Understanding Artificial Neural Networks

PIERIAN DATA

ANN

- A large part of this section will focus on theory behind many of the ideas we will implement with code.
- Let’s do a quick review of how we will gradually build an understanding of artificial neural networks.

ANN

- Theory Topics
  - Perceptron Model to Neural Networks
  - Activation Functions
  - Cost Functions
  - Feed Forward Networks
  - BackPropagation

ANN

- Coding Topics
  - TensorFlow 2.0 Keras Syntax
  - ANN with Keras
    - Regression
    - Classification
  - Exercises for Keras ANN
  - Tensorboard Visualizations

Let’s get started!

Perceptron Model

Perceptron model

- To begin understanding deep learning, we will build up our model abstractions:
  - Single Biological Neuron
  - Perceptron
  - Multi-layer Perceptron Model
  - Deep Learning Neural Network

Perceptron model

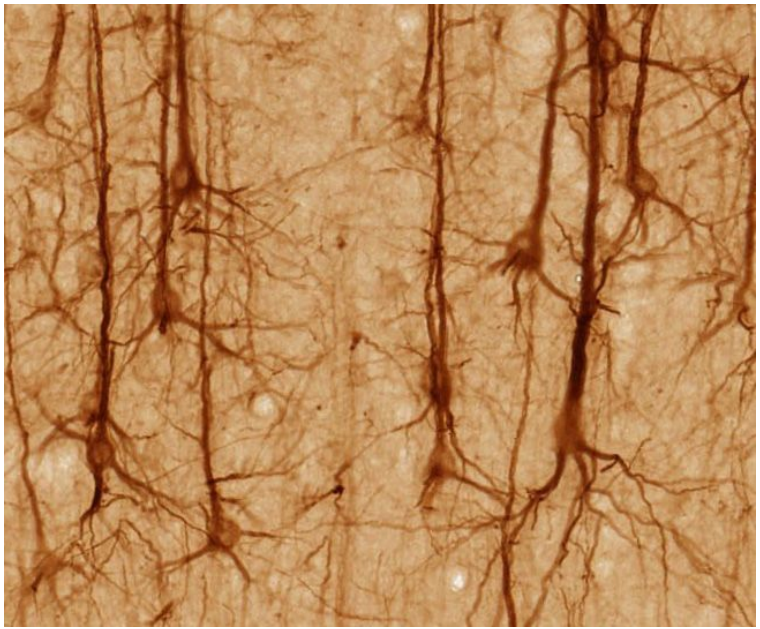
- As we learn about more complex models, we'll also introduce concepts, such as:
  - Activation Functions
  - Gradient Descent
  - BackPropagation

Perceptron model

- If the whole idea behind deep learning is to have computers artificially mimic biological natural intelligence, we should probably build a general understanding of how biological neurons work!

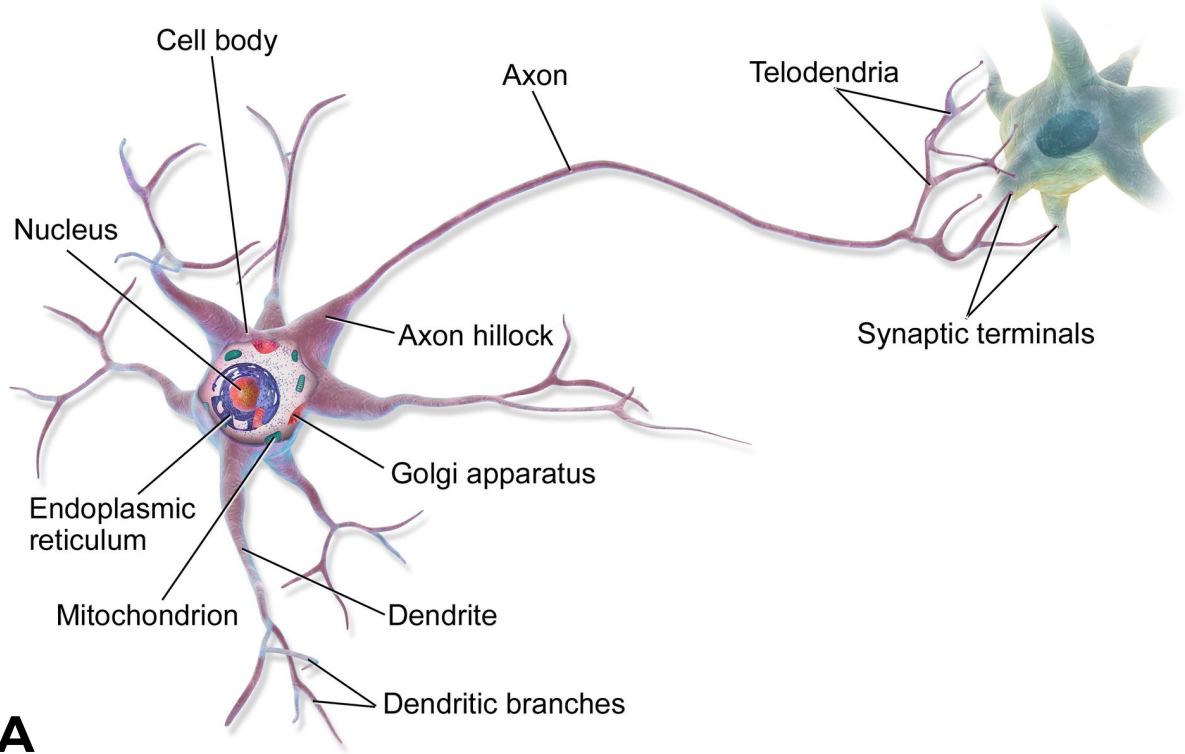
Perceptron model

- Stained Neurons in cerebral cortex



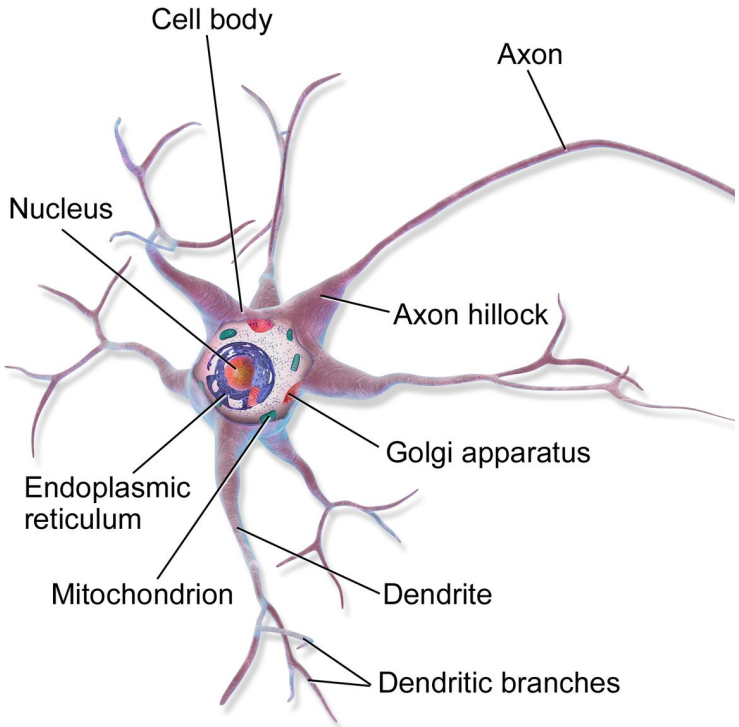
Perceptron model

- Illustration of biological neurons



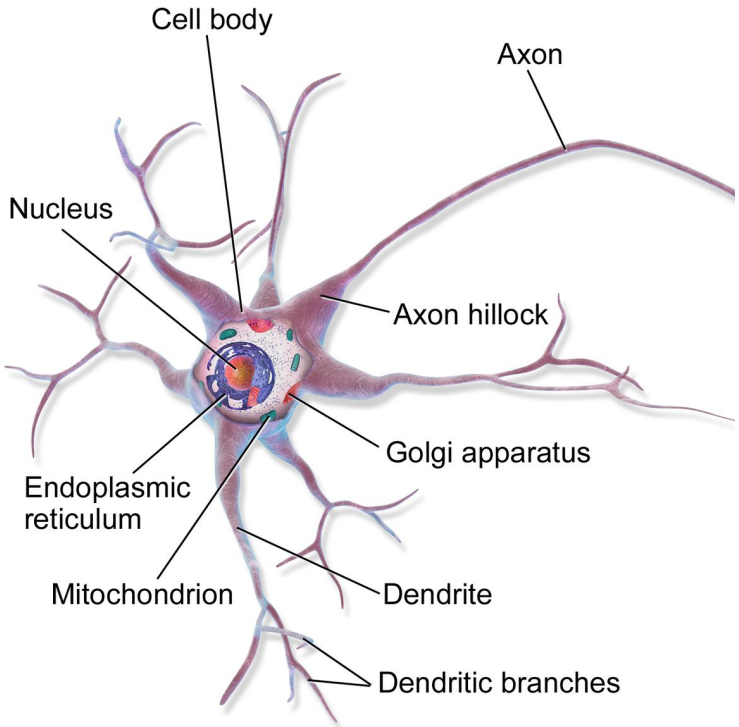
Perceptron model

- Illustration of biological neurons



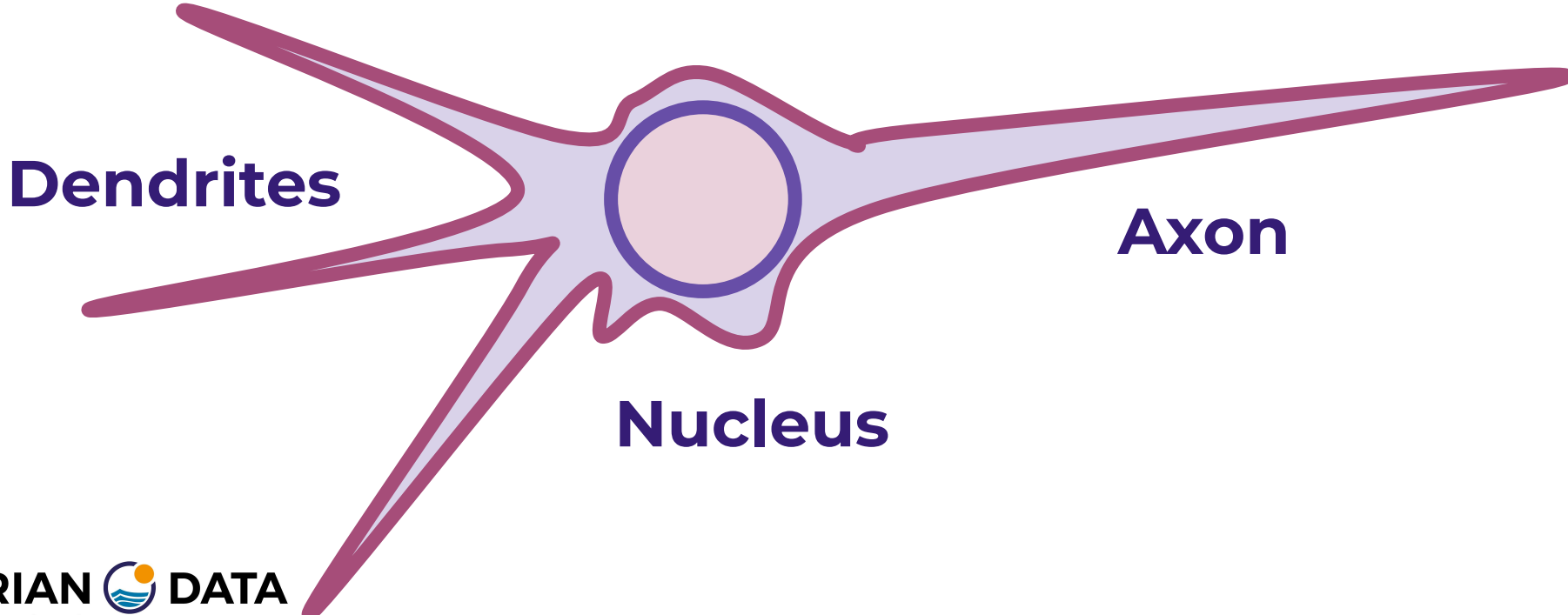
Perceptron model

- Let’s really simplify this!



Perceptron model

- Simplified Biological Neuron Model



Perceptron model

- A perceptron was a form of neural network introduced in 1958 by Frank Rosenblatt.
- Amazingly, even back then he saw huge potential:
    - "...perceptron may eventually be able to learn, make decisions, and translate languages."

Perceptron model

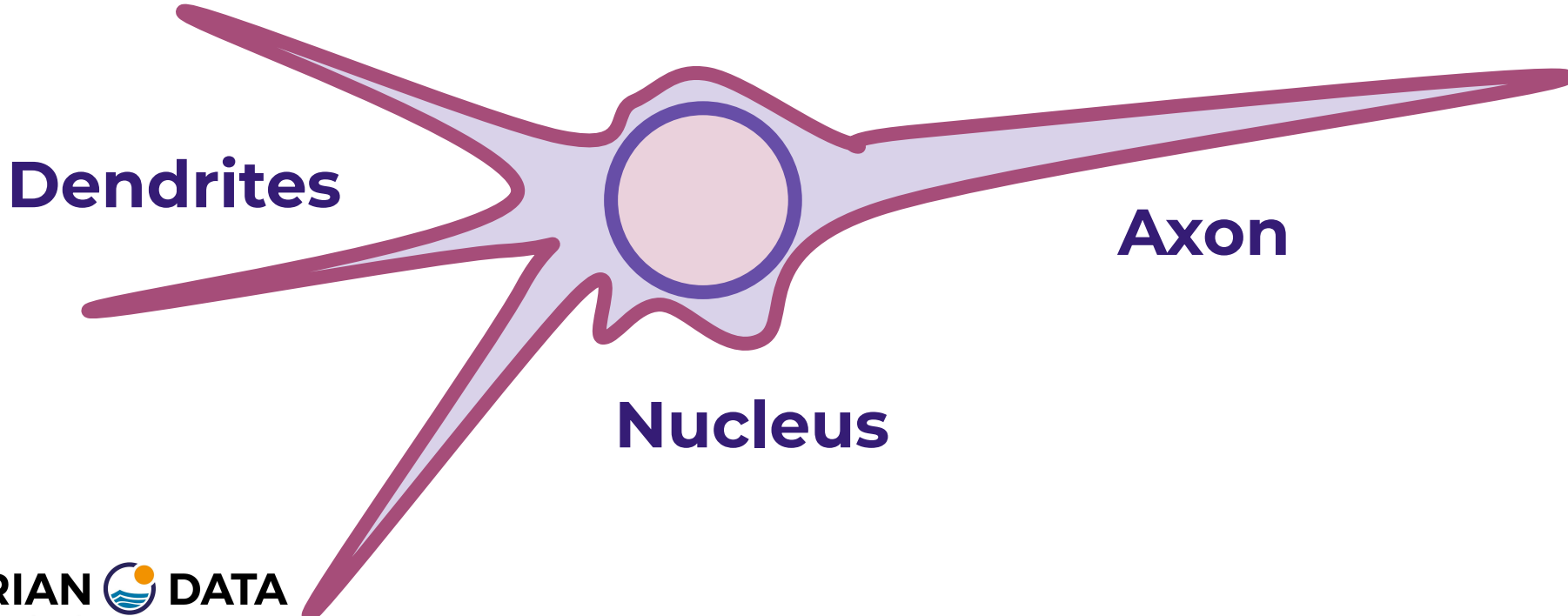
- However, in 1969 Marvin Minsky and Seymour Papert's published their book ***Perceptrons***.
- It suggested that there were severe limitations to what perceptrons could do.
- This marked the beginning of what is known as the AI Winter, with little funding into AI and Neural Networks in the 1970s.

Perceptron model

- Fortunately for us, we now know the amazing power of neural networks, which all stem from the simple perceptron model, so let’s head back and convert our simple biological neuron model into the perceptron model.

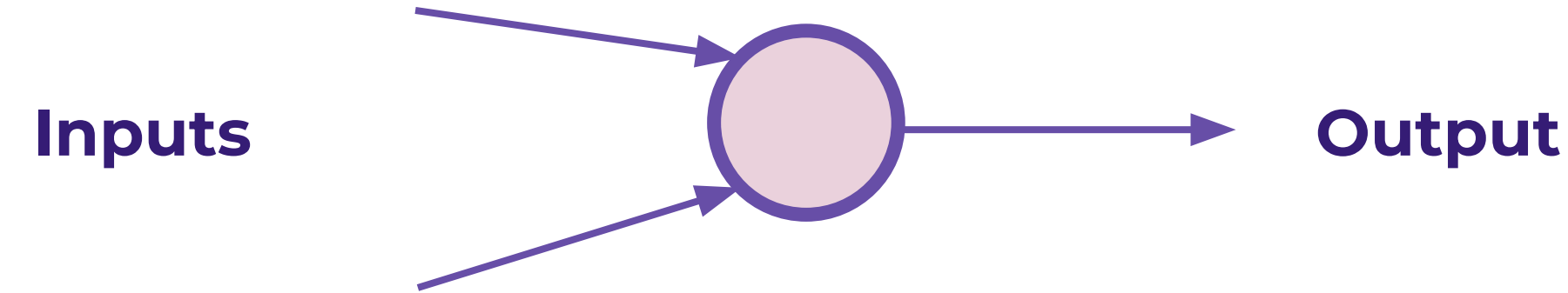
Perceptron model

- Perceptron Model



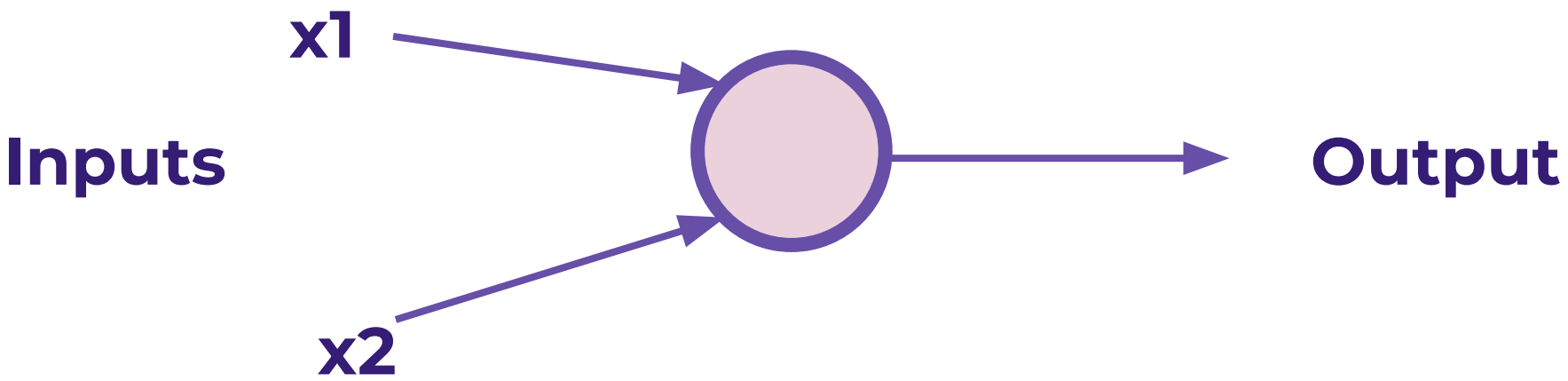
Perceptron model

- Perceptron Model



Perceptron model

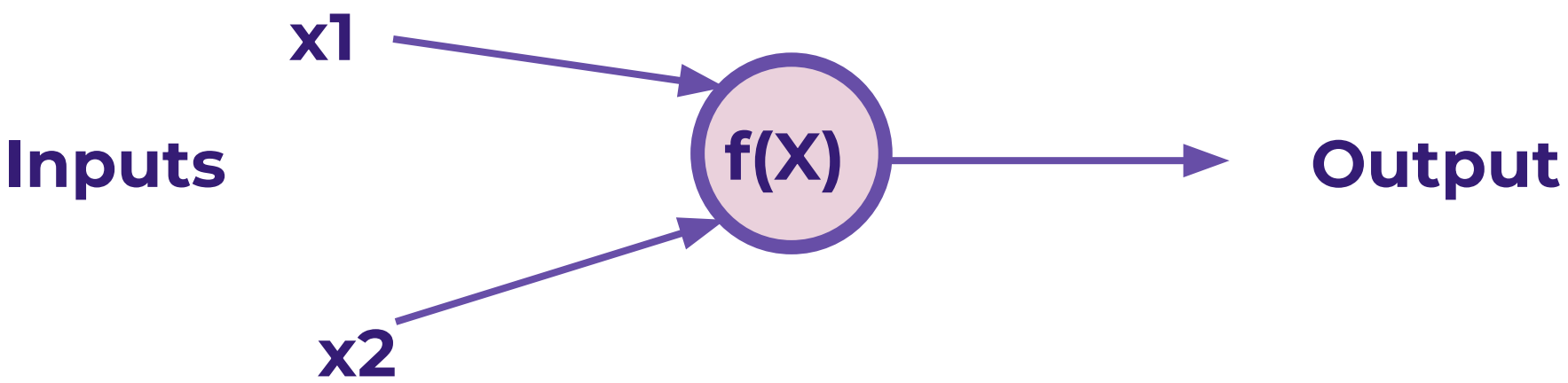
- Let's work through a simple example



PIERIAN DATA

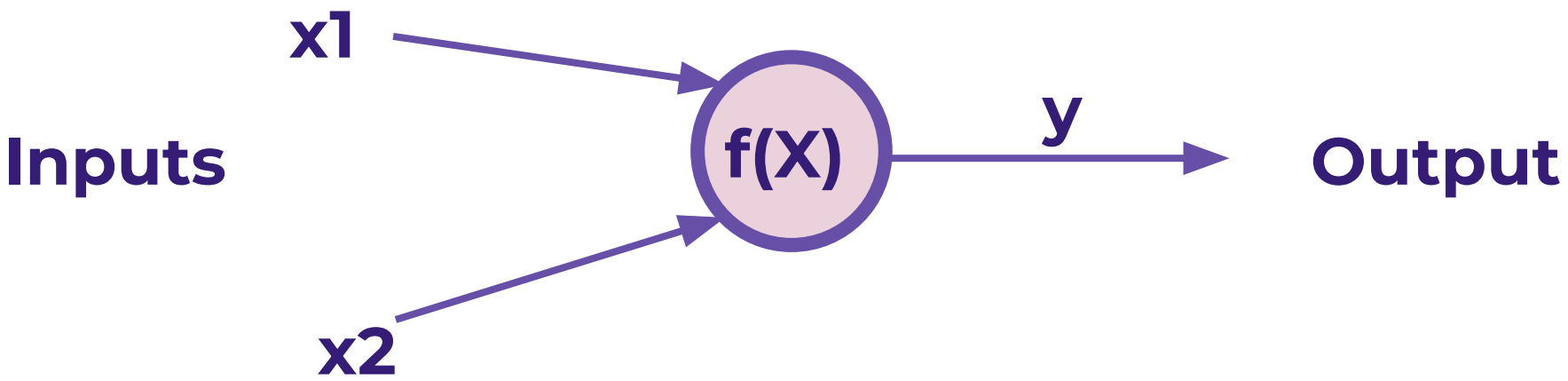
Perceptron model

- Let's work through a simple example



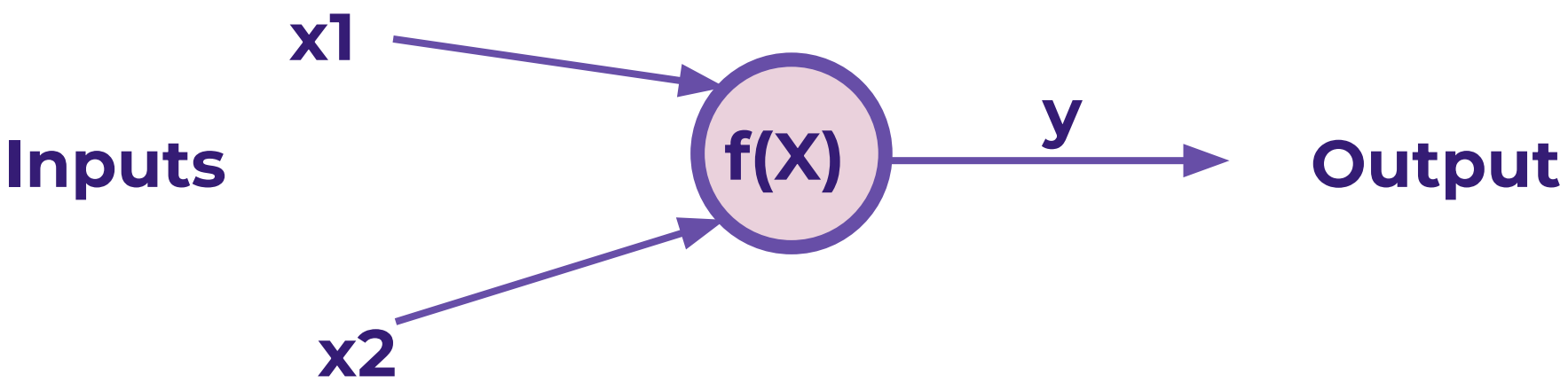
Perceptron model

- Let's work through a simple example



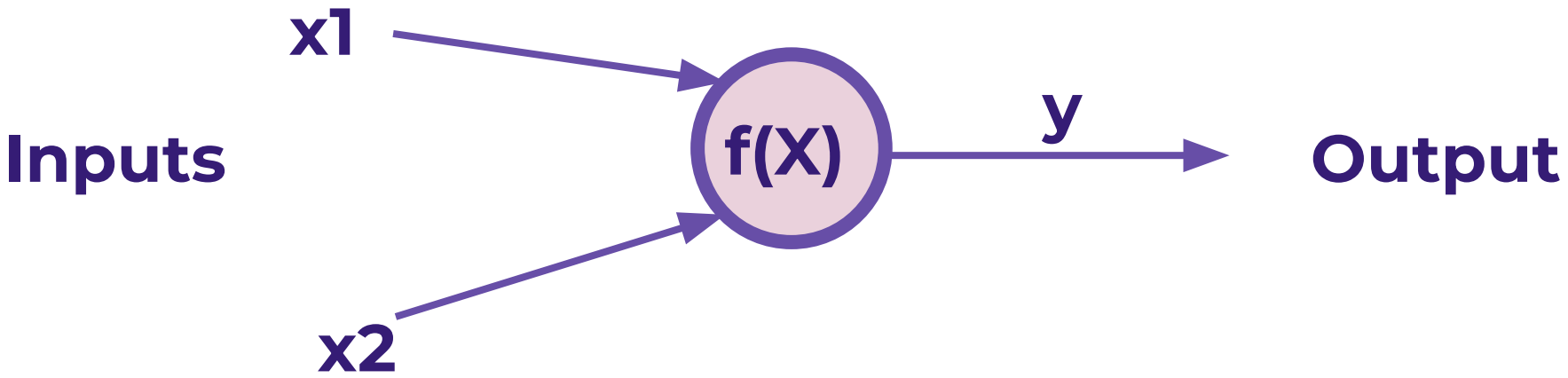
Perceptron model

- If f(X) is just a sum, then y=x1+x2



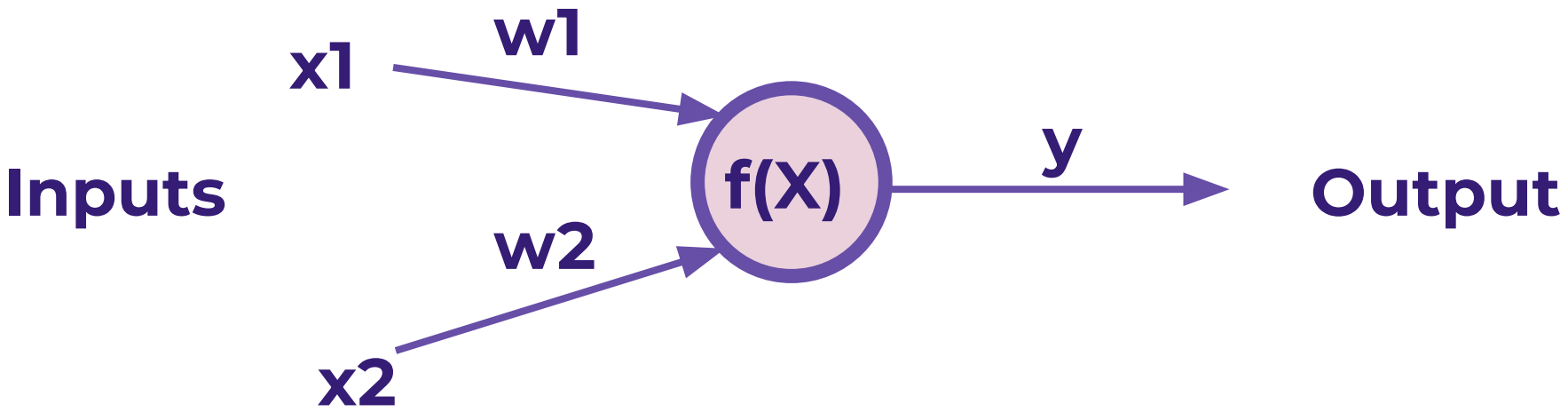
Perceptron model

- Realistically, we would want to be able to adjust some parameter in order to “learn”



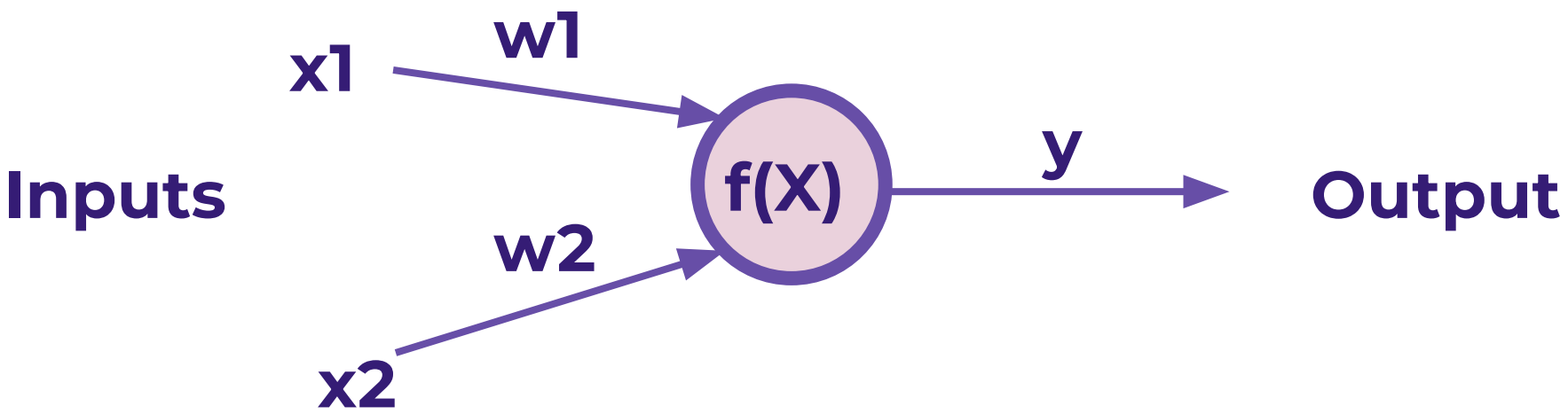
Perceptron model

- Let's add an adjustable weight we multiply against x



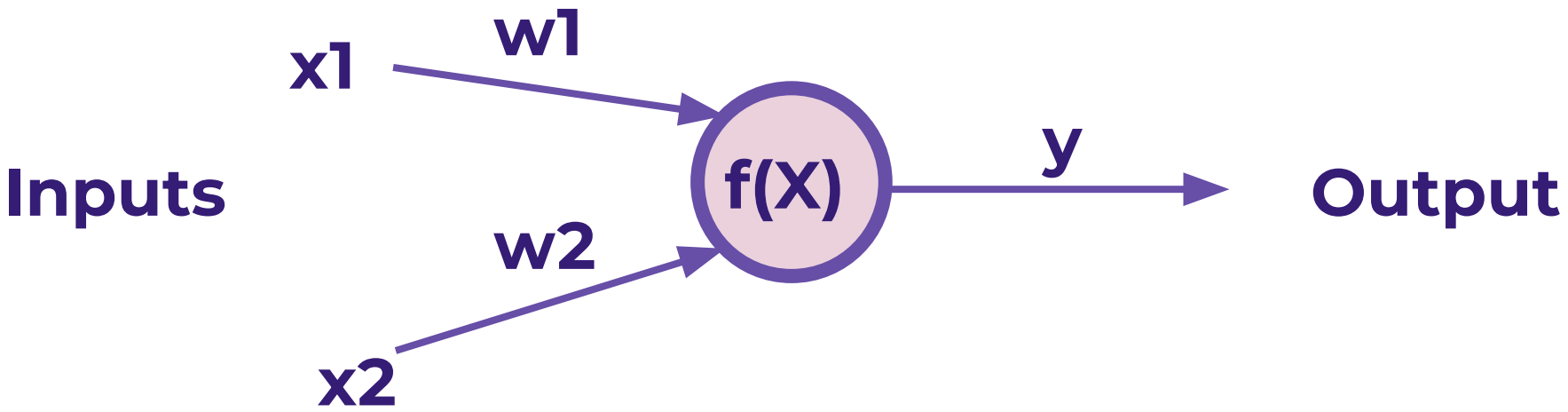
Perceptron model

- Now **y = x1w1 + x2w2**



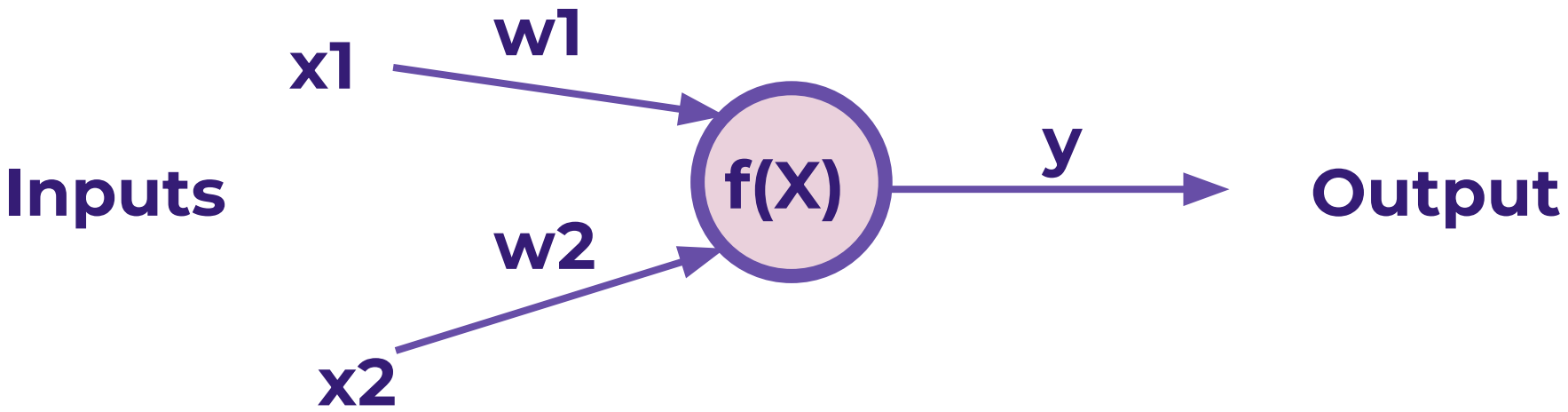
Perceptron model

- We could update the **weights** to effect **y**



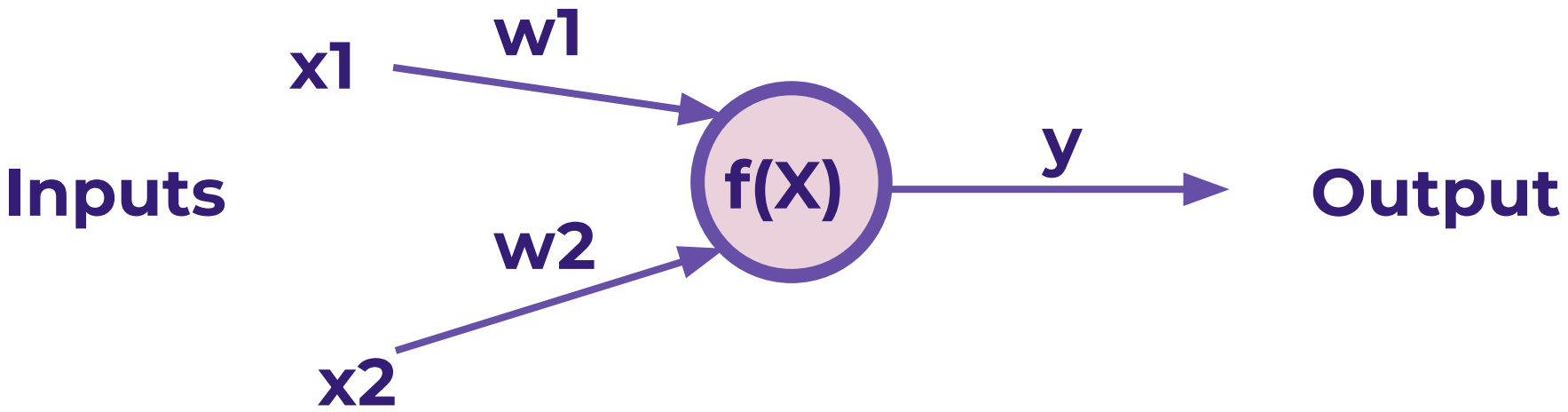
Perceptron model

- But what if an **x** is zero? **w** won't change anything!



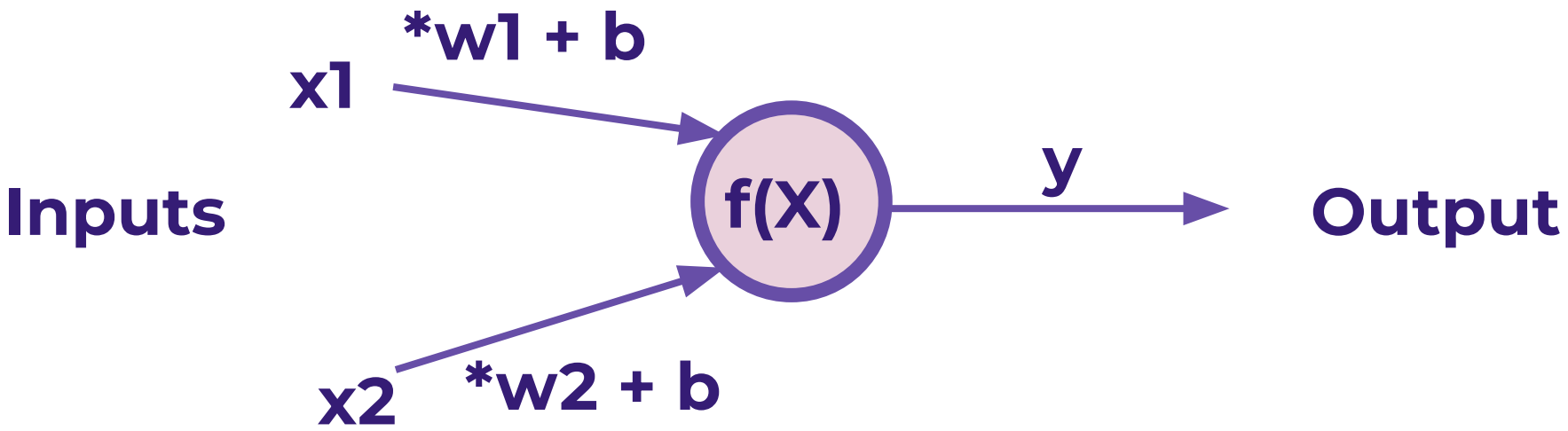
Perceptron model

- Let's add in a **bias** term **b** to the inputs.



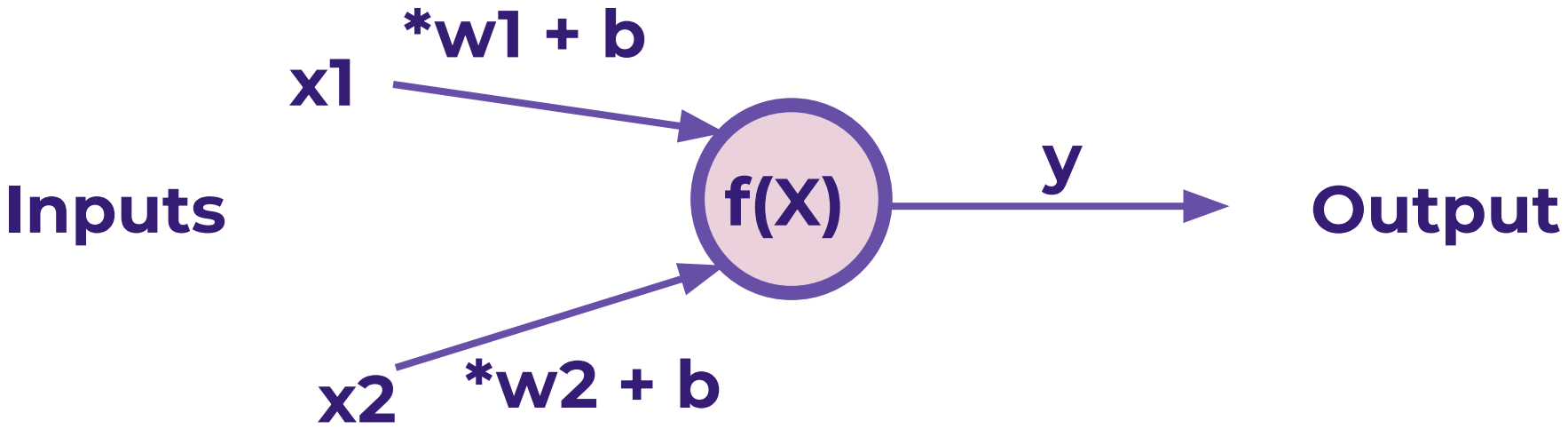
Perceptron model

- Let's add in a **bias** term **b** to the inputs.



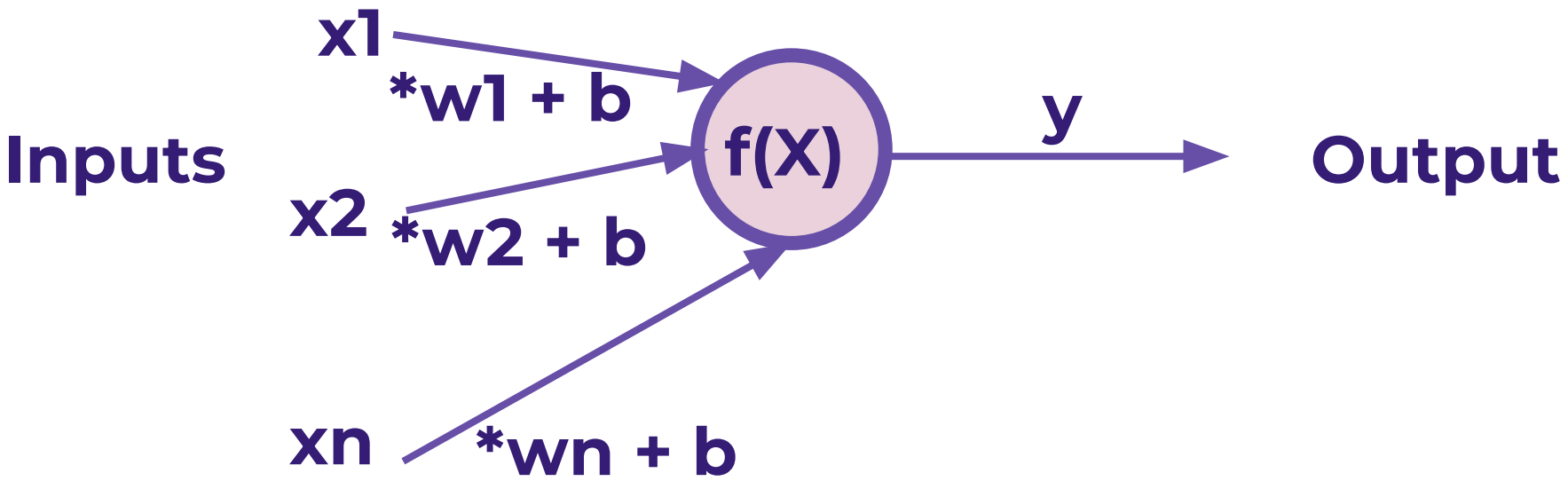
Perceptron model

- y = (x1w1 + b) + (x2w2 + b)



Perceptron model

- We can expand this to a generalization:



Perceptron model

- We've been able to model a biological neuron as a simple perceptron! Mathematically our generalization was:

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$

Perceptron model

- Later on we will see how we can expand this model to have X be a **tensor** of information (an n-dimensional matrix).

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$

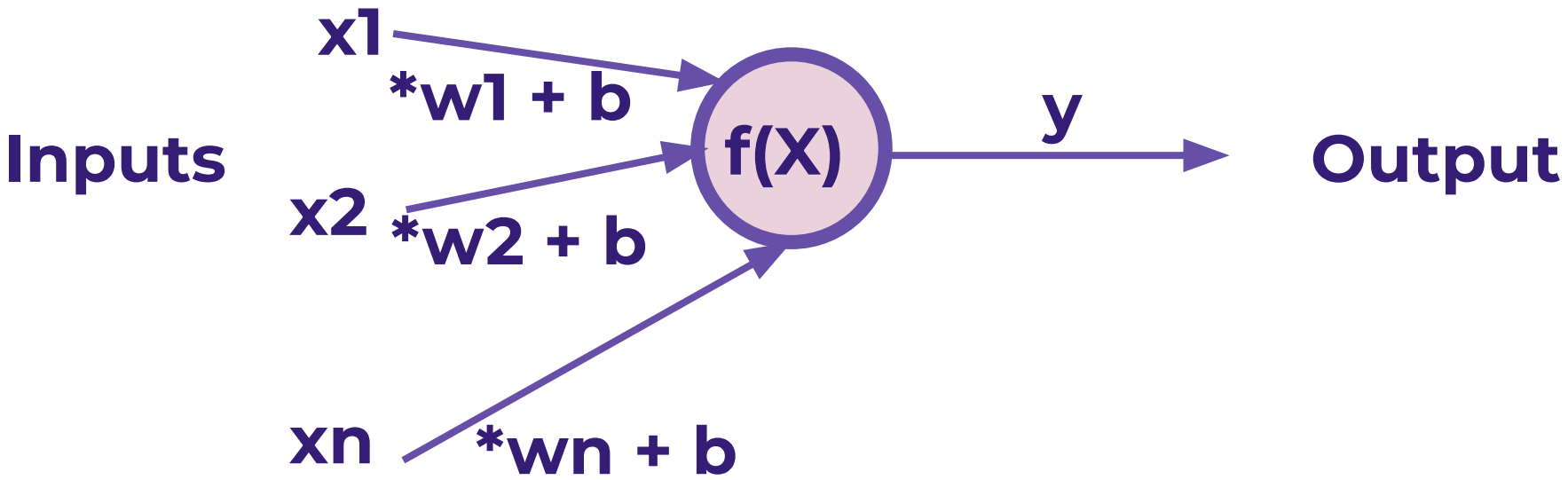
Perceptron model

- Also we'll see we can even simplify the bias to be at a layer level instead of a bias per input.

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$

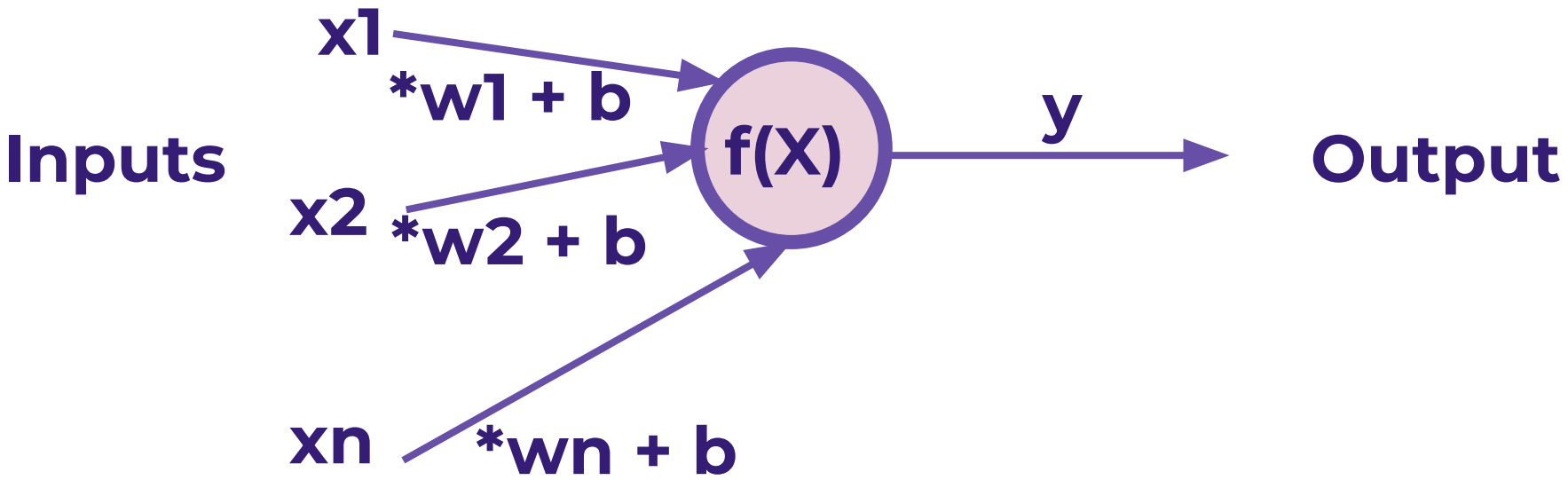
Perceptron model

- Notice how at the end we sum a bunch of biases...



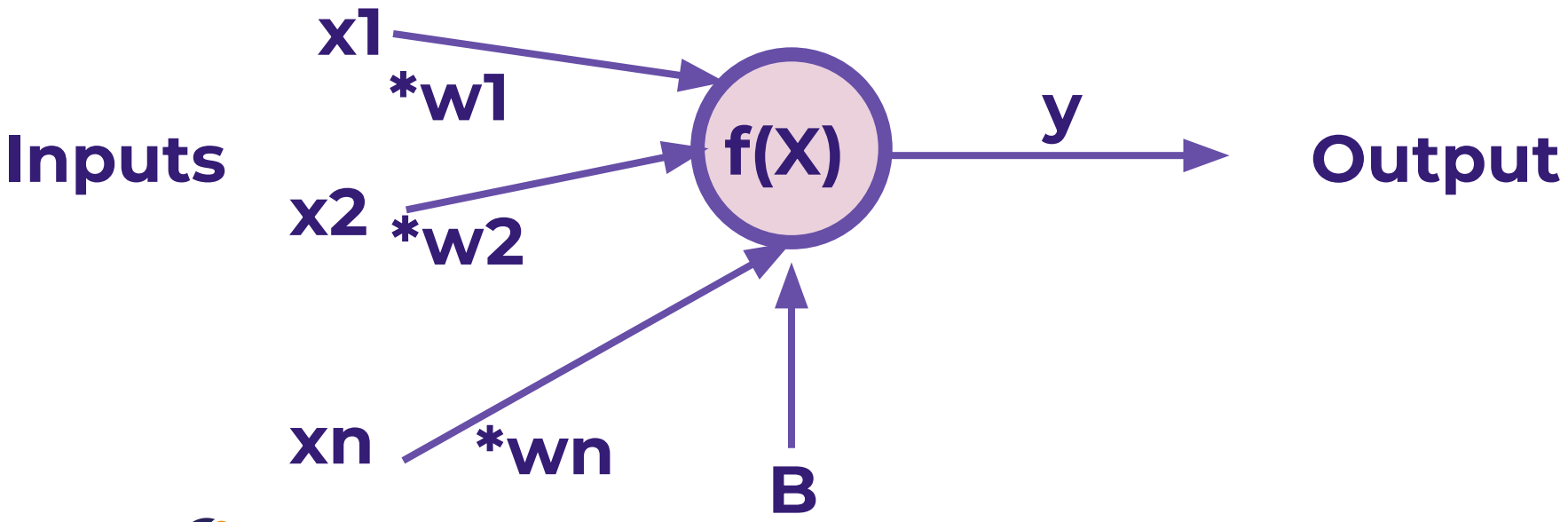
Perceptron model

- Theoretically for any number of biases, there exists a bias that is the sum.



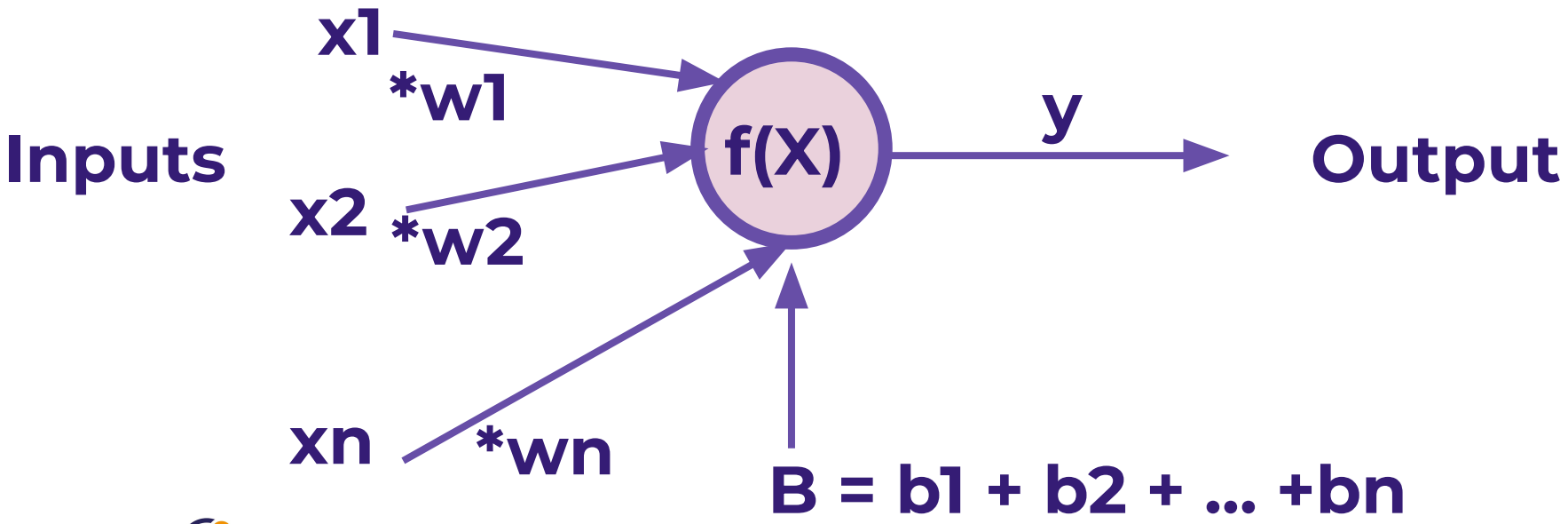
Perceptron model

- Theoretically for any number of biases, there exists a bias that is the sum.



Perceptron model

- Theoretically for any number of biases, there exists a bias that is the sum.



Perceptron model

- Let’s review what we learned:
    - We understand the very basics of a biological neuron
    - We saw how we can create a simple perceptron model replicating the core concepts behind a neuron.

Neural Networks

Neural Networks

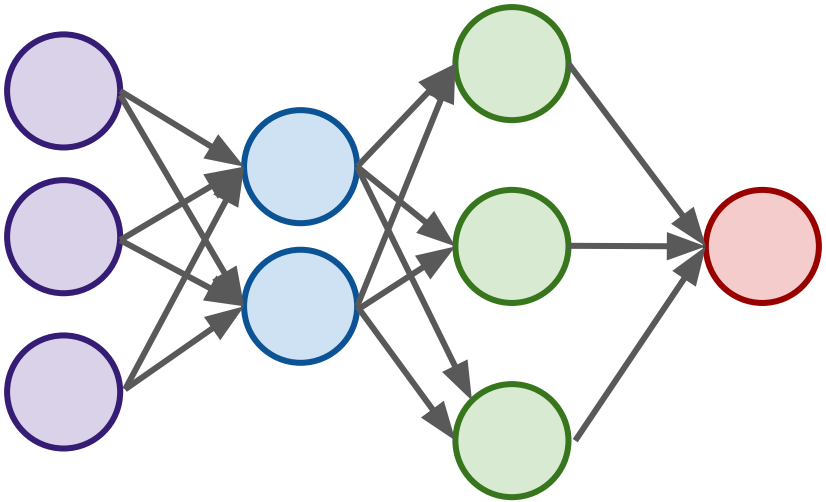
- A single perceptron won't be enough to learn complicated systems.
- Fortunately, we can expand on the idea of a single perceptron, to create a multi-layer perceptron model.

Neural Networks

- A single perceptron won’t be enough to learn complicated systems.
- Fortunately, we can expand on the idea of a single perceptron, to create a multi-layer perceptron model.
- We’ll also introduce the idea of activation functions.

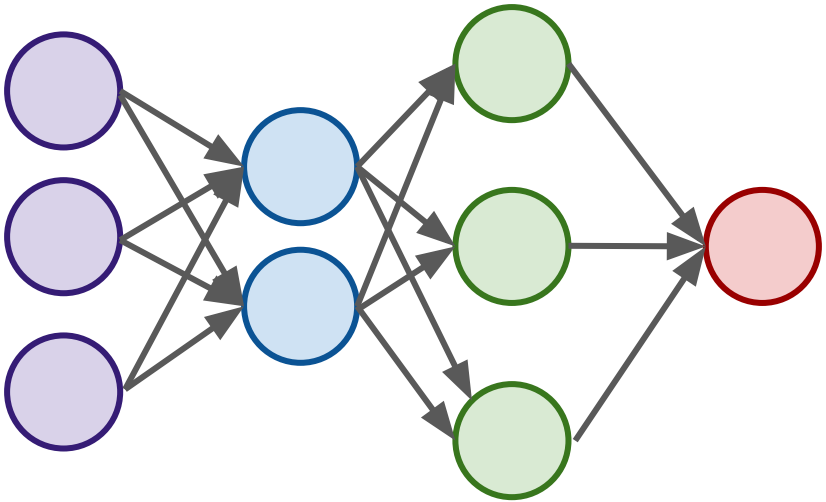
Neural Networks

- To build a network of perceptrons, we can connect layers of perceptrons, using a **multi-layer perceptron model**.



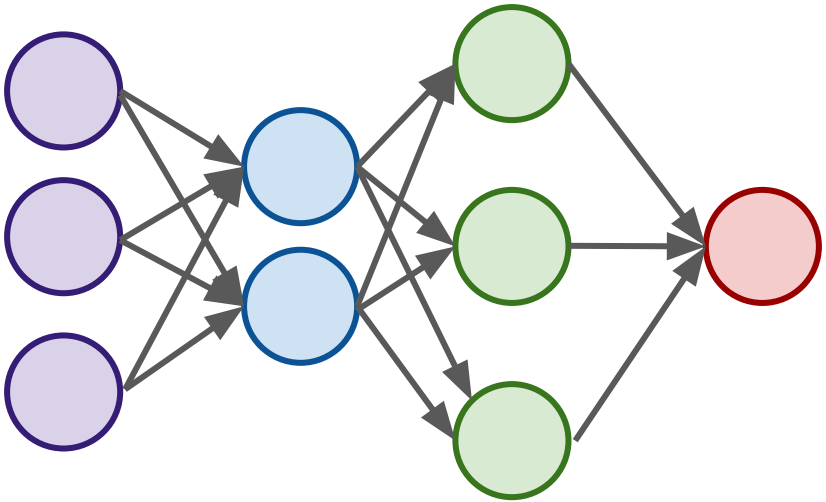
Neural Networks

- The outputs of one perceptron are directly fed into as inputs to another perceptron.



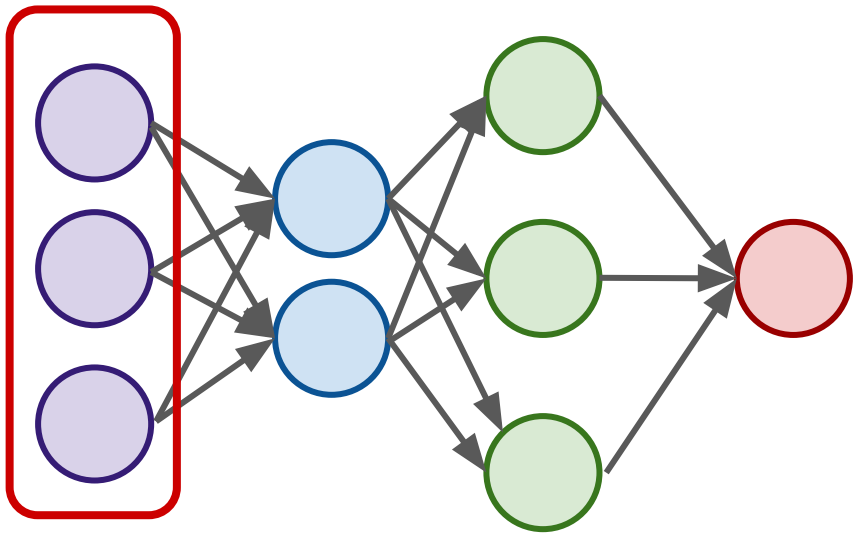
Neural Networks

- This allows the network as a whole to learn about interactions and relationships between features.



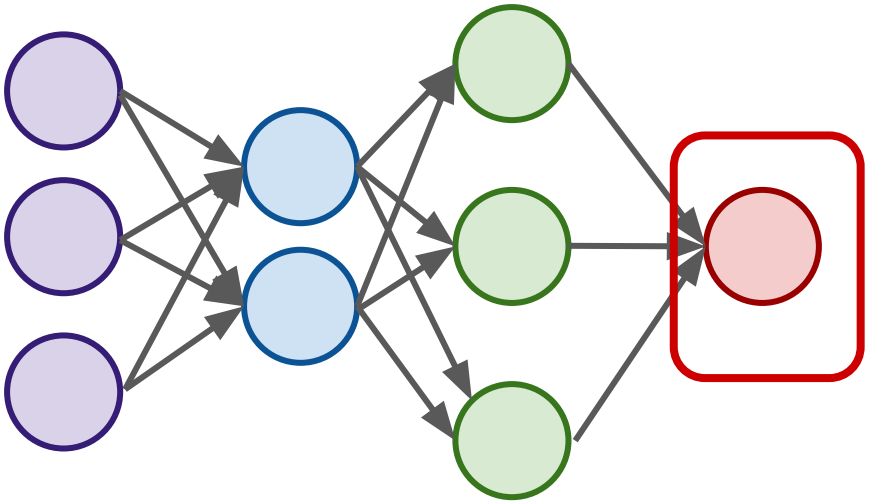
Neural Networks

- The first layer is the **input layer**



Neural Networks

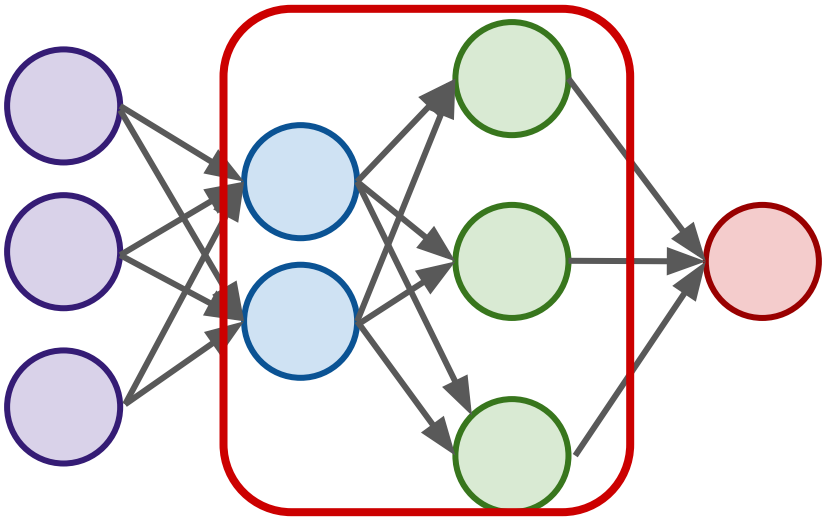
- The last layer is the **output layer.**
- Note: This last layer can be more than one neuron



PIERIAN DATA

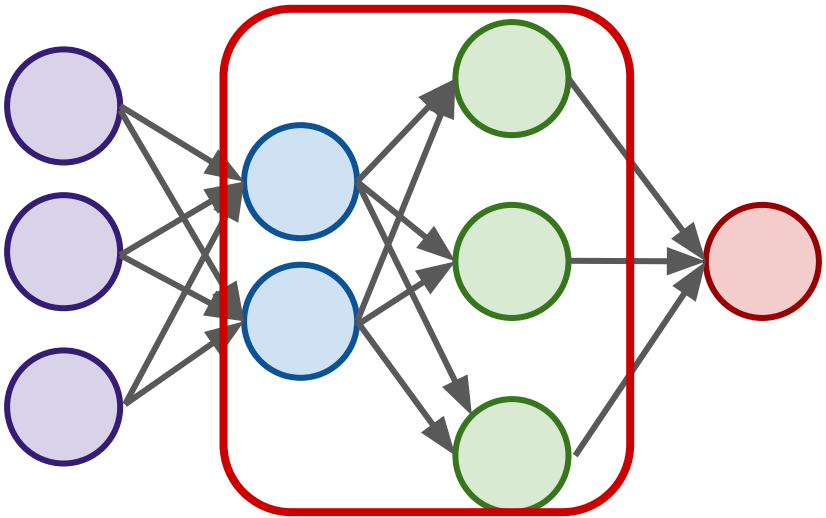
Neural Networks

- Layers in between the input and output layers are the **hidden layers.**



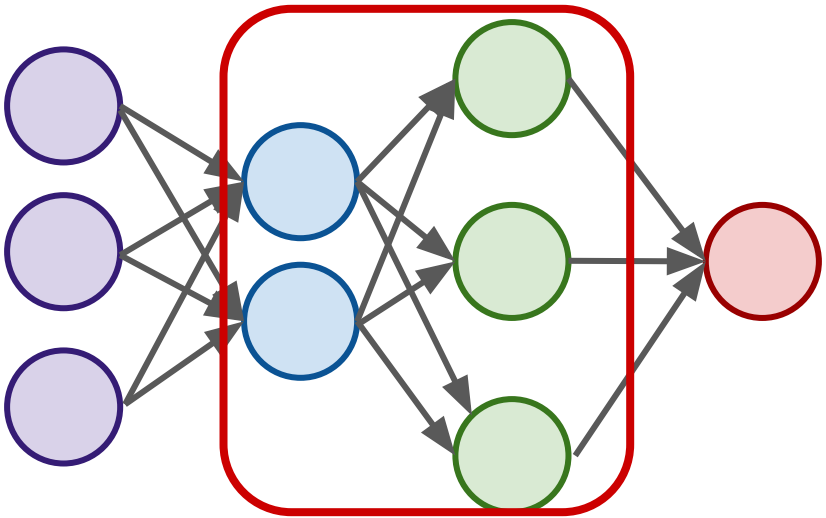
Neural Networks

- Hidden layers are difficult to interpret, due to their high interconnectivity and distance away from known input or output values.



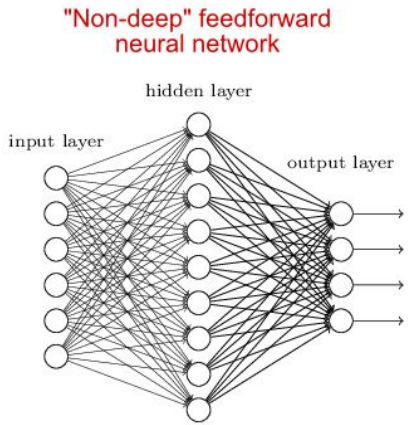
Neural Networks

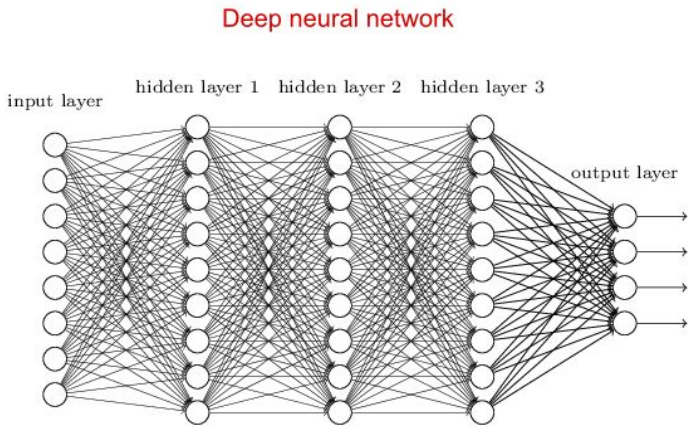
- Neural Networks become **“deep neural networks”** if then contain 2 or more hidden layers.



Neural Networks

- Neural Networks become **“deep neural networks”** if then contain 2 or more hidden layers.





Neural Networks

- Terminology:
  - Input Layer: First layer that directly accepts real data values
  - Hidden Layer: Any layer between input and output layers
  - Output Layer: The final estimate of the output.

Neural Networks

- What is incredible about the neural network framework is that it can be used to approximate any function.
- Zhou Lu and later on Boris Hanin proved mathematically that Neural Networks can approximate any convex continuous function.

Neural Networks

- For more details on this check out the Wikipedia page for “Universal Approximation Theorem”

Neural Networks

- Previously in our simple model we saw that the perceptron itself contained a very simple summation function f(x).
- For most use cases however that won't be useful, we'll want to be able to set constraints to our output values, especially in classification tasks.

Neural Networks

- In a classification tasks, it would be useful to have all outputs fall between 0 and 1.
- These values can then present probability assignments for each class.
- In the next lecture, we'll explore how to use **activation functions** to set boundaries to output values from the neuron.

Activation Functions

Neural Networks

- Recall that inputs **x** have a weight **w** and a bias term **b** attached to them in the perceptron model.
- Which means we have
    - **x*w + b**

Neural Networks

- Which means we have
  - **x*w + b**
  - Clearly **w** implies how much weight or strength to give the incoming input.
  - We can think of **b** as an offset value, making **x*w** have to reach a certain threshold before having an effect.

Neural Networks

- For example if **b= -10**
  - **x*w + b**
  - Then the effects of **x*w** won't really start to overcome the bias until their product surpasses 10.
  - After that, then the effect is solely based on the value of **w.**
  - Thus the term "bias"

Neural Networks

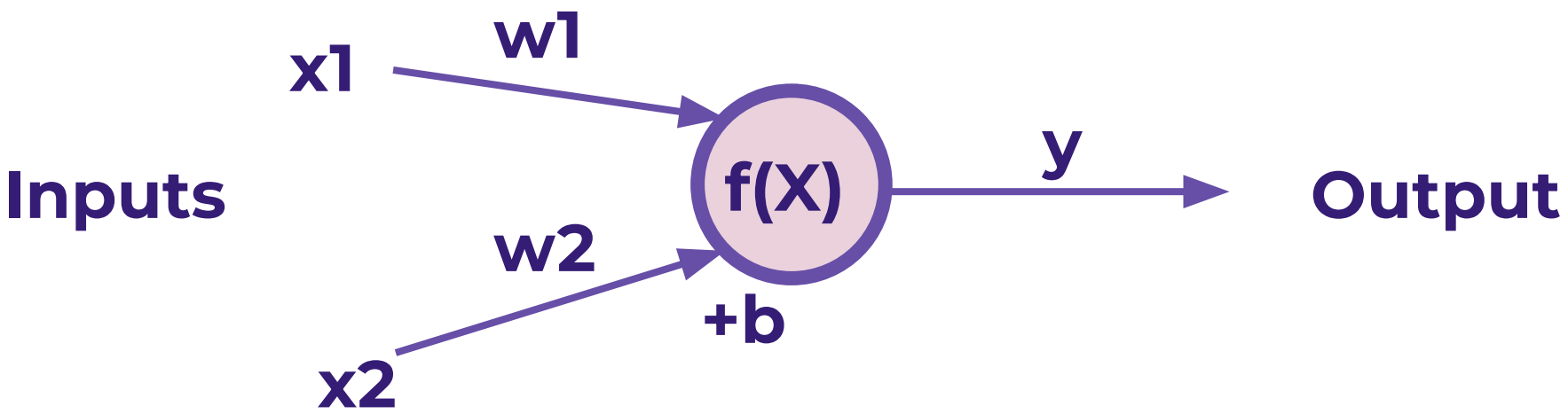
- Next we want to set boundaries for the overall output value of:
    - **x*w + b**
- We can state:
    - **z = x*w + b**
- And then pass **z** through some activation function to limit its value.

Neural Networks

- A lot of research has been done into activation functions and their effectiveness.
- Let’s explore some common activation functions.

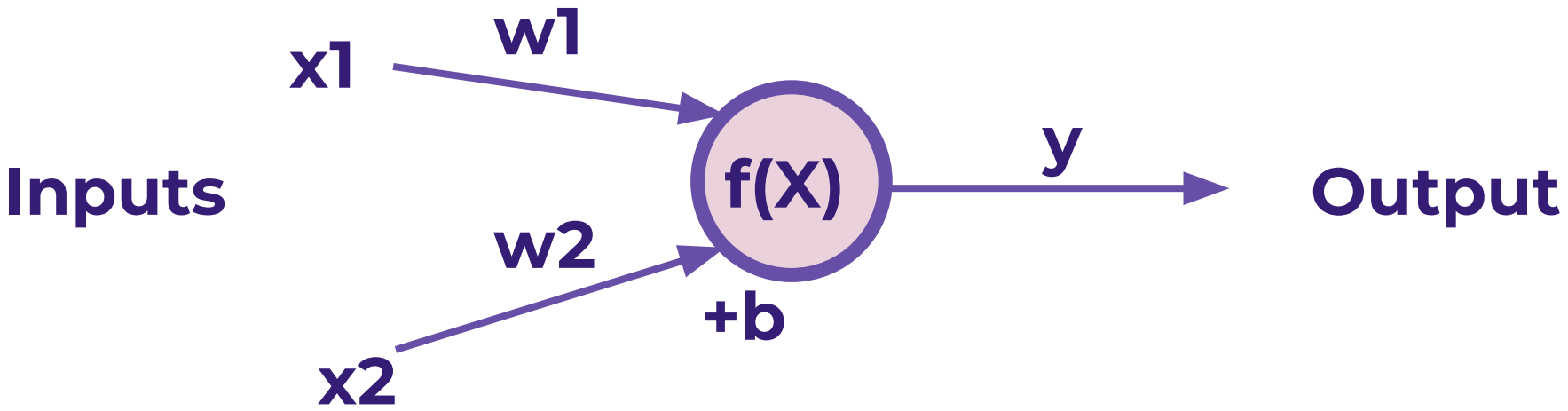
Perceptron model

- Recall our simple perceptron has an f(X)



Perceptron model

- If we had a binary classification problem, we would want an output of either 0 or 1.

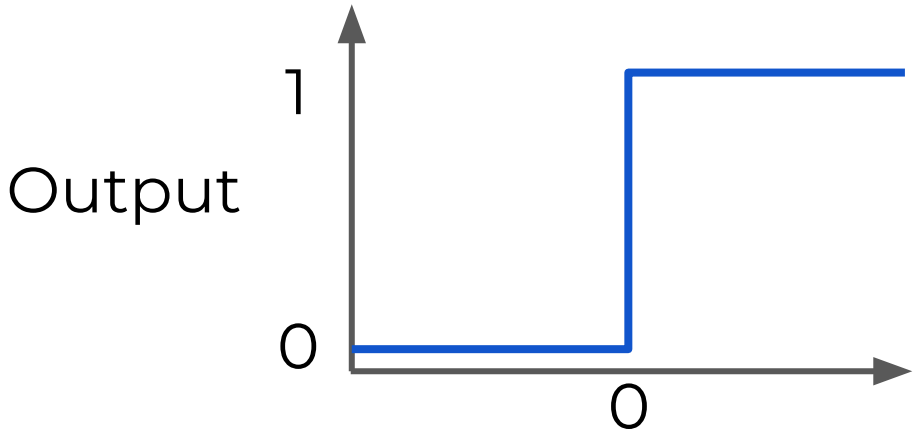


Neural Networks

- To avoid confusion, let's define the total inputs as a variable **z.**
- Where **z = wx + b**
- In this context, we'll then refer to activation functions as **f(z)**.
- Keep in mind, you will often see these variables capitalized **f(Z)** or **X** to denote a tensor input consisting of multiple values.

Deep Learning

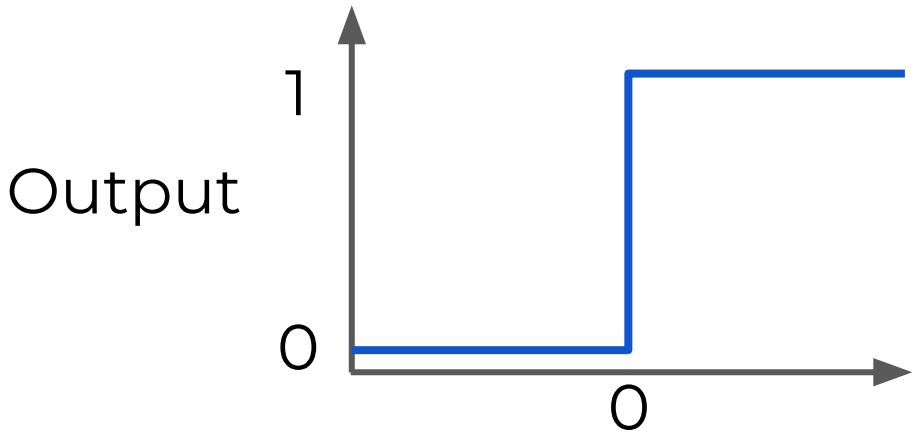
- The most simple networks rely on a basic **step function** that outputs 0 or 1.



$$z = wx + b$$

Deep Learning

- Regardless of the values, this always outputs 0 or 1.

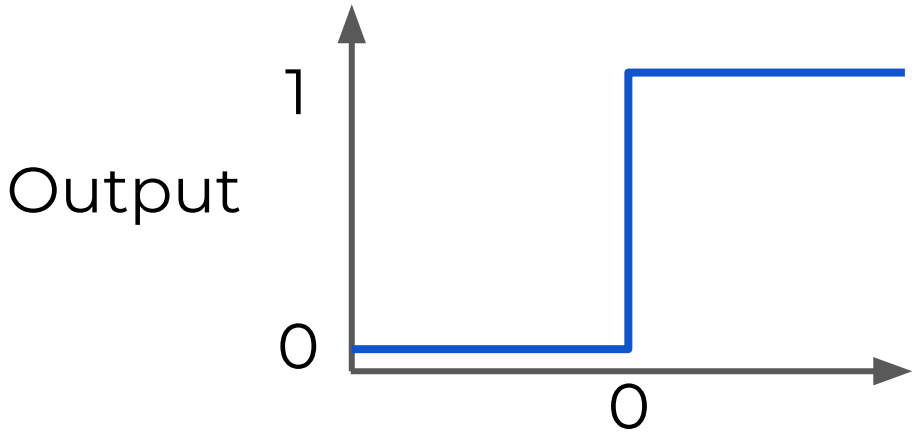


$$z = wx + b$$

PIERIAN DATA

Deep Learning

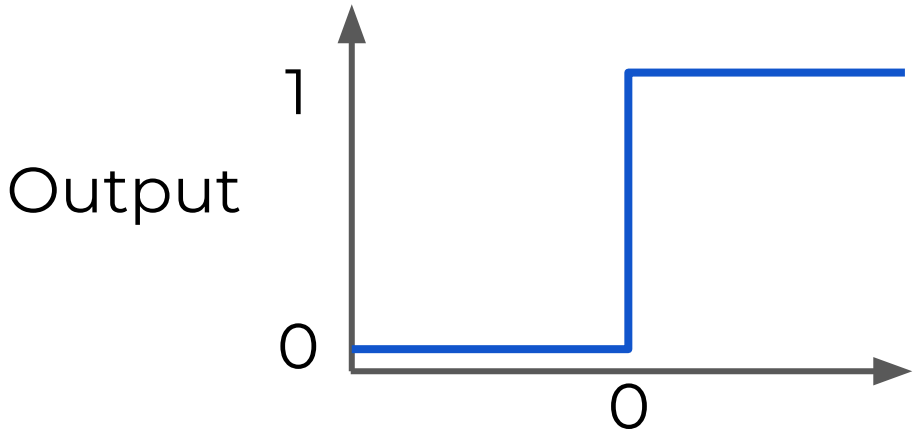
- This sort of function could be useful for classification (0 or 1 class).



$$z = wx + b$$

Deep Learning

- However this is a very “strong” function, since small changes aren’t reflected.

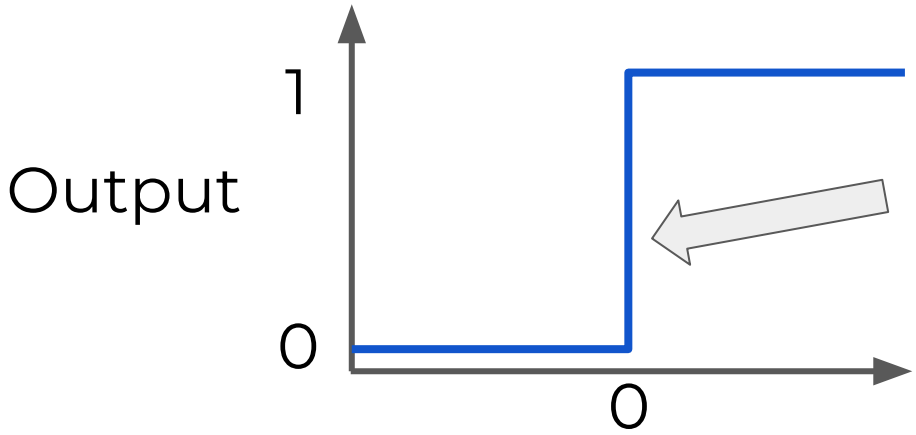


$$z = wx + b$$

PIERIAN DATA

Deep Learning

- There is just an immediate cut off that splits between 0 and 1.

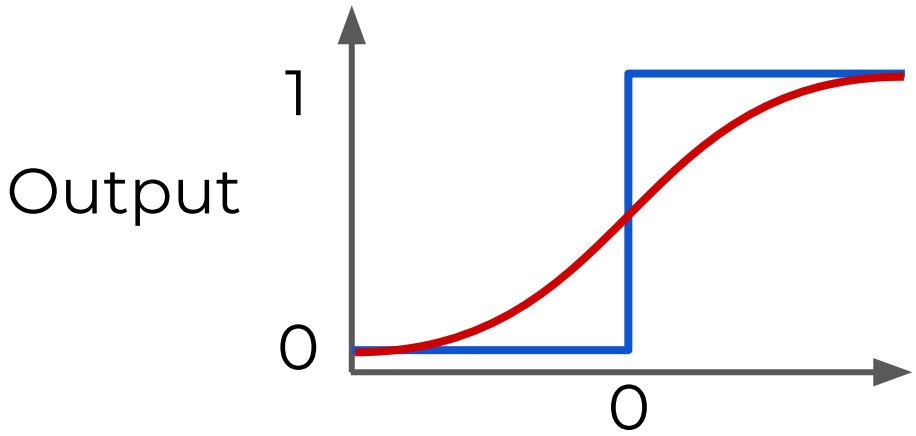


$$z = wx + b$$

PIERIAN DATA

Deep Learning

- It would be nice if we could have a more dynamic function, for example the red line!

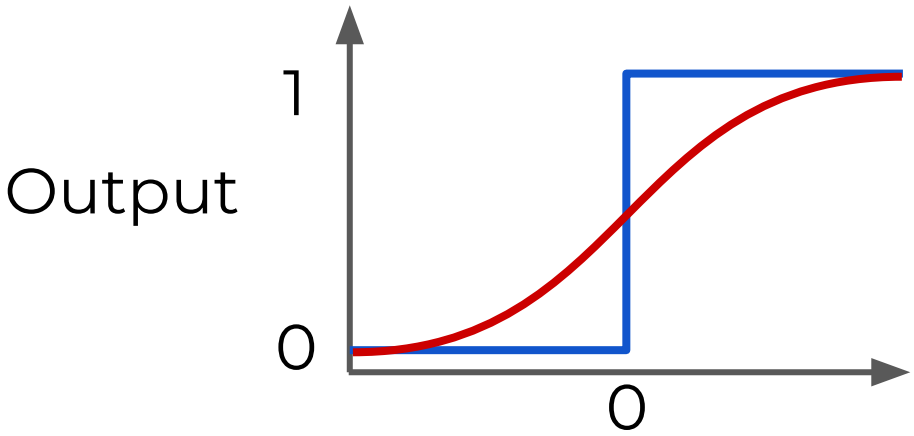


$$z = wx + b$$

PIERIAN DATA

Deep Learning

- Lucky for us, this is the sigmoid function!

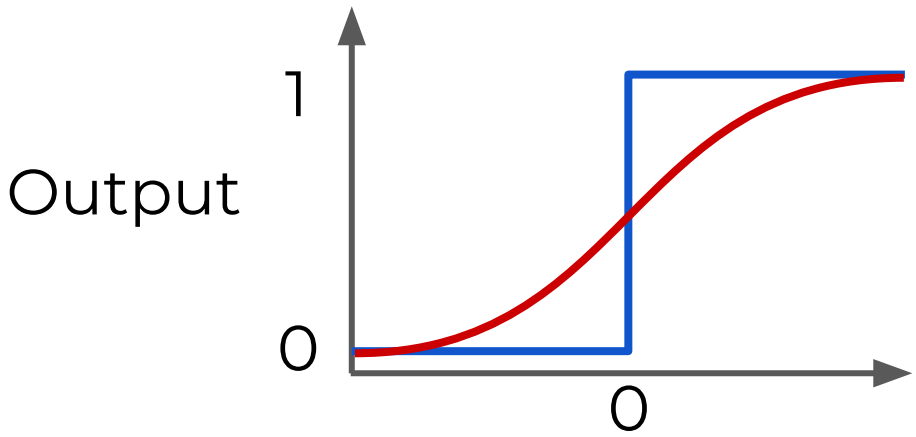


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

$$z = wx + b$$

Deep Learning

- Changing the activation function used can be beneficial depending on the task!

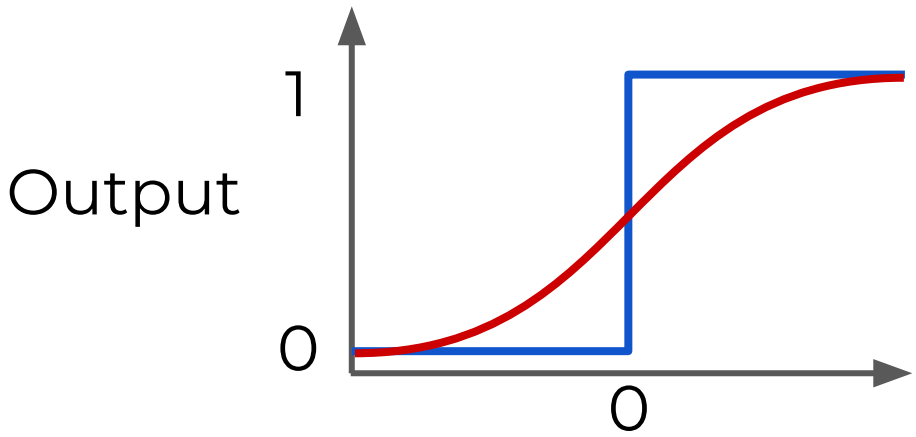


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

z = wx + b

Deep Learning

- This still works for classification, and will be more sensitive to small changes.

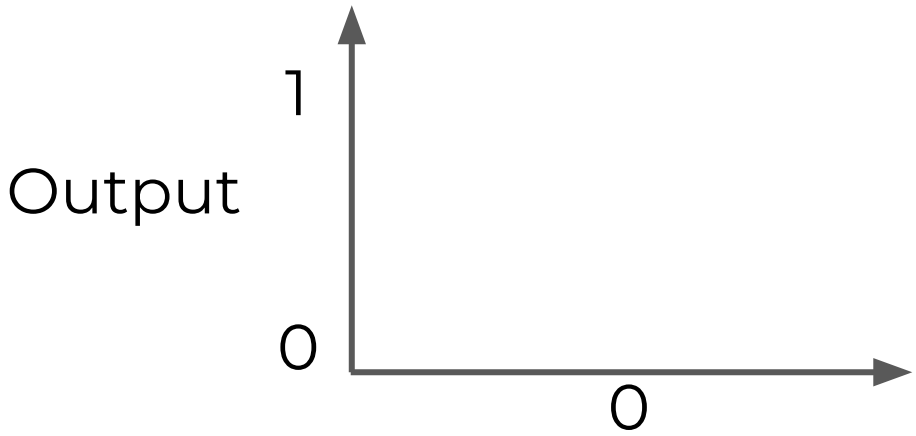


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

z = wx + b

Deep Learning

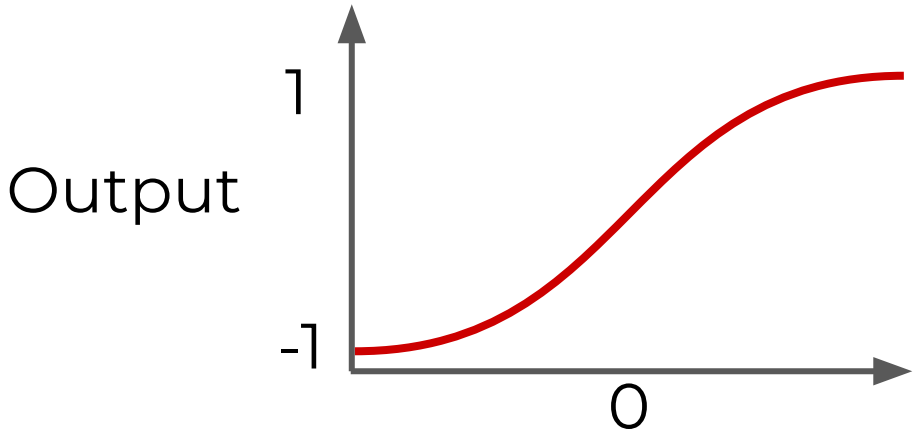
- Let’s discuss a few more activation functions that we’ll encounter!



$$z = wx + b$$

Deep Learning

- Hyperbolic Tangent: tanh(z)



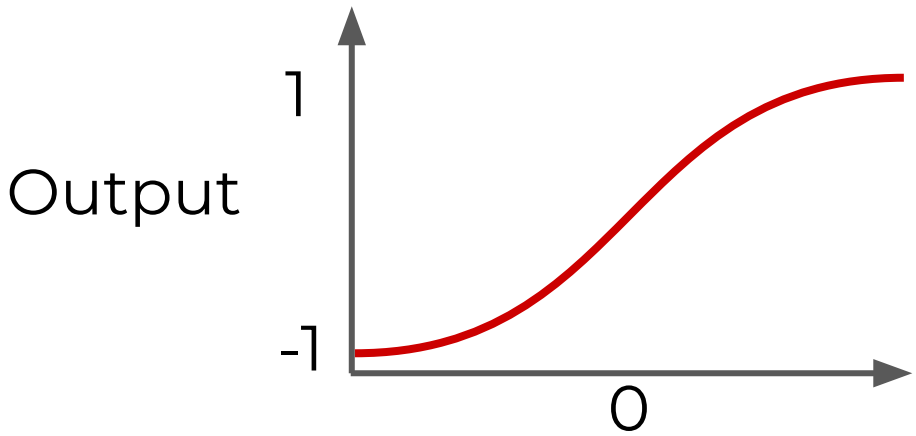
$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

$$\sinh x = \frac{\mathbf{e}^{x} - \mathbf{e}^{-x}}{2}$$

$$\tanh x = \frac{\sinh x}{\cosh x}$$

Deep Learning

- Hyperbolic Tangent: tanh(z)
- Outputs between -1 and 1 instead of 0 to 1



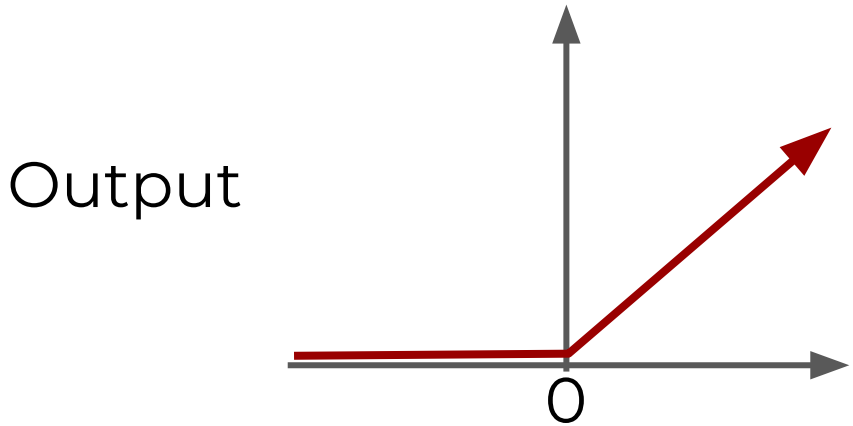
$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

$$\sinh x = \frac{\mathbf{e}^{x} - \mathbf{e}^{-x}}{2}$$

$$\tanh x = \frac{\sinh x}{\cosh x}$$

Deep Learning

- Rectified Linear Unit (ReLU): This is actually a relatively simple function: max(0,z)



$z = wx + b$

Deep Learning

- ReLu has been found to have very good performance, especially when dealing with the issue of **vanishing gradient**.
- We'll often default to ReLu due to its overall good performance.

Deep Learning

- For a full list of possible activation functions check out:
- **en.wikipedia.org/wiki/Activation_function**

Multi-Class Activation Functions

Deep Learning

- Notice all these activation functions make sense for a single output, either a continuous label or trying to predict a binary classification (either a 0 or 1).
- But what should we do if we have a multi-class situation?

Deep Learning

- There are 2 main types of multi-class situations
  - Non-Exclusive Classes
    - A data point can have multiple classes/categories assigned to it
  - Mutually Exclusive Classes
    - Only one class per data point.

Deep Learning

- Non-Exclusive Classes
  - A data point can have multiple classes/categories assigned to it
  - Photos can have multiple tags (e.g. beach, family, vacation, etc...)

Deep Learning

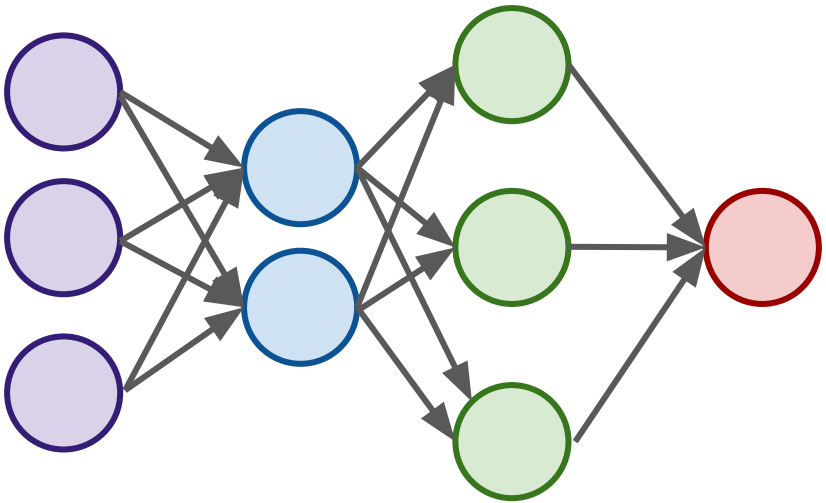
- Mutually Exclusive Classes
  - A data point can only have one class/category assigned to it
  - Photos can be categorized as being in grayscale (black and white) or full color photos. A photo can not be both at the same time.

Deep Learning

- Organizing Multiple Classes
  - The easiest way to organize multiple classes is to simply have 1 output node per class.

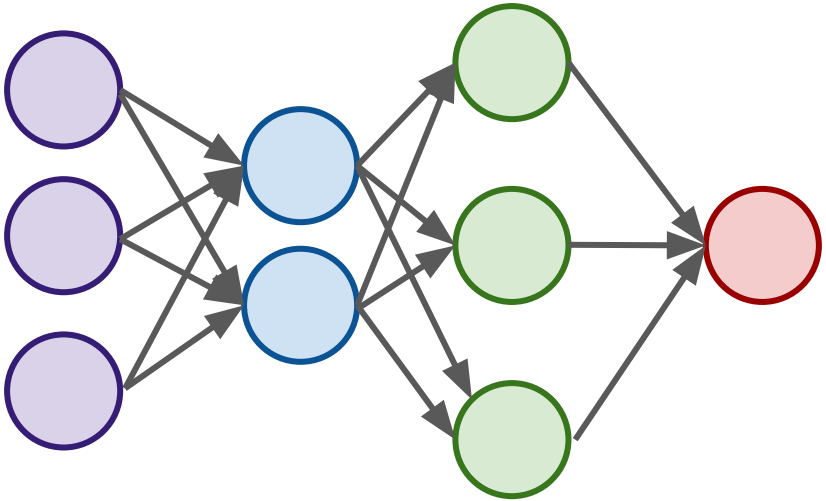
Neural Networks

- Previously we thought of the last output layer as a single node.



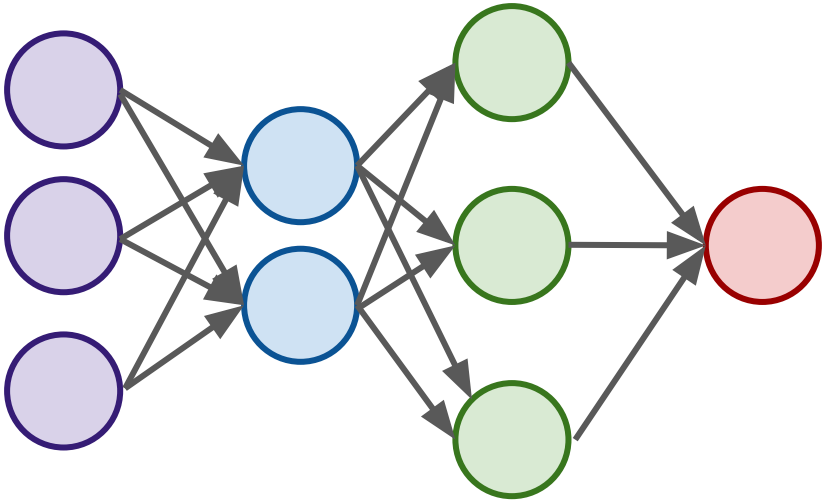
Neural Networks

- This single node could output a continuous regression value or binary classification (0 or 1).



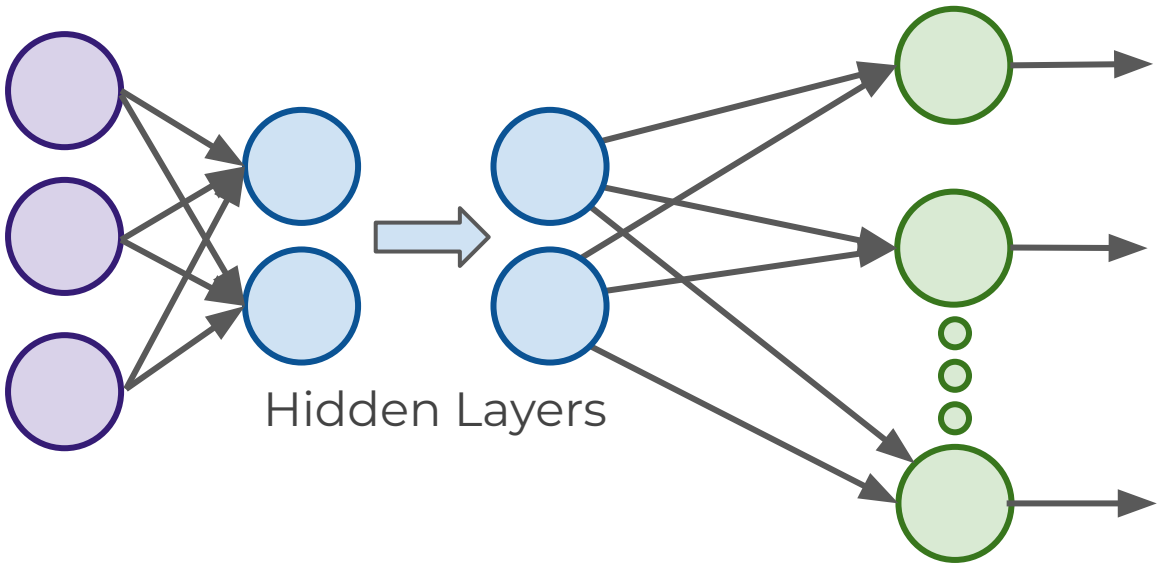
Neural Networks

- Let’s expand this output layer to work for the case of multi-classification.



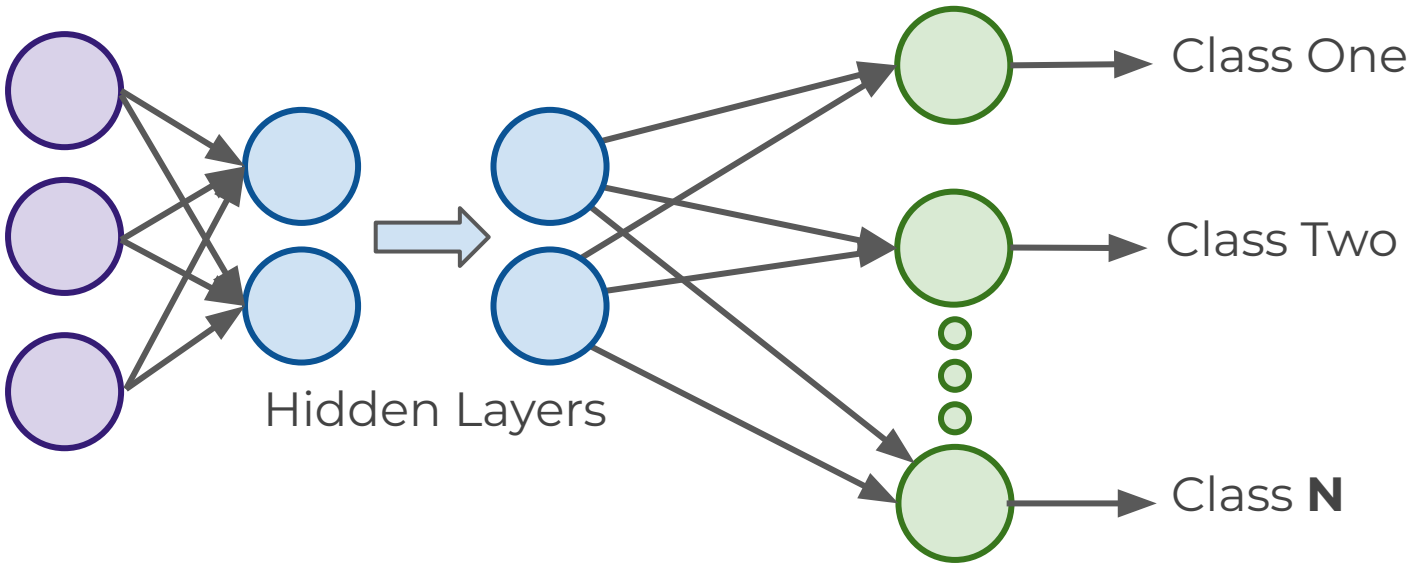
Multiclass Classification

- Organizing for Multiple Classes



Multiclass Classification

- Organizing for Multiple Classes



Deep Learning

- Organizing Multiple Classes
  - This means we will need to organize categories for this output layer.
  - We can’t just have categories like “red”, “blue”, “green”, etc…

Deep Learning

- Organizing Multiple Classes
  - Instead we use **one-hot encoding**
  - Let’s take a look at what this looks like for mutually exclusive classes.

Deep Learning

- Mutually Exclusive Classes

Data Point 1	**RED**
Data Point 2	**GREEN**
Data Point 3	**BLUE**
...	**...**
Data Point N	**RED**

Deep Learning

- Mutually Exclusive Classes

Data Point 1	RED
Data Point 2	GREEN
Data Point 3	BLUE
...	...
Data Point N	RED

	RED	GREEN	BLUE
Data Point 1	1	0	0
Data Point 2	0	1	0
Data Point 3	0	0	1
...	...	...	...
Data Point N	1	0	0

Deep Learning

- Non-Exclusive Classes

Data Point 1	A,B
Data Point 2	A
Data Point 3	C,B
...	...
Data Point N	B

	A	B	C
Data Point 1	1	1	0
Data Point 2	1	0	0
Data Point 3	0	1	1
...	...	...	...
Data Point N	0	1	0

Deep Learning

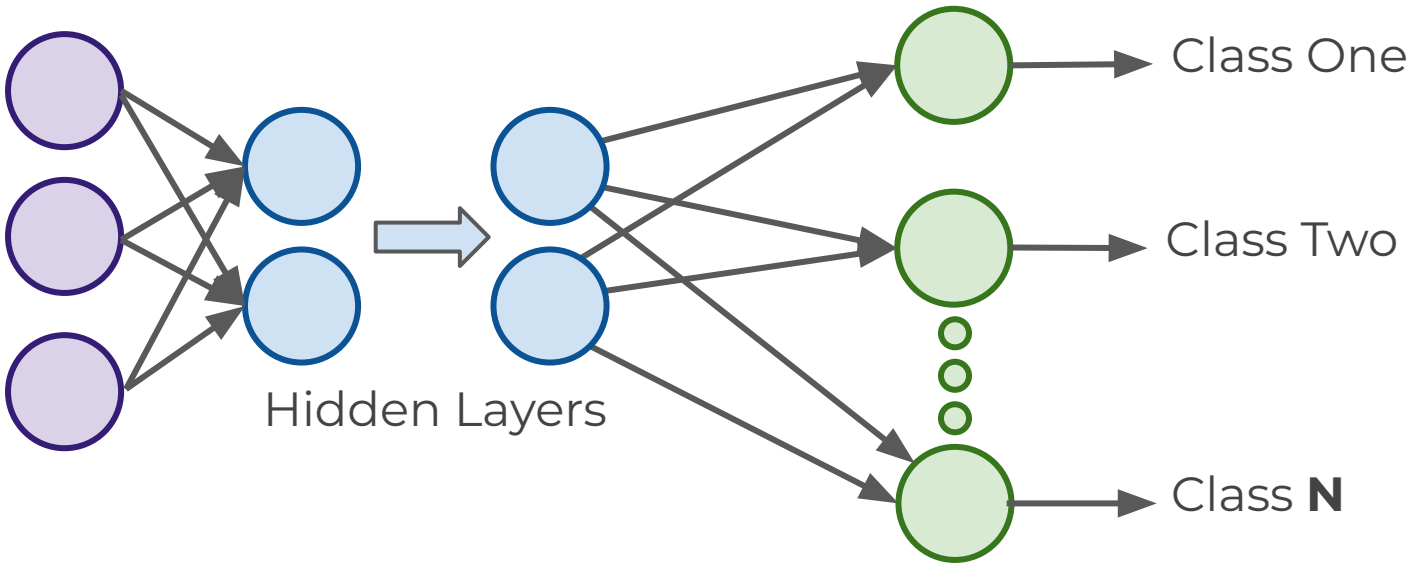
- Now that we have our data correctly organized, we just need to choose the correct classification activation function that the last output layer should have.

Deep Learning

- Non-exclusive
  - Sigmoid function
    - Each neuron will output a value between 0 and 1, indicating the probability of having that class assigned to it.

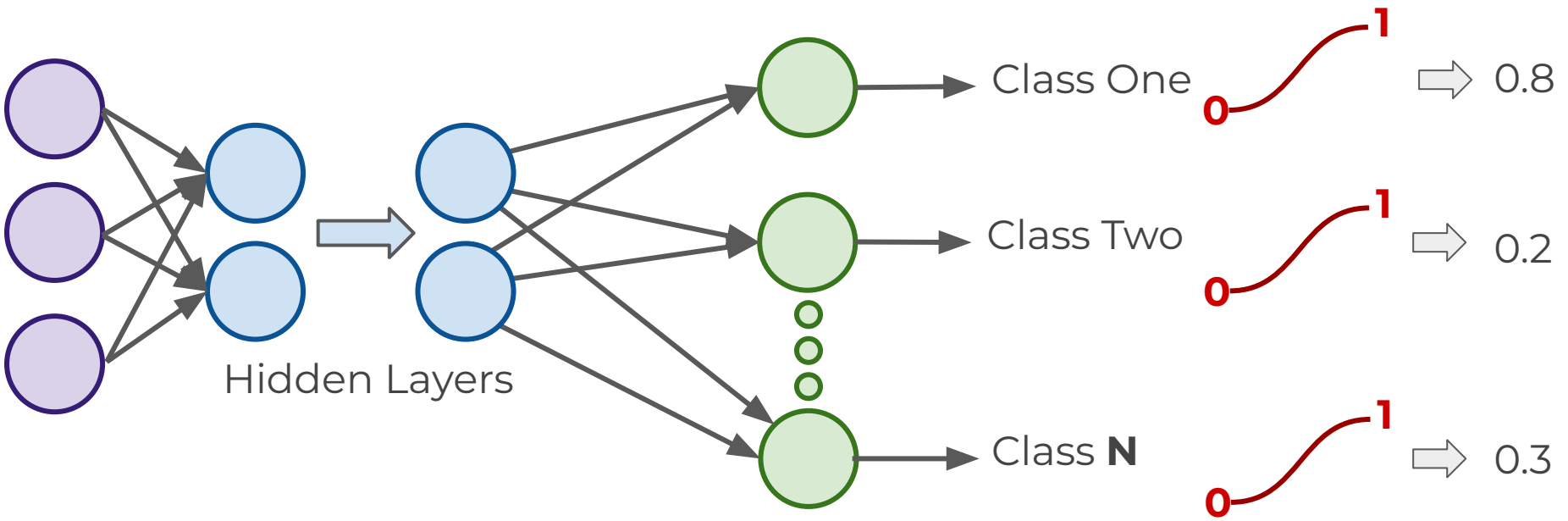
Multiclass Classification

- Sigmoid Function for Non-Exclusive Classes



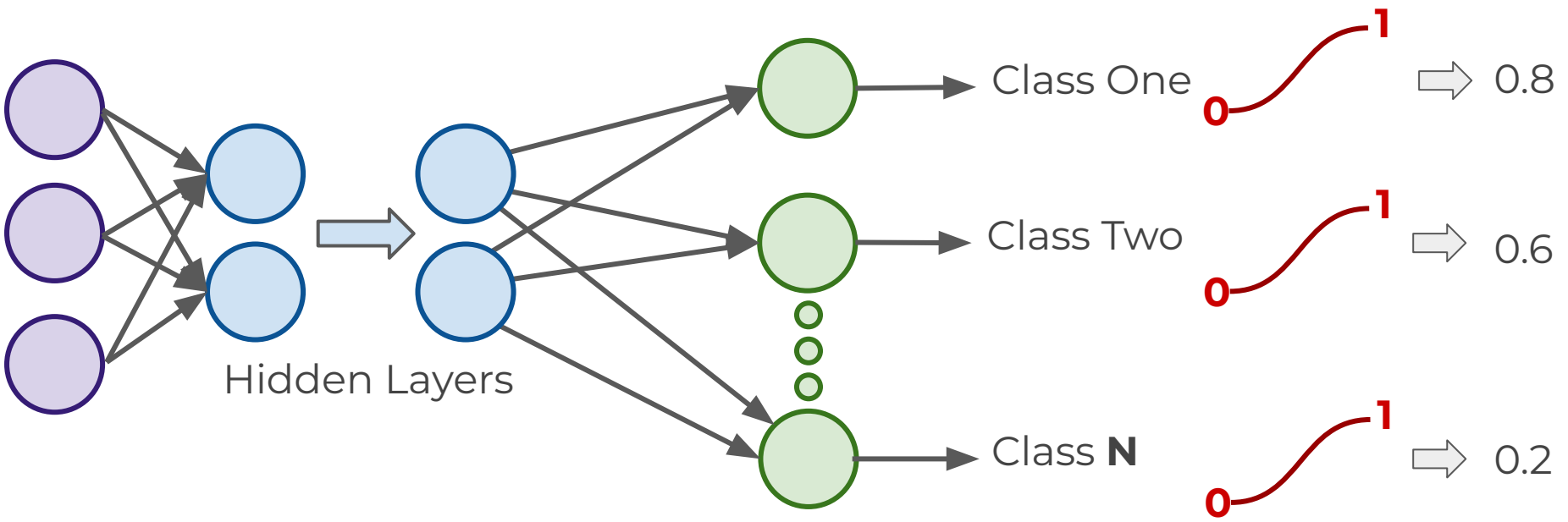
Multiclass Classification

- Sigmoid Function for Non-Exclusive Classes



Multiclass Classification

- Sigmoid Function for Non-Exclusive Classes



Deep Learning

- Non-exclusive
  - Sigmoid function
    - Keep in mind this allows each neuron to output independent of the other classes, allowing for a single data point fed into the function to have multiple classes assigned to it.

Deep Learning

- Mutually Exclusive Classes
  - But what do we do when each data point can only have a single class assigned to it?
  - We can use the **softmax function** for this!

Deep Learning

- Softmax Function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K$$

Deep Learning

- Mutually Exclusive Classes
  - Softmax function calculates the probabilities distribution of the event over **K** different events.
  - This function will calculate the probabilities of each target class over all possible target classes.

Deep Learning

- Mutually Exclusive Classes
  - The range will be 0 to 1, and **the sum of all the probabilities will be equal to one.**
  - The model returns the probabilities of each class and the target class chosen will have the highest probability.

Deep Learning

- Mutually Exclusive Classes
  - The main thing to keep in mind is that if you use softmax for multi-class problems you get this sort of output:
    - [Red , Green , Blue]
    - [0.1 , 0.6 , 0.3]

Deep Learning

- Mutually Exclusive Classes
  - The probabilities for each class all sum up to 1. We choose the highest probability as our assignment.
    - [Red , Green , Blue]
    - [0.1 , 0.6 , 0.3]

Deep Learning

- Review
  - Perceptrons expanded to neural network model
  - Weights and Biases
  - Activation Functions
  - Time to learn about Cost Functions!

Cost Functions and Gradient Descent

Deep Learning

- We now understand that neural networks take in inputs, multiply them by weights, and add biases to them.
- Then this result is passed through an activation function which at the end of all the layers leads to some output.

Deep Learning

- This output $\hat{\mathbf{y}}$ is the model's estimation of what it predicts the label to be.
- So after the network creates its prediction, how do we evaluate it?
- And after the evaluation how can we update the network's weights and biases?

Deep Learning

- We need to take the estimated outputs of the network and then compare them to the real values of the label.
- Keep in mind this is using the training data set during the fitting/training of the model.

Deep Learning

- The cost function (often referred to as a loss function) must be an average so it can output a single value.
- We can keep track of our loss/cost during training to monitor network performance.

Deep Learning

- We'll use the following variables:
  - y to represent the true value
  - a to represent neuron's prediction
  - In terms of weights and bias:
    - w*x + b = z
    - Pass z into activation function σ(z) = a

Deep Learning

- One very common cost function is the quadratic cost function:

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

Deep Learning

- We simply calculate the difference between the real values y(x) against our predicted values a(x).

$$C = \frac{1}{2n} \sum_{x} \|y(x) - a^{L}(x)\|^2$$

Deep Learning

- Note: The notation shown here corresponds to vector inputs and outputs, since we will be dealing with a **batch** of training points and predictions.

$$C = \frac{1}{2n} \sum_{x} \| y(x) - a^{L}(x) \|^2$$

Deep Learning

- Notice how squaring this does 2 useful things for us, keeps everything positive and **punishes** large errors!

$$C = \frac{1}{2n} \sum_{x} \|y(x) - a^{L}(x)\|^2$$

Deep Learning

- We can think of the cost function as:

$$C(W, B, S^r, E^r)$$

Deep Learning

- **W** is our neural network's weights, **B** is our neural network's biases, $\mathbf{S^r}$ is the input of a single training sample, and $\mathbf{E^r}$ is the desired output of that training sample.

$$C(W, B, S^r, E^r)$$

Deep Learning

- Notice how that information was all encoded in our simplified notation.
- The **a(x)** holds information about weights and biases.

$$C = \frac{1}{2n} \sum_{x} \|y(x) - a^{L}(x)\|^2$$

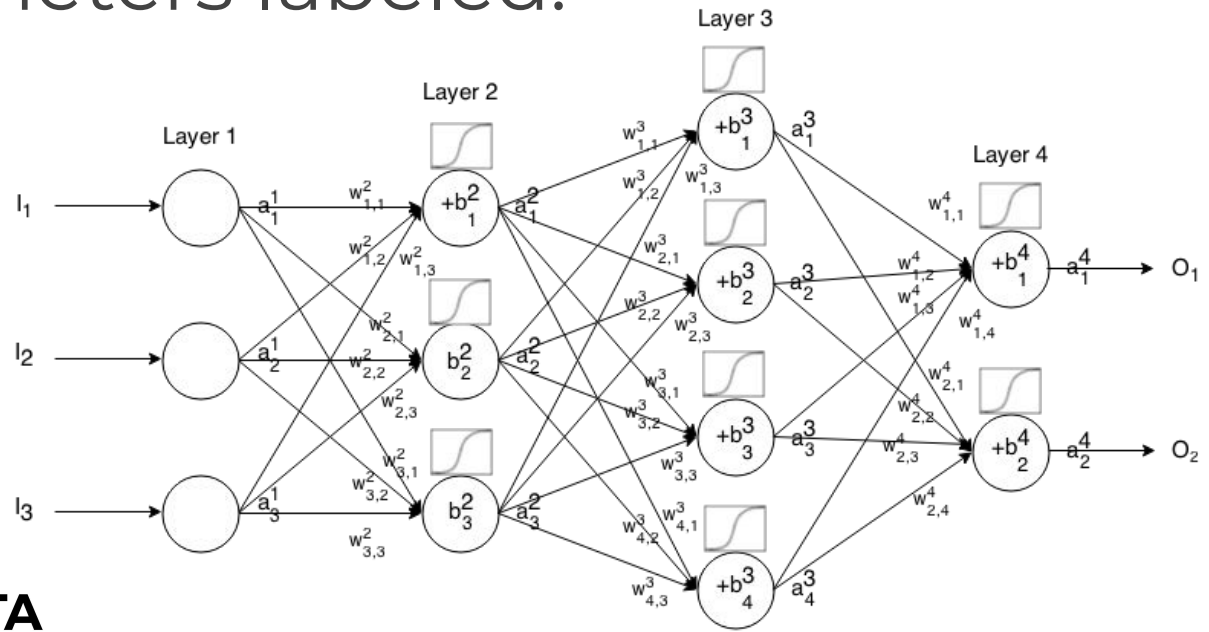
Deep Learning

- This means that if we have a huge network, we can expect **C** to be quite complex, with huge vectors of weights and biases.

$$C(W, B, S^r, E^r)$$

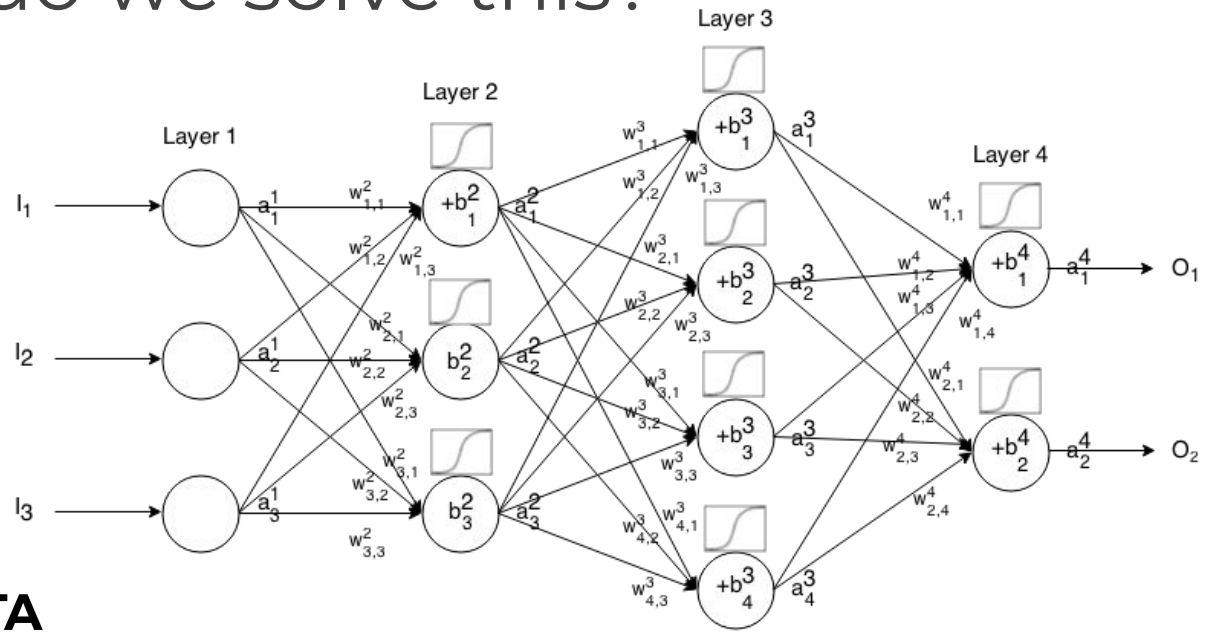
Deep Learning

- Here is a small network with all its parameters labeled:



Deep Learning

- That is a lot to calculate!
- How do we solve this?



PIERIAN DATA

Deep Learning

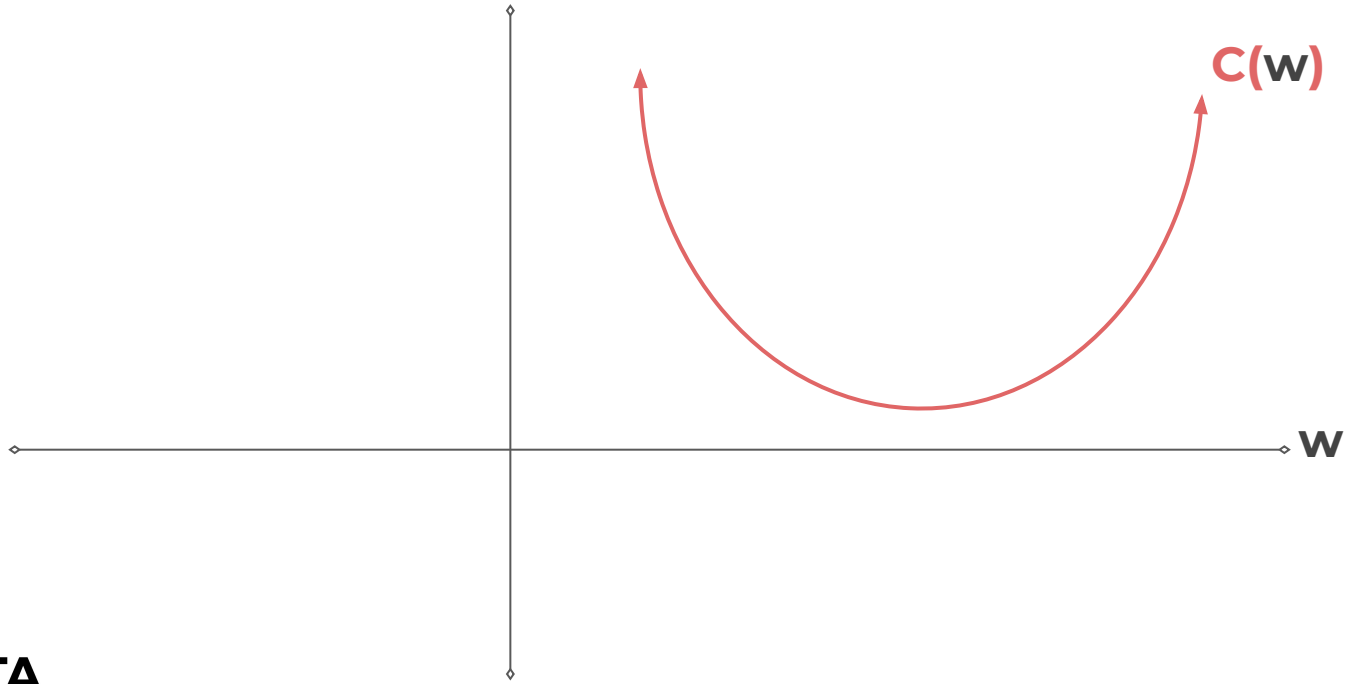
- In a real case, this means we have some cost function **C** dependent lots of weights!
    - **C(w1,w2,w3,....wn)**
- How do we figure out which weights lead us to the lowest cost?

Deep Learning

- For simplicity, let's imagine we only had one weight in our cost function **w.**
- We want to **minimize** our loss/cost (overall error).
- Which means we need to figure out what value of **w** results in the minimum of **C(w)**

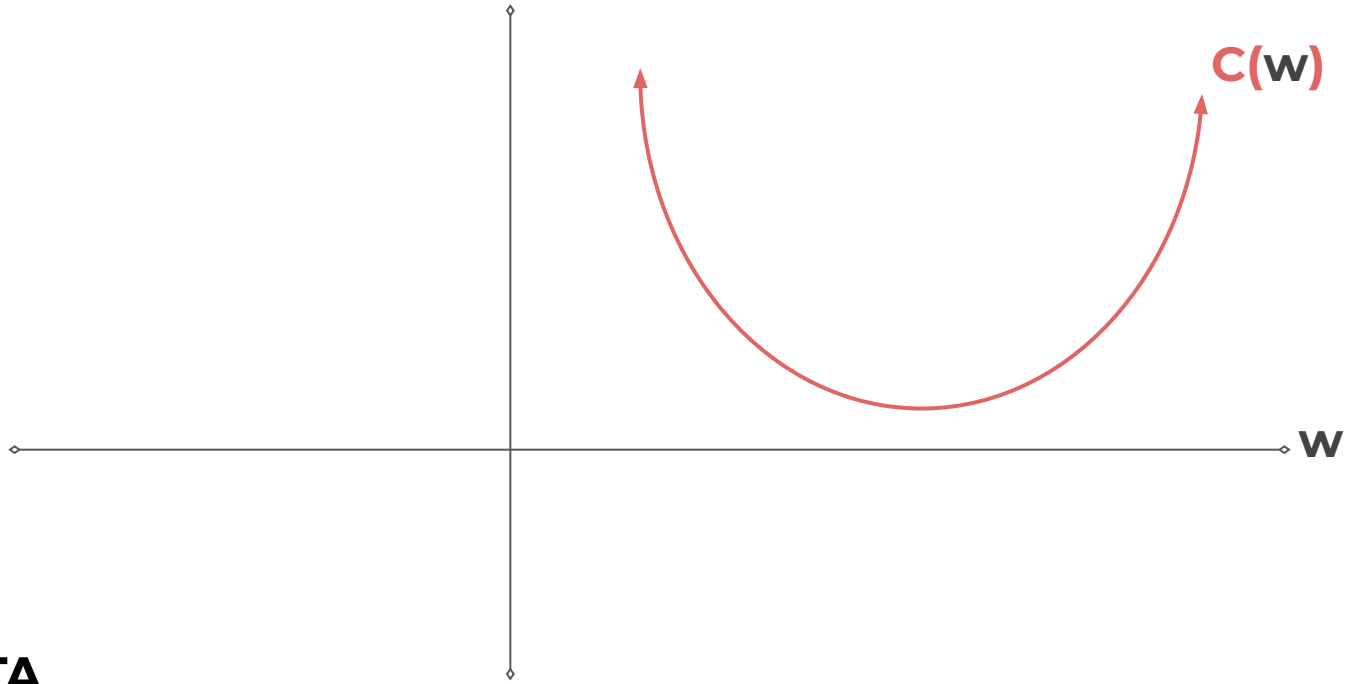
Deep Learning

- Our “simple” function **C(w)**



Deep Learning

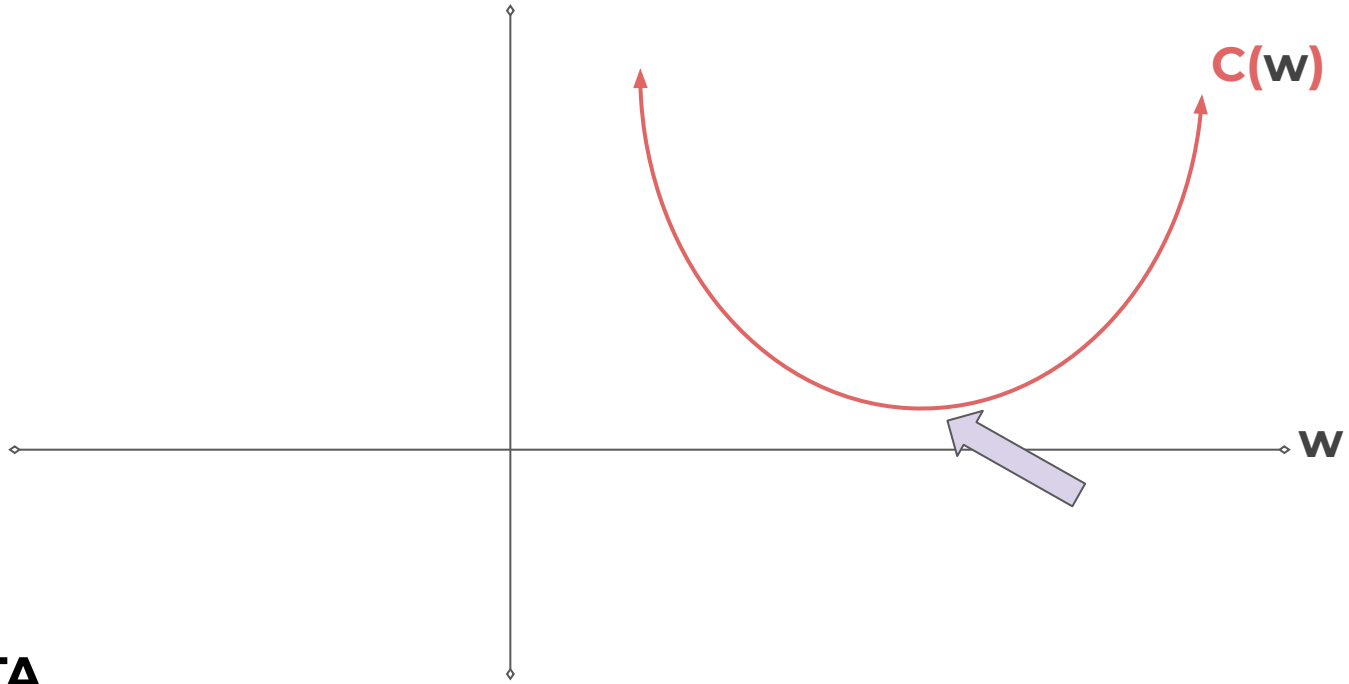
- What value of **w** minimizes our cost?



PIERIAN DATA

Deep Learning

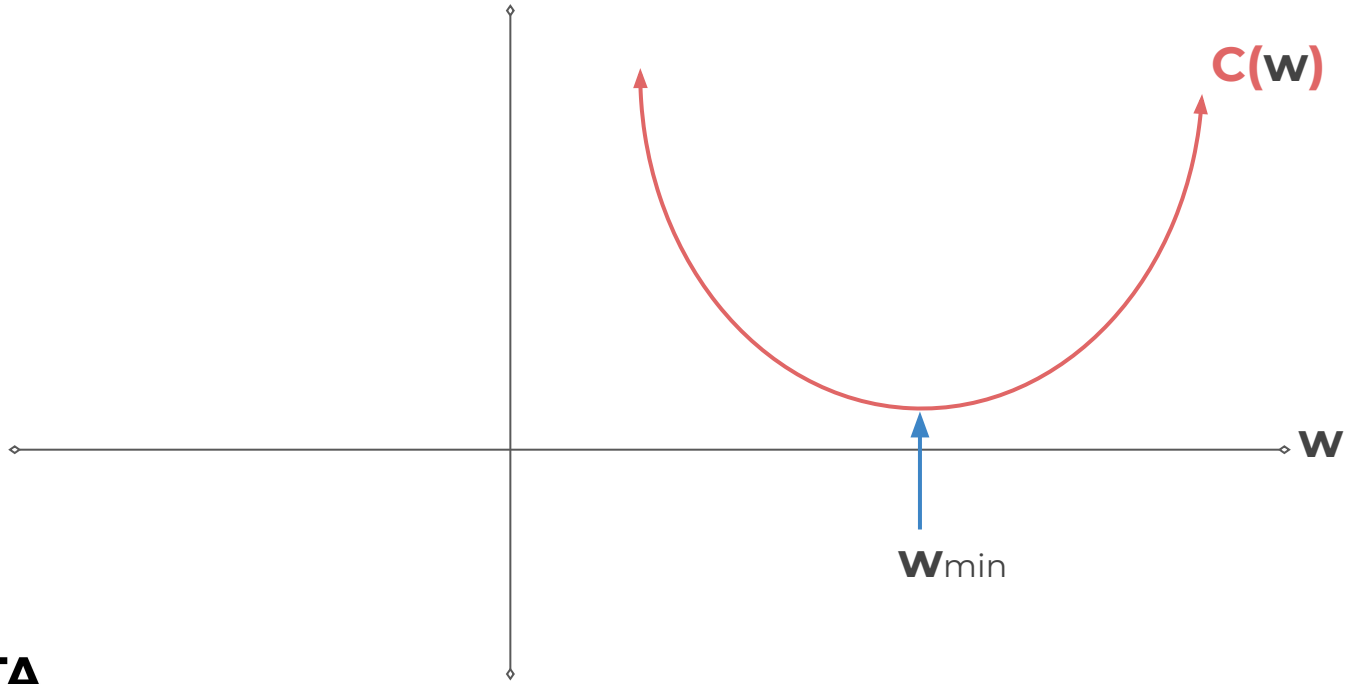
- What value of **w** minimizes our cost?



PIERIAN DATA

Deep Learning

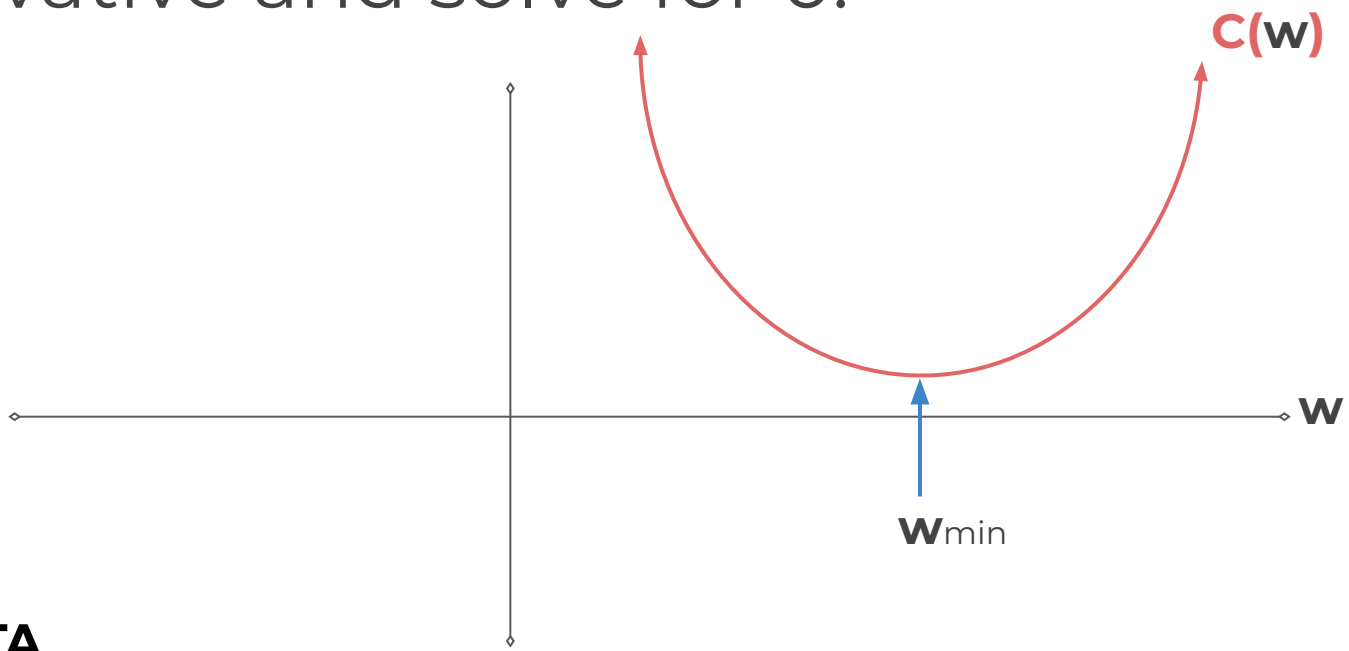
- What value of **w** minimizes our cost?



PIERIAN DATA

Deep Learning

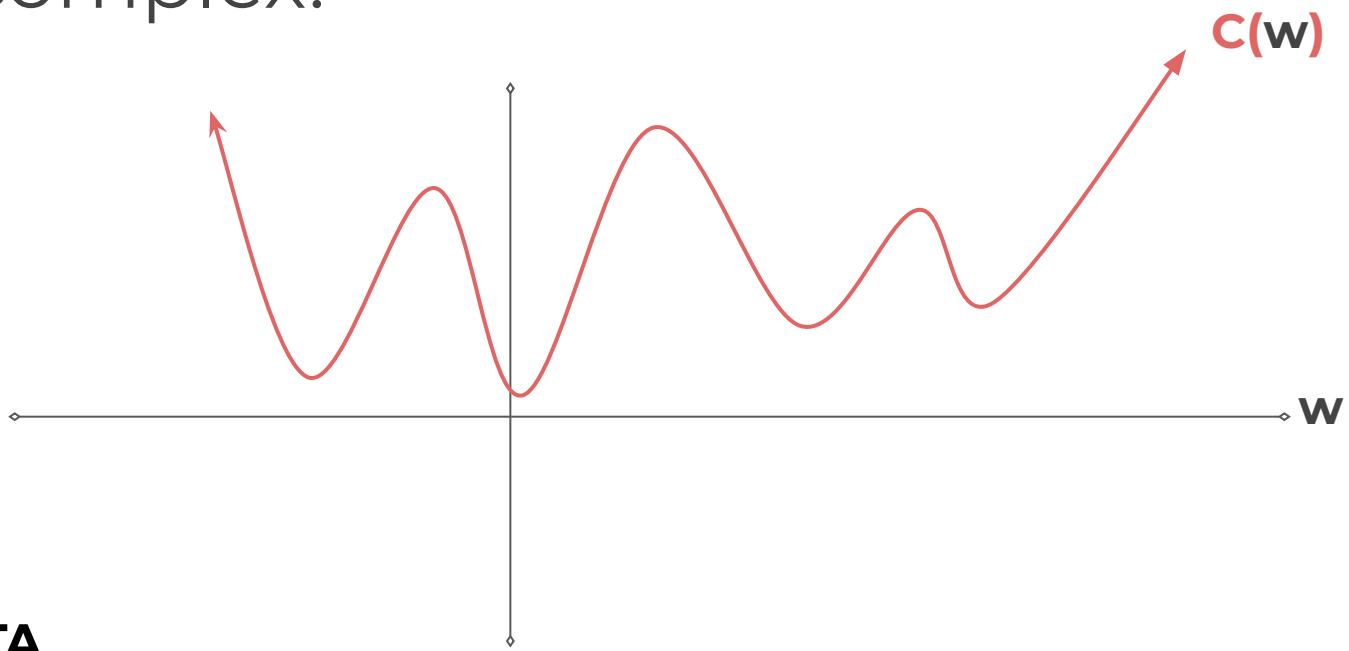
- Students of calculus know we could take a derivative and solve for 0.



PIERIAN DATA

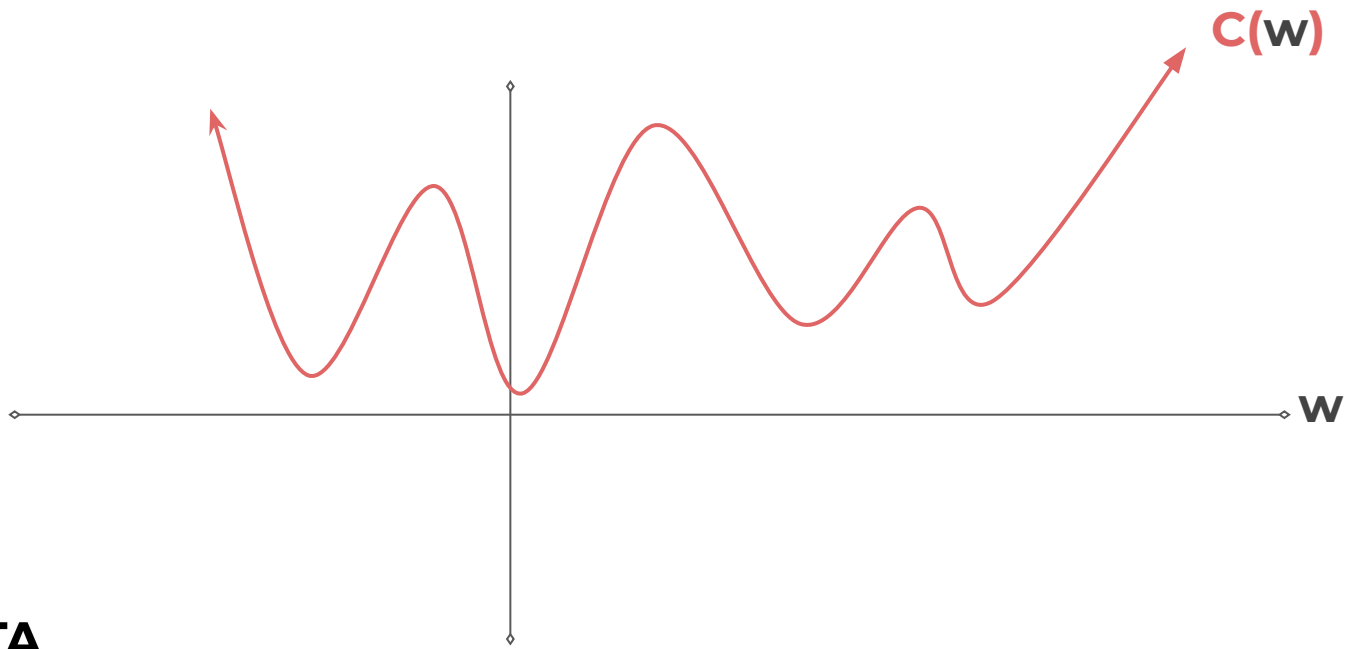
Deep Learning

- But recall our real cost function will be very complex!



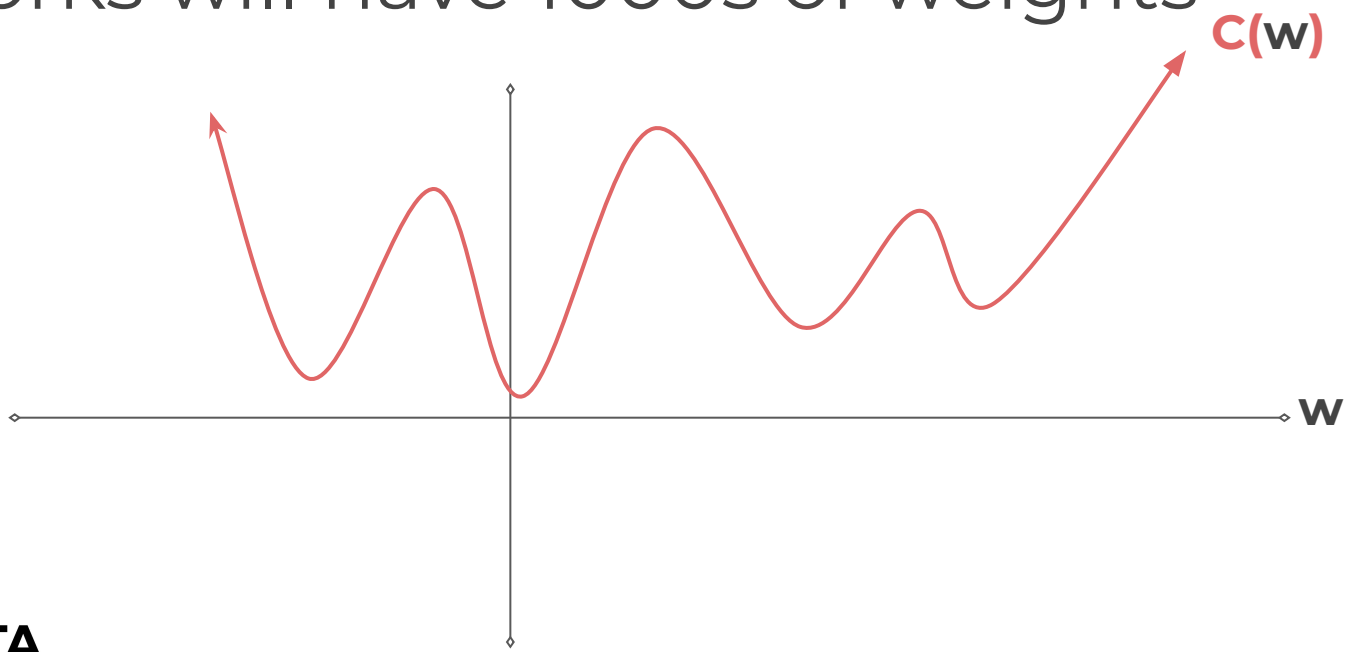
Deep Learning

- And it will be **n-dimensional!**



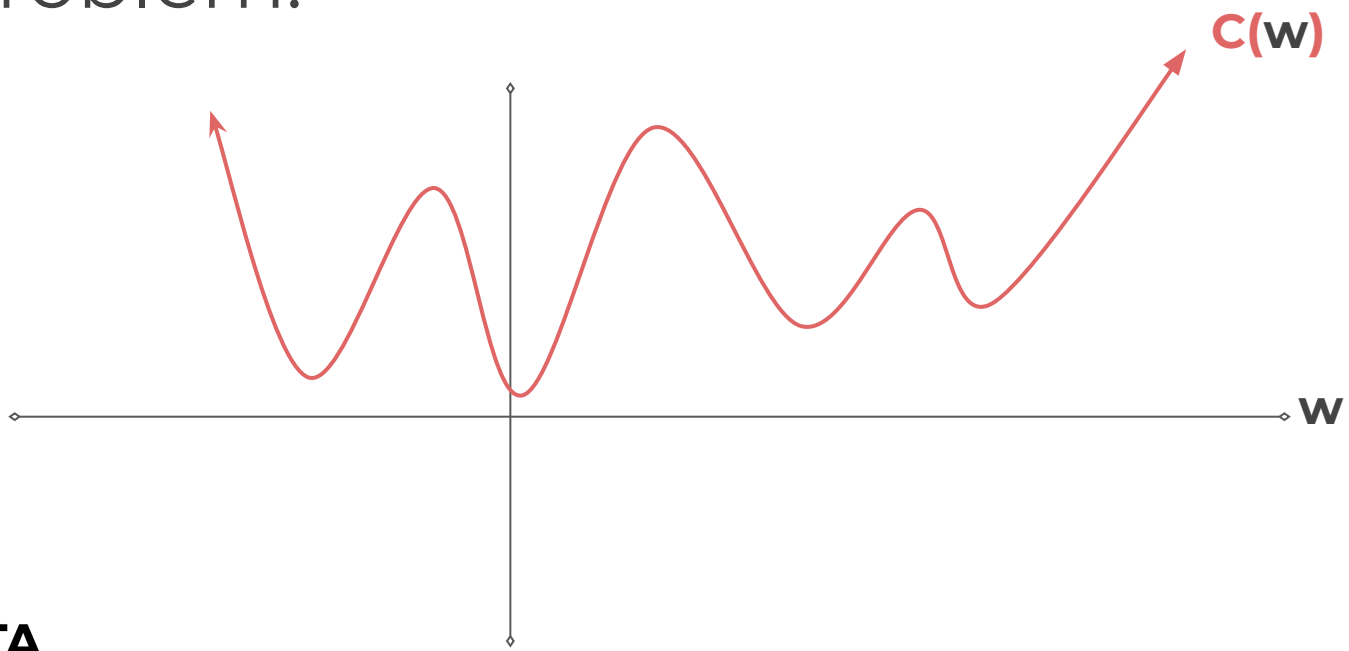
Deep Learning

- And it will be **n-dimensional** since our networks will have 1000s of weights



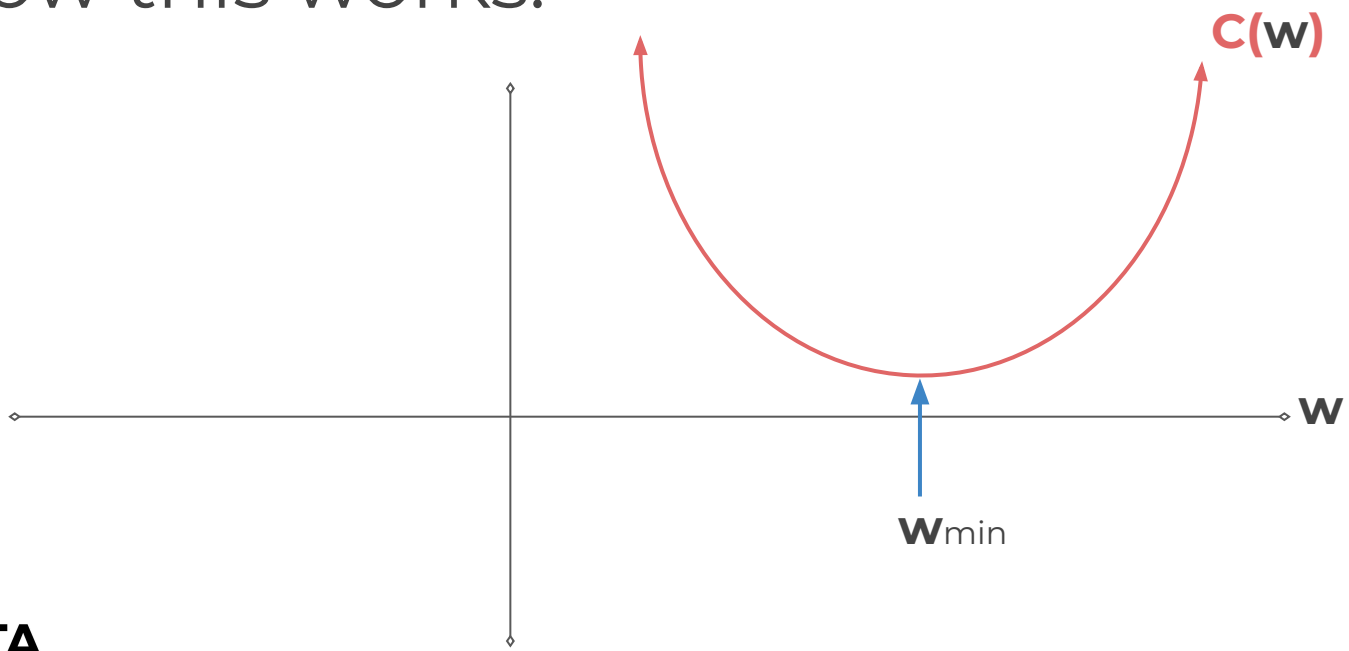
Deep Learning

- We can use **gradient descent** to solve this problem.



Deep Learning

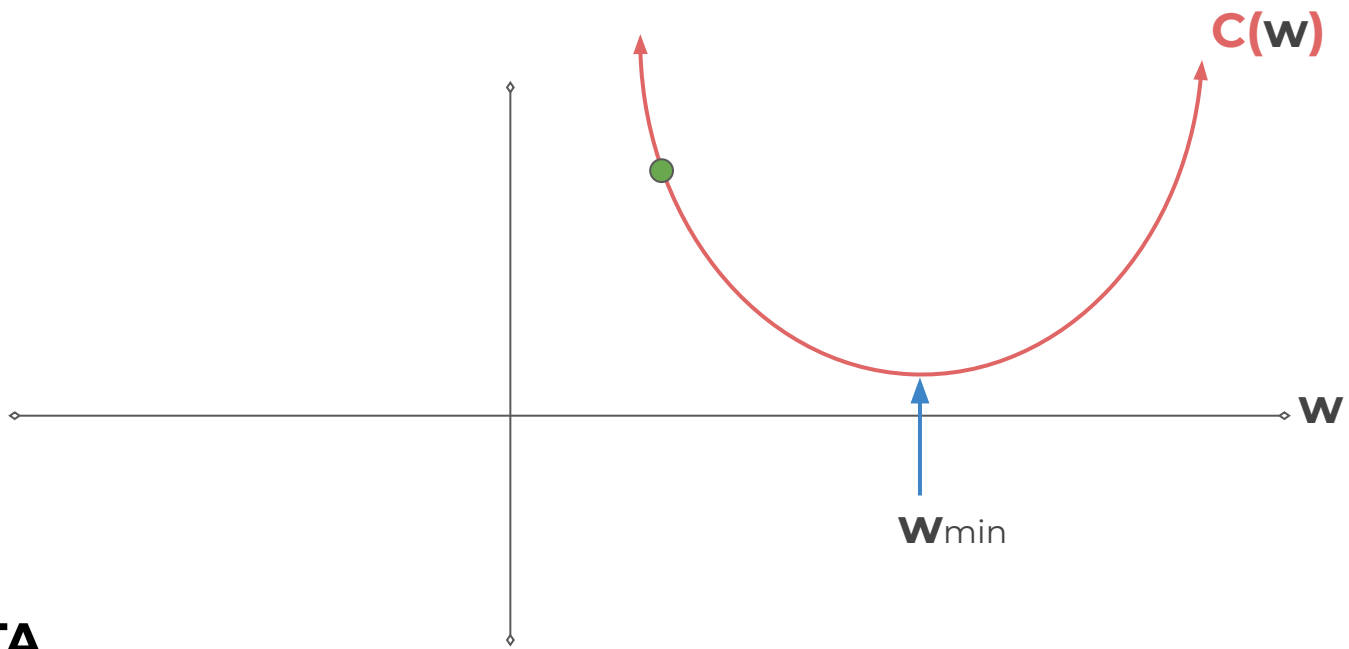
- Let’s go back to our simplified version to see how this works.



PIERIAN DATA

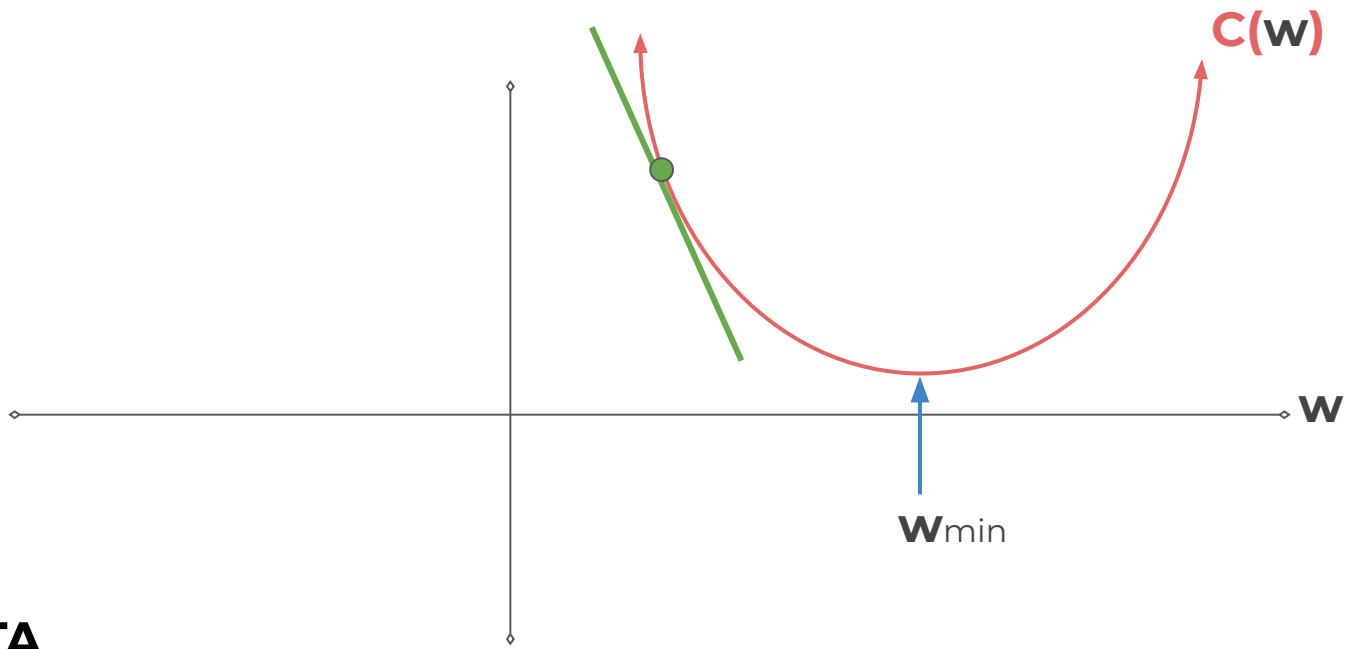
Deep Learning

- We can calculate the slope at a point



Deep Learning

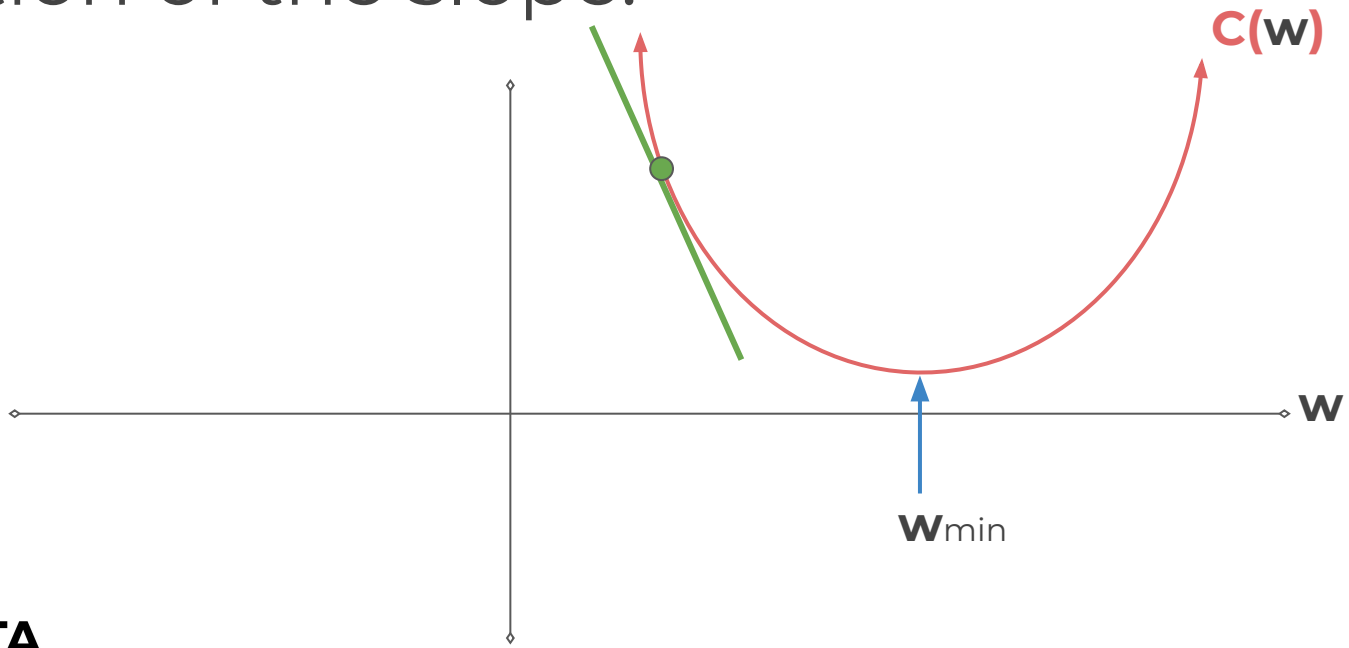
- We can calculate the slope at a point



PIERIAN DATA

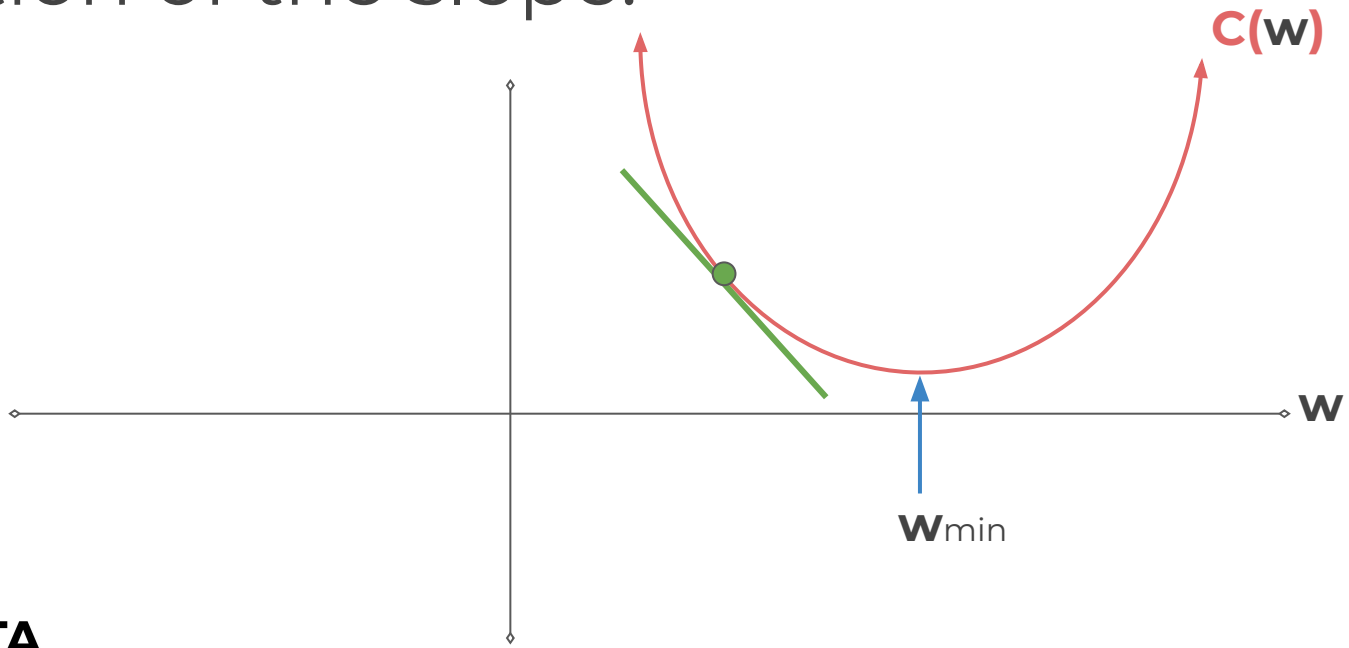
Deep Learning

- Then we move in the downward direction of the slope.



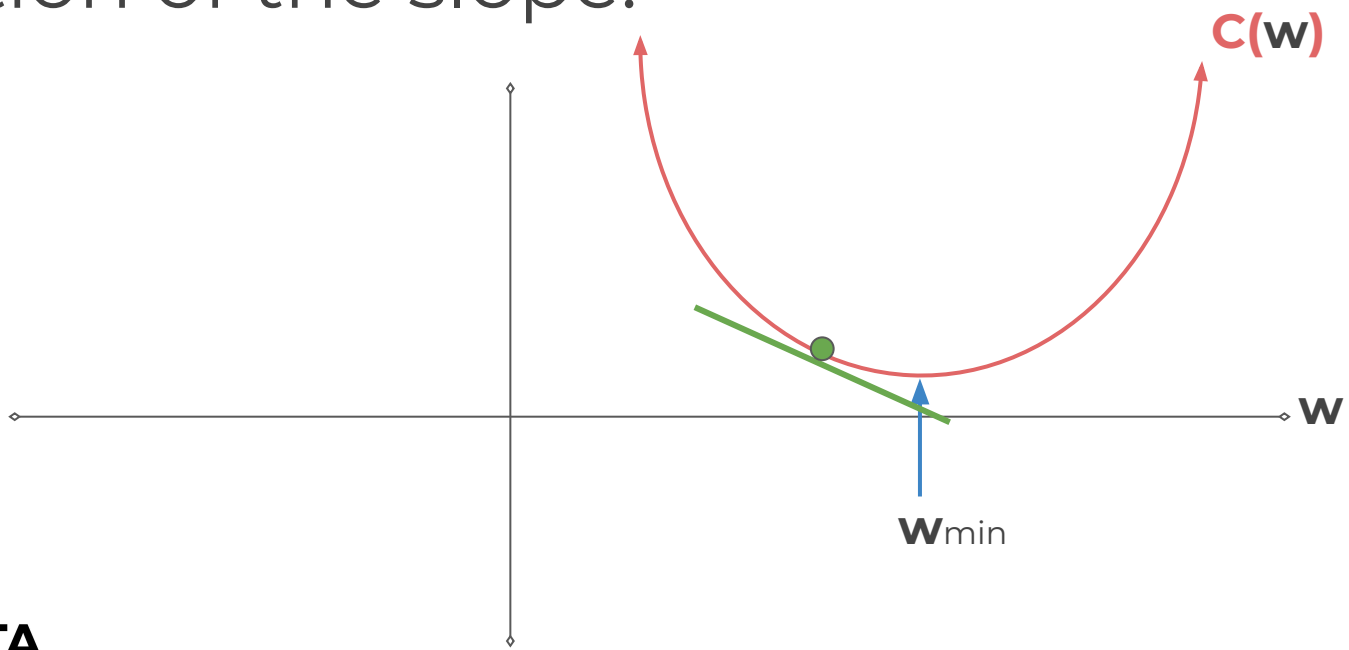
Deep Learning

- Then we move in the downward direction of the slope.



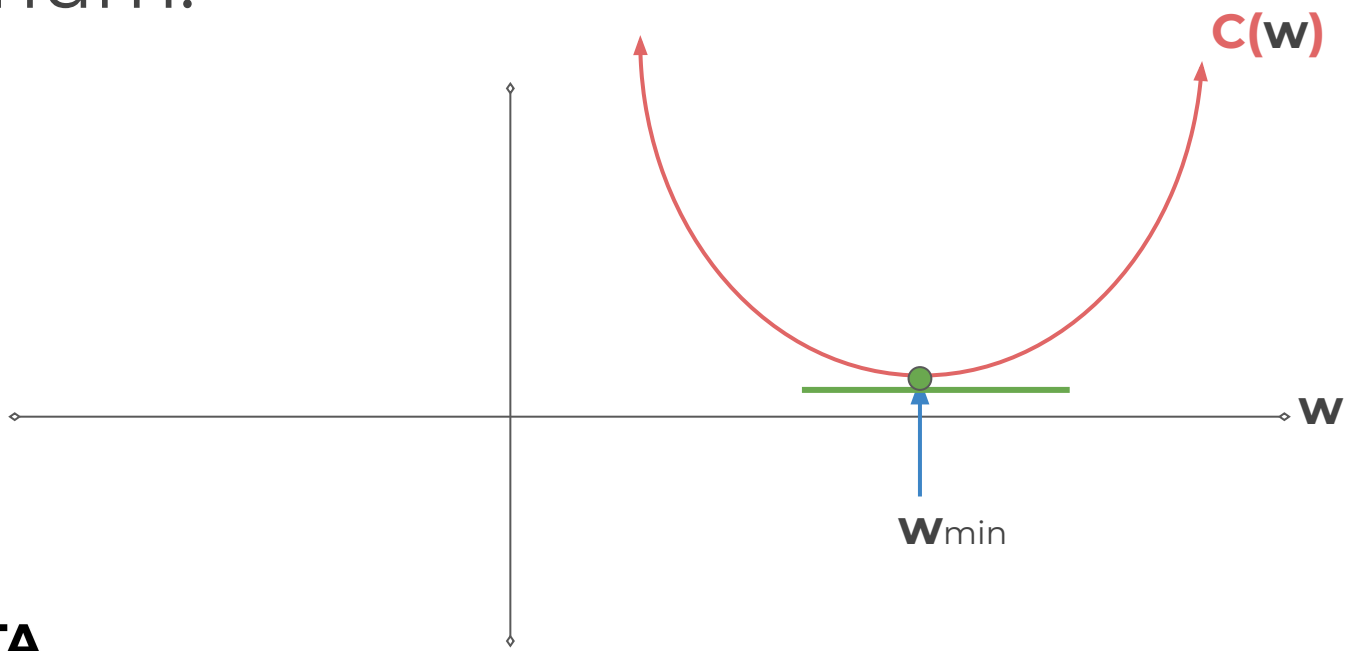
Deep Learning

- Then we move in the downward direction of the slope.



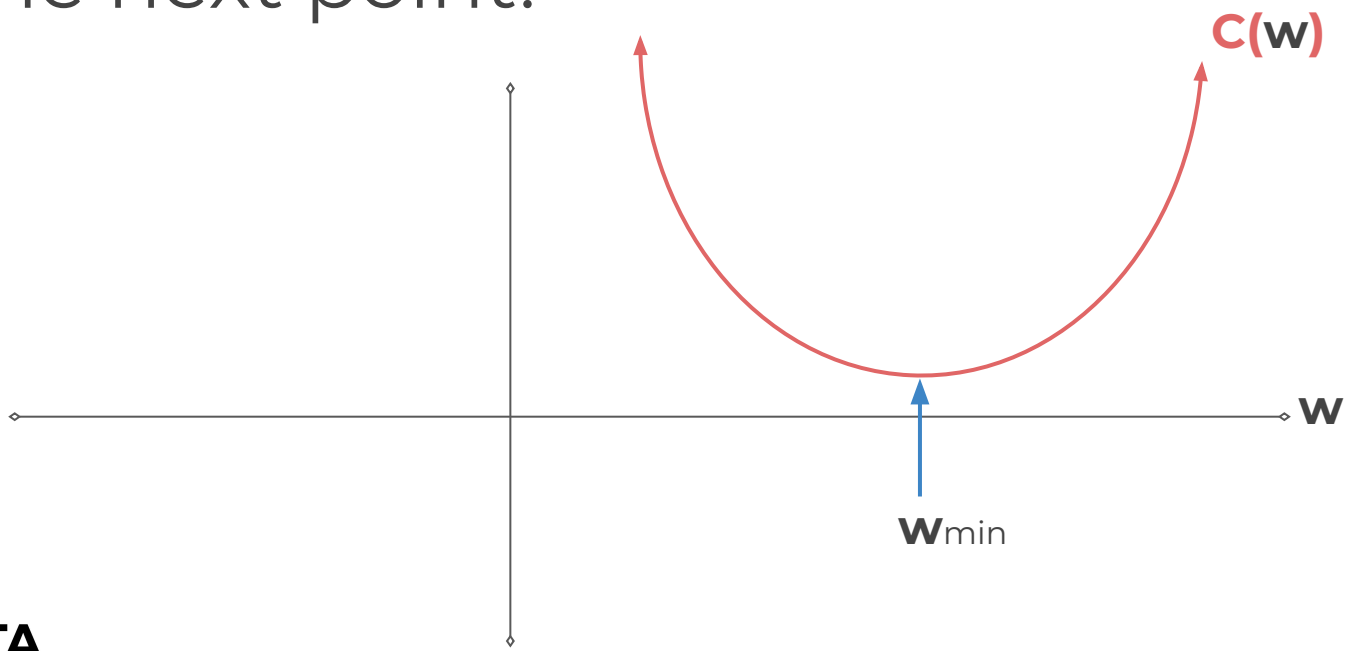
Deep Learning

- Until we converge to zero, indicating a minimum.



Deep Learning

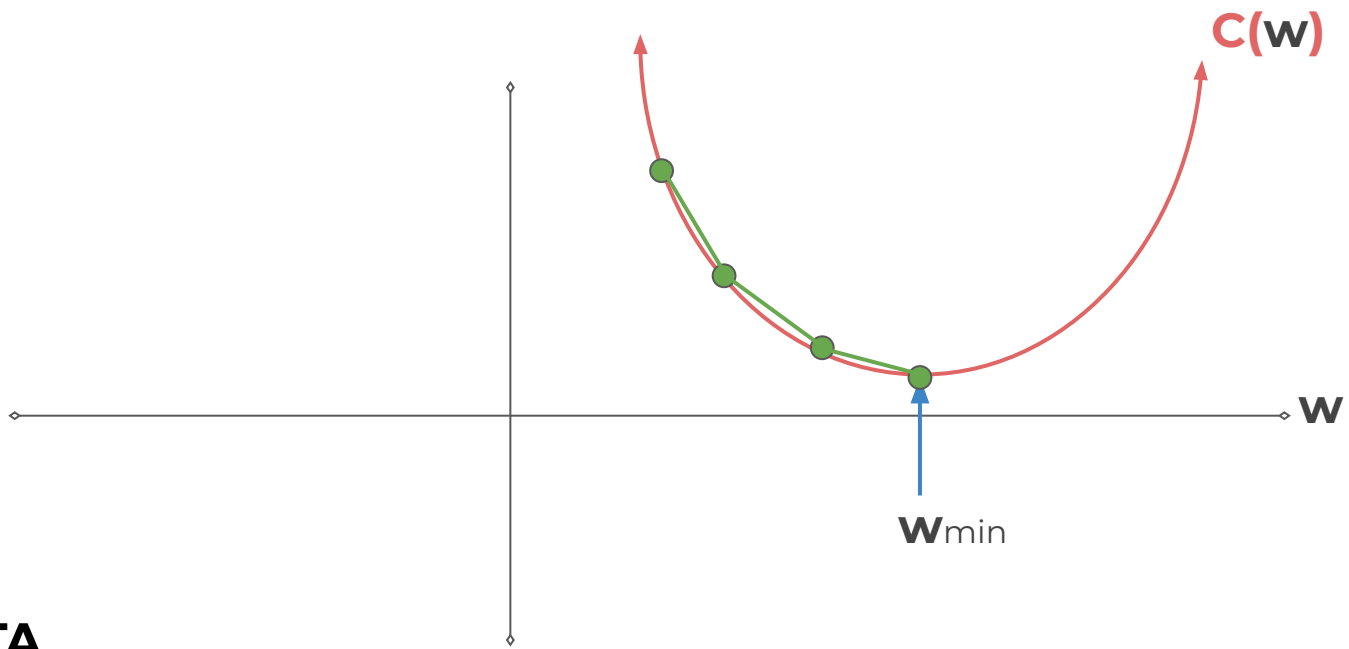
- We could have changed our step size to find the next point!



PIERIAN DATA

Deep Learning

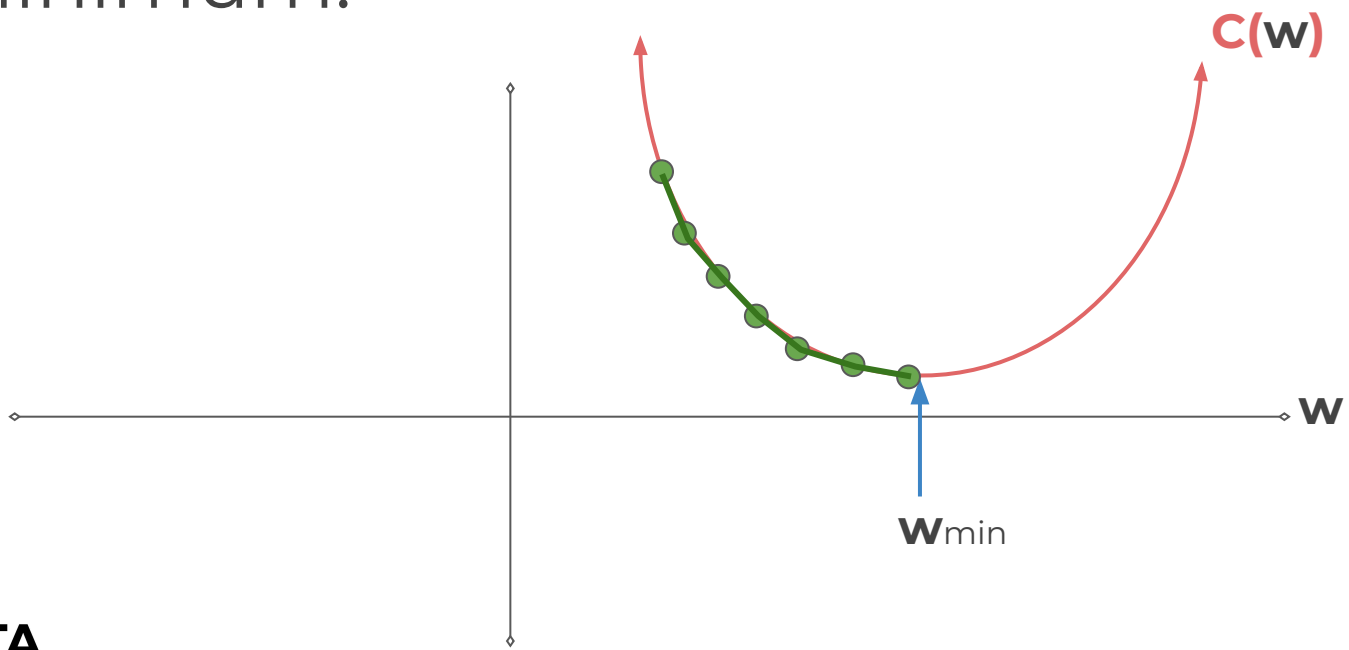
- Our steps:



PIERIAN DATA

Deep Learning

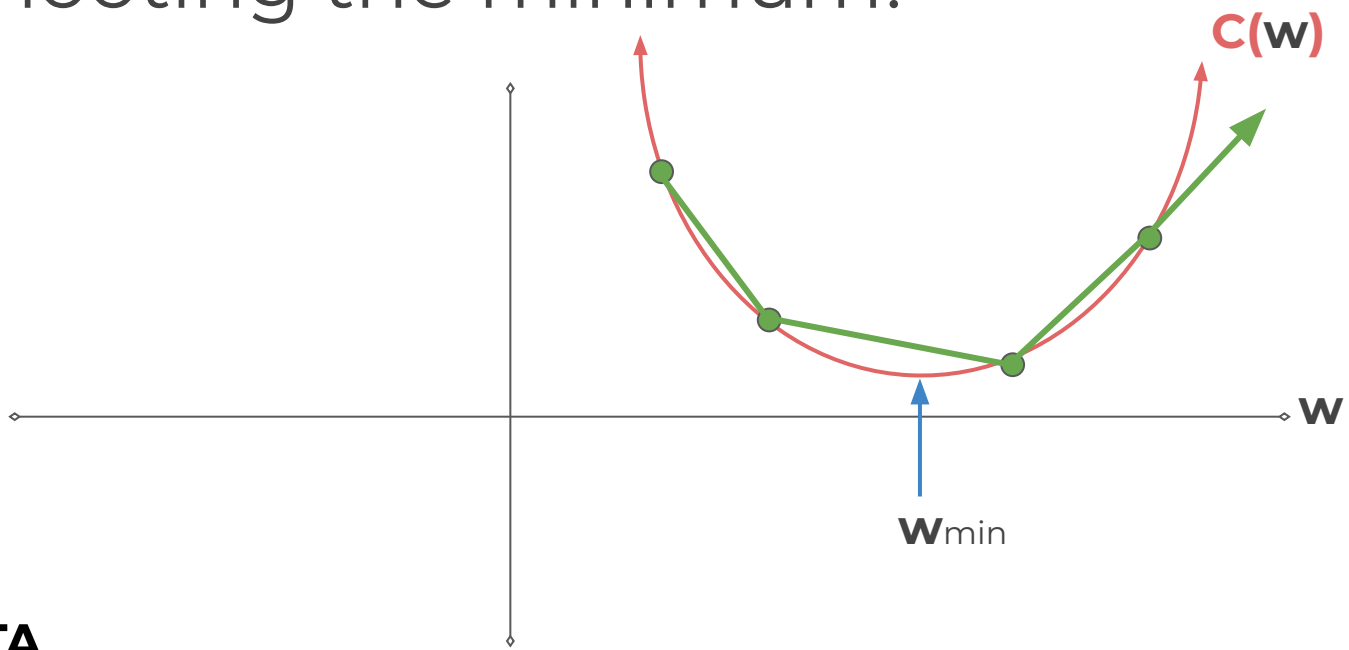
- Smaller steps sizes take longer to find the minimum.



PIERIAN DATA

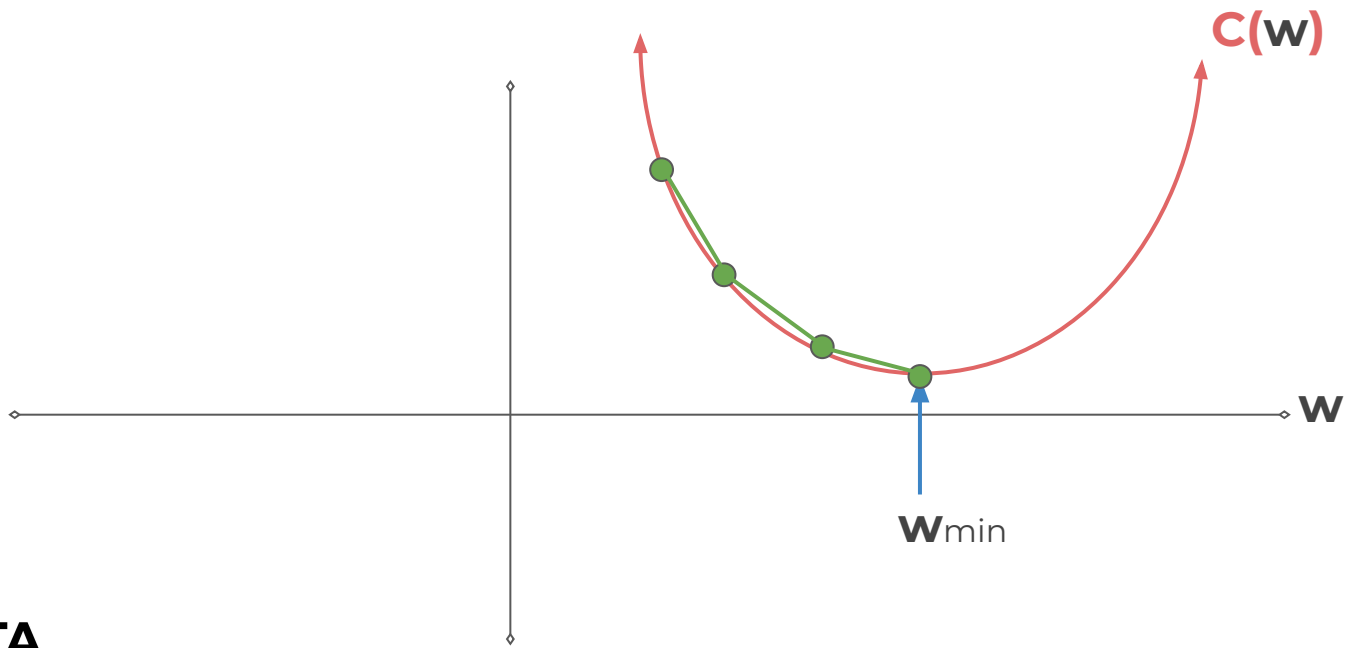
Deep Learning

- Larger steps are faster, but we risk overshooting the minimum!



Deep Learning

- This step size is known as the **learning rate.**



PIERIAN DATA

Deep Learning

- The learning rate we showed in our illustrations was constant (each step size was equal)
- But we can be clever and adapt our step size as we go along.

Deep Learning

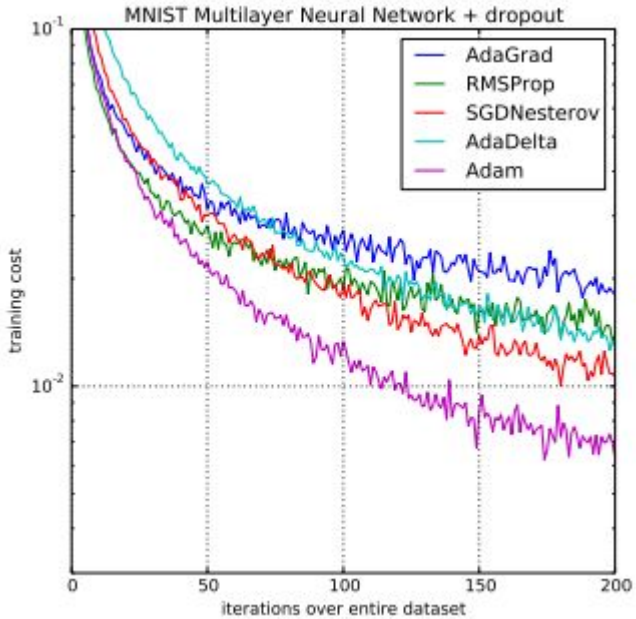
- We could start with larger steps, then go smaller as we realize the slope gets closer to zero.
- This is known as **adaptive gradient descent.**

Deep Learning

- In 2015, Kingma and Ba published their paper: “Adam: A Method for Stochastic Optimization“.
- Adam is a much more efficient way of searching for these minimums, so you will see us use it for our code!

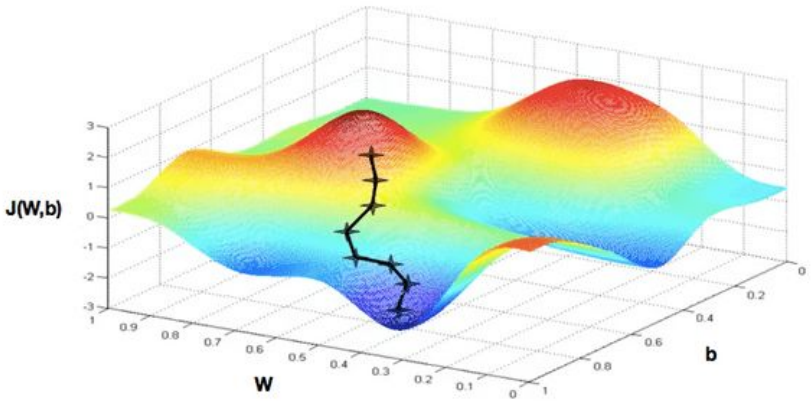
Deep Learning

- Adam versus other gradient descent algorithms:



Deep Learning

- Realistically we're calculating this descent in an n-dimensional space for all our weights.



Deep Learning

- When dealing with these N-dimensional vectors (tensors), the notation changes from **derivative** to **gradient**.
- This means we calculate

  $\nabla C(w1,w2,...wn)$

Deep Learning

- For classification problems, we often use the **cross entropy** loss function.
- The assumption is that your model predicts a probability distribution p(y=i) for each class i=1,2,...,C.

Deep Learning

- For a binary classification this results in:

$$-(y\log(p)+(1-y)\log(1-p))$$

- For **M** number of classes > 2

$$-\sum_{c=1}^{M} y_{o,c}\log(p_{o,c})$$

Deep Learning

- Review:
  - Cost Functions
  - Gradient Descent
  - Adam Optimizer
  - Quadratic Cost and Cross-Entropy

Deep Learning

- So far we understand how networks can take in input , effect that input with weights, biases, and activation functions to produce an estimated output.
- Then we learned how to evaluate that output.

Deep Learning

- The last thing we need to learn about theory is:
    - Once we get our cost/loss value, how do we actually go back and adjust our weights and biases?
- This is **backpropagation**, and it is what we are going to cover next!

Backpropagation

Deep Learning

- The last theory topic we will cover is **backpropagation.**
- We'll start by building an intuition behind backpropagation, and then we'll dive into the calculus and notation of backpropagation.

Deep Learning

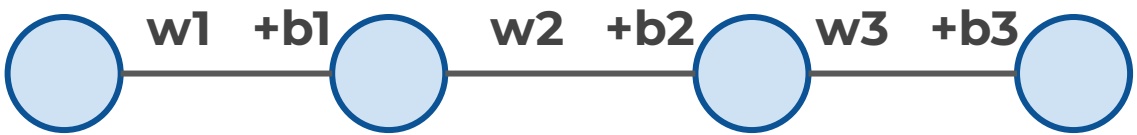
- Fundamentally, we want to know how the cost function results changes with respect to the weights in the network, so we can update the weights to minimize the cost function

Deep Learning

- Let’s begin with a very simple network, where each layer only has 1 neuron

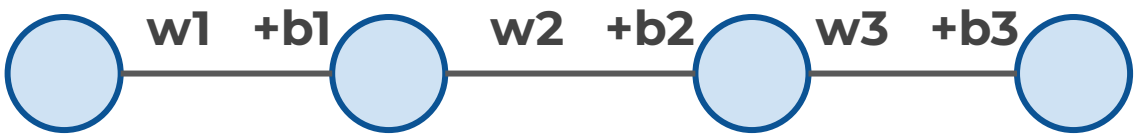
Deep Learning

- Each input will receive a weight and bias



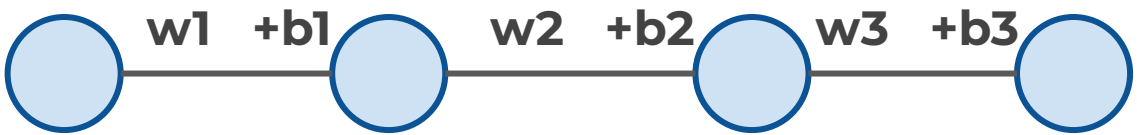
Deep Learning

- This means we have:
  - **C(w1,b1,w2,b2,w3,b3)**



Deep Learning

- We've already seen how this process propagates forward.
- Let's start at the end to see the backpropagation.

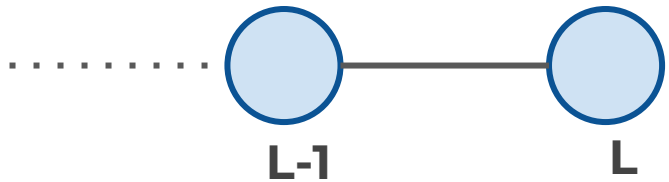


Deep Learning

- Let’s say we have **L** layers, then our notation becomes:

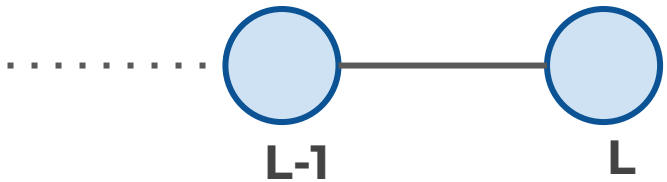
Deep Learning

- Focusing on these last two layers, let's define **z=wx+b**
- Then applying an activation function we'll state: **a = σ(z)**



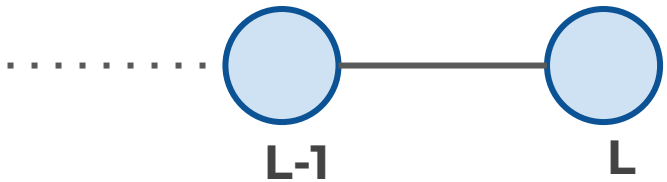
Deep Learning

- This means we have:
  - $z^L = w^L a^{L-1} + b^L$



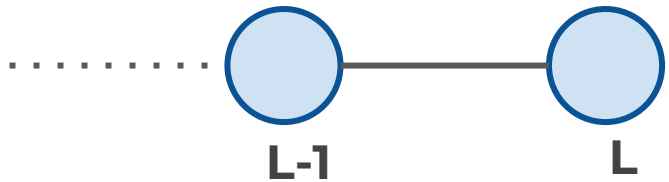
Deep Learning

- This means we have:
  - $z^L = w^L a^{L-1} + b^L$
  - $a^L = \sigma(z^L)$



Deep Learning

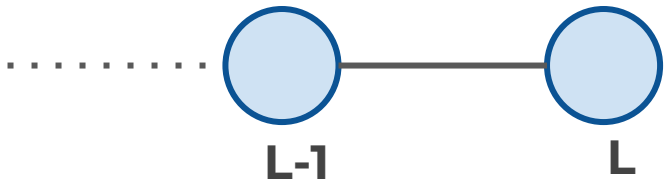
- This means we have:
  - $z^L = w^L a^{L-1} + b^L$
  - $a^L = \sigma(z^L)$
  - $C_0(...) = (a^L - y)^2$



Deep Learning

- We want to understand how sensitive is the cost function to changes in **w**:

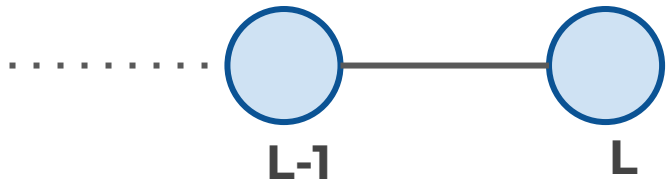
$$\frac{\partial C_0}{\partial w^L}$$



Deep Learning

- Using the relationships we already know along with the chain rule:

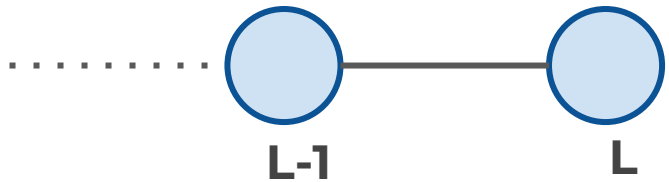
$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$



Deep Learning

- We can calculate the same for the bias terms:

$$\frac{\partial C_0}{\partial b^L} = \frac{\partial z^L}{\partial b^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$



Deep Learning

- The main idea here is that we can use the gradient to go back through the network and adjust our weights and biases to minimize the output of the error vector on the last output layer.

Deep Learning

- Using some calculus notation, we can expand this idea to networks with multiple neurons per layer.
- Hadamard Product

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

Deep Learning

- Given this notation and backpropagation, we have a few main steps to training neural networks.
- Note! You do not need to fully understand these intricate details to continue with the coding portions.

Deep Learning

- Step 1: Using input **x** set the activation function **a** for the input layer.
  - **z = w x +b**
  - **a = σ(z)**
- This resulting **a** then feeds into the next layer (and so on).

Deep Learning

- Step 2: For each layer, compute:
  - $z^L = w^L a^{L-1} + b^L$
  - $a^L = \sigma(z^L)$

Deep Learning

- Step 3: We compute our error vector:
  - $\delta^L = \nabla_a C \odot \sigma'(z^L)$

Deep Learning

- Step 3: We compute our error vector:
  - $\delta^L = \nabla_a C \odot \sigma'(z^L)$
    - $\nabla_a C = (a^L - y)$
    - **Expressing the rate of change of C with respect to the output activations**

Deep Learning

- Step 3: We compute our error vector:
  - $\delta^L = (a^L - y) \odot \sigma'(z^L)$

Deep Learning

- Step 3: We compute our error vector:
  - $\delta^L = (a^L - y) \odot \sigma'(z^L)$
- Now let's write out our error term for a layer in terms of the error of the next layer (since we're moving backwards).
- Font Note: **l**owercase **L**
- Font Note: Number **1**

Deep Learning

- Step 4: Backpropagate the error:
  - For each layer: L−1,L−2,... we compute (note the lowercase L (l)):
    - $\delta^l=(w^{l+1})^T\delta^{l+1}\odot\sigma'(z^l)$
    - $(w^{l+1})^T$ is the transpose of the weight matrix of **l+1** layer

Deep Learning

- Step 4: Backpropagate the error:
  - This is the generalized error for any layer **l**:
    - $\delta^l=(w^{l+1})^T\delta^{l+1}\odot\sigma'(z^l)$
    - $(w^{l+1})^T$ is the transpose of the weight matrix of L+1 layer

Deep Learning

- Step 4: When we apply the transpose weight matrix, $\mathbf{(w^{l+1})^T}$ we can think intuitively of this as moving the error backward through the network, giving us some sort of measure of the error at the output of the lth layer.

Deep Learning

- Step 4: We then take the Hadamard product $\odot\sigma'(z^l)$. This moves the error backward through the activation function in layer l, giving us the error δ^l in the weighted input to layer l.

Deep Learning

- The gradient of the cost function is given by:
  - For each layer: L−1,L−2,... we compute

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Deep Learning

- This then allows us to adjust the weights and biases to help minimize that cost function.
- Check out the external links for more details!

TensorFlow and Keras

Deep Learning

- Before we begin learning how to code our own neural networks, let’s quickly clarify the differences between TensorFlow and Keras!

Deep Learning

- TensorFlow is an open-source deep learning library developed by Google, with TF 2.0 being officially released in late 2019.

Deep Learning

- TensorFlow has a large ecosystem of related components, including libraries like Tensorboard, Deployment and Production APIs, and support for various programming languages.

Deep Learning

- Keras is a high-level python library that can use a variety of deep learning libraries underneath, such as: TensorFlow, CNTK, or Theano.

Deep Learning

- TensorFlow 1.x had a complex python class system for building models, and due to the huge popularity of Keras, when TF 2.0 was released, TF adopted Keras as the official API for TF.

Deep Learning

- While Keras still also remains as a separate library from Tensorflow, it can also now officially be imported through TF, so there is now need to additionally install it.

Deep Learning

- The Keras API is easy to use and builds models by simply adding layers on top of each other through simple calls.
- Let’s now explore the basics of the Keras API for TensorFlow!

Keras Classification Code Along - Part One

PIERIAN DATA

Deep Learning

- This lecture will show how to perform a classification task with TensorFlow.
- We will also focus on how to identify and deal with overfitting through Early Stopping Callbacks and Dropout Layers.

Deep Learning

- Early Stopping
  - Keras can automatically stop training based on a loss condition on the validation data passed during the model.fit() call.

Deep Learning

- Dropout Layers
  - Dropout can be added to layers to “turn off” neurons during training to prevent overfitting.

Deep Learning

- Dropout Layers
  - Each Dropout layer will “drop” a user-defined percentage of neuron units in the previous layer every batch.

Introduction to Project

Deep Learning

- It’s time for a project! You will build a model that will attempt to predict whether or not someone will pay back their loan based on historical information.

Deep Learning

- You have 3 options for this project
  - Completely Solo
  - Exercise Guide Notebook
  - Code Along with Solution Lectures

Deep Learning

- Completely Solo
  - Read the introduction in the Exercise notebook, then proceed with your own methods to build a predictive model.

Deep Learning

- Exercise Guide Notebook
  - Follow the written steps in the notebook to complete tasks that guide you through building a predictive model.

Deep Learning

- Code along with solutions lecture
  - Skip the next overview lecture and code along with us as we guide you through our solution video for the project.

Deep Learning

- This is a large project!
- To reflect a realistic situation, we will spend a lot of time performing feature engineering and analyzing our data.
- Let’s explore the project guide notebook in the next lecture!

Keras Project Exercise Solutions

Exploratory Data Analysis

PIERIAN DATA

Keras Project Exercise Solutions

Data Preprocessing: Missing Data

PIERIAN DATA

Keras Project Exercise Solutions

Categorical Data

PIERIAN DATA

Keras Project Exercise Solutions

Data Preprocessing

PIERIAN DATA

Keras Project Exercise Solutions

Creating and Training a Model

PIERIAN DATA

Keras Project Exercise Solutions

Model Evaluation

PIERIAN DATA

Keras Project Exercise Solutions

Model Evaluation

PIERIAN DATA

Tensorboard

Deep Learning

- Tensorboard is a visualization tool from Google designed to work in conjunction with TensorFlow to visualize various aspects of your model.

Deep Learning

- Here we will simply understand how to view the Tensorboard dashboard in our browser and analyze an existing model.
- **NOTE - This lecture requires that you understand file paths and the location of your notebook or .py file!**

Deep Learning

- Keep in mind, Tensorboard is a separate library from TensorFlow.
- Google Collab Users can follow with Google's official guide and pre-made notebook:

https://www.tensorflow.org/tensorboard/tensorboard_in_notebooks