater than some threshold
- The pattern of activations was then used as input for the next layer in much the same way
- Today we cover an alternative method that activates neurons based on distance within the input space
- To understand this we will briefly go over two concepts (one of which was discussed previously), weight space and receptive fields

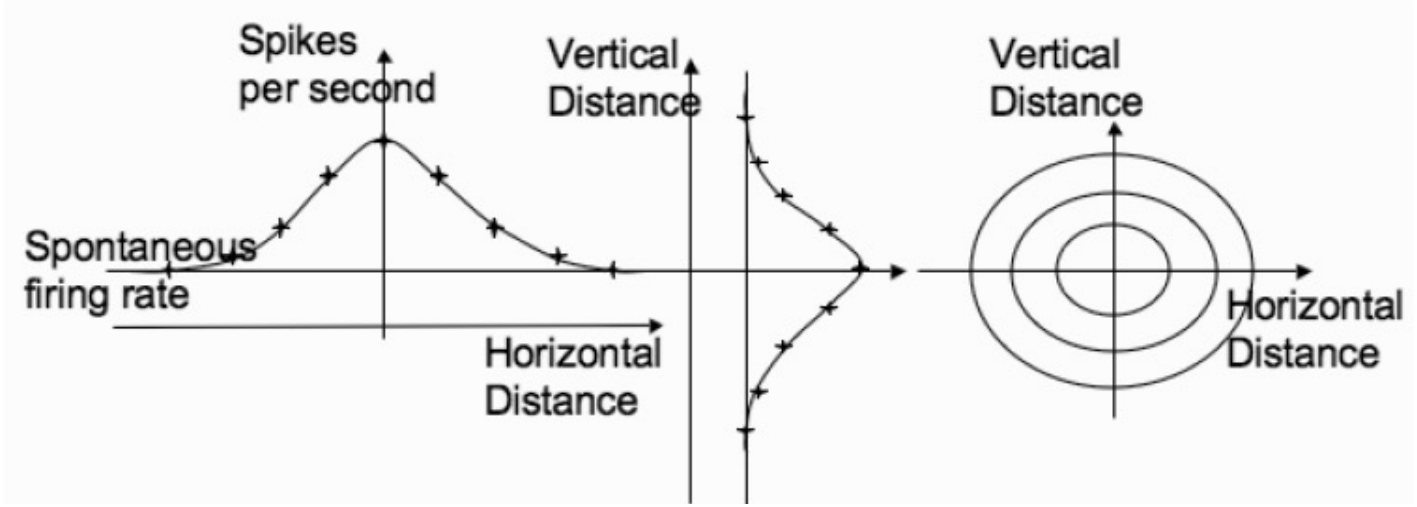
WEIGHT SPACE

- As discussed in chapter 2, weight space is essentially the plot of a neural networks nodes as a set of coordinates
- Here each axis represents a weight of the network and the nodes position along each represents the strength of the weight for that node
- Assuming inputs are normalised to be within the same range as the weights, we can also plot input neurons within this space
- This gives us a new way of learning, whereby we can measure the distance between inputs and nodes to determine how close an input needs to be in order to activate the neuron



RECEPTIVE FIELDS

- Fancy way of describing the distance at which a neuron can detect an input within weight space
- Book explains this with a retina analogy (weird I know)
- Basically if a star appeared within the left side of your view and you turned your head towards it the light-sensitive rods on that side will start to deactivate as it leaves their view
- However rods within the centre/right side of your retina would start to activate (spike) as the light got closer to them
- Diagram helps to understand this a lot more:

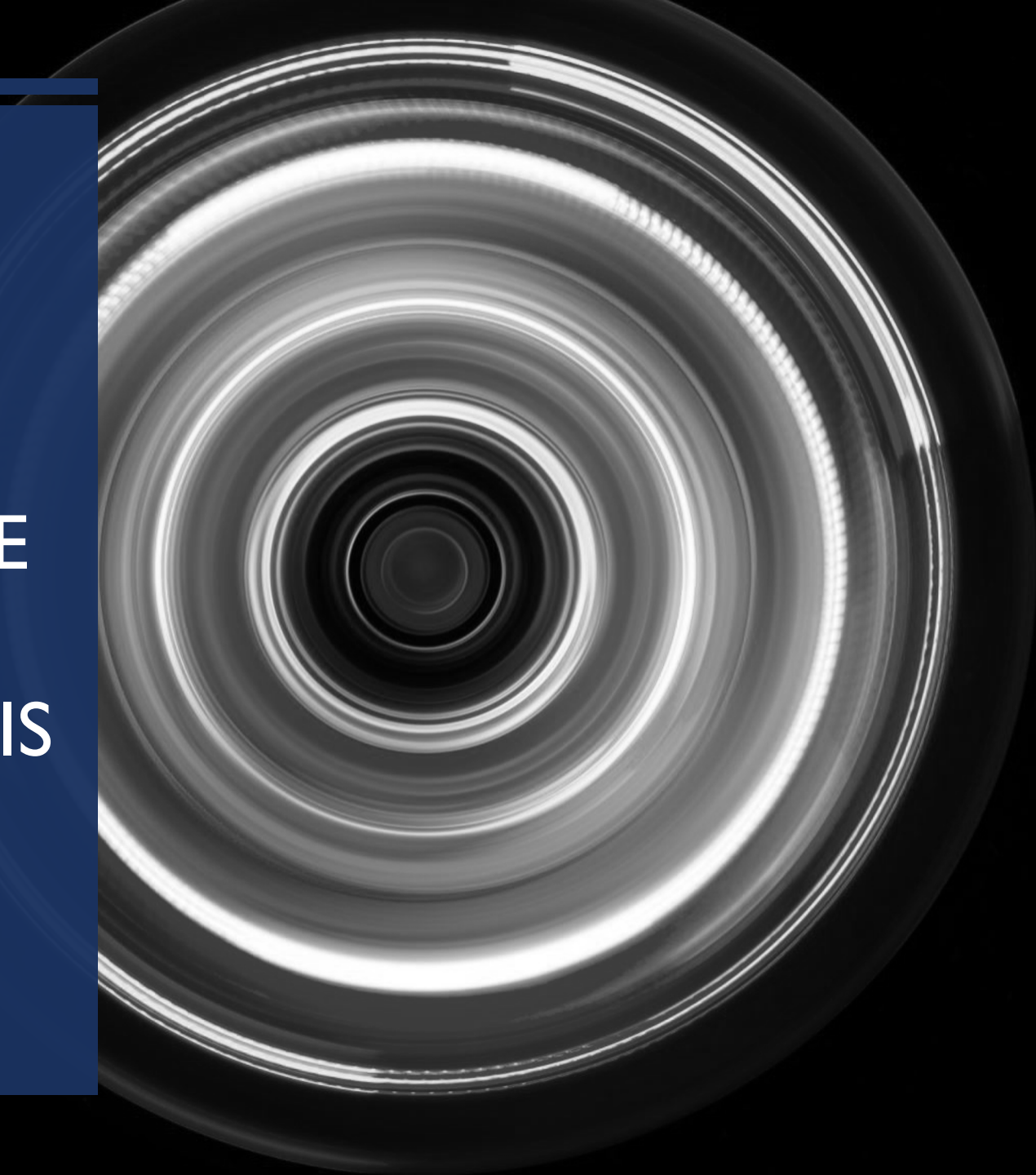


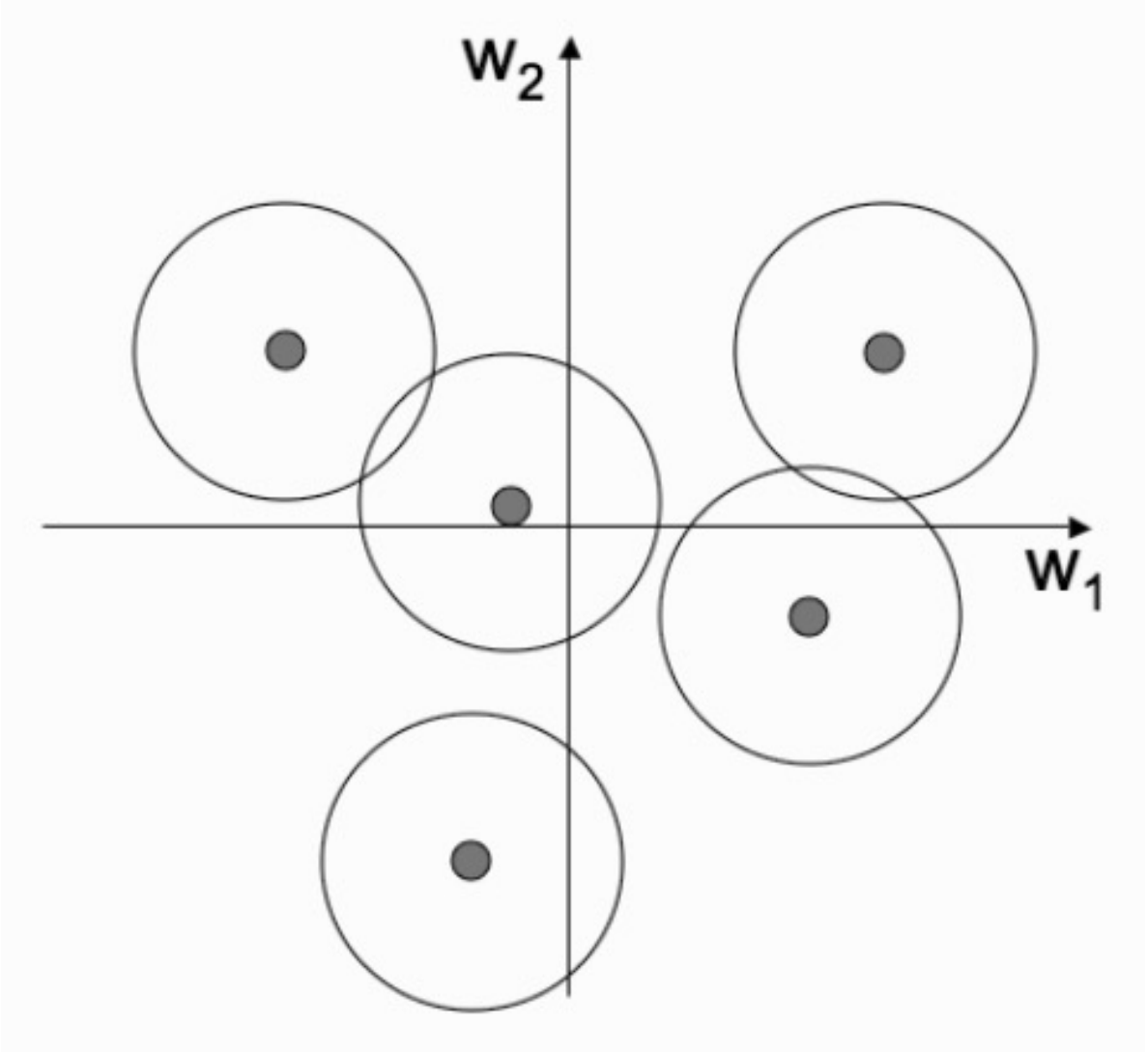
TRANSLATING THIS TO NEURAL NETWORKS

- So how do we define the receptive field for each neuron?
- Answer: Use a Gaussian for the activation function
- This can be represented with the following equation, where x and w represent the input and hidden neurons location in weight space respectively, and sigma controls the width of the gaussian

$$g(\mathbf{x}, \mathbf{w}, \sigma) = \exp\left(\frac{-\|\mathbf{x} - \mathbf{w}\|^2}{2\sigma^2}\right)$$

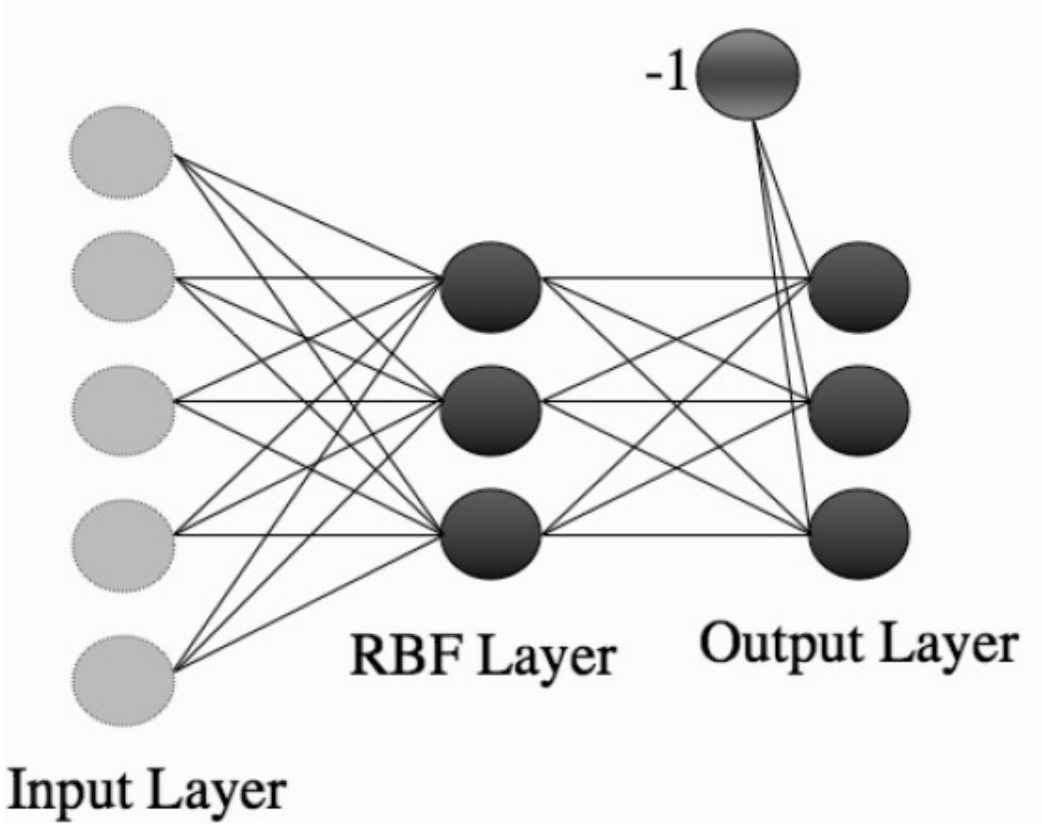
HOW CAN WE VISUALISE THE EFFECT OF THIS FUNCTION?




w_2
w_1

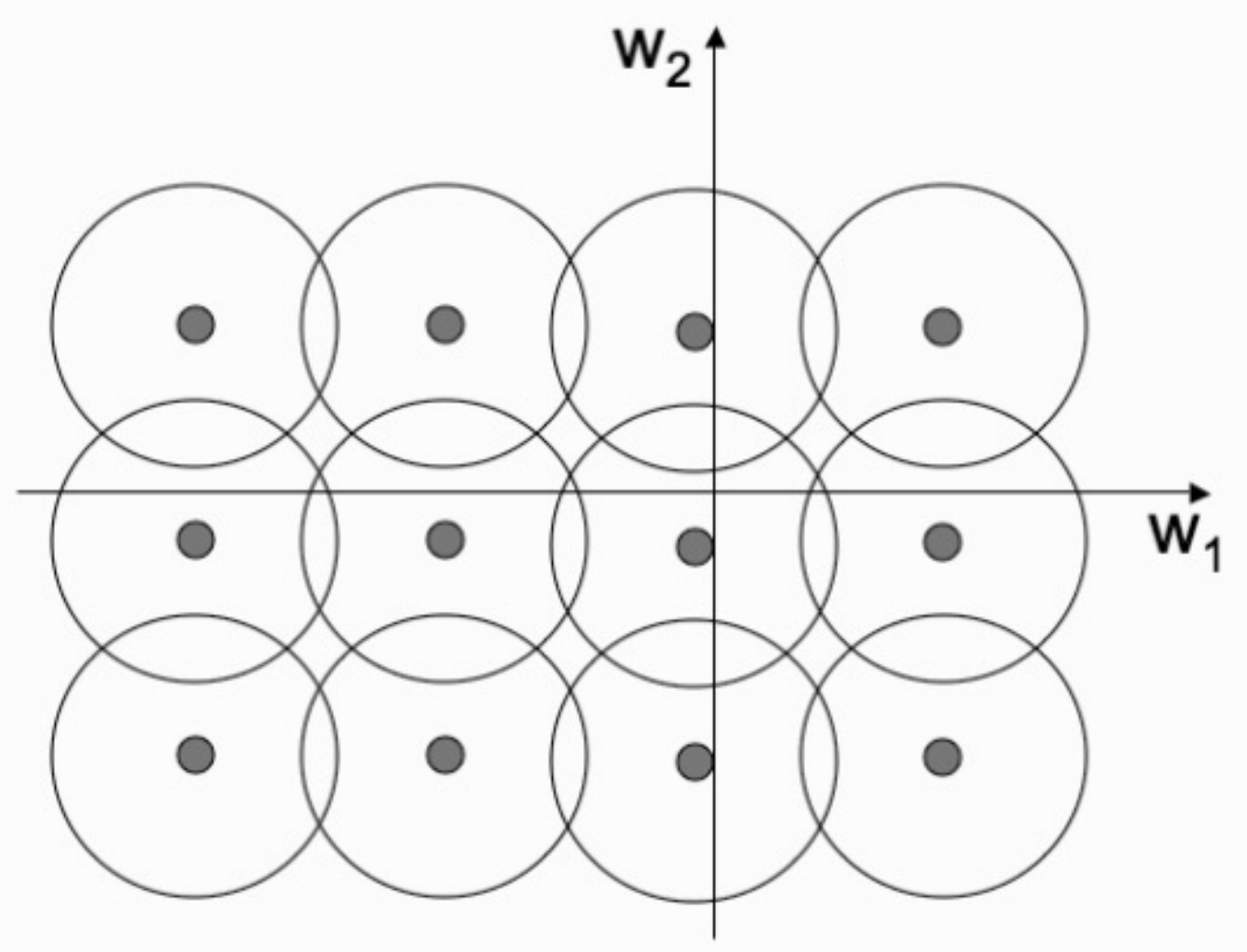
THE RADIAL BASIS FUNCTION (RBF) NETWORK

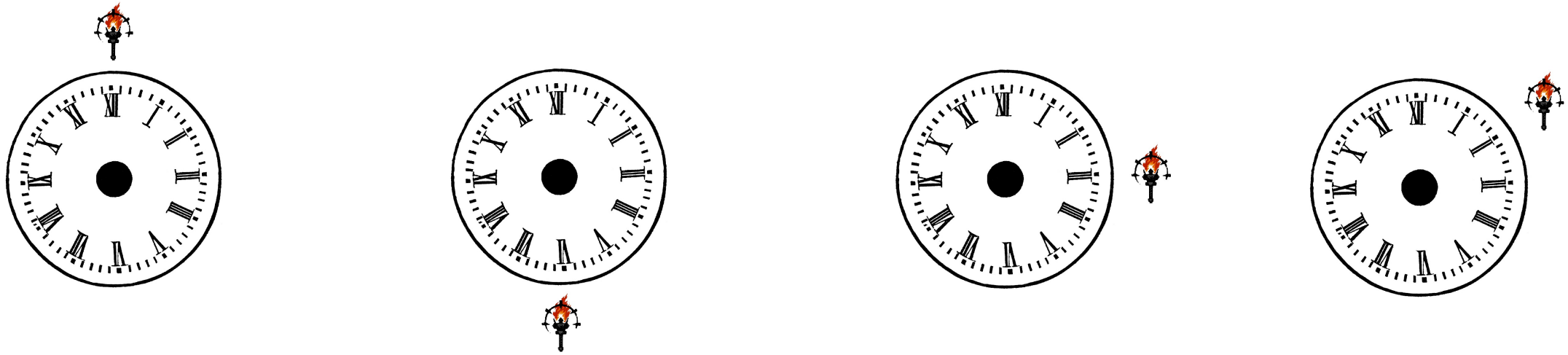
- So now that we have discussed the concept of distance based neuron activations and how to achieve them, we move on to an architecture that implements this idea, the RBF
- The RBF simply makes use of a hidden layer consisting of those neurons we saw in the last diagram
- The output layer is kept simple by using a normal perceptron that connects to the previous hidden layer via a set of weights
- Because we already know how to train a perceptron, training the output layers weights is straightforward
- The difficult part is training the weights (and therefore positions) of the hidden nodes



WHAT MAKES THIS ARCHITECTURE GOOD?

- For a start, it already improves upon the perceptron because here the outputs of the hidden layer are a non-linear function of the inputs, making the RBF a universal approximator just like the MLP
- To prove this claim though, you would want to make sure the entire weight space is covered with RBF nodes, ensuring the network can respond to all inputs appropriately (see the figure to the right)
- Compared to MLP, RBF networks use only one hidden layer and, instead of separating input space into global hyperplanes, instead try to match functions locally
- Book explains this with another analogy that describes moving in a certain direction based on the position of a light source…





LIGHT SOURCE ANALOGY EXPLAINED

TRAINING THE RBF

- For RBF we can use backpropagation via differentiating relevant activation functions, however here we have the benefit of simpler alternatives
- Remember that the RBF hidden nodes find a non-linear representation of the input and the output layer finds a linear combination of the hidden nodes to perform classification
- So we can split training into two parts, positioning the RBF nodes (first layer weights) and training the output as a perceptron
- Two main ways to position the RBF nodes:
  - Select positions based on random datapoints (Yes, really)
  - Use the results of k-means on the input to initialise RBF centres (Probably better)

ALTERNATIVLEY TRAINING FOR THE OUTPUT

- Because of the separation of learning, we can actually do something pretty wild with the output weights
- Let's say we took the activations of all hidden nodes and assembled them into a matrix G, where Gij is the activation of node j for input I
- The output of the whole network would be y = GW where W are the output layer weights
- Assuming we have the true output t we can perform the inverse of G to get the best possible weights for W like so: W = G^-1*t
- However, inverse only works if a matrix is square which would require the number of hidden nodes to match the input, which we don't want anyway as there would be serious overfitting
- Thankfully we can instead use what is known as a pseudo inverse G^+…

BRIEF EXPLANATION OF THE PSEUDO-INVERSE

- In short, pseudo-inverse G^+ = (G^T * G)^-1 * G^T
- Since the point of inverse G^-1 to a matrix G is that G^-1 * G = i where i is the identity matrix, the pseudo-inverse similarly satisfies G^+ * G = i
- If G is square the pseudo-inverse is the same as the true inverse
- Thankfully when coding you can just call np.linalg.pinv from the numpy library and not worry about all this

DECIDING THE SIZE OF RECEPTIVE FIELD

- If you remember the gaussian equation earlier you might recall the sigma variable defining the width of each RBF nodes receptive field
- We can just give all nodes the same size and test different sizes using a validation set to see which one works
- However it would be better to size them in a way that covers the entire weight space as we discussed before
- To do this we set the width based on maximum distance between locations of nodes (d) and the number of nodes (M), resulting in: $\sigma = d/\sqrt{2M}$
- Can also use normalise Gaussians so that there is atleast always one input firing: the node that is closest to the input, even if it's far away, resulting in a modified version of the gaussian equation: $g(\mathbf{x}, \mathbf{w}, \sigma) = \frac{\exp(-\|\mathbf{x} - \mathbf{w}\|/2\sigma^2)}{\sum_i \exp(-\|\mathbf{x} - \mathbf{w}_i\|/2\sigma^2)}$

RESULTS OF THE RBF

- Similarly to the MLP, RBF achieved over 90% classification accuracy on iris dataset when suing five RBF centres
- With both architectures the authors unfortunately failed to answer the question of how to pick the optimal number of hidden nodes without having to resort to repetitive network training
- The distance between input and hidden neurons in weight space for RBF was calculated using Euclidean distance, and it was found that increasing input dimensionality had a negative effect on the network
- Basically as dimensions increase covering the whole space with a fixed set of nodes becomes more difficult, so more are needed