Deep Learning for Time Series

PIERIAN DATA

Section Overview

- In this section of the course we will focus on learning about Neural Networks and how they can be used for Time Series Forecasting.
- Let’s explore what topics this section will cover!

Section Overview

- Perceptron Model
- Neural Networks
- Keras Basics for Regression Task
- Recurrent Neural Networks
- LSTM and GRU Neurons
- Time Series Forecasting with RNN

Section Overview

- Keep in mind Neural Networks in general tend to be “black boxes” so it’s very difficult to interpret them beyond their performance metrics.
- ARIMA based models are much easier to understand and work with (and often perform better!)

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

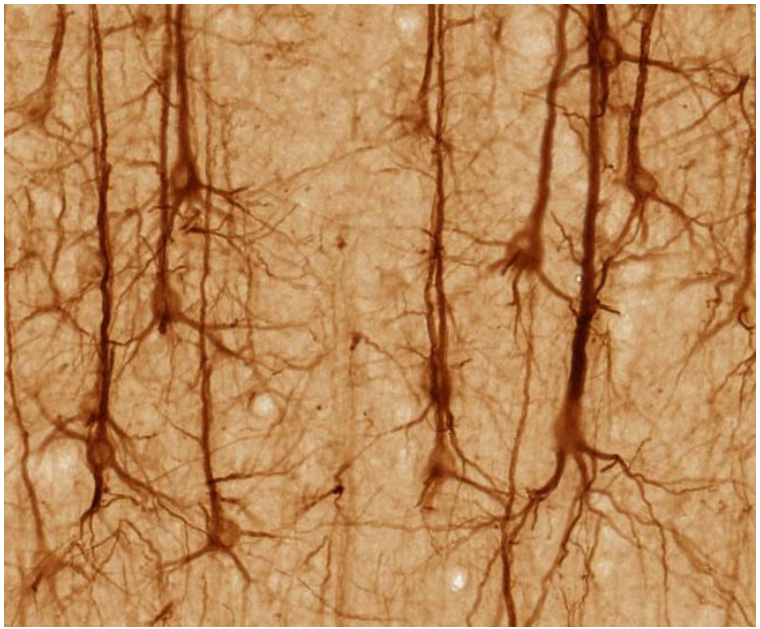
- As we learn about more complex models, we'll also introduce concepts, such as:
  - Activation Functions
  - Gradient Descent
  - BackPropagation

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!

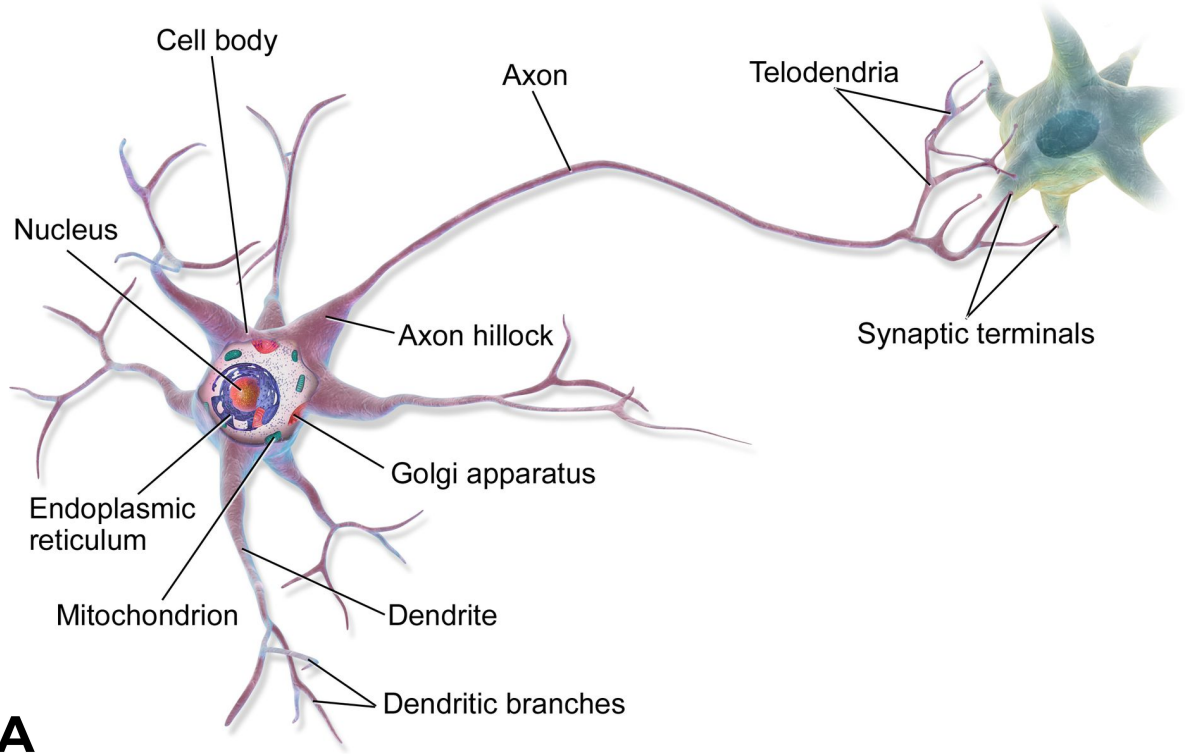
Perceptron model

- Stained Neurons in cerebral cortex



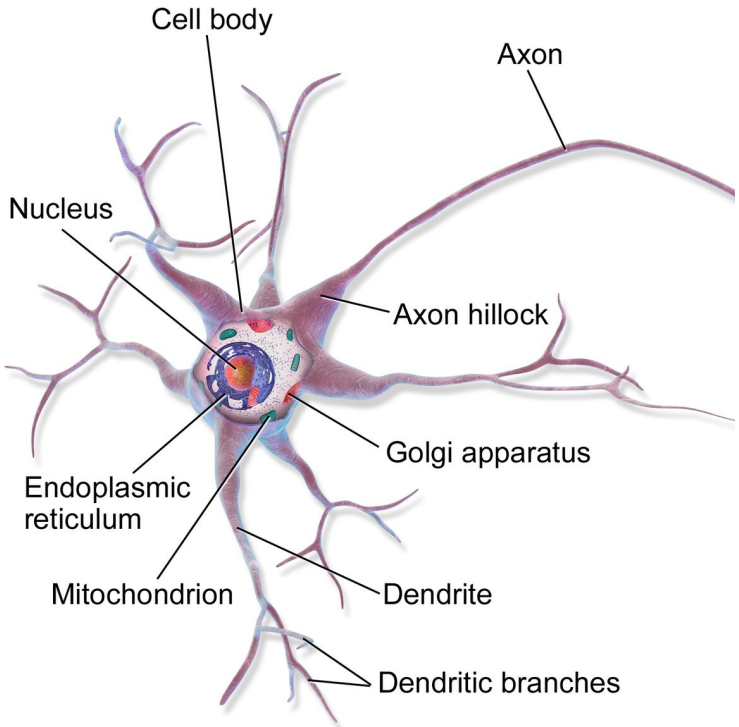
Perceptron model

- Illustration of biological neurons



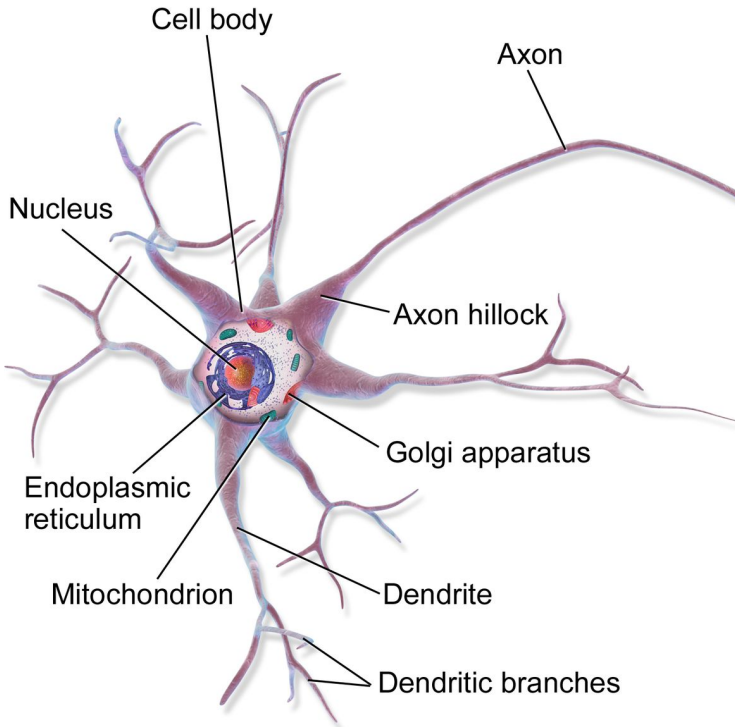
Perceptron model

- Illustration of biological neurons



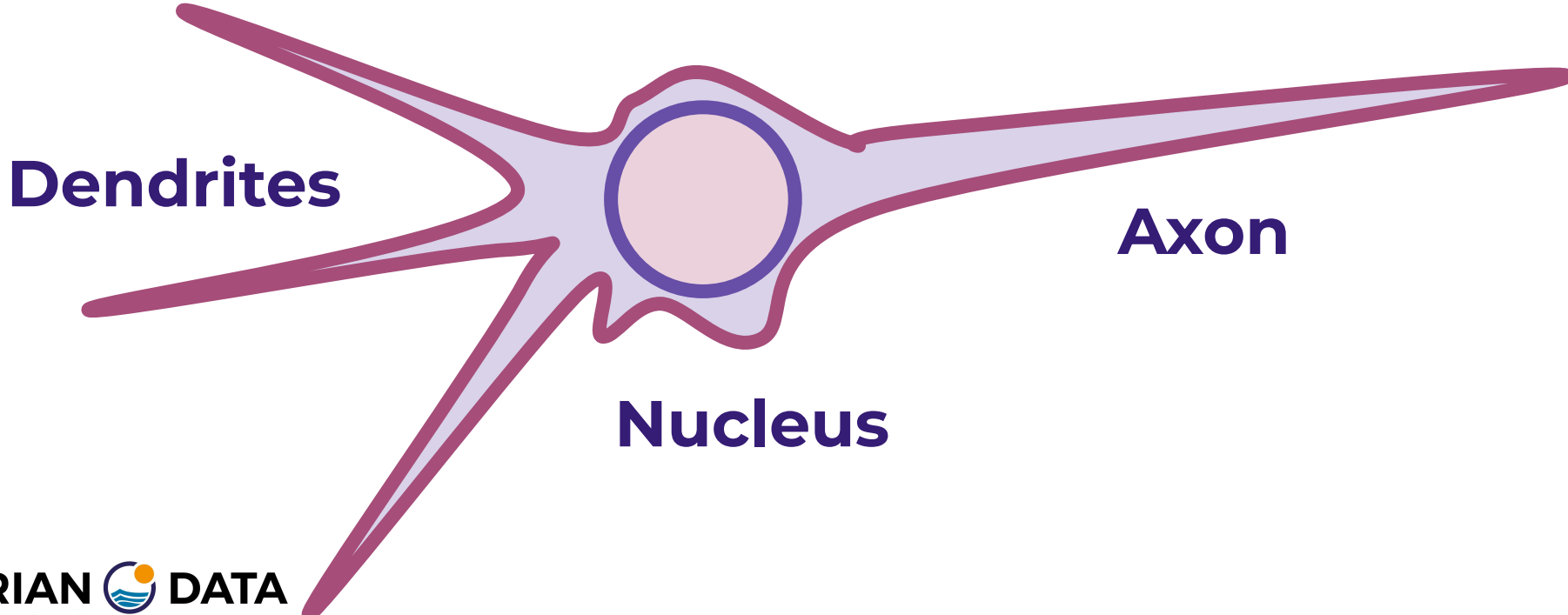
Perceptron model

- 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

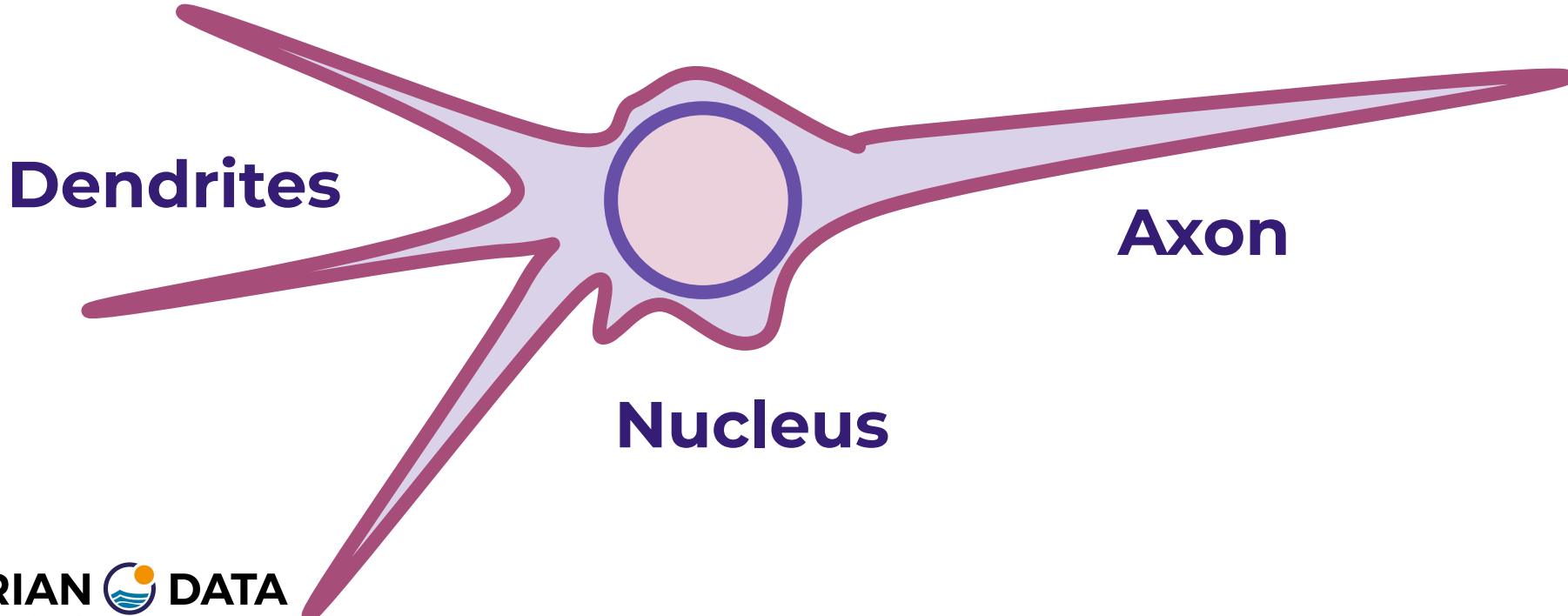
- However, in 1969 Marvin Minsky and Seymour Papert's published their book ***Perceptrons***.
- It suggested that there were severe limitations to what perceptrons could do.
- This marked the beginning of what is known as the AI Winter, with little funding into AI and Neural Networks in the 1970s.

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.

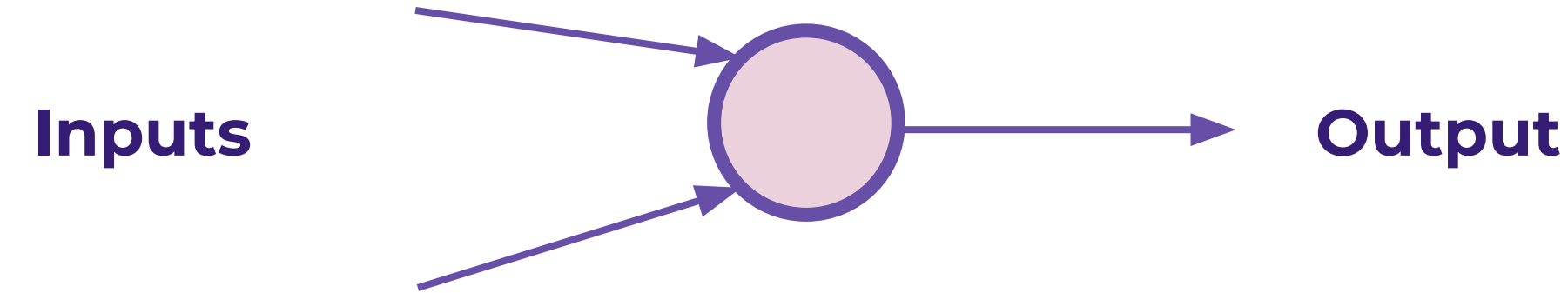
Perceptron model

- Perceptron Model



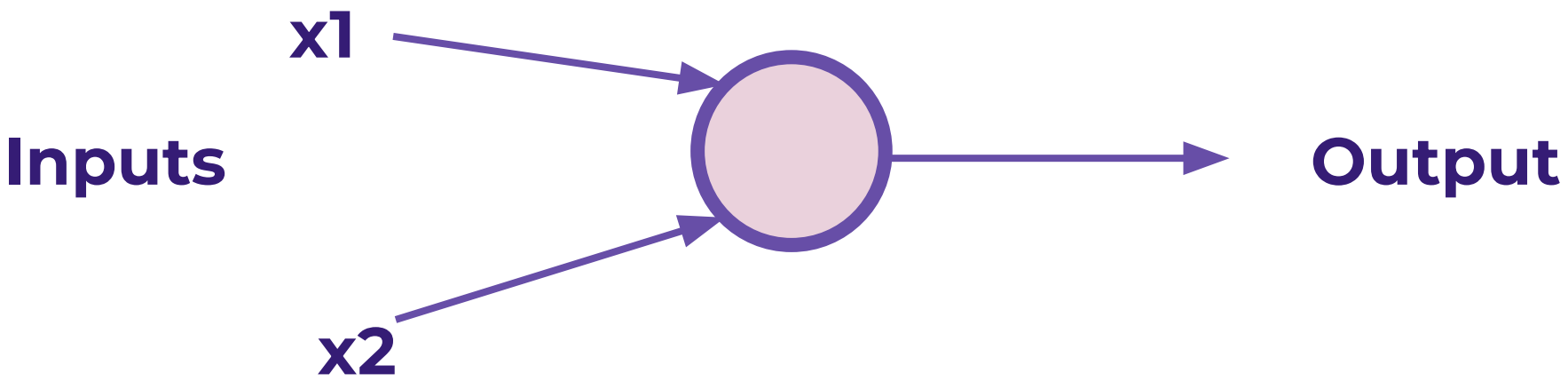
Perceptron model

- Perceptron Model



Perceptron model

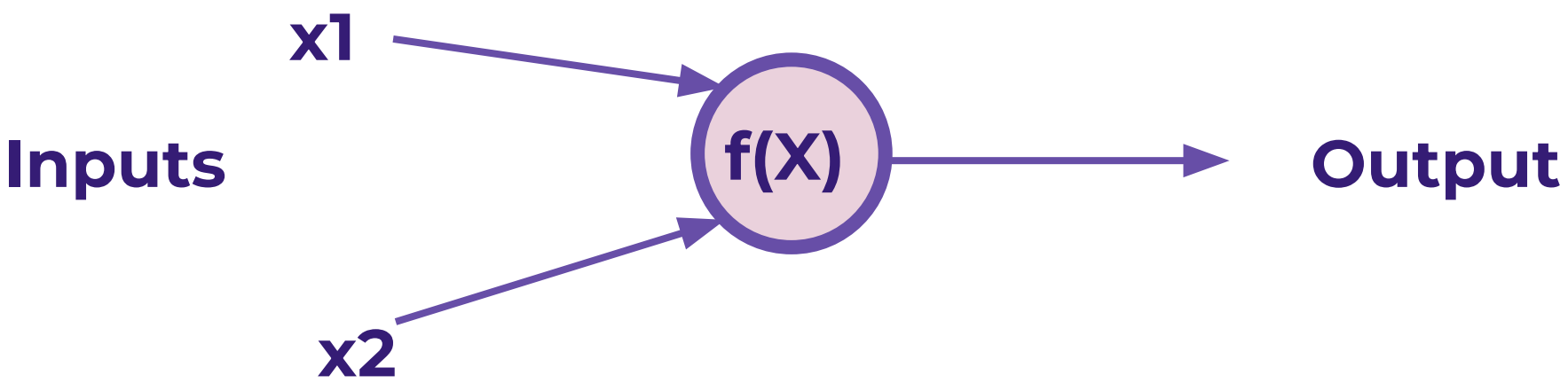
- Let's work through a simple example



PIERIAN DATA

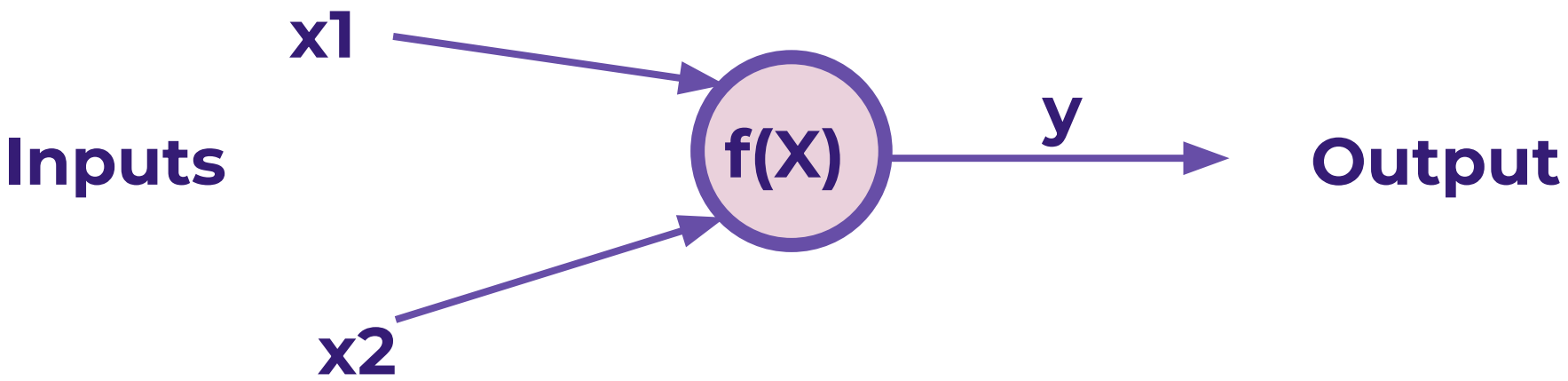
Perceptron model

- Let's work through a simple example



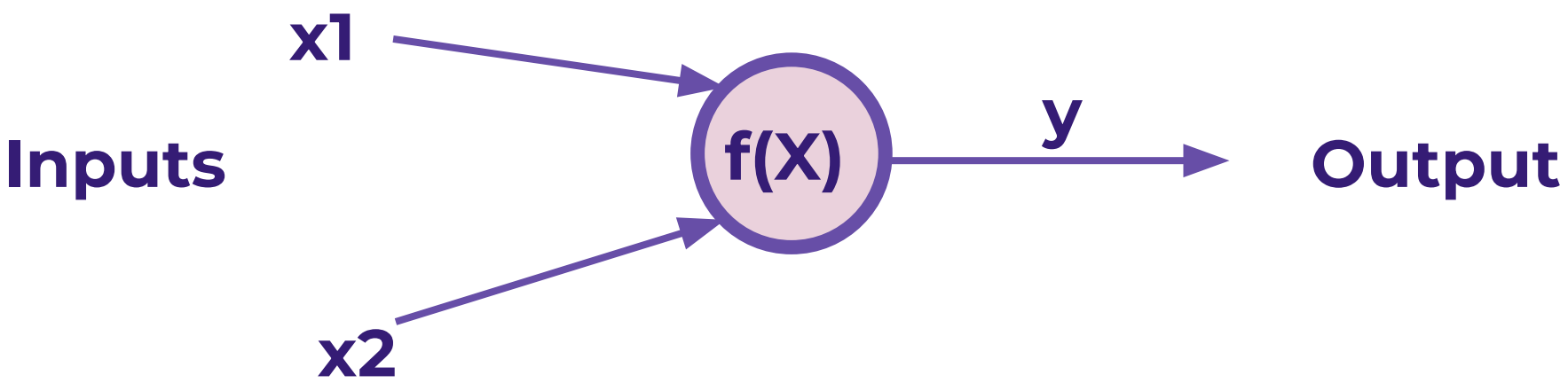
Perceptron model

- Let's work through a simple example



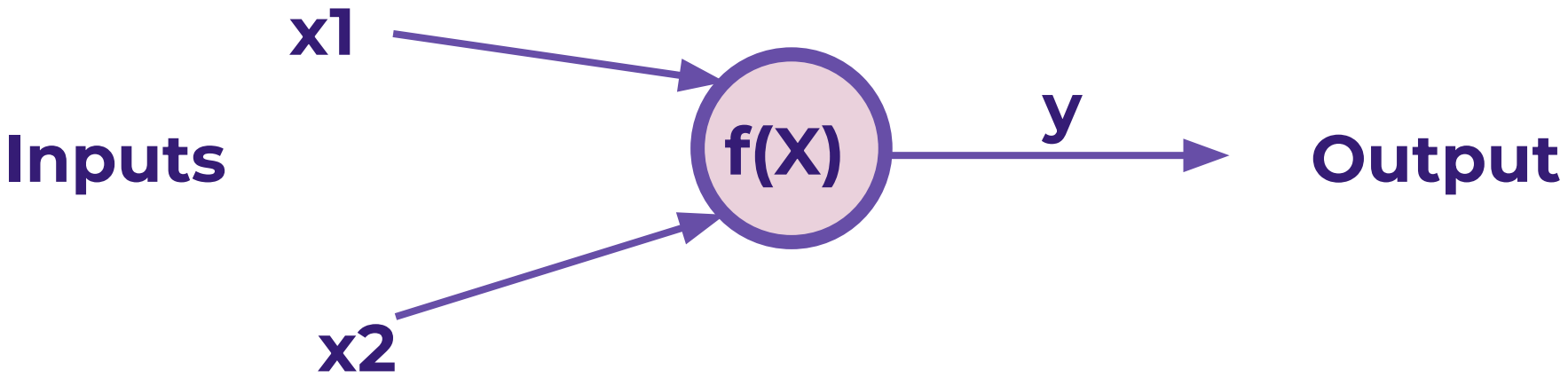
Perceptron model

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



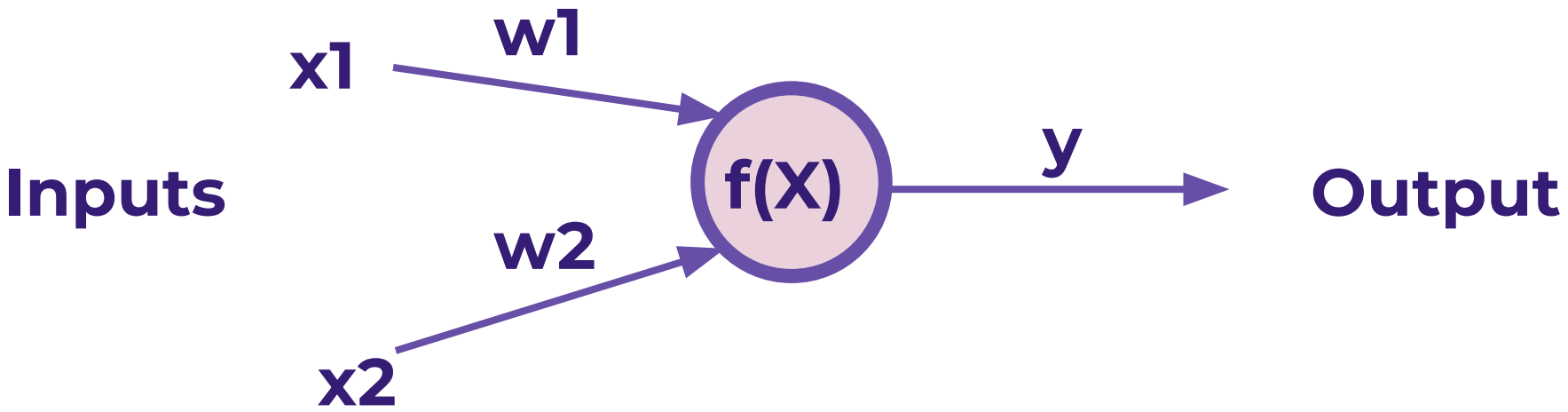
Perceptron model

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



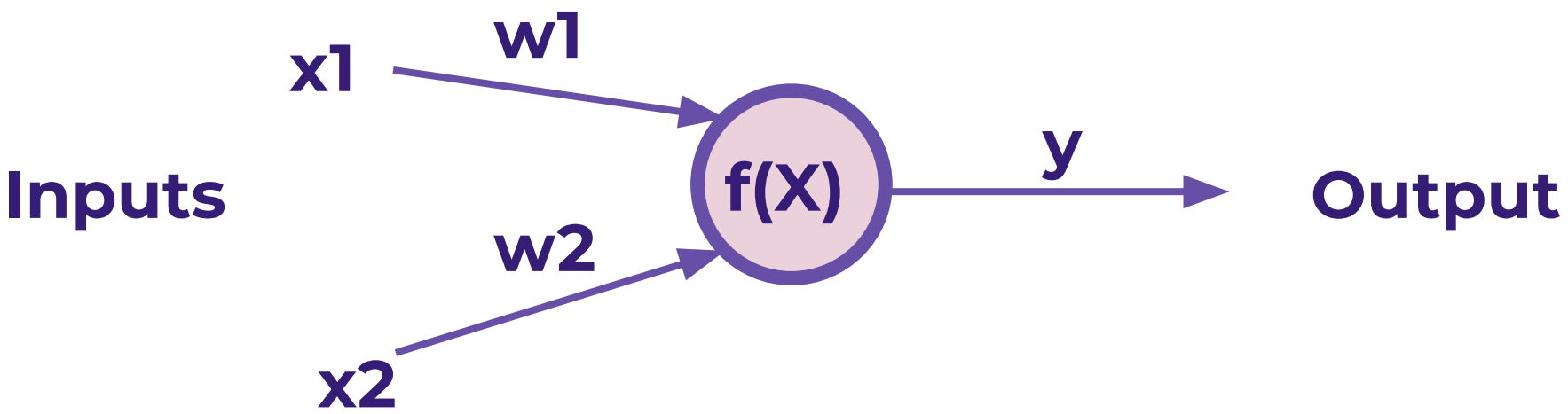
Perceptron model

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



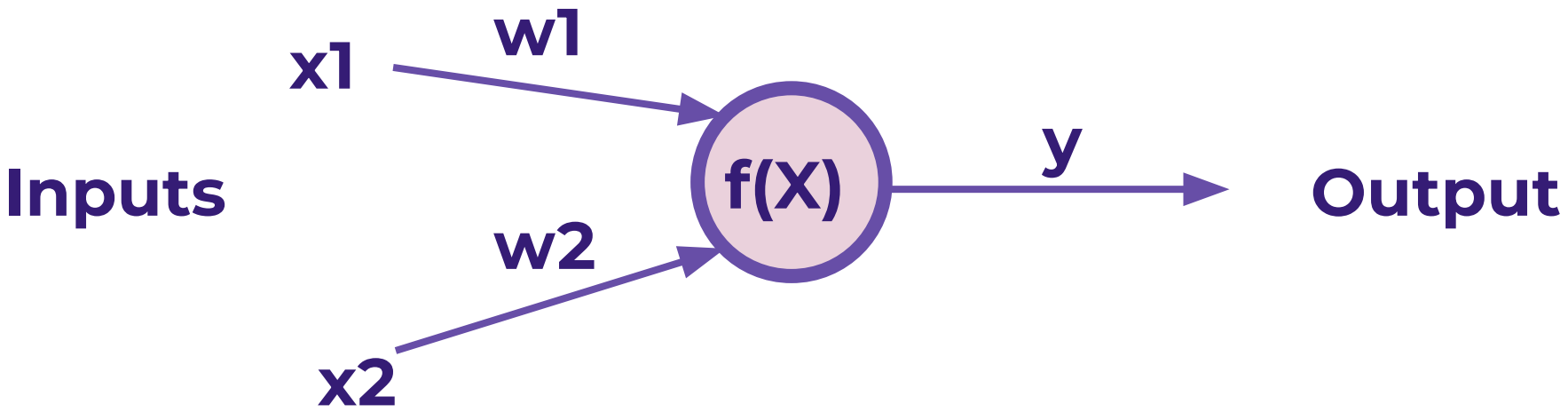
Perceptron model

- Now **y = x1w1 + x2w2**



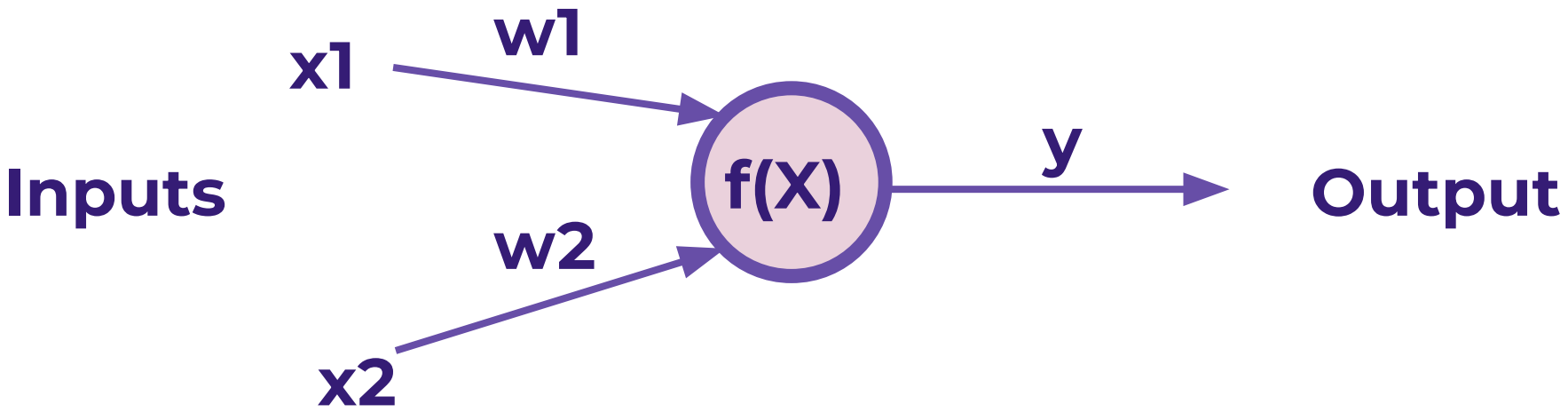
Perceptron model

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



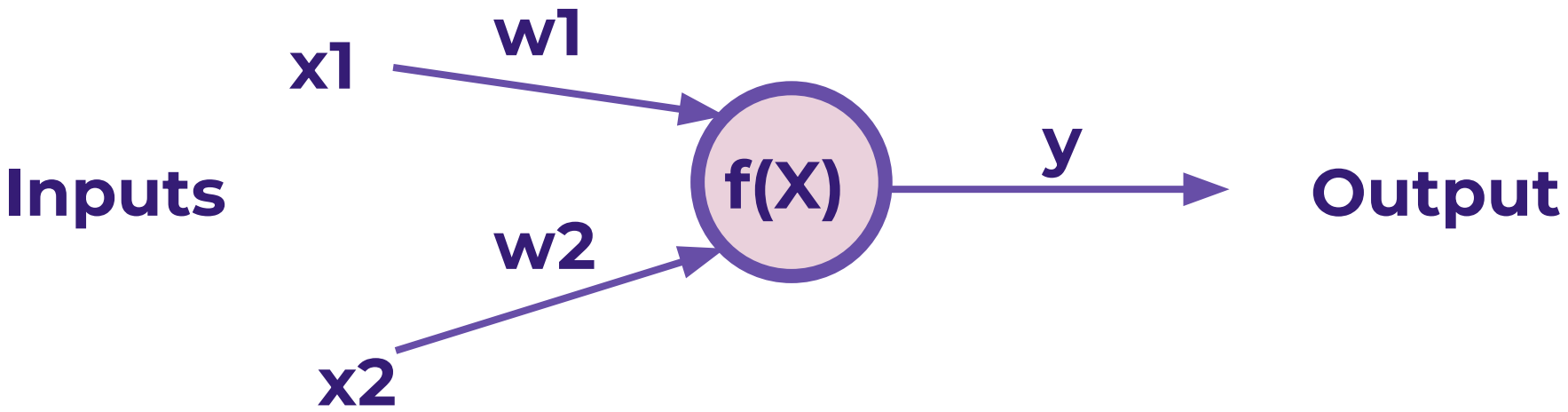
Perceptron model

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



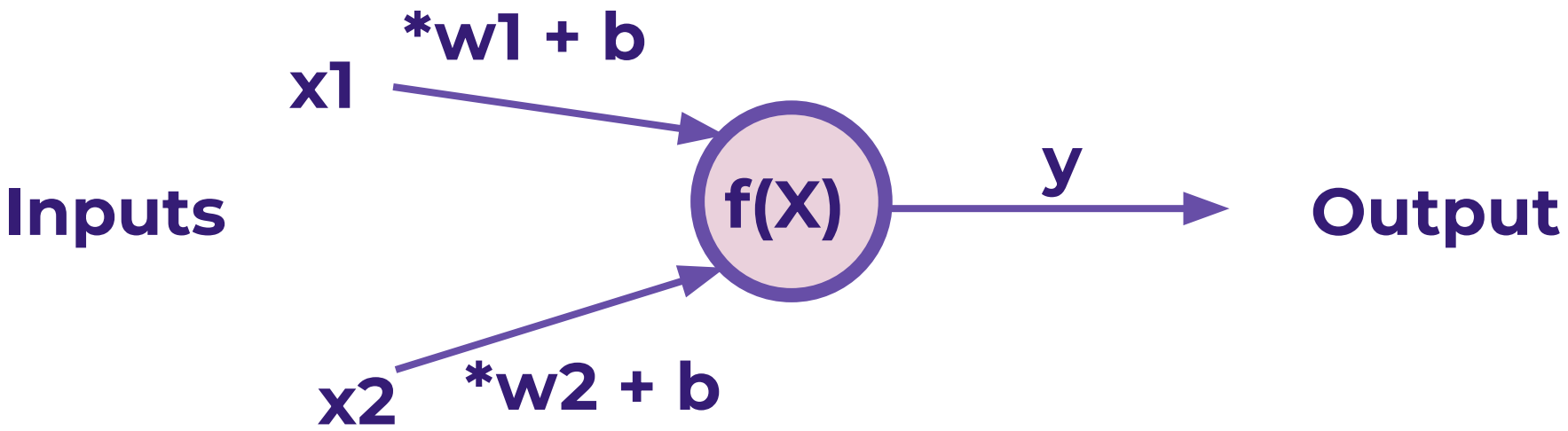
Perceptron model

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



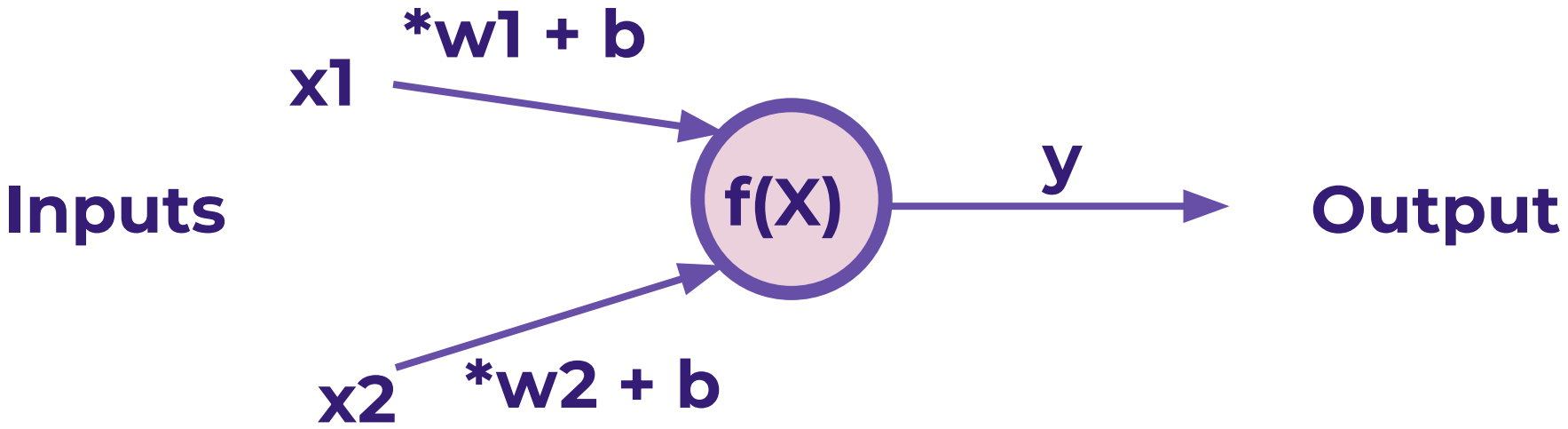
Perceptron model

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



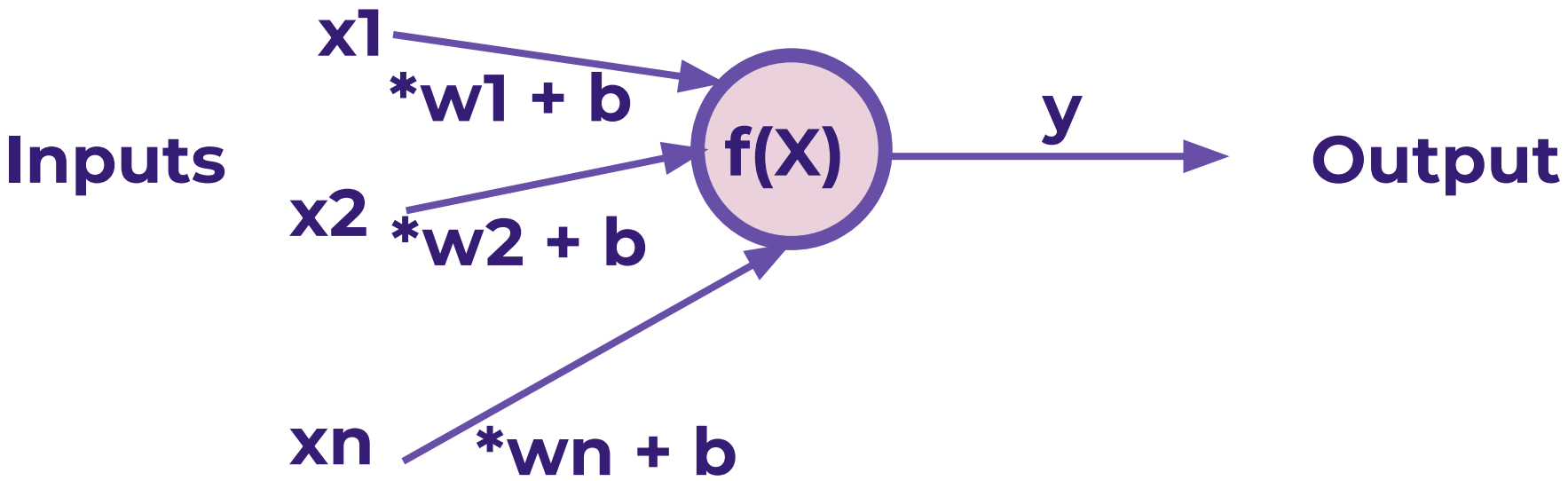
Perceptron model

- 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

- 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

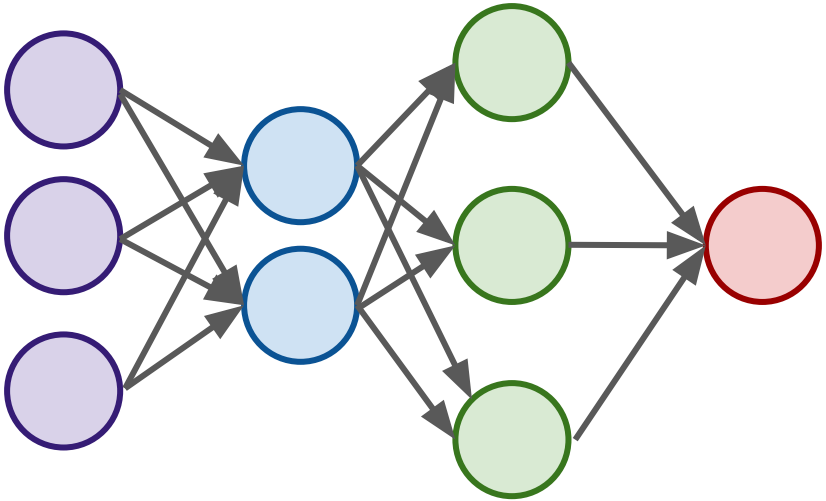
- A single perceptron won't be enough to learn complicated systems.
- Fortunately, we can expand on the idea of a single perceptron, to create a multi-layer perceptron model.

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.

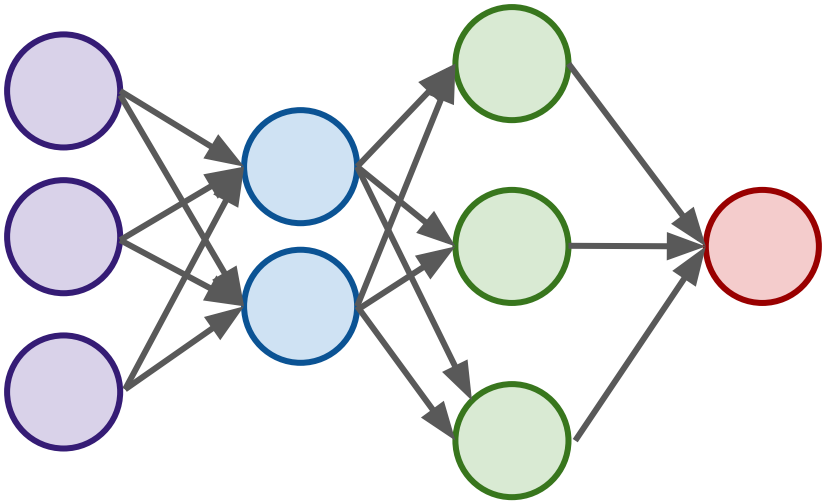
Neural Networks

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



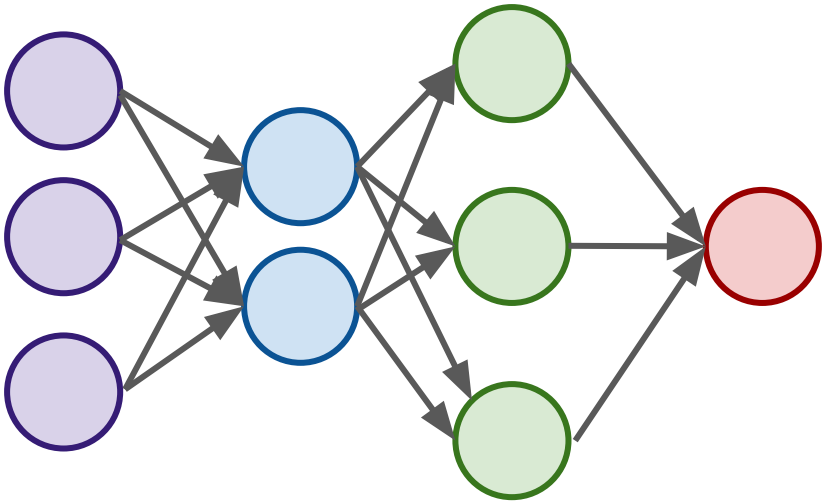
Neural Networks

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



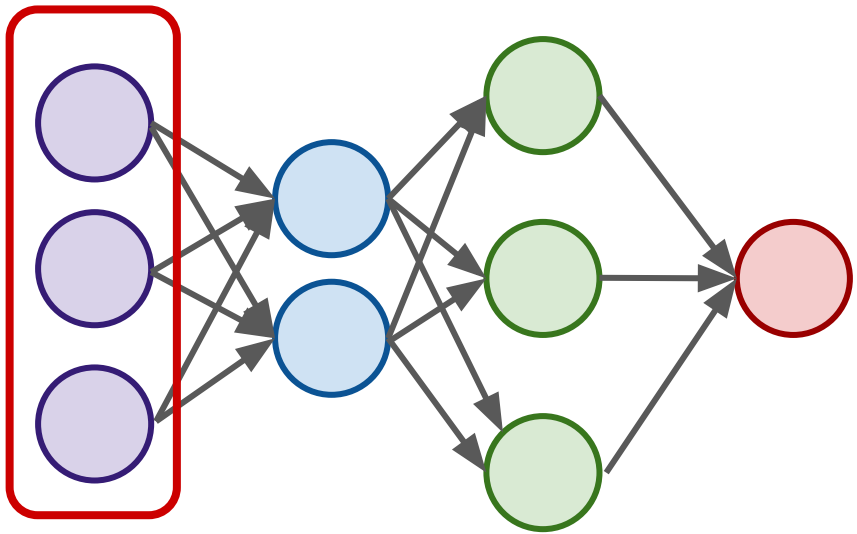
Neural Networks

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



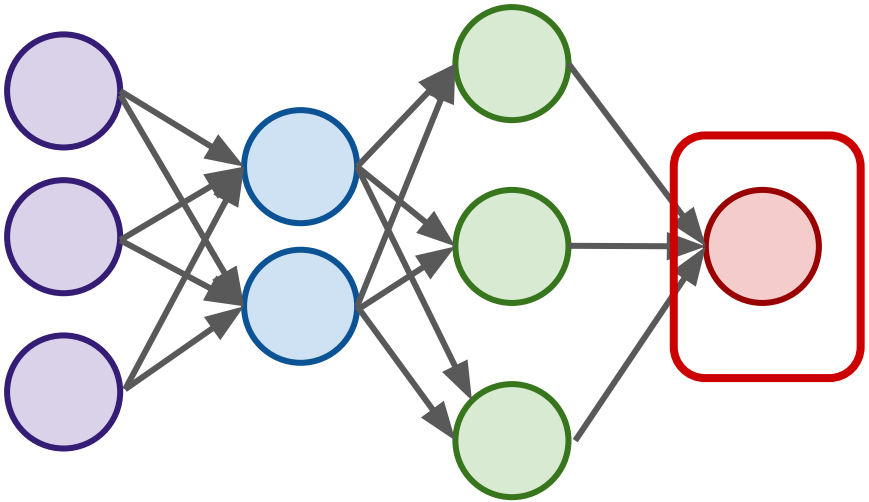
Neural Networks

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



Neural Networks

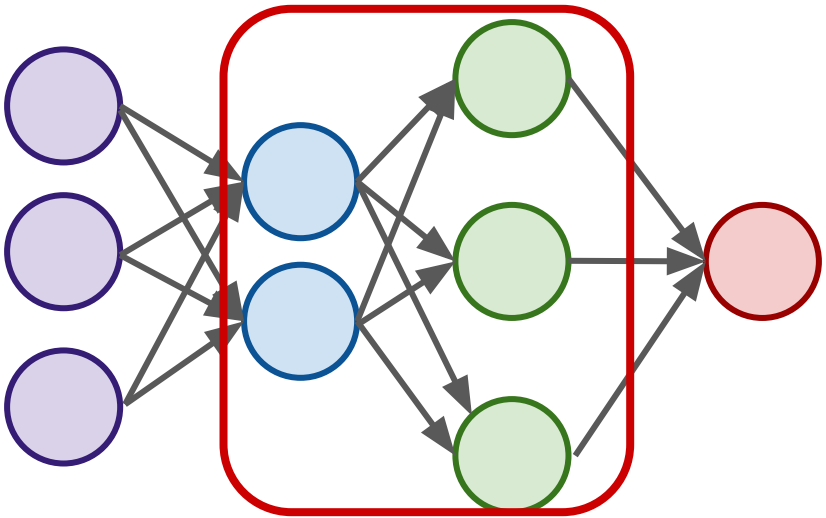
- The last layer is the **output layer.**
- Note: This last layer can be more than one neuron



PIERIAN DATA

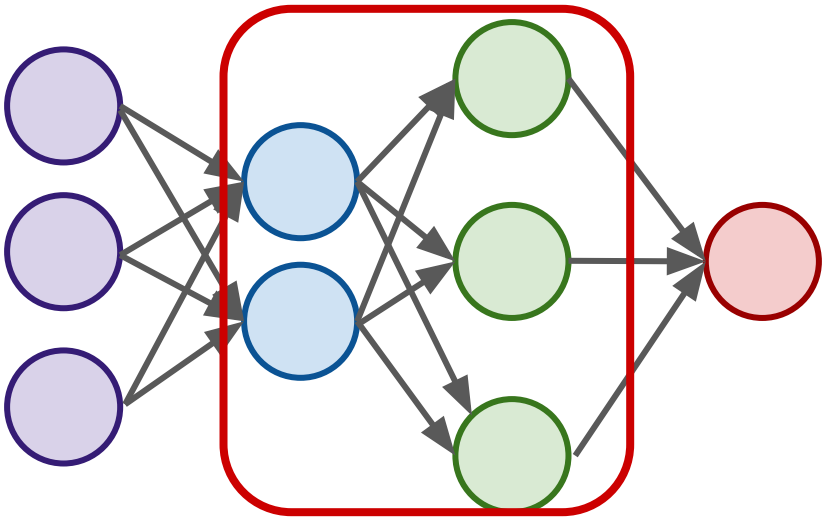
Neural Networks

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



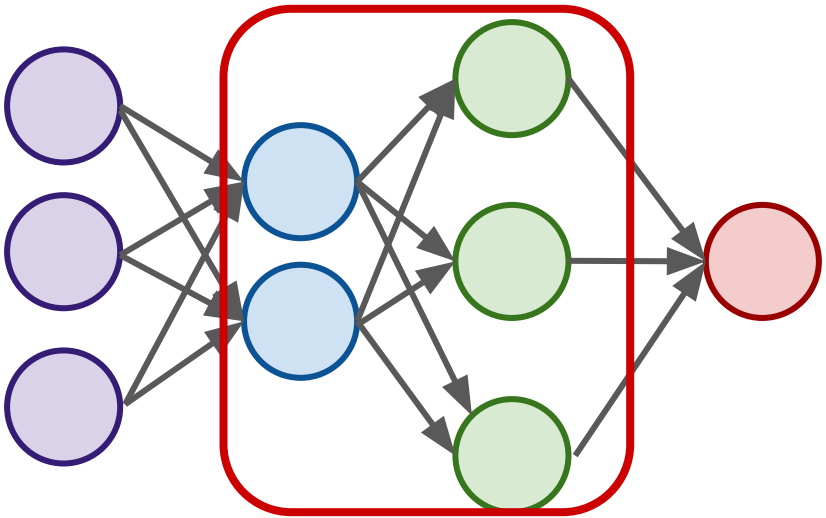
Neural Networks

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



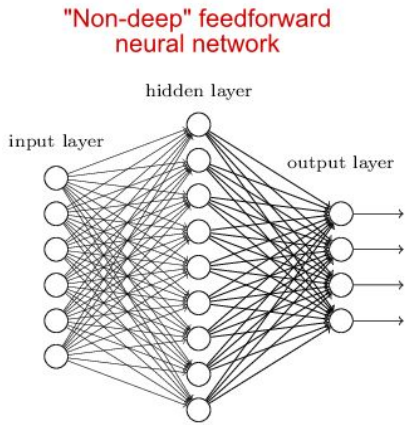
Neural Networks

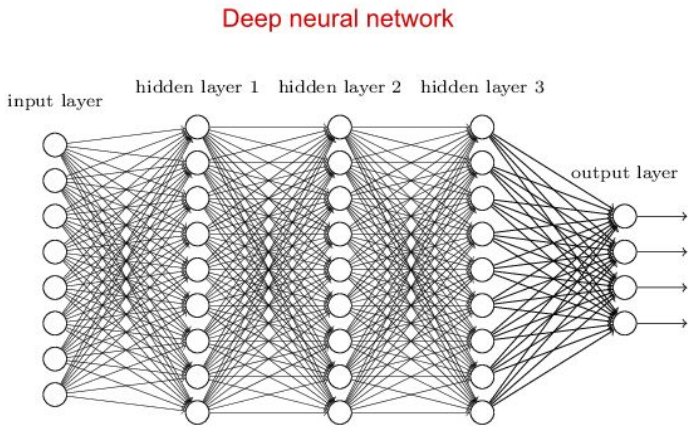
- Neural Networks become **“deep neural networks”** if then contain 2 or more hidden layers.



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

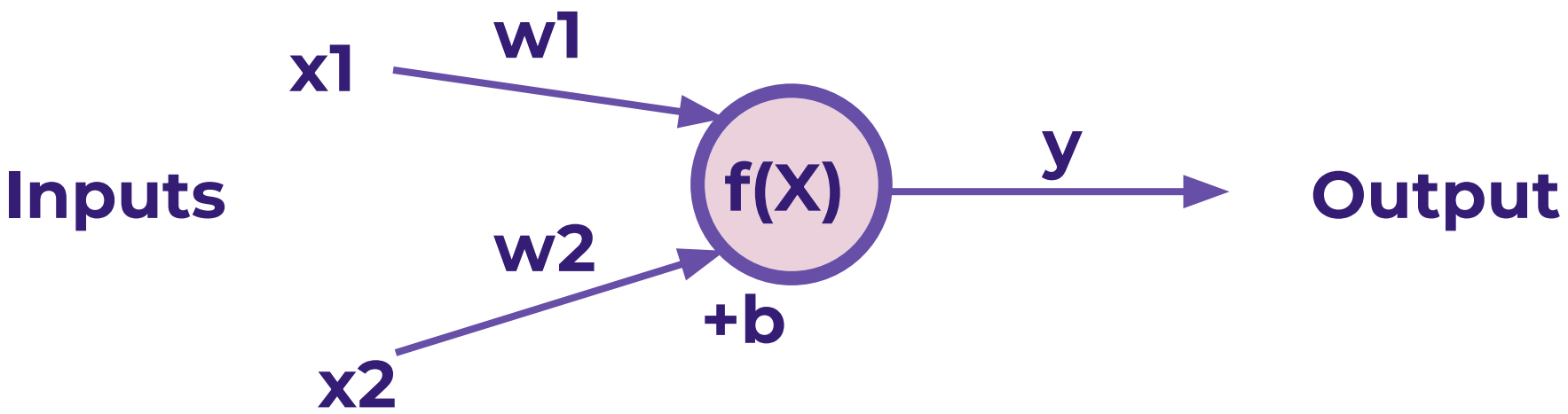
- Next we want to set boundaries for the overall output value of:
    - **x*w + b**
- We can state:
    - **z = x*w + b**
- And then pass **z** through some activation function to limit its value.

Neural Networks

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

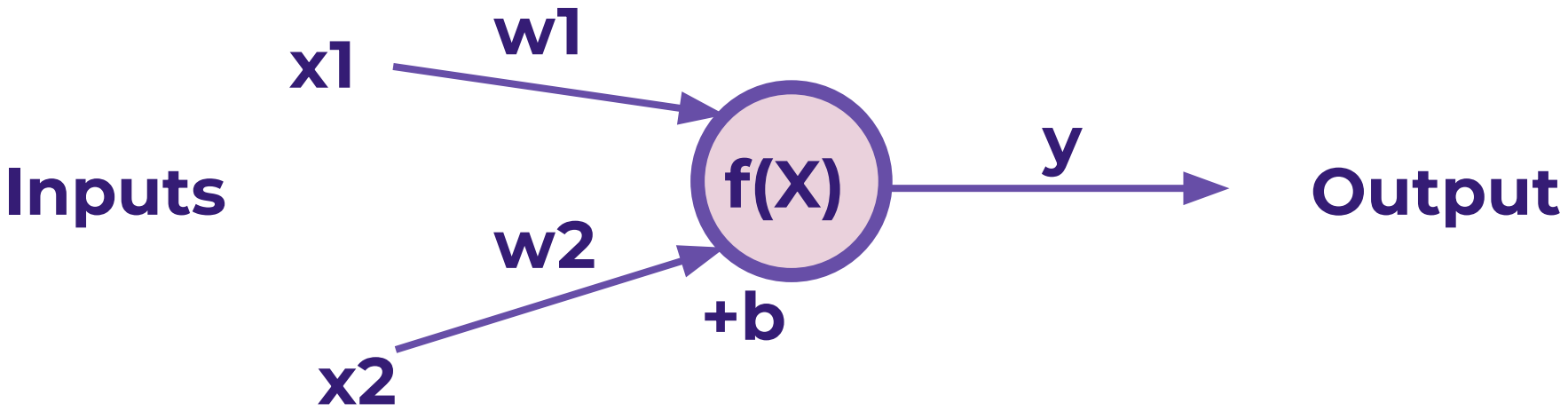
Perceptron model

- 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.

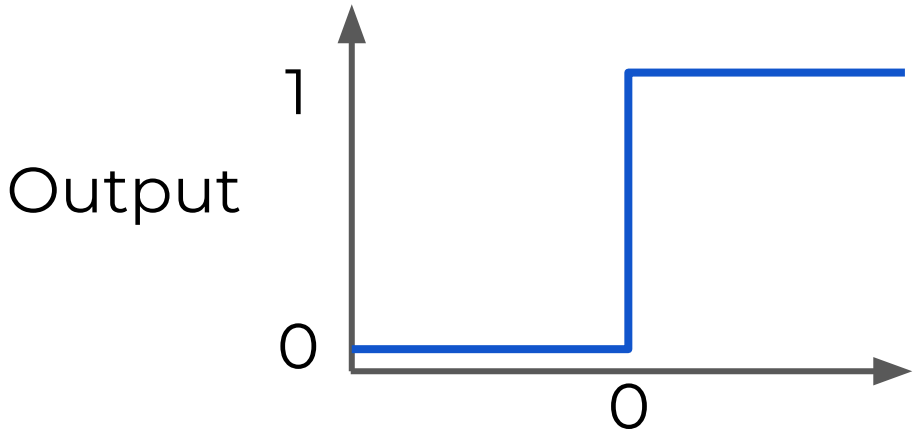


Neural Networks

- 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

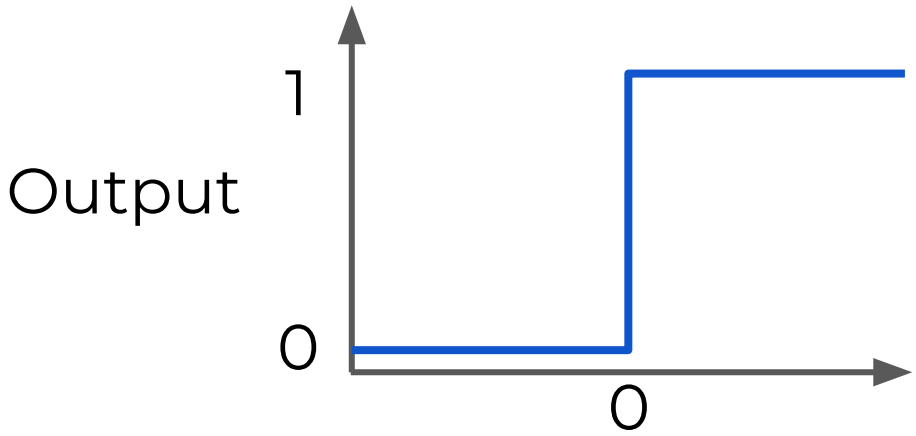
- The most simple networks rely on a basic **step function** that outputs 0 or 1.



$$z = wx + b$$

Deep Learning

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

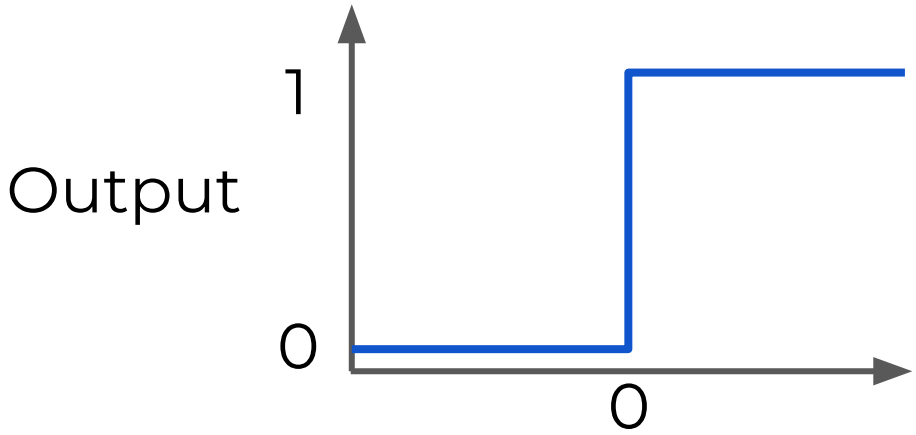


$$z = wx + b$$

PIERIAN DATA

Deep Learning

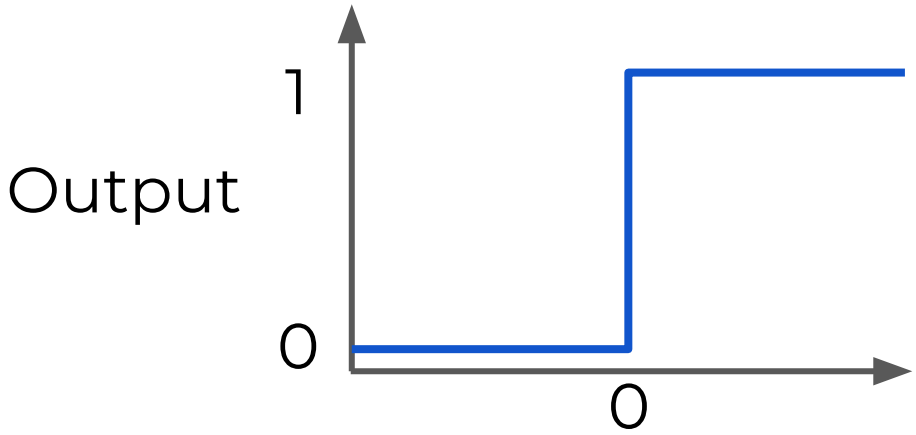
- This sort of function could be useful for classification (0 or 1 class).



$$z = wx + b$$

Deep Learning

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

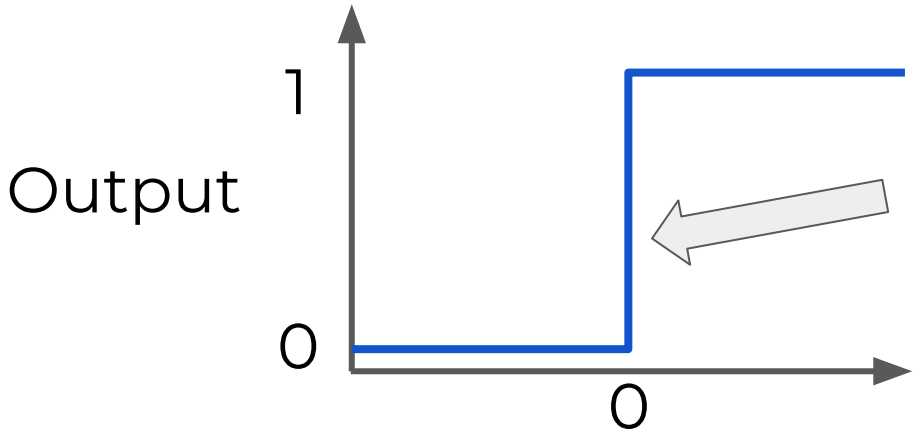


$$z = wx + b$$

PIERIAN DATA

Deep Learning

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

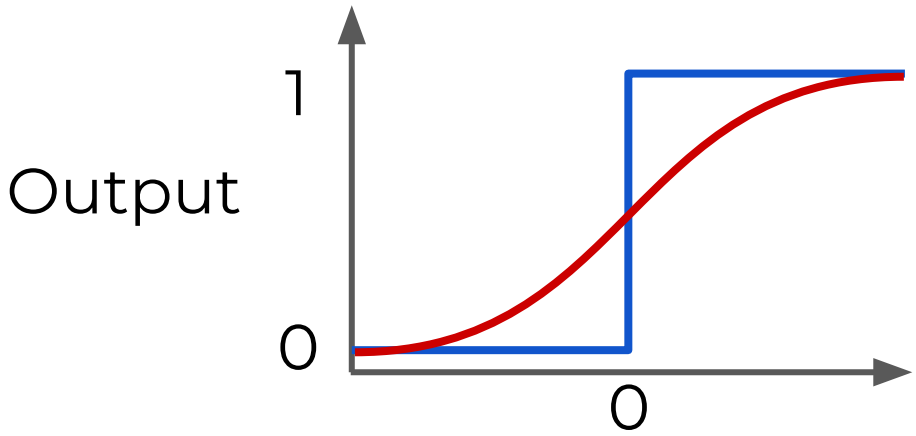


$$z = wx + b$$

PIERIAN DATA

Deep Learning

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

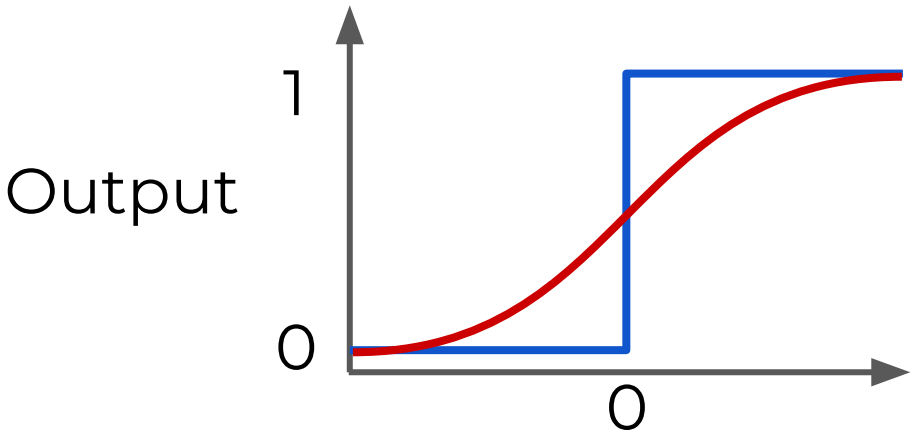


$$z = wx + b$$

PIERIAN DATA

Deep Learning

- Lucky for us, this is the sigmoid function!

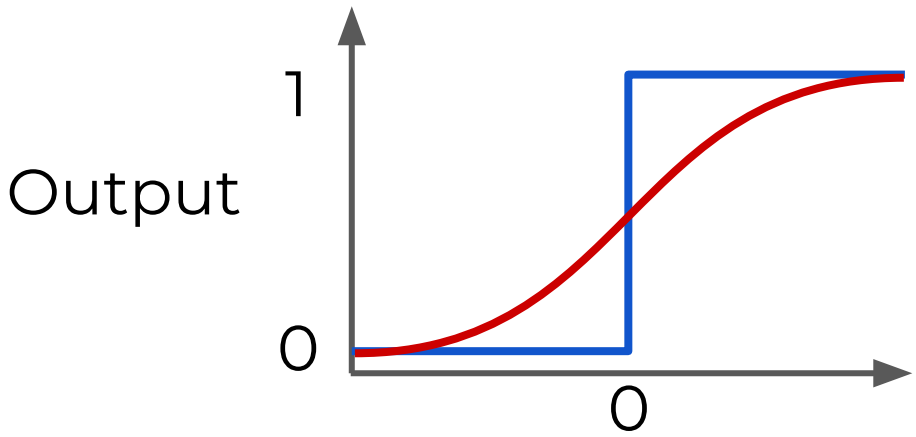


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

$$z = wx + b$$

Deep Learning

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

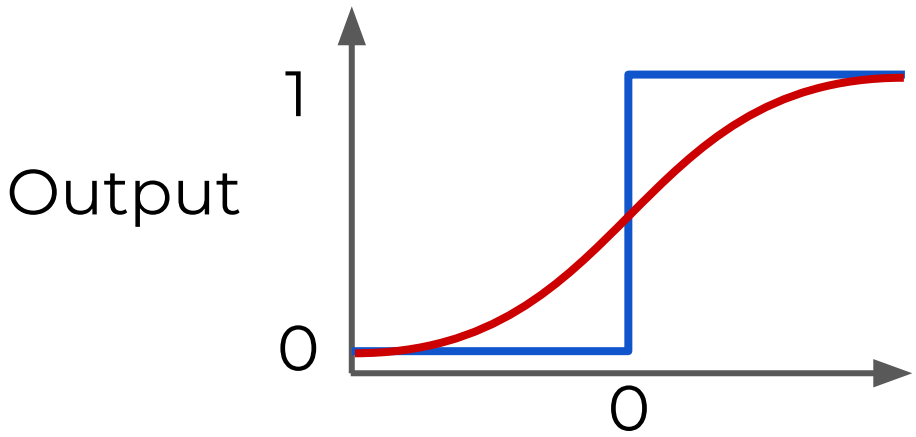


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

z = wx + b

Deep Learning

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

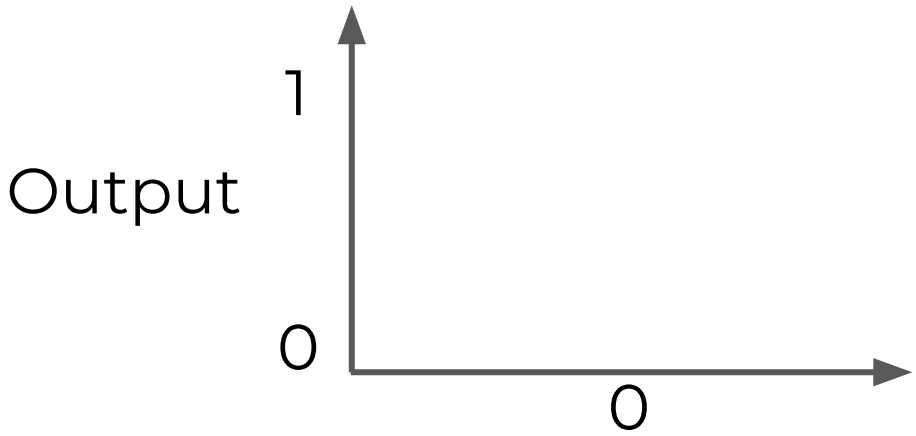


$$f(z) = \frac{1}{1 + e^{(-z)}}$$

z = wx + b

Deep Learning

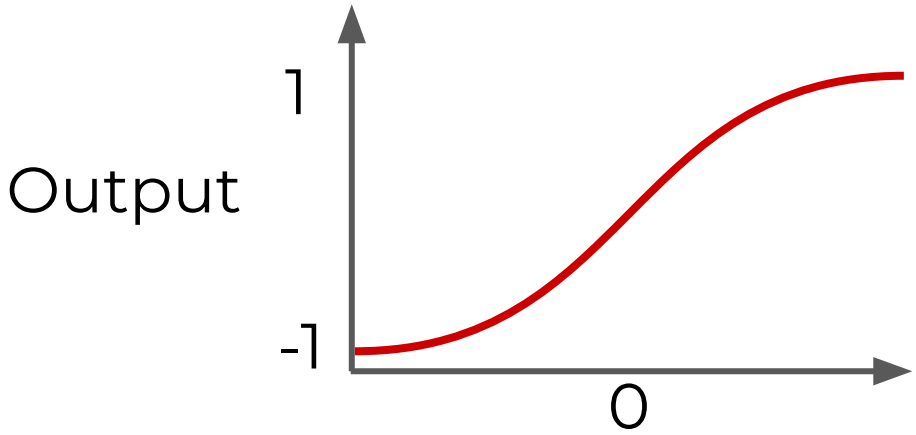
- Let’s discuss a few more activation functions that we’ll encounter!



$$z = wx + b$$

Deep Learning

- Hyperbolic Tangent: tanh(z)



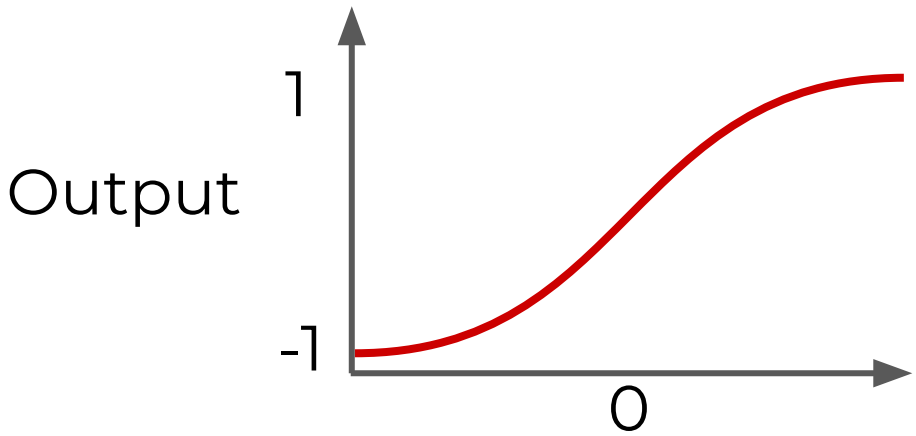
$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

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

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

Deep Learning

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



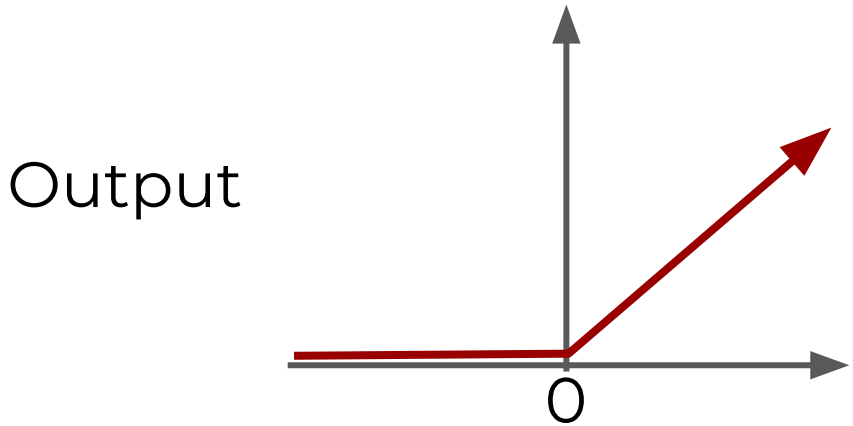
$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

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

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

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**

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.

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

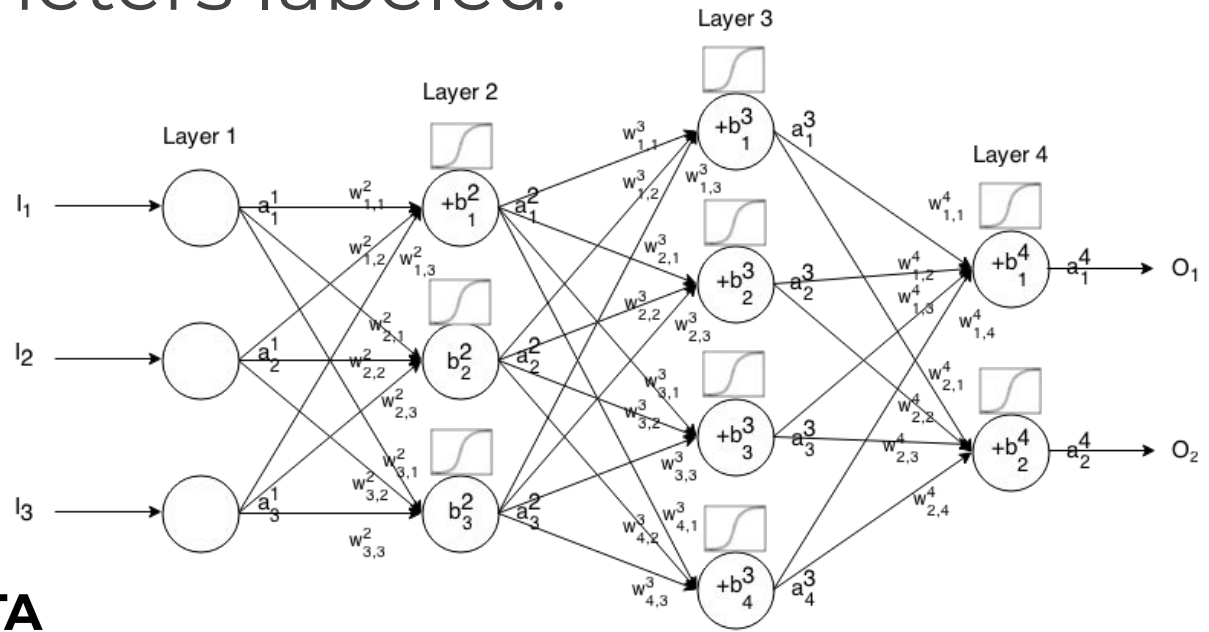
Deep Learning

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

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

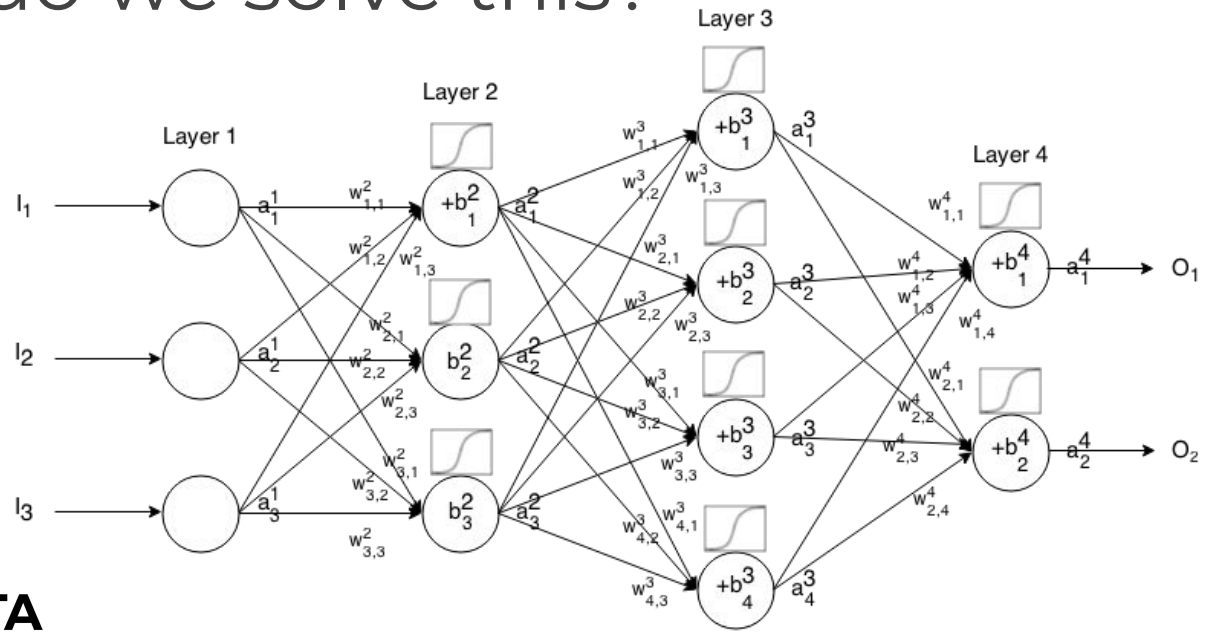
Deep Learning

- 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

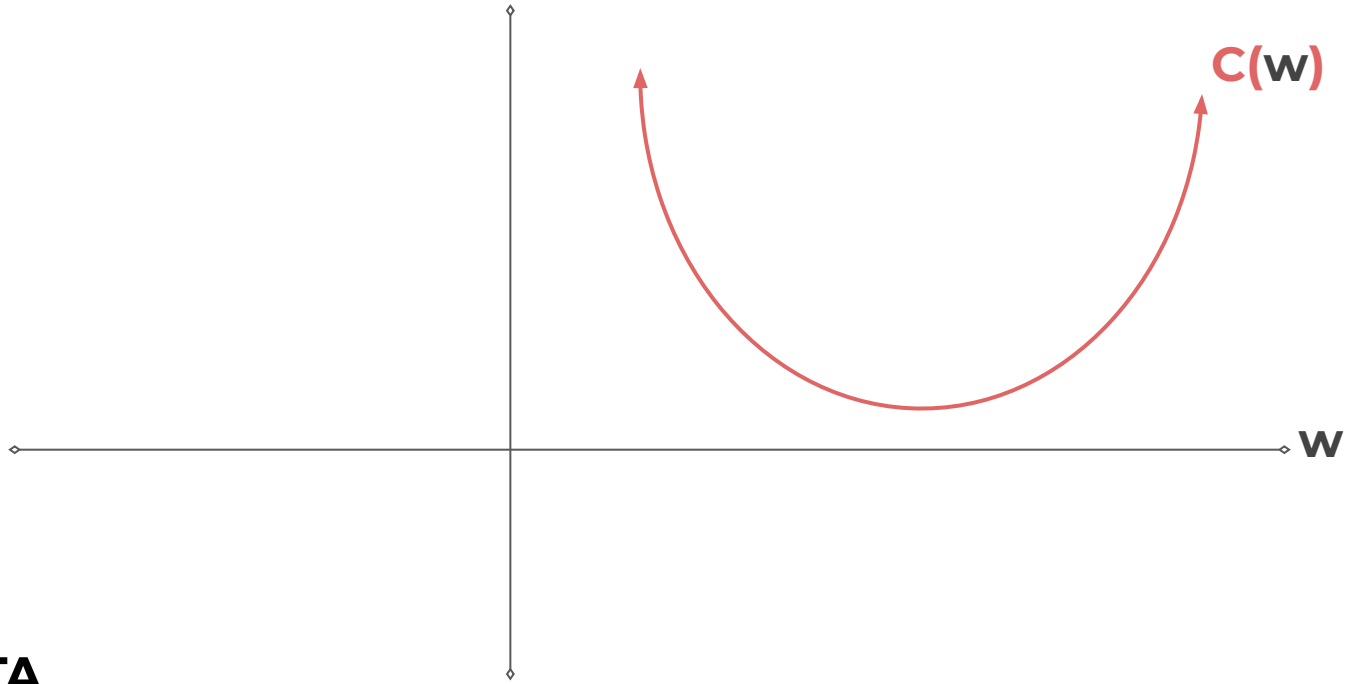
- In a real case, this means we have some cost function **C** dependent lots of weights!
    - **C(w1,w2,w3,....wn)**
- How do we figure out which weights lead us to the lowest cost?

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)**

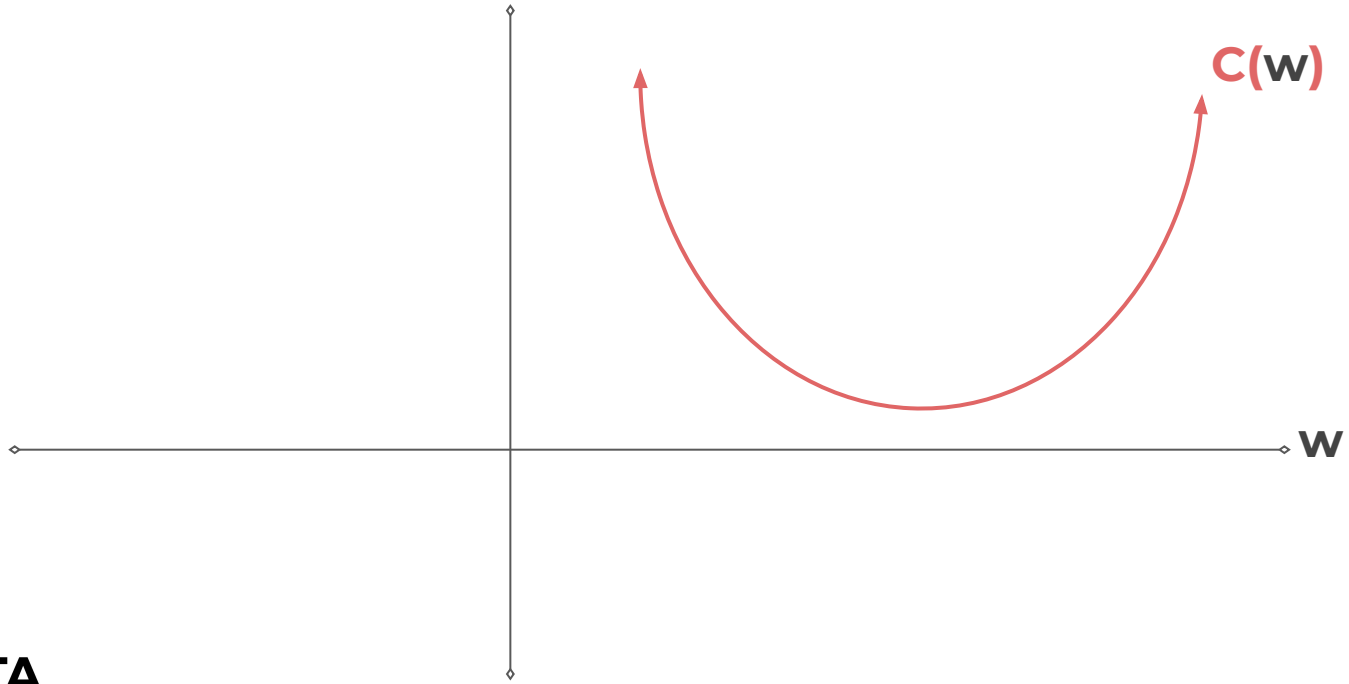
Deep Learning

- Our “simple” function **C(w)**



Deep Learning

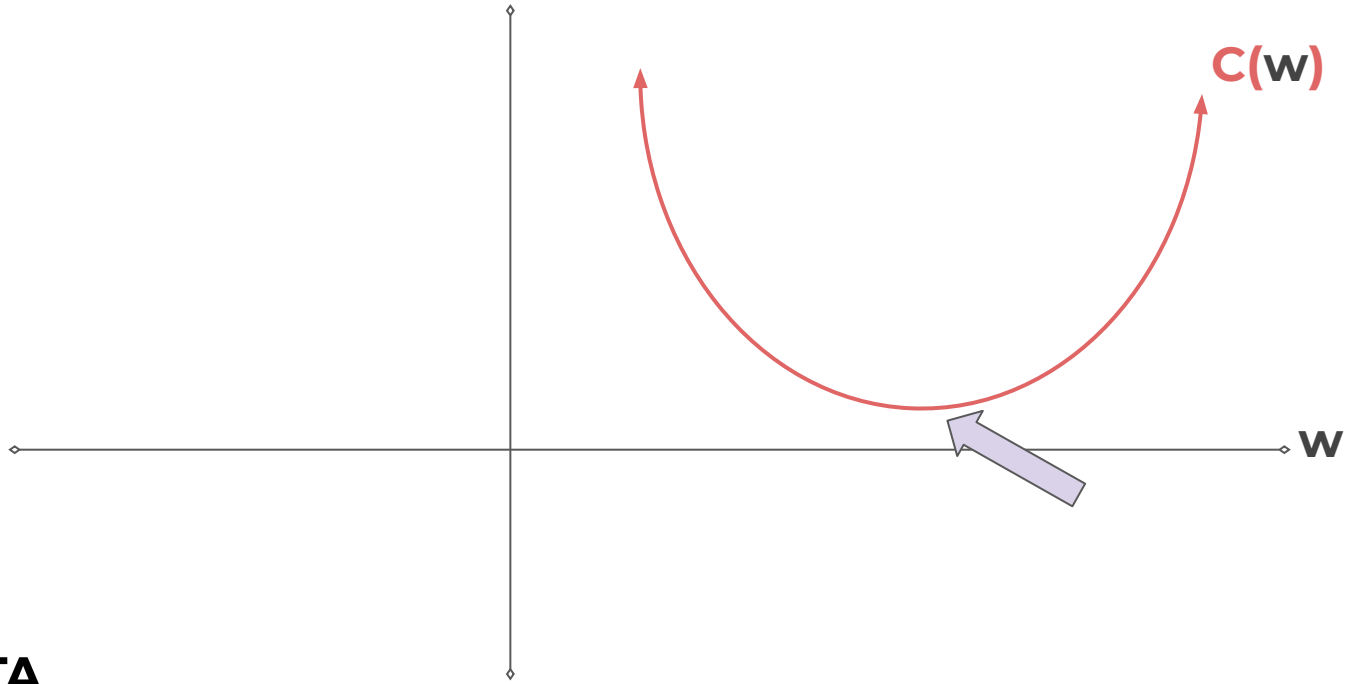
- What value of **w** minimizes our cost?



PIERIAN DATA

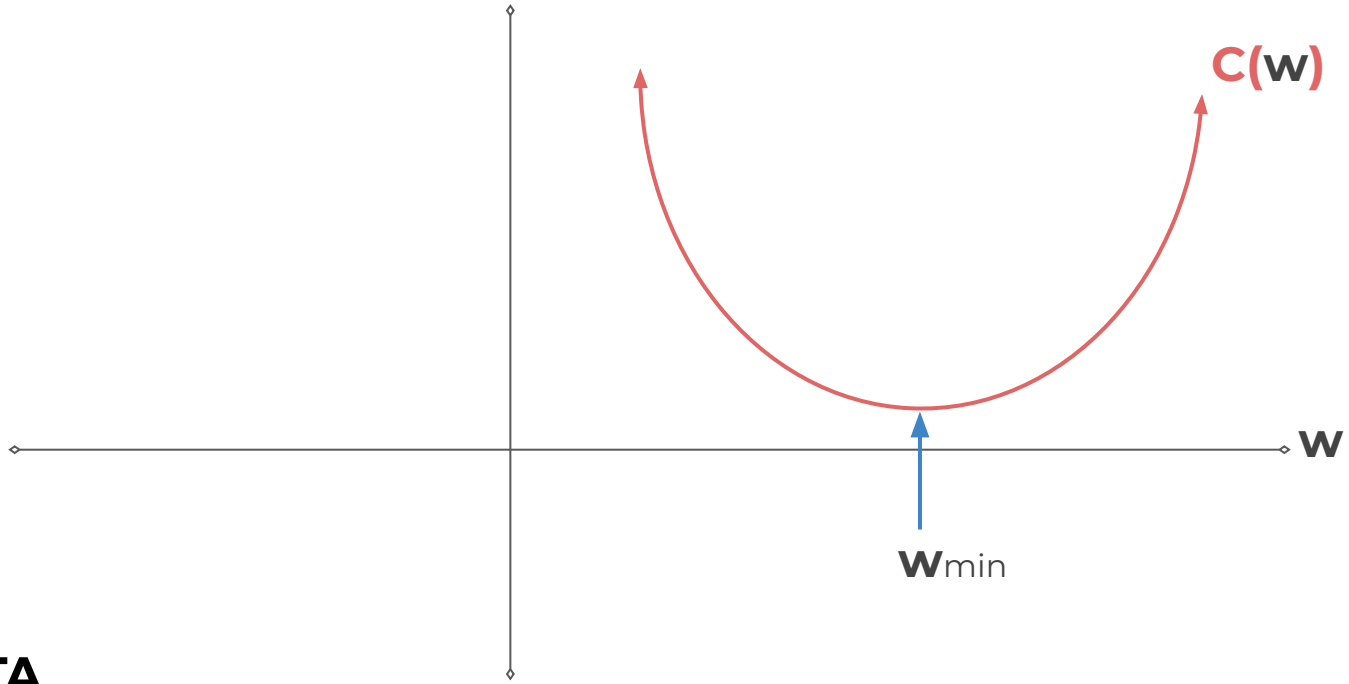
Deep Learning

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



Deep Learning

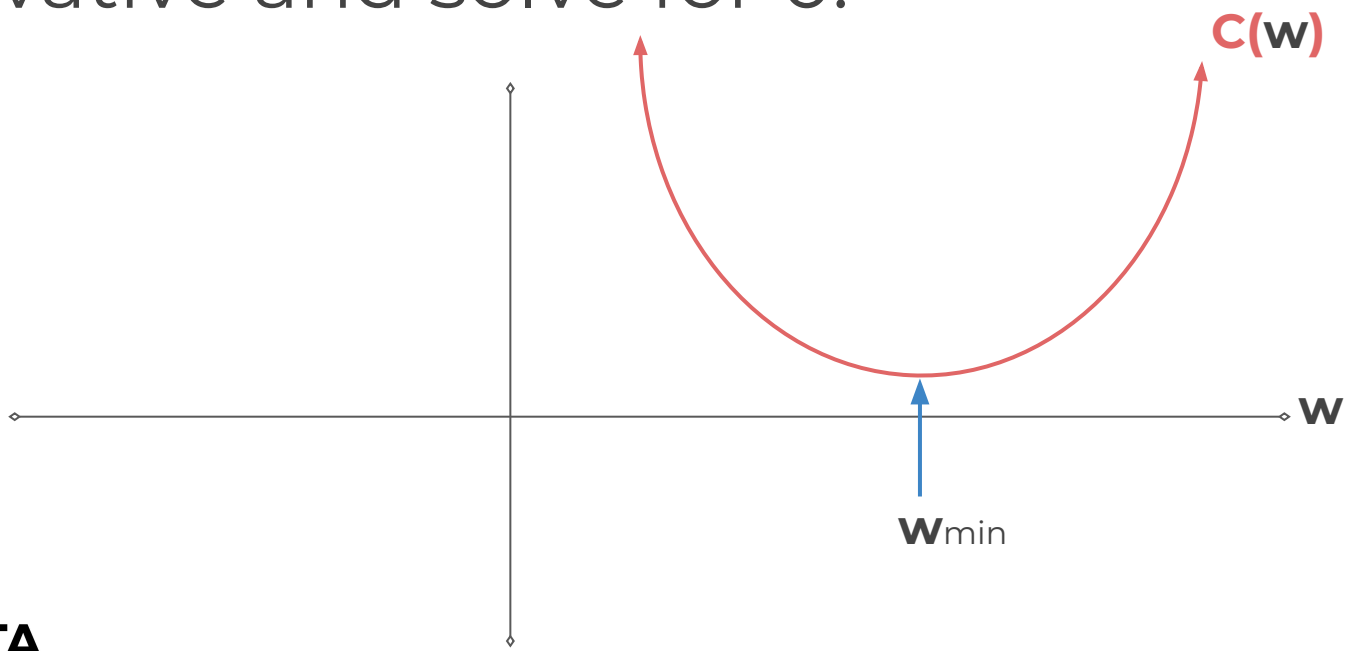
- What value of **w** minimizes our cost?



PIERIAN DATA

Deep Learning

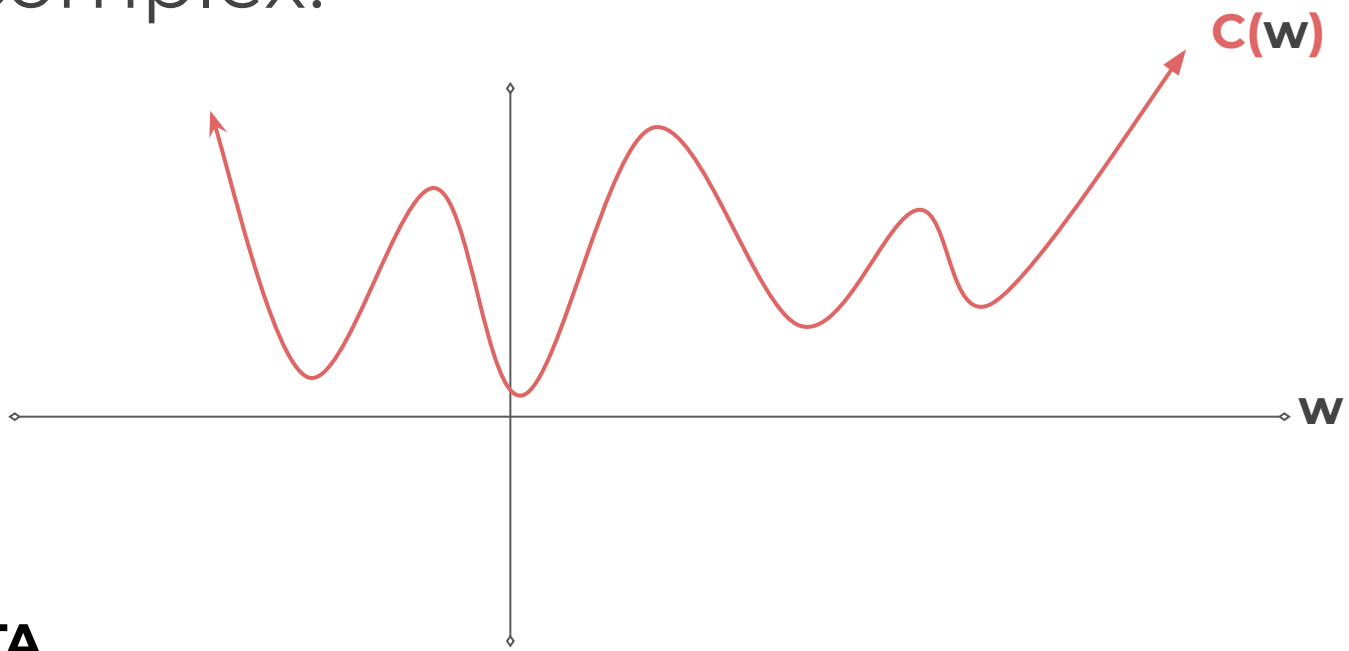
- Students of calculus know we could take a derivative and solve for 0.



PIERIAN DATA

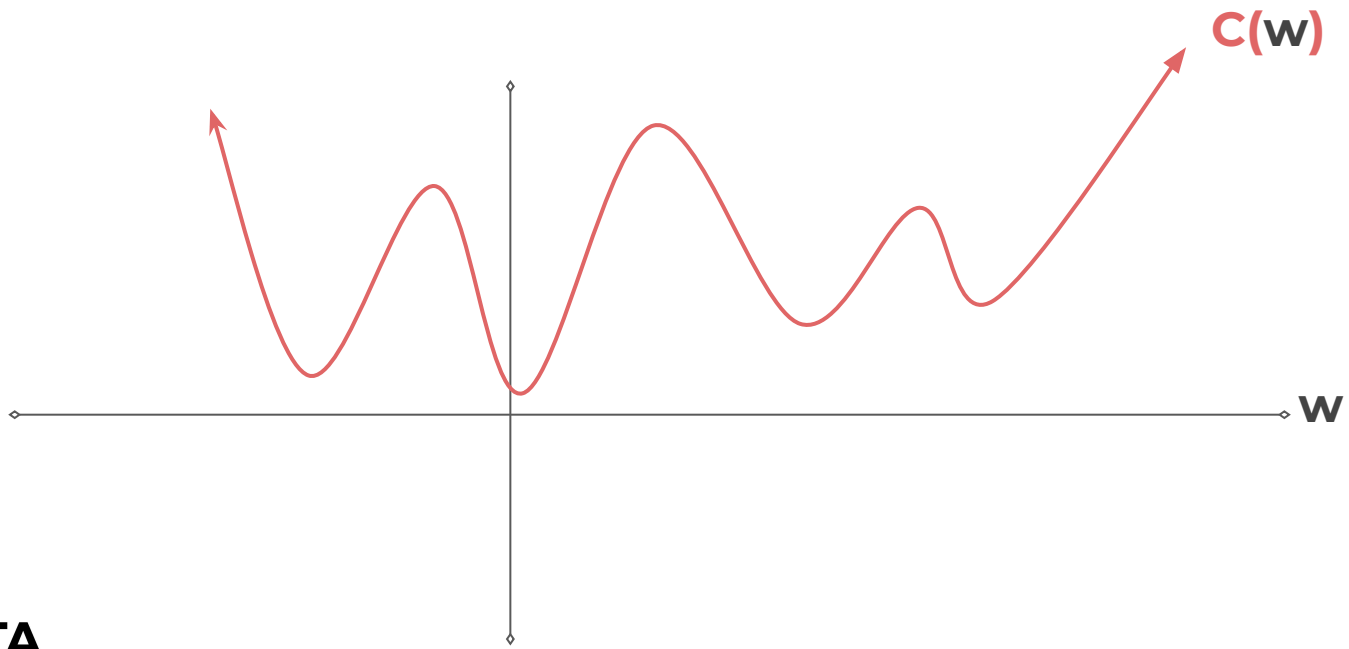
Deep Learning

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



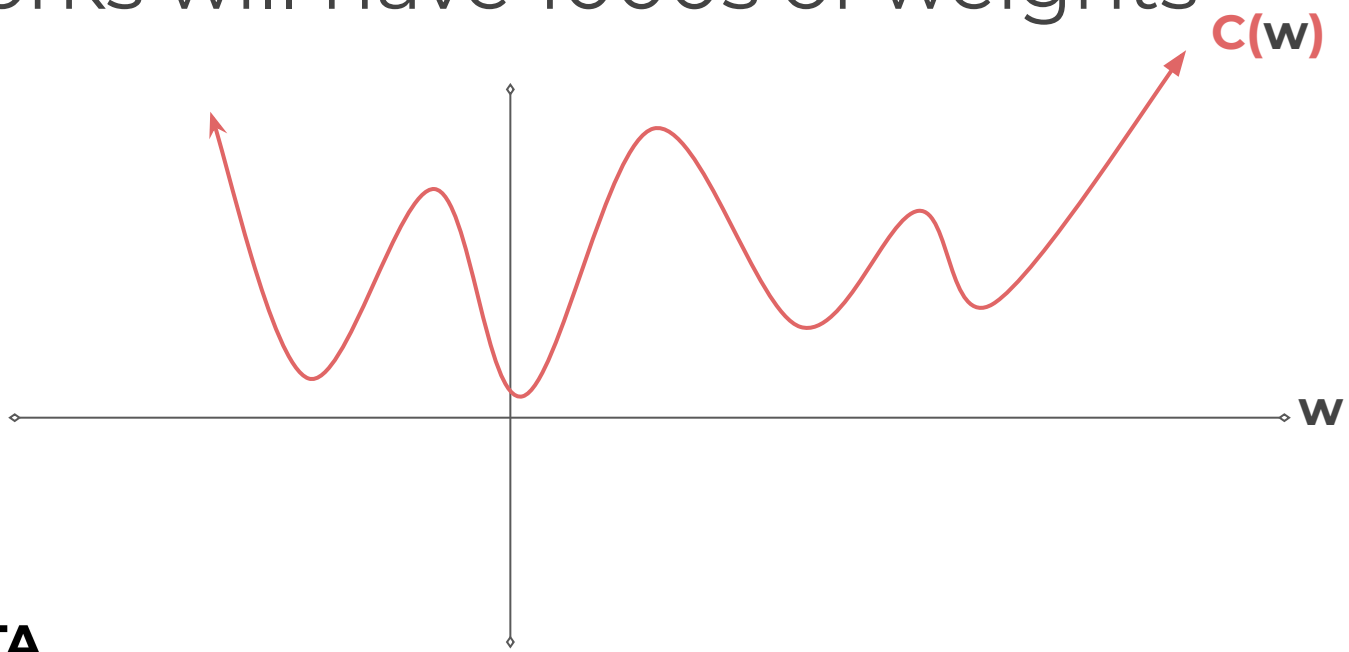
Deep Learning

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



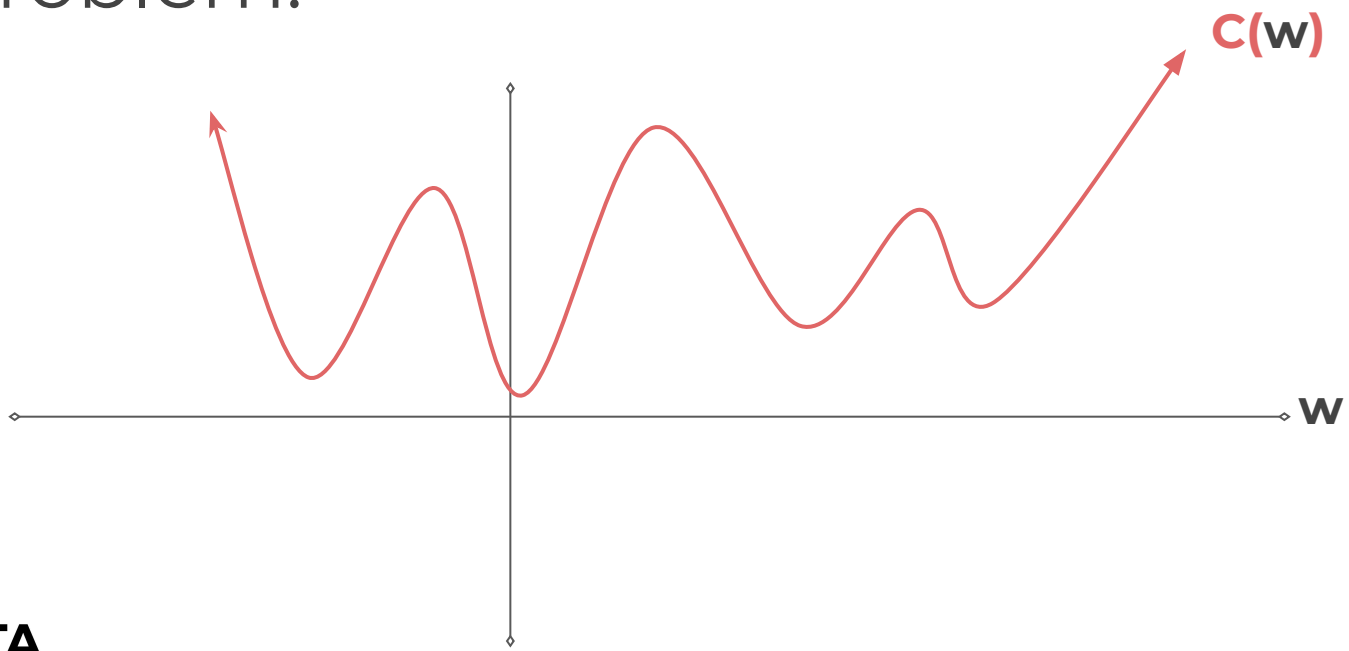
Deep Learning

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



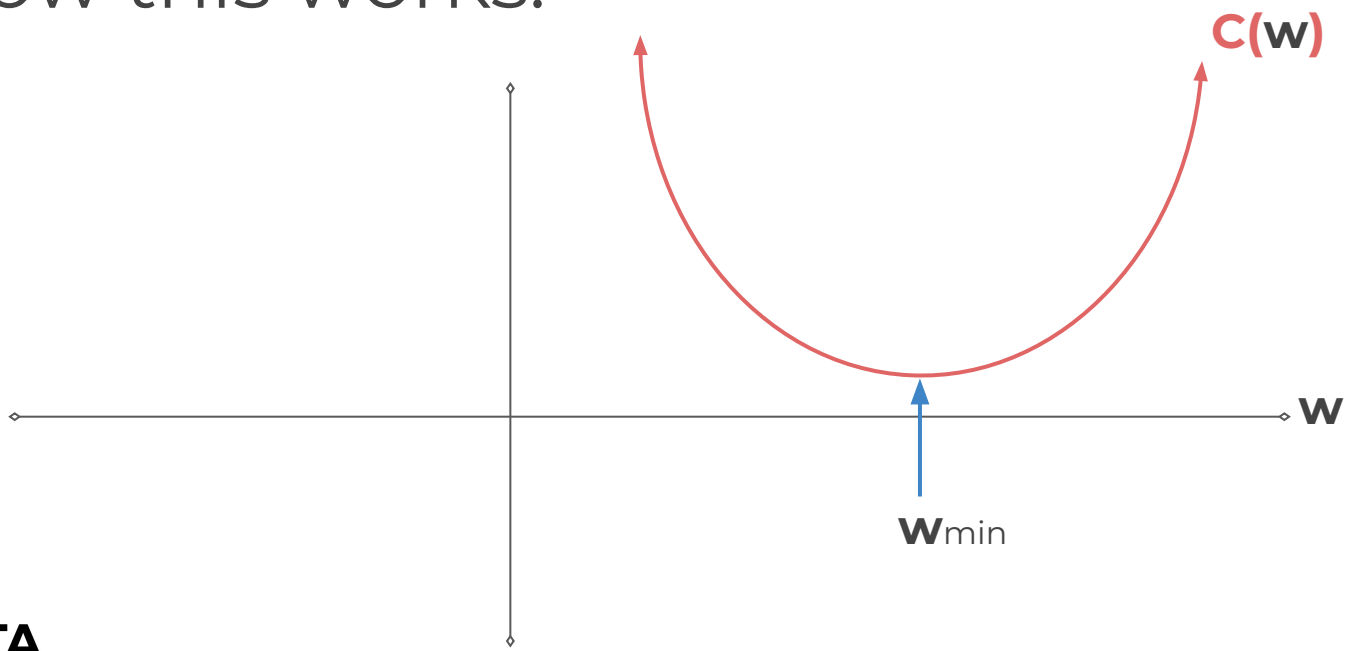
Deep Learning

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



Deep Learning

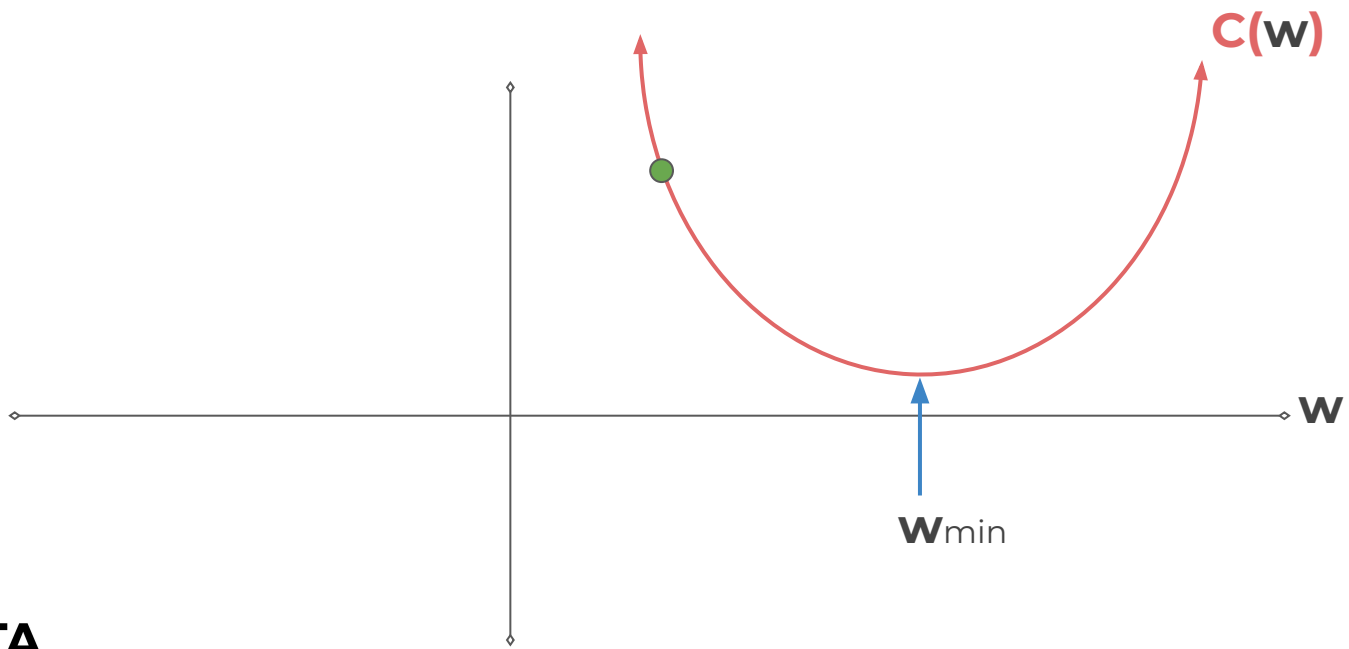
- Let’s go back to our simplified version to see how this works.



PIERIAN DATA

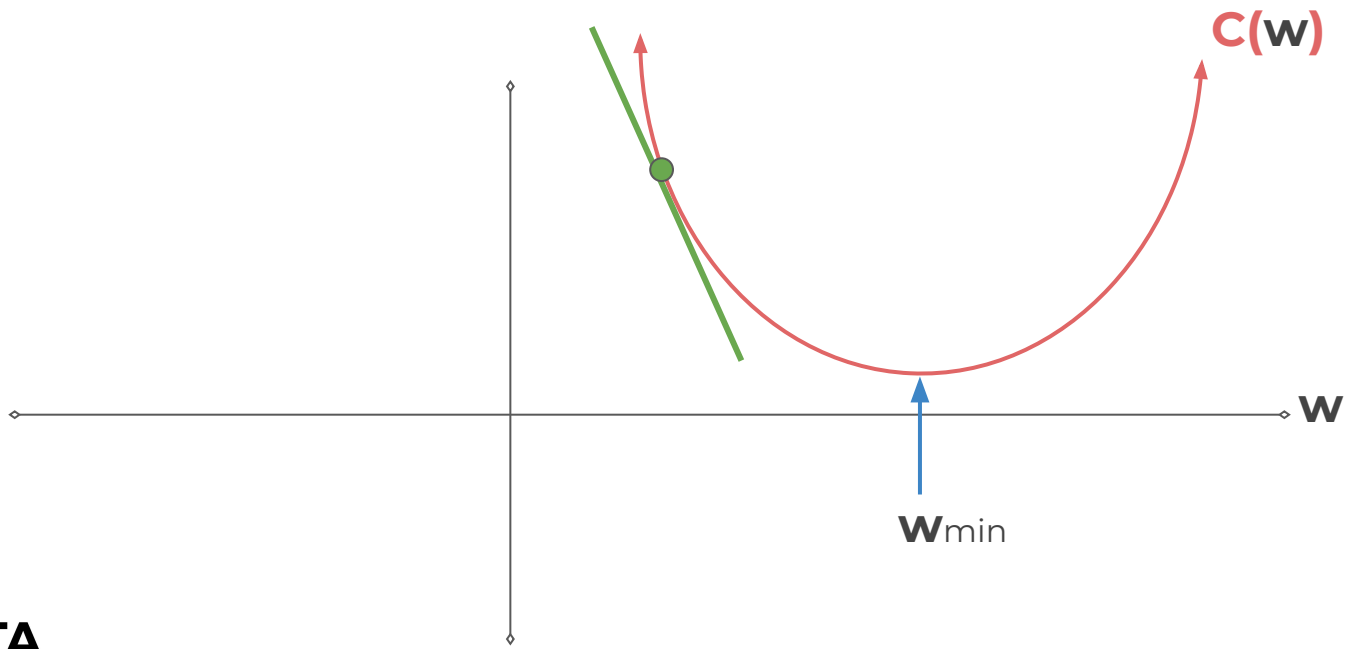
Deep Learning

- We can calculate the slope at a point



Deep Learning

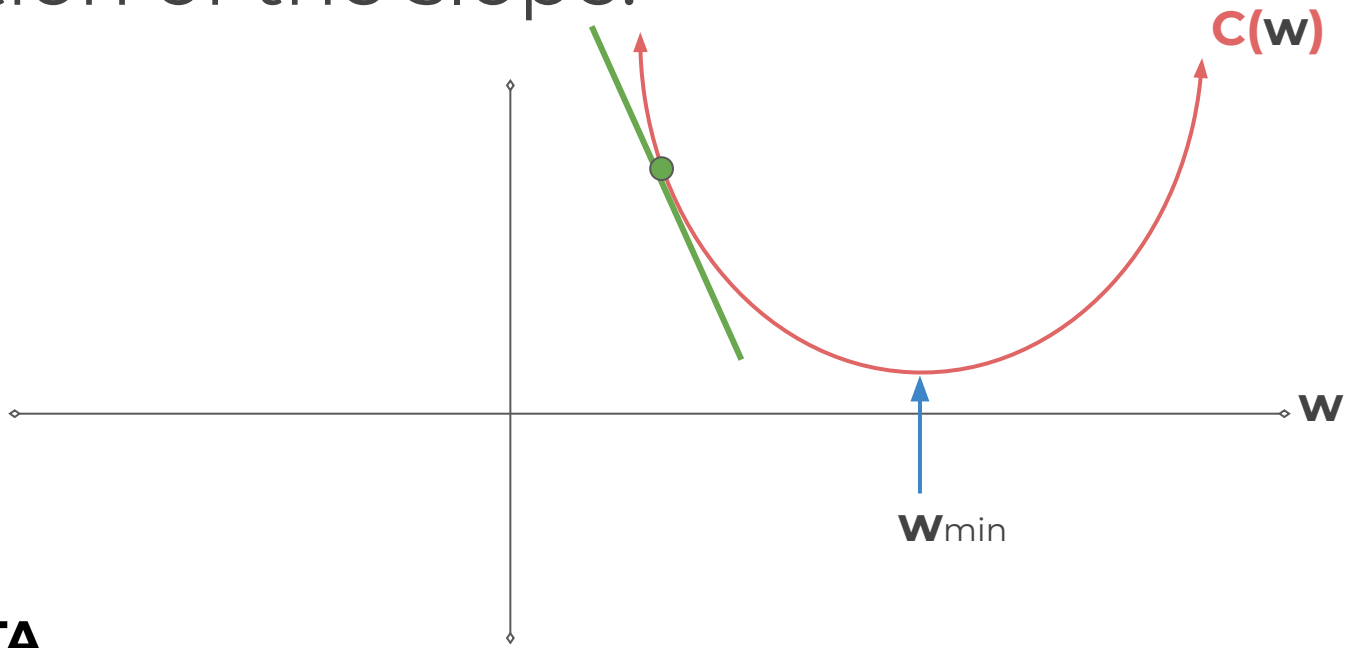
- We can calculate the slope at a point



PIERIAN DATA

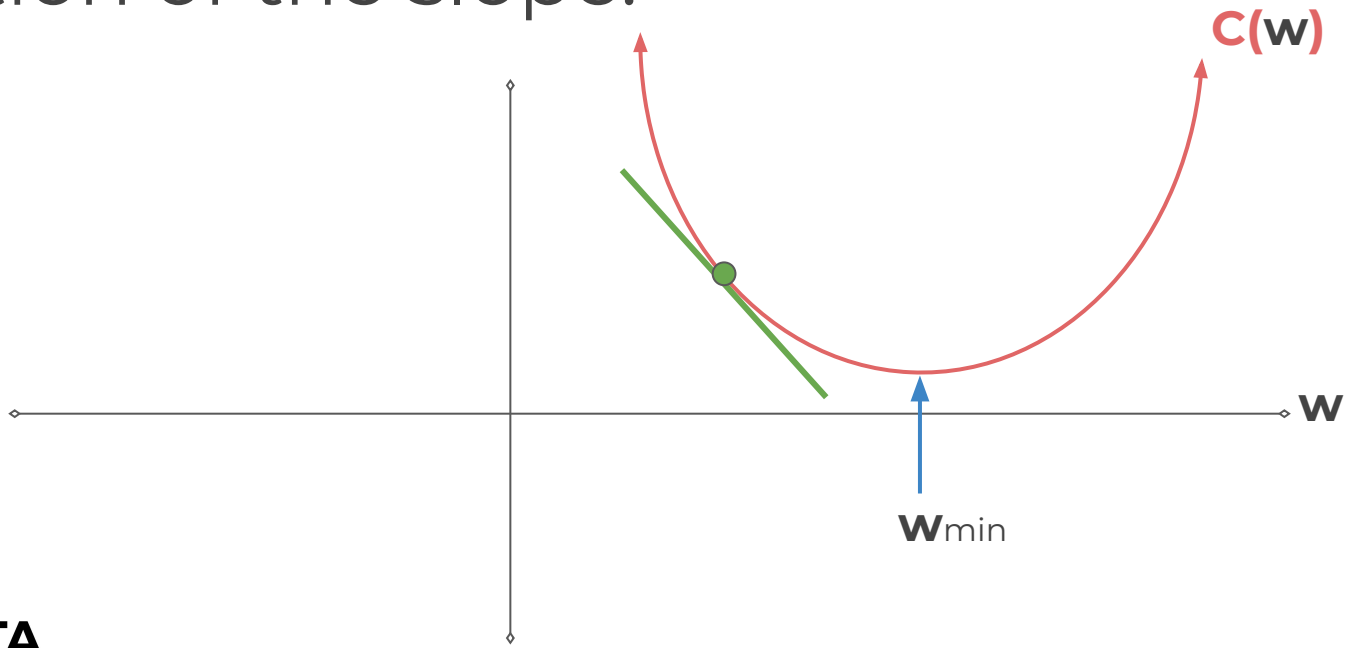
Deep Learning

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



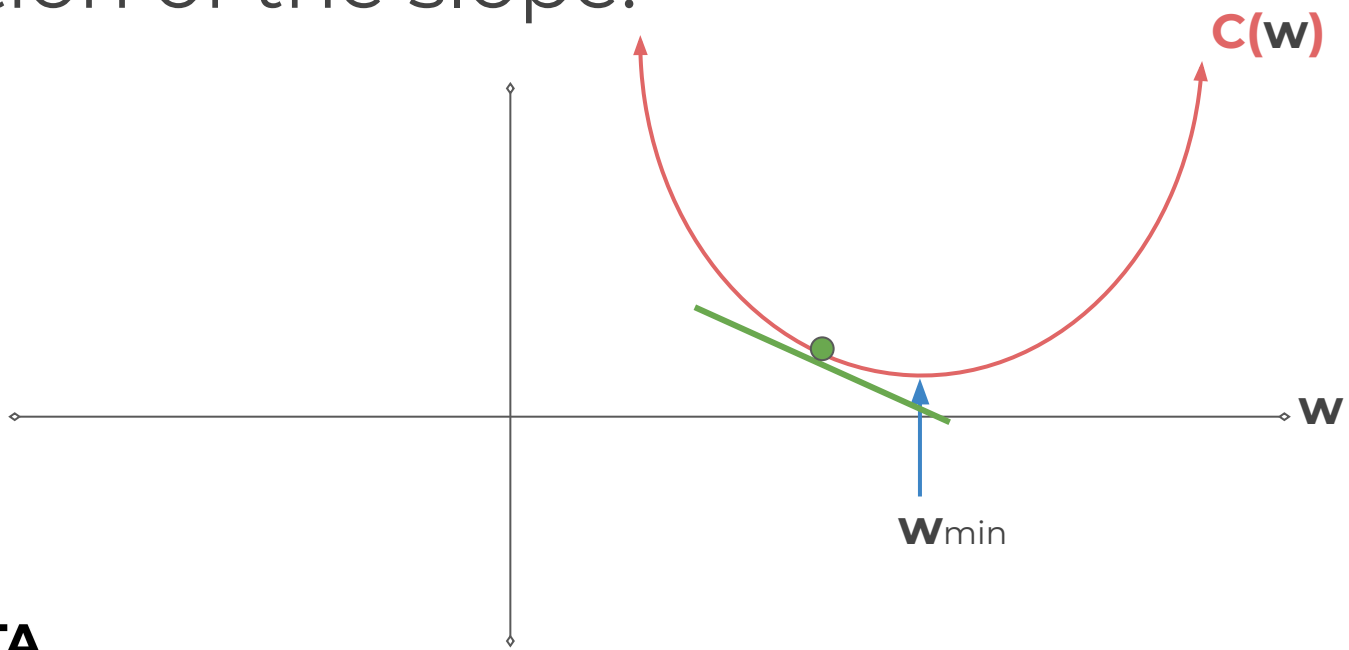
Deep Learning

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



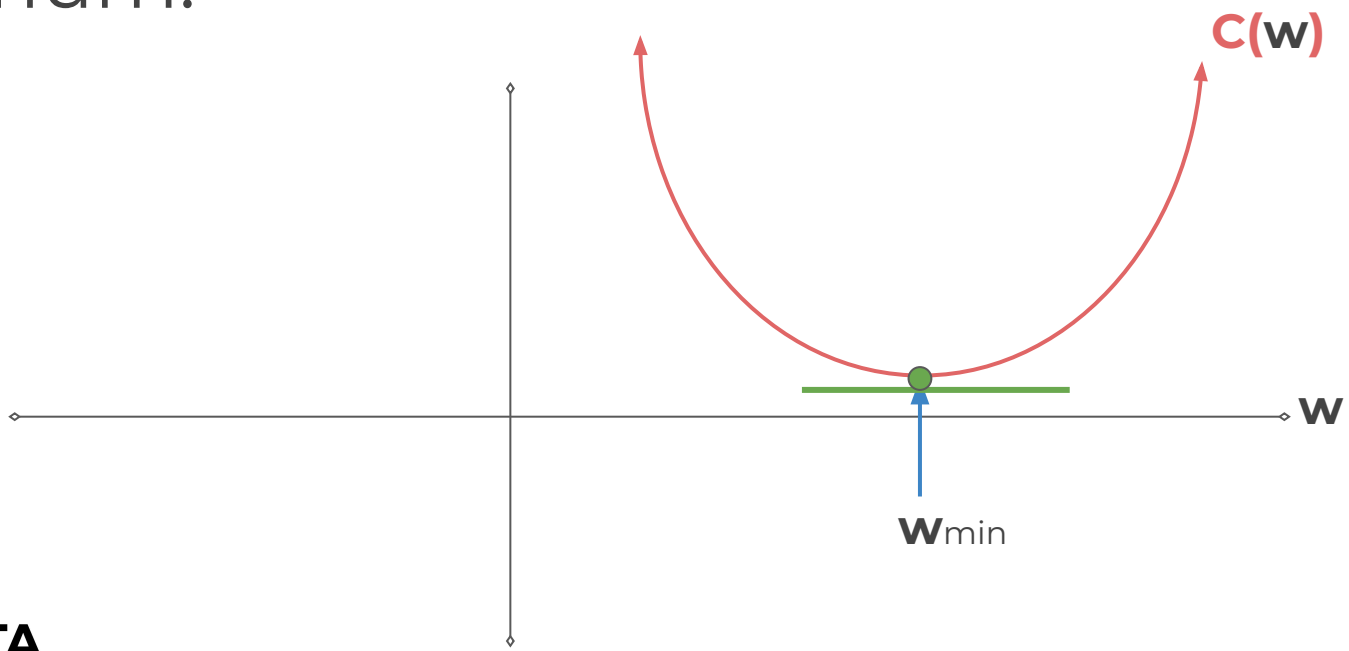
Deep Learning

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



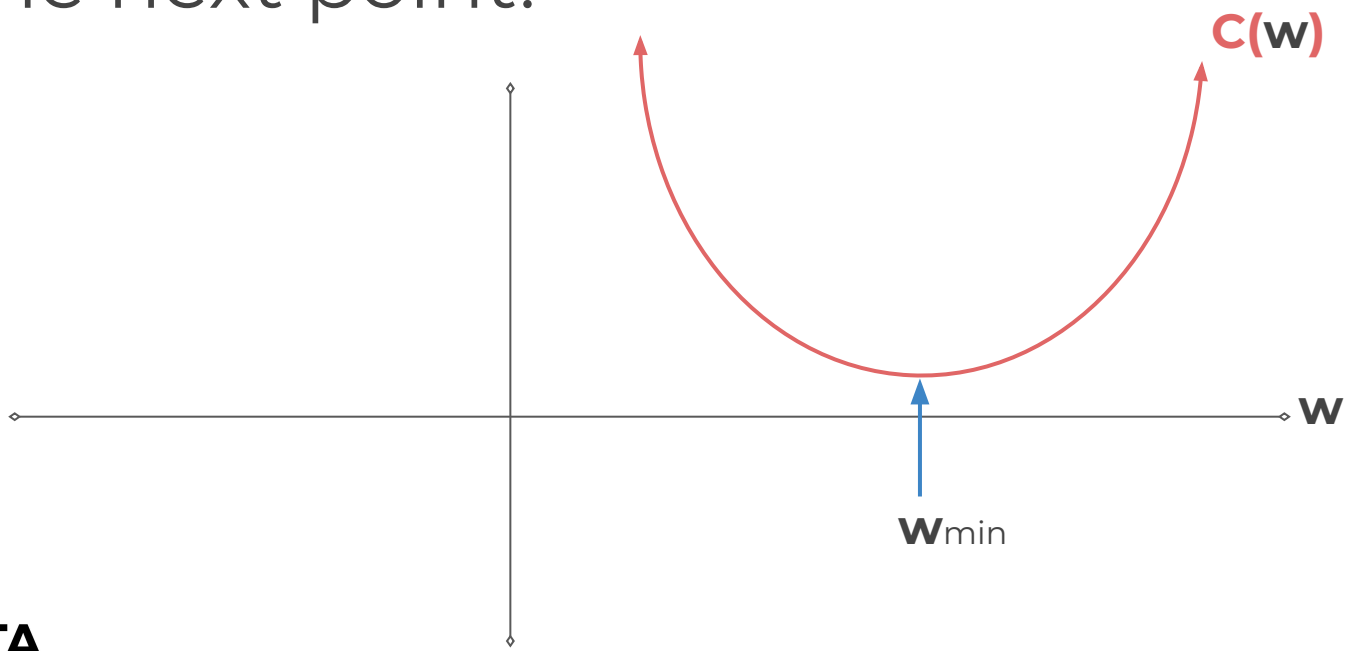
Deep Learning

- Until we converge to zero, indicating a minimum.



Deep Learning

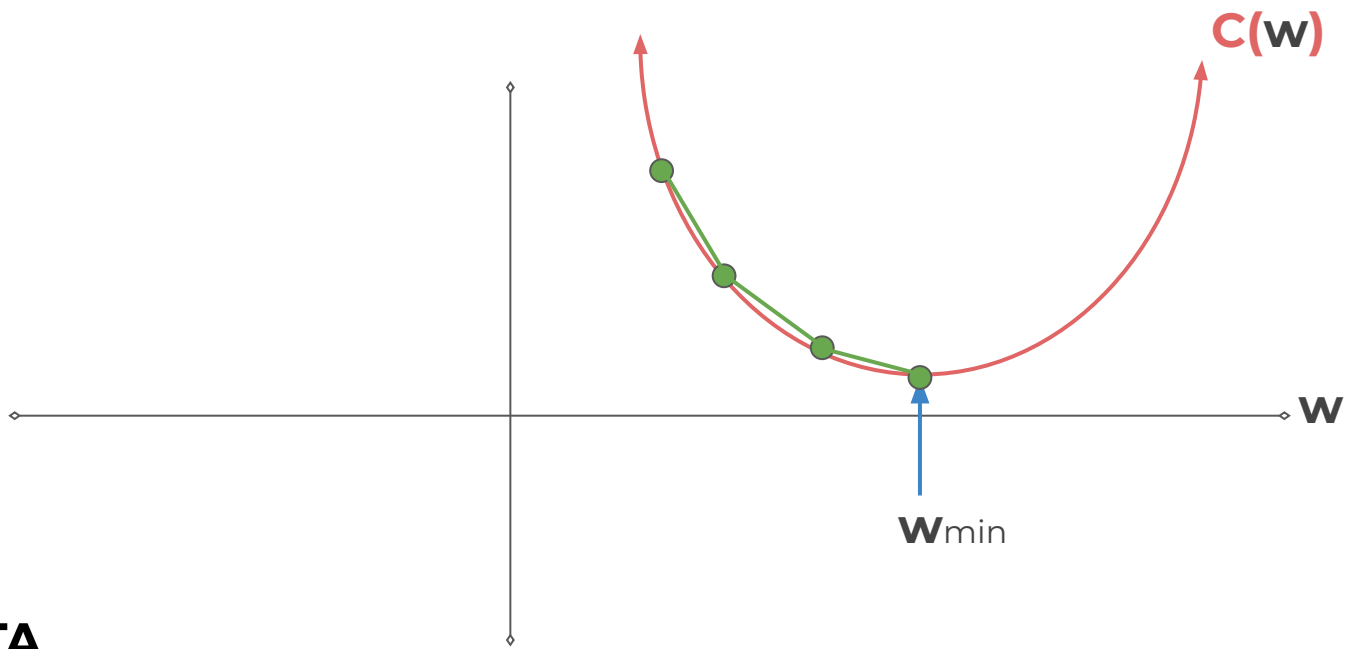
- We could have changed our step size to find the next point!



PIERIAN DATA

Deep Learning

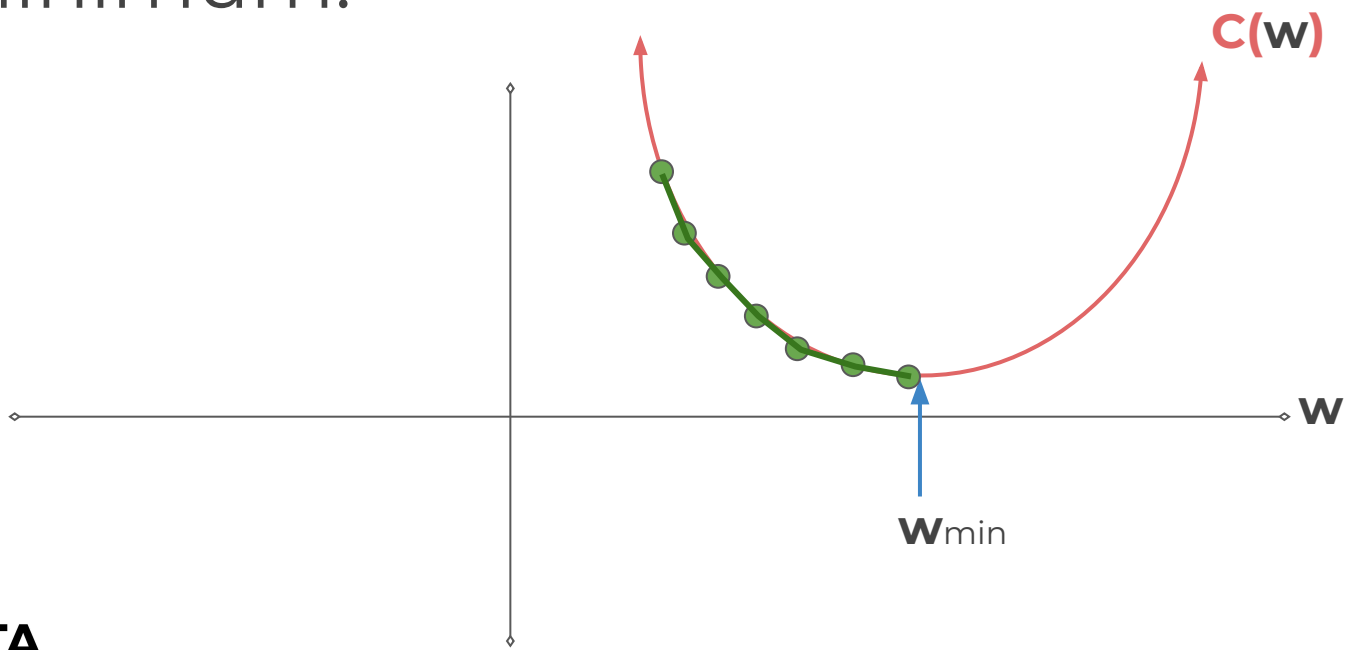
- Our steps:



PIERIAN DATA

Deep Learning

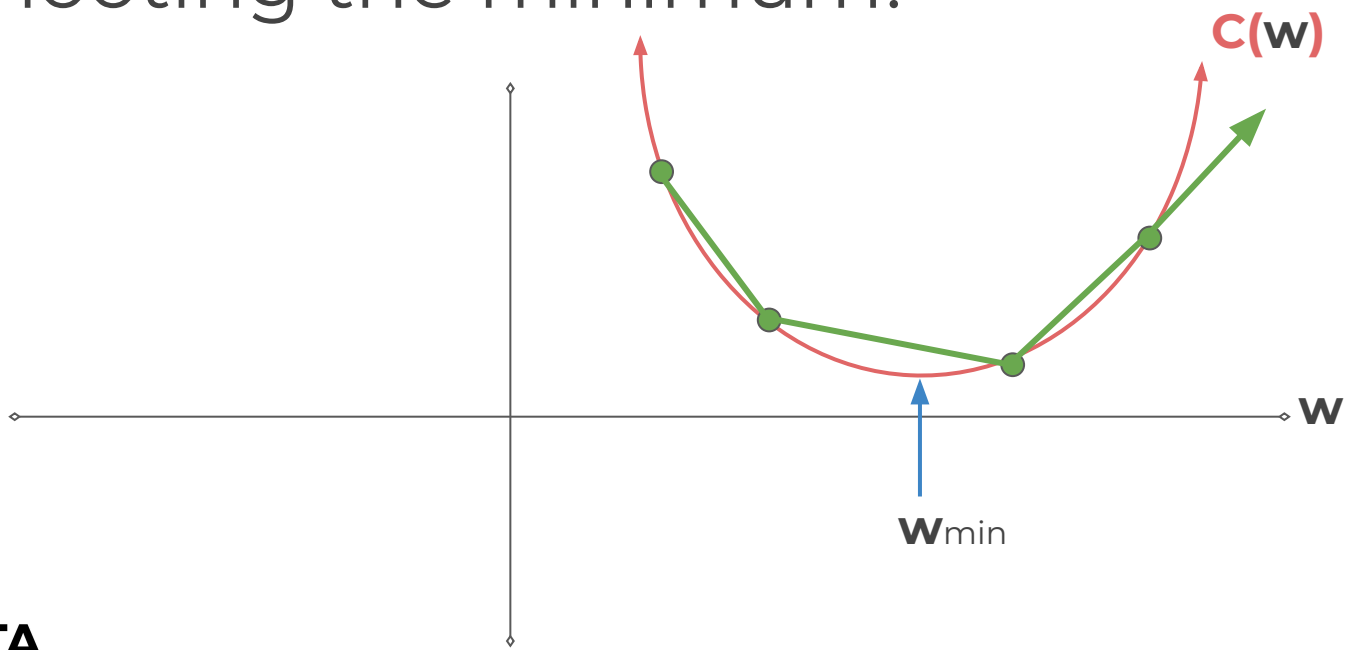
- Smaller steps sizes take longer to find the minimum.



PIERIAN DATA

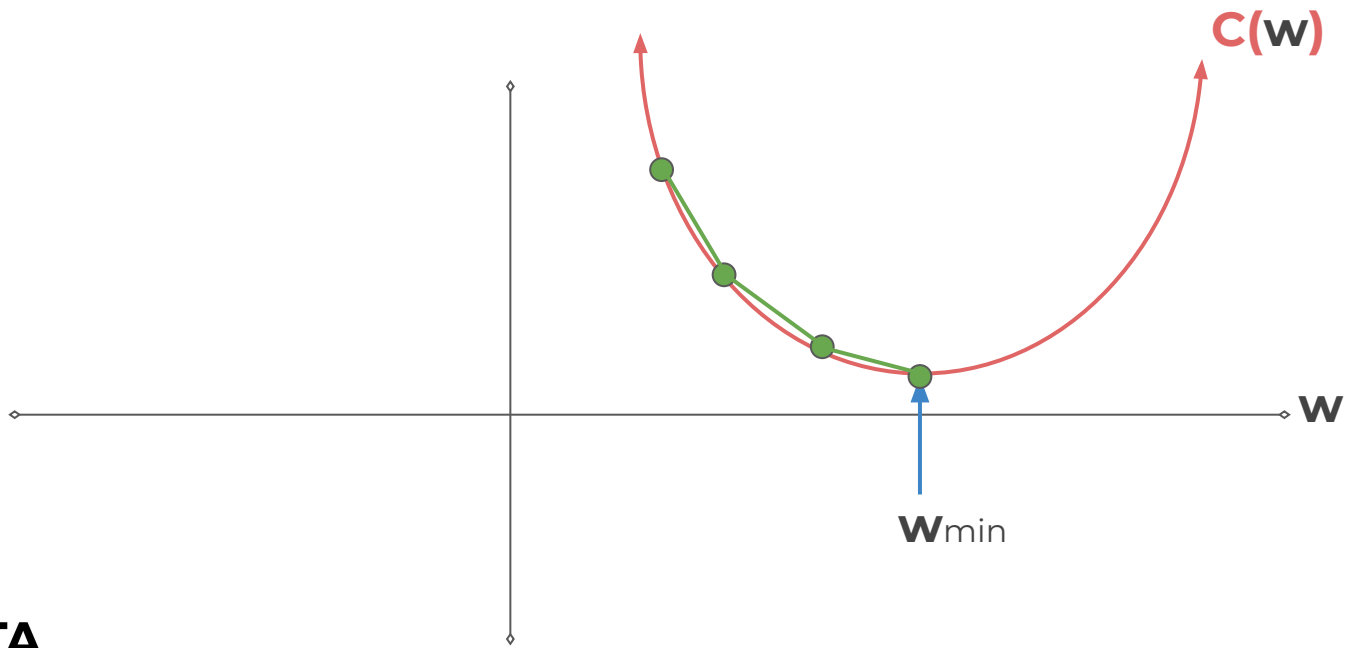
Deep Learning

- 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

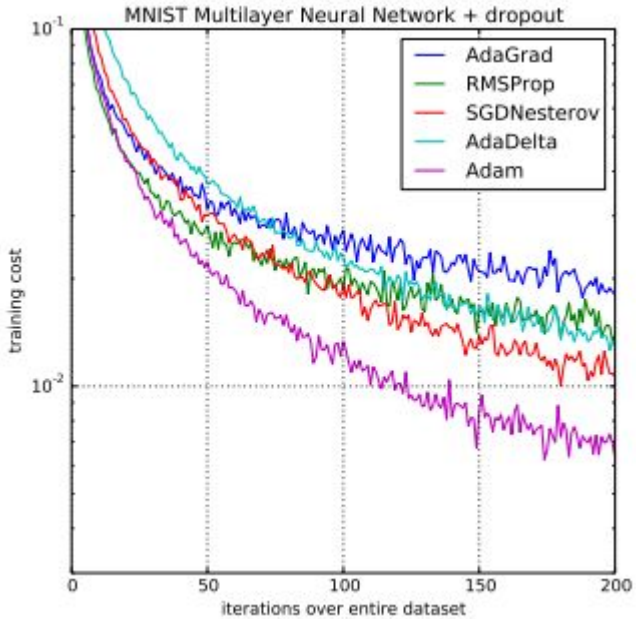
- We could start with larger steps, then go smaller as we realize the slope gets closer to zero.
- This is known as **adaptive gradient descent.**

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!

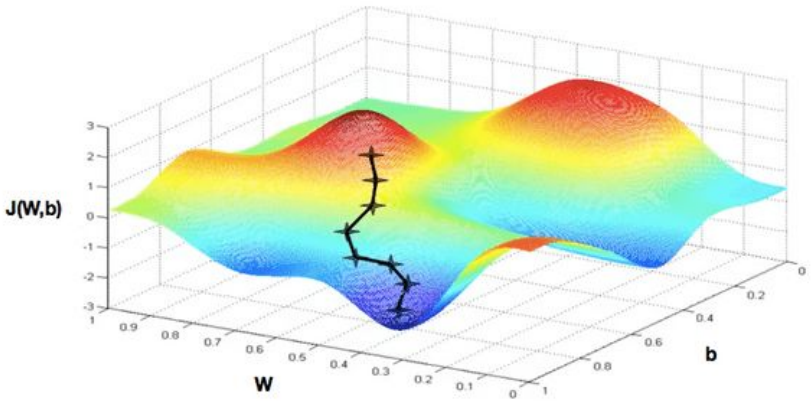
Deep Learning

- 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

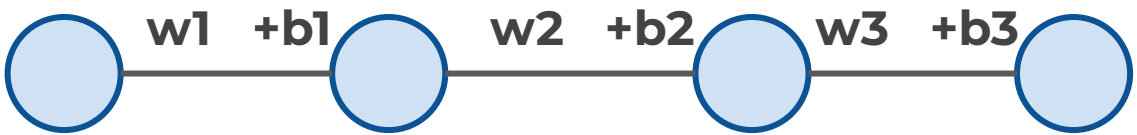
- Fundamentally, we want to know how the cost function results changes with respect to the weights in the network, so we can update the weights to minimize the cost function

Deep Learning

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

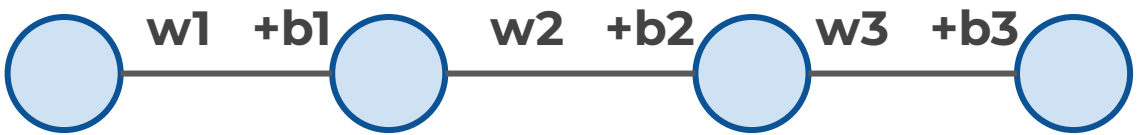
Deep Learning

- Each input will receive a weight and bias



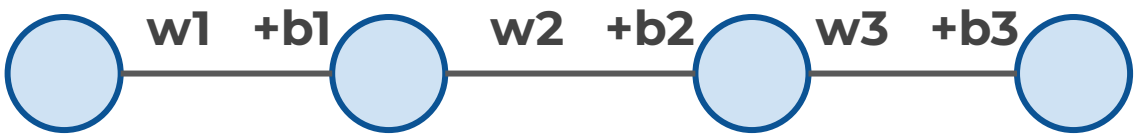
Deep Learning

- 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.

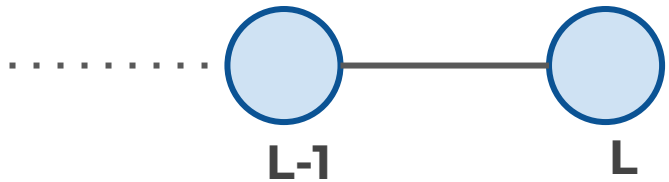


Deep Learning

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

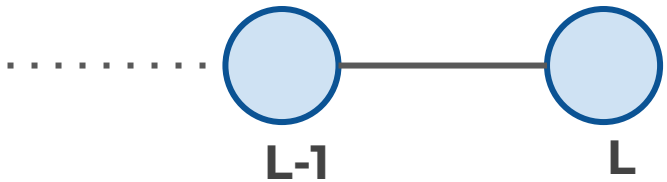
Deep Learning

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



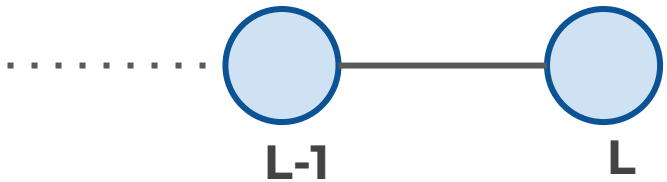
Deep Learning

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



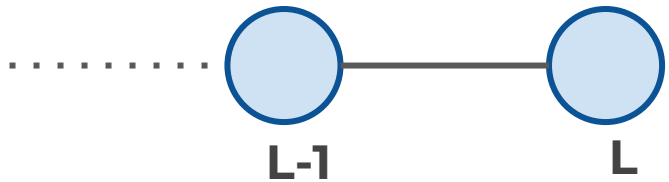
Deep Learning

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



Deep Learning

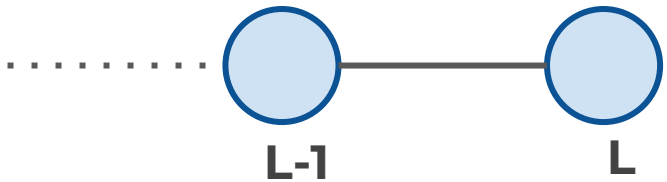
- This means we have:
  - $z^L = w^L a^{L-1} + b^L$
  - $a^L = \sigma(z^L)$
  - $C_0(...) = (a^L - y)^2$



Deep Learning

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

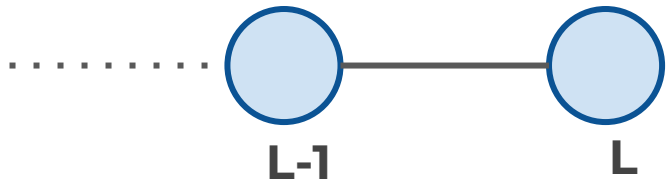
$$\frac{\partial C_0}{\partial w^L}$$



Deep Learning

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

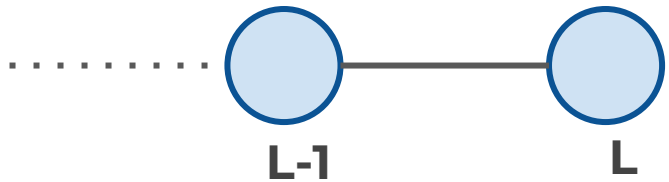
$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$



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!

Keras Basics

Section Overview

- Let’s learn the basics of Keras and building a Neural Network.
- To create an NN model with Keras, we first define the Sequential model object, then add layers to it.

Section Overview

- Afterwards we fit the model to the training data for a chosen number of epochs.
- An epoch is one full pass through all the training data. Typically the training data is split into batches, instead of being passed all at once to the Network.

Recurrent Neural Networks Theory

Deep Learning

- Examples of Sequences
    - Time Series Data (Sales)
    - Sentences
    - Audio
    - Car Trajectories
    - Music

Deep Learning

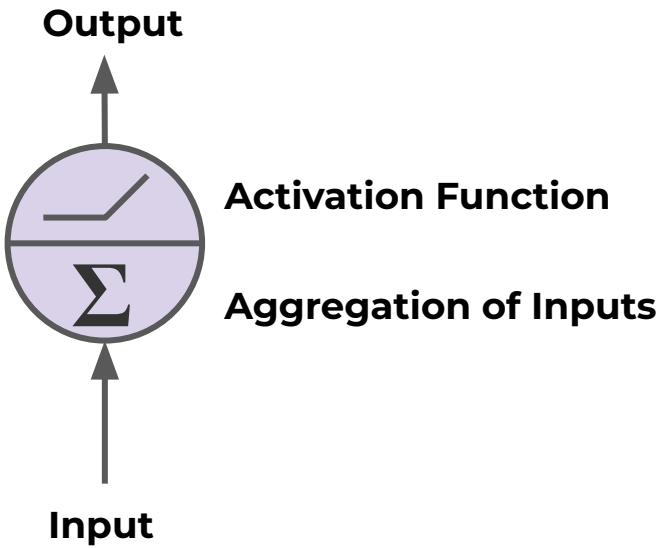
- Let’s imagine a sequence:
  - [1,2,3,4,5,6]
  - Would you be able to predict a similar sequence shifted one time step into the future?
    - [2,3,4,5,6,7]

Deep Learning

- To do this properly, we need to somehow let the neuron “know” about its previous history of outputs.
- One easy way to do this is to simply feed its output back into itself as an input!

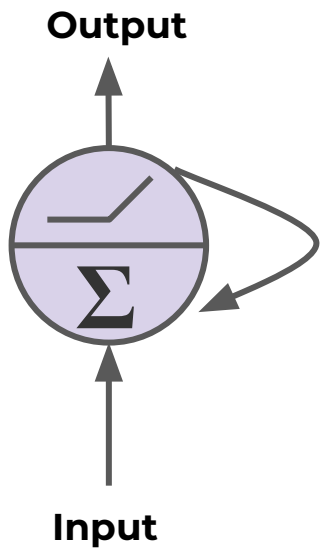
Deep Learning

- Normal Neuron in Feed Forward Network



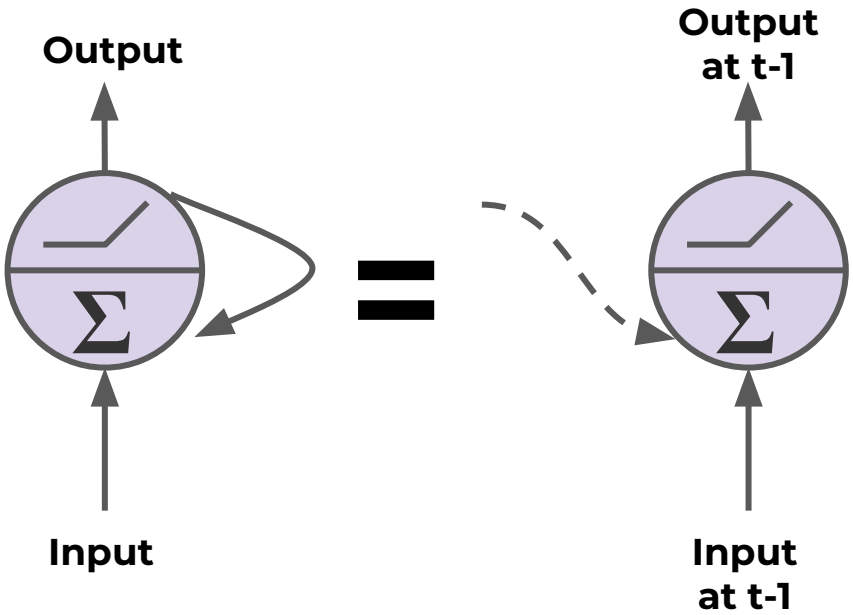
Deep Learning

- Recurrent Neuron - Sends output back to itself!
  - Let's see what this looks like over time!



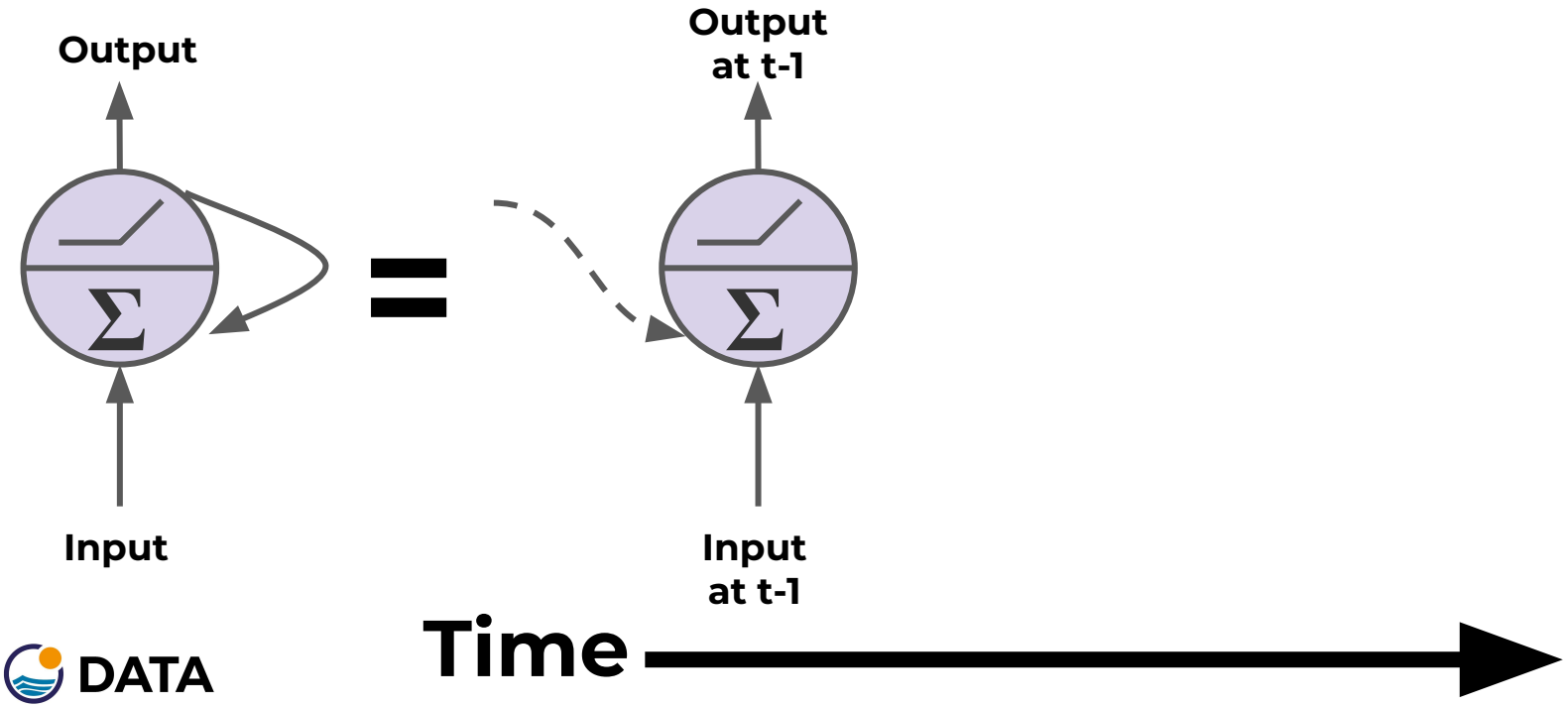
Deep Learning

- Recurrent Neuron



Deep Learning

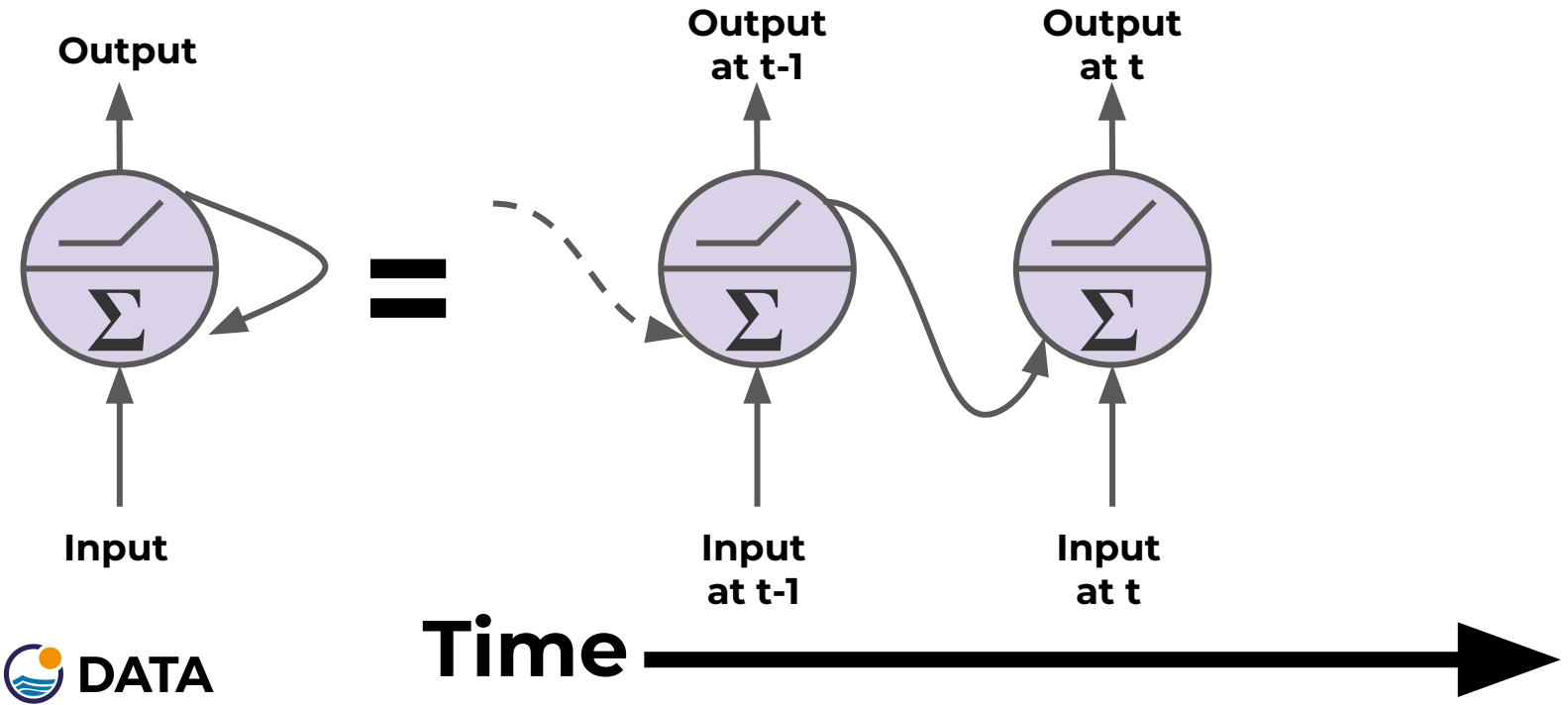
- Recurrent Neuron

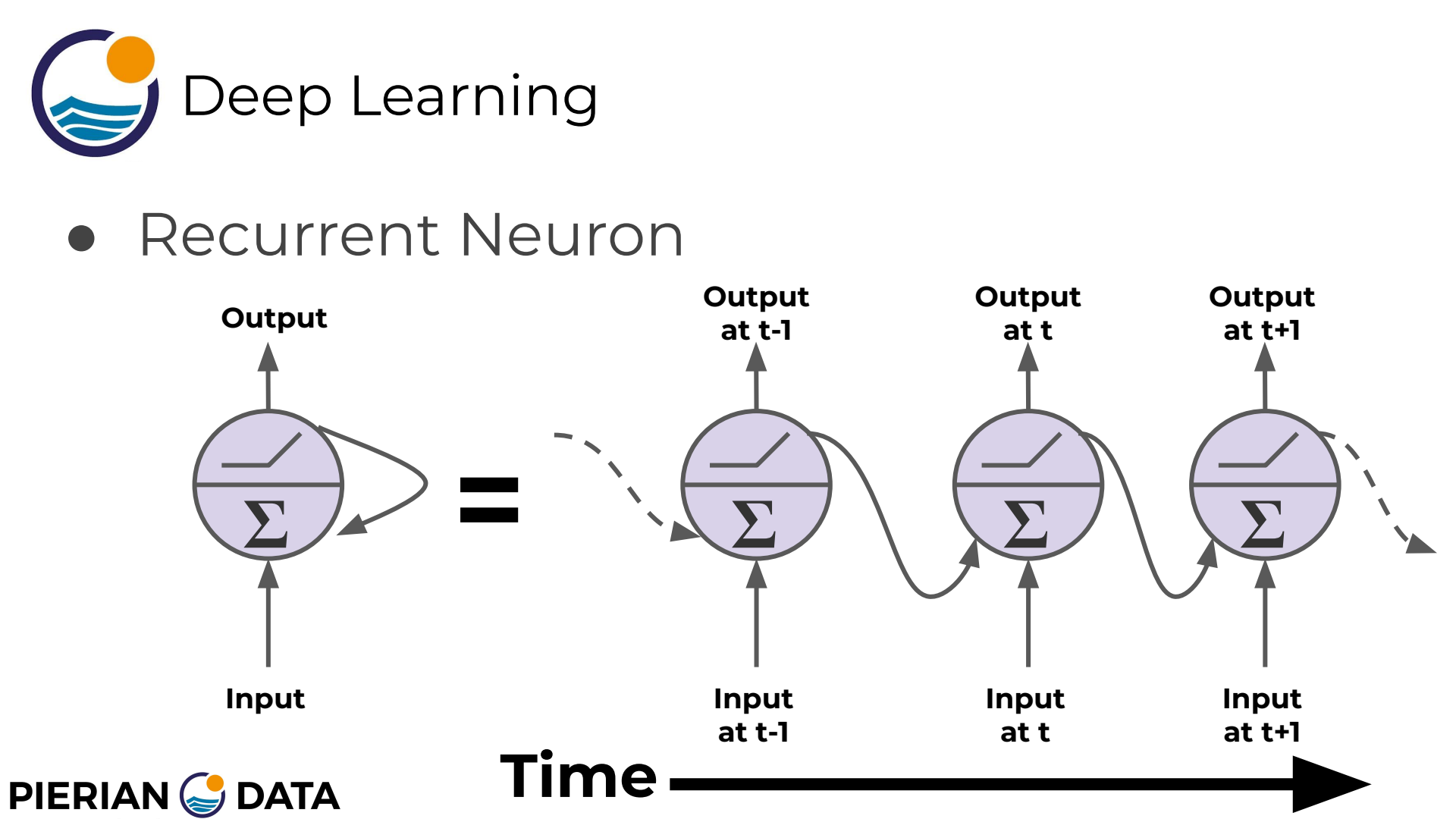


PIERIAN DATA

Deep Learning

- Recurrent Neuron




Deep Learning
Recurrent Neuron
Output
Input
=
Output at t-1
Output at t
Output at t+1
Input at t-1
Input at t
Input at t+1
Time
PIERIAN DATA

Deep Learning

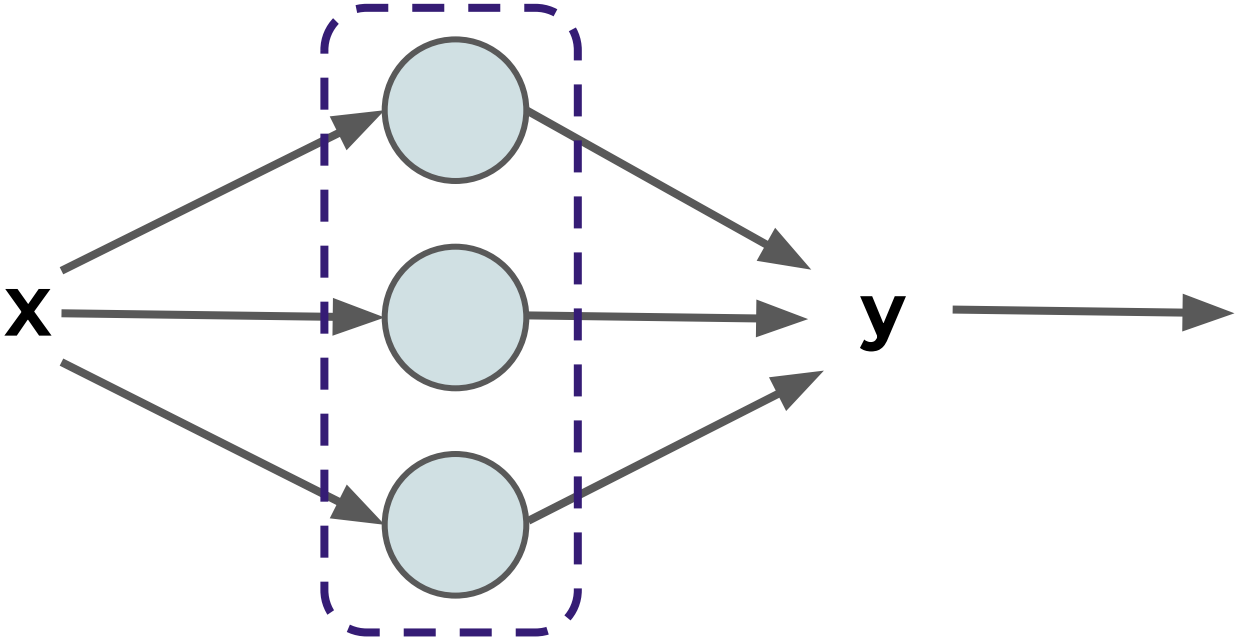
- Cells that are a function of inputs from previous time steps are also known as *memory cells*.
- RNN are also flexible in their inputs and outputs, for both sequences and single vector values.

Deep Learning

- We can also create entire layers of Recurrent Neurons...

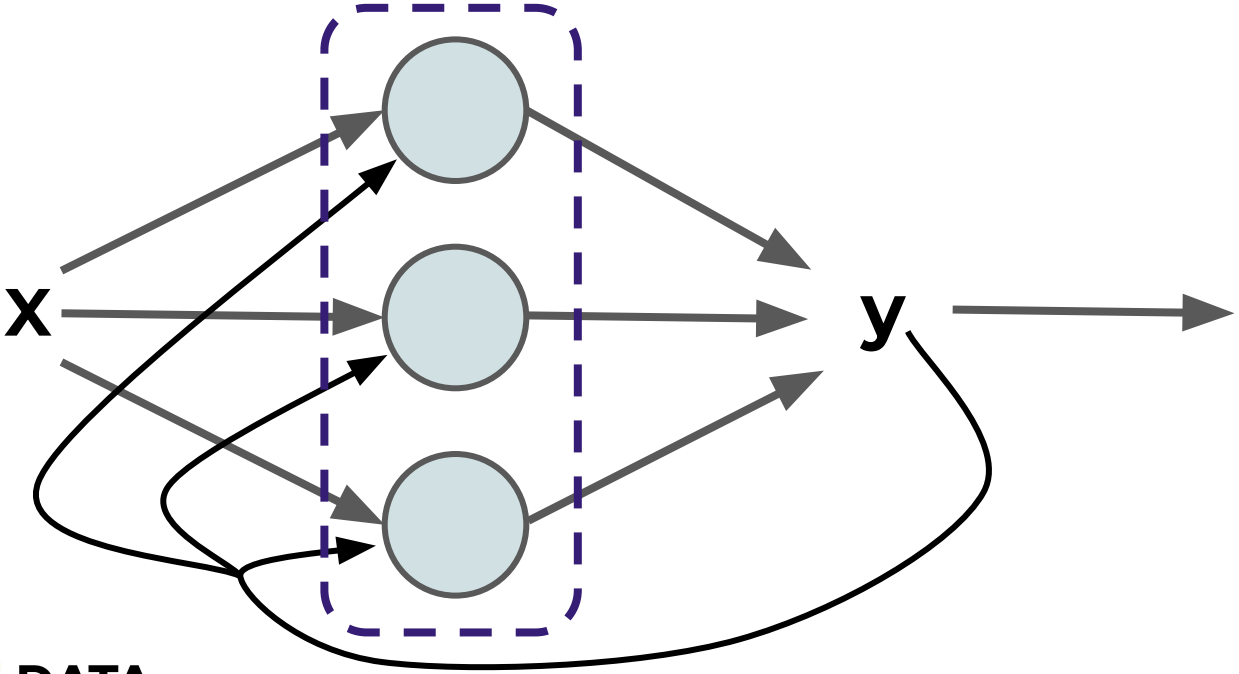
Deep Learning

- ANN Layer with 3 Neurons:



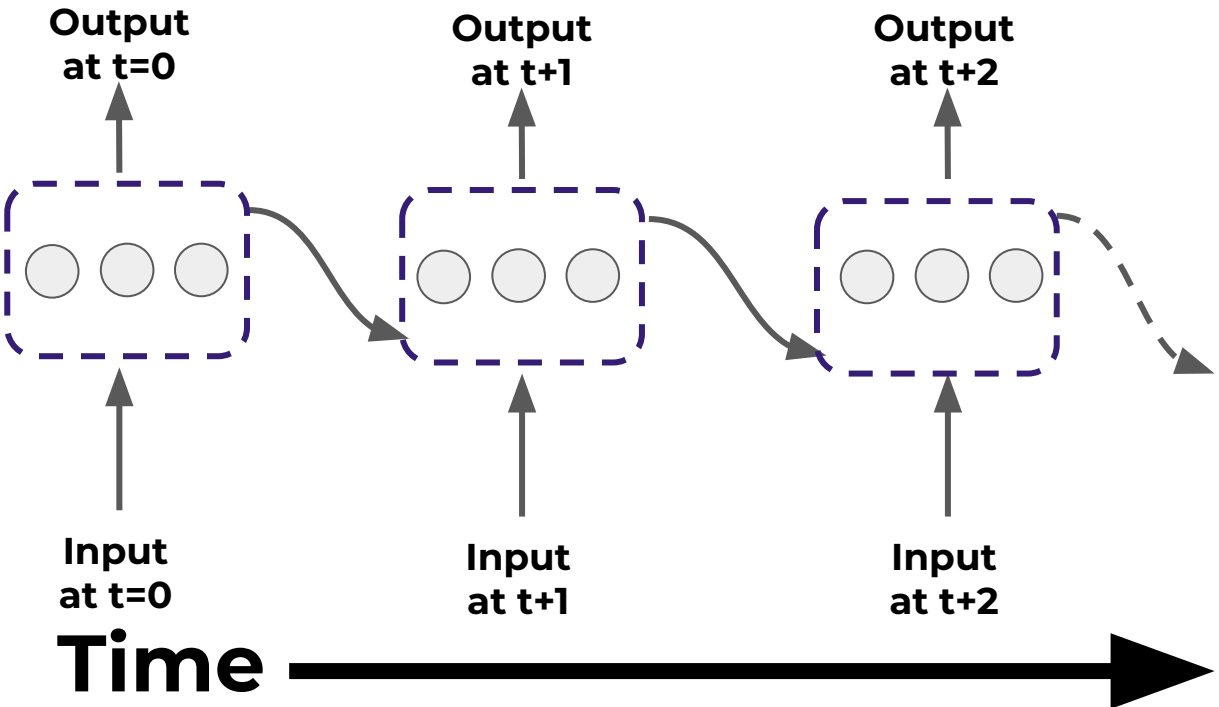
Deep Learning

- RNN Layer with 3 Neurons:



Deep Learning

- “Unrolled” layer.

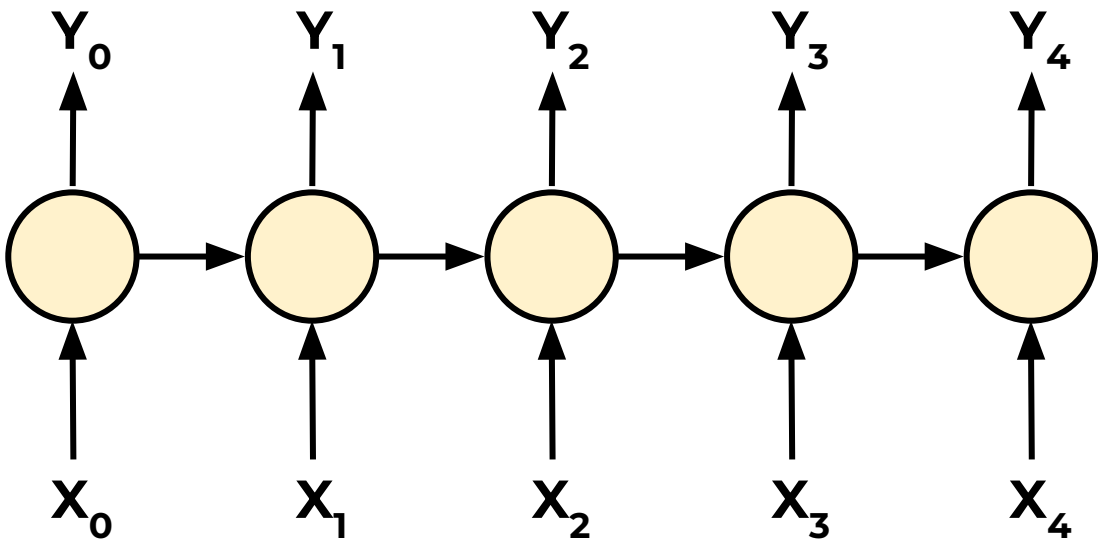


Deep Learning

- RNN are also very flexible in their inputs and outputs.
- Let’s see a few examples.

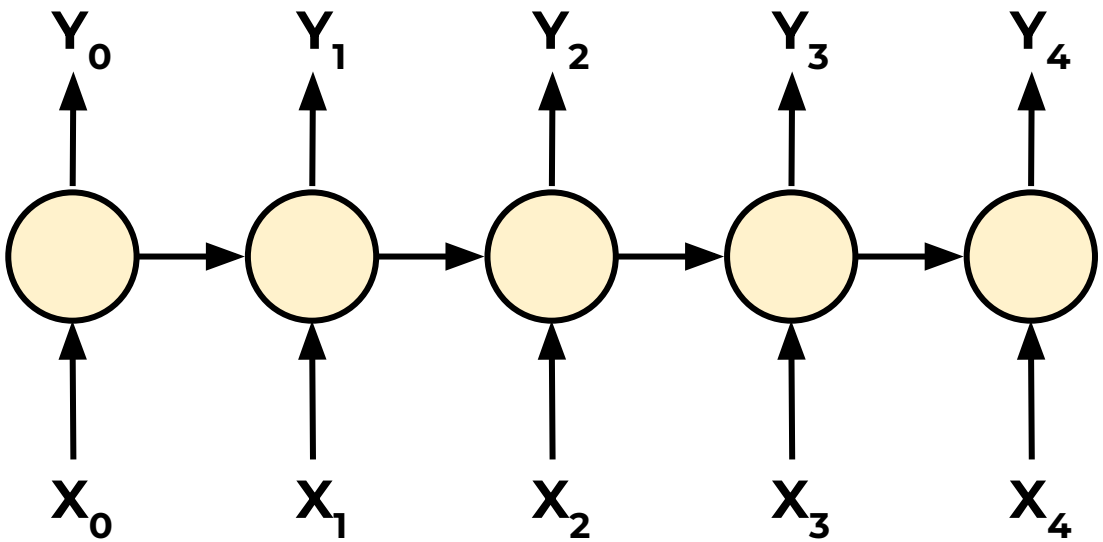
Deep Learning

- Sequence to Sequence (Many to Many)



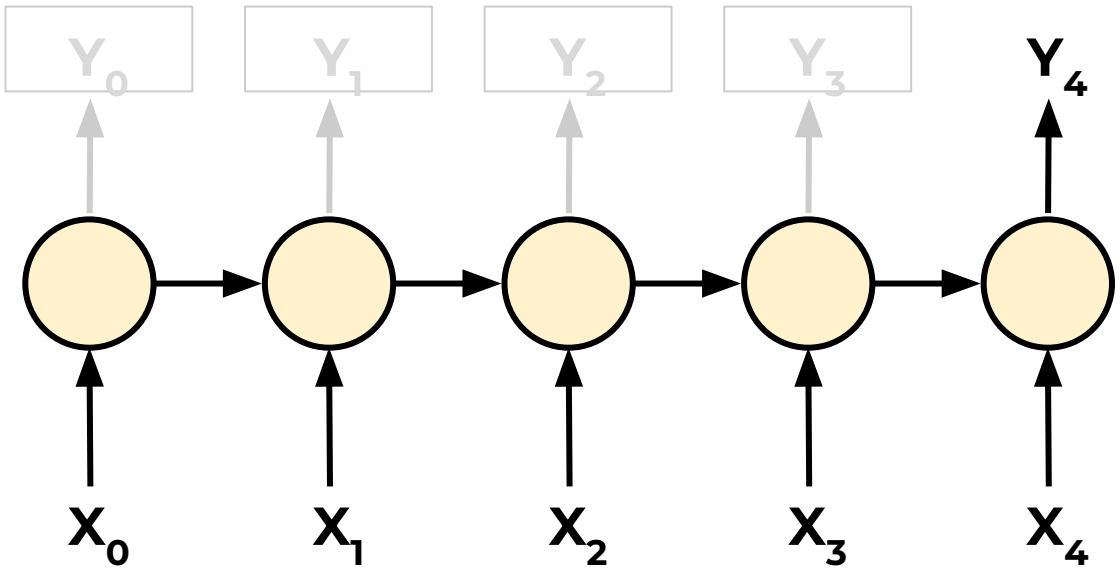
Deep Learning

- Given 5 previous words, predict the next 5



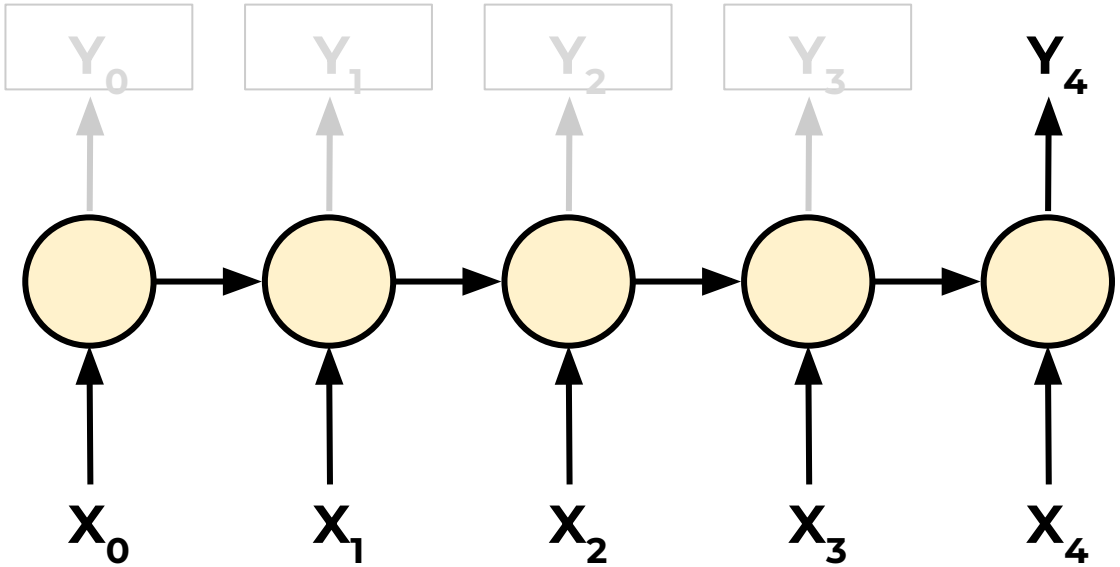
Deep Learning

- Sequence to Vector (Many to One)



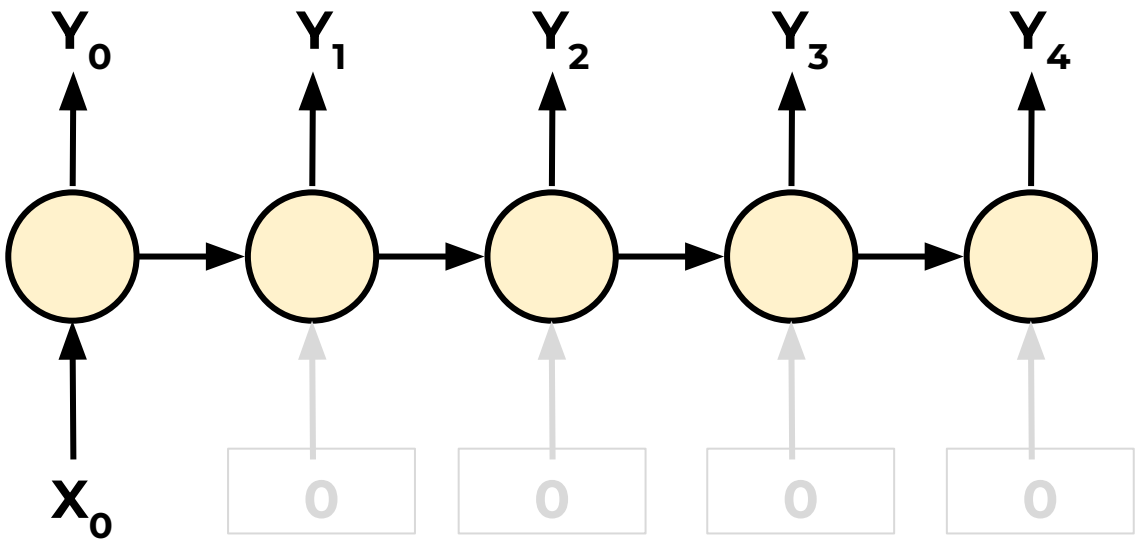
Deep Learning

- Given 5 previous words, predict next word



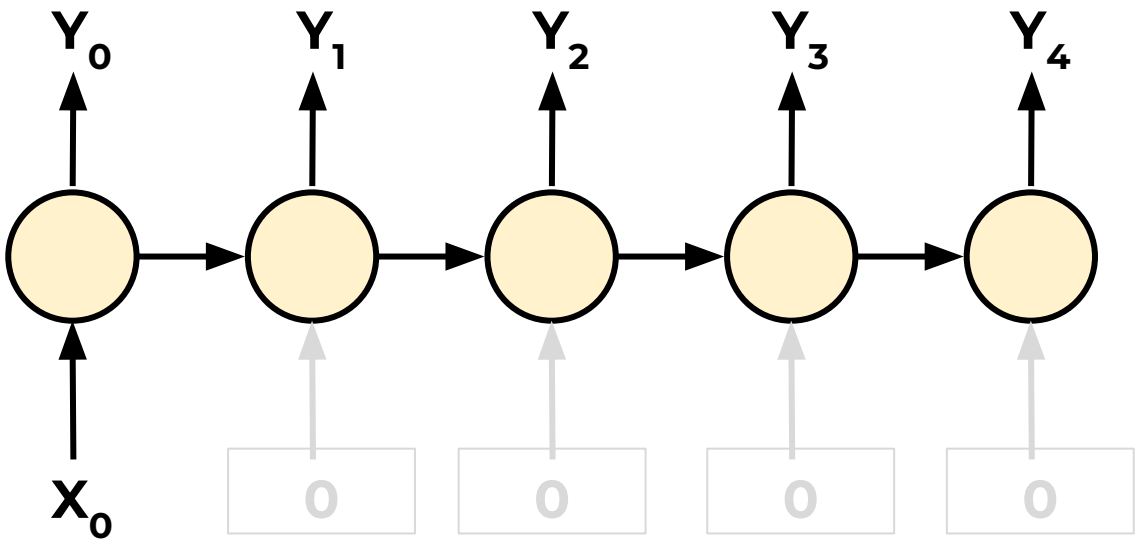
Deep Learning

- Vector to Sequence (One to Many)



Deep Learning

- Given 1 word predict the next 5 words



Deep Learning

- A basic RNN has a major disadvantage, we only really “remember” the previous output.
- It would be great it we could keep track of longer history, not just short term history.

Deep Learning

- Another issue that arises during training is the “vanishing gradient”.
- Let’s explore vanishing gradients in more detail before moving on to discussing LSTM (Long Short Term Memory Units).

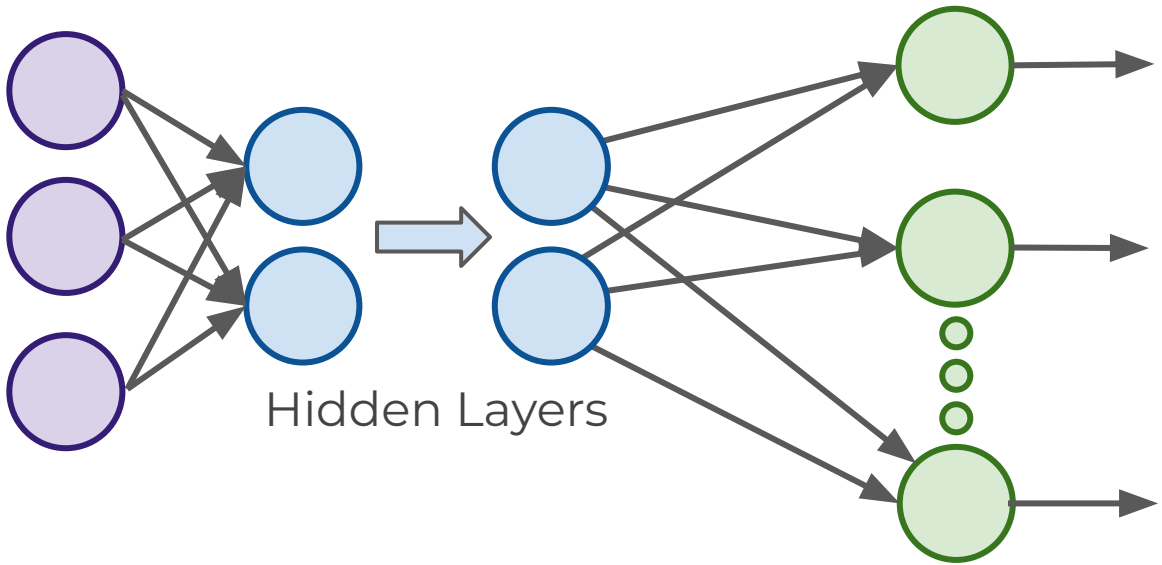
Exploding and Vanishing Gradients

Deep Learning

- As our networks grow deeper and more complex, we have 2 issues arise:
    - Exploding Gradients
    - Vanishing Gradients
- Recall that the gradient is used in our calculation to adjust weights and biases in our network.

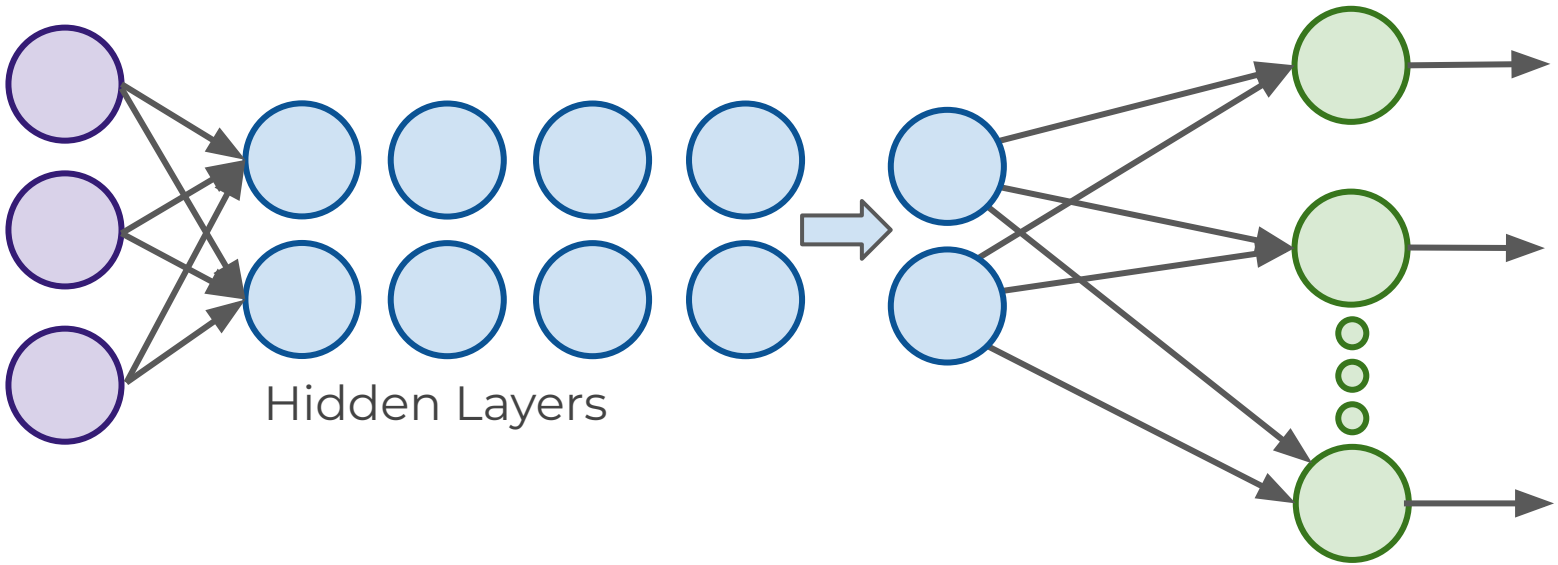
Deep Learning

- Let’s think about a network.



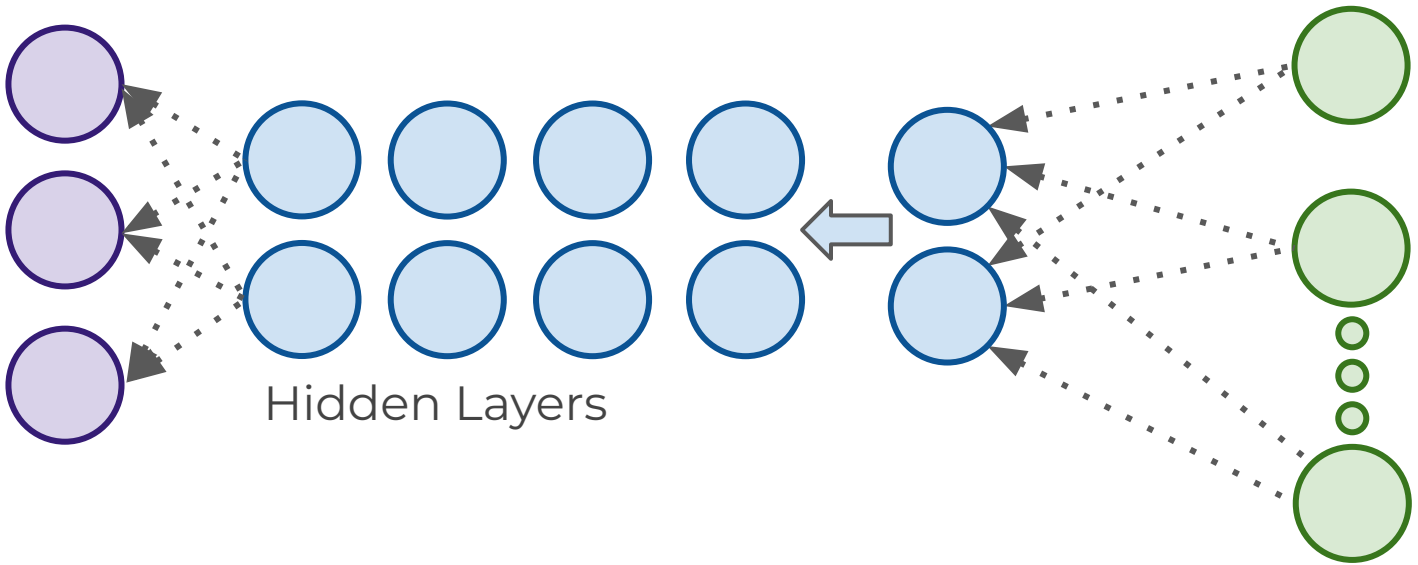
Deep Learning

- For complex data we need deep networks



Deep Learning

- Issues can arise during backpropagation



Deep Learning

- Backpropagation goes backwards from the output to the input layer, propagating the error gradient.
- For deeper networks issues can arise from backpropagation, vanishing and exploding gradients!

Deep Learning

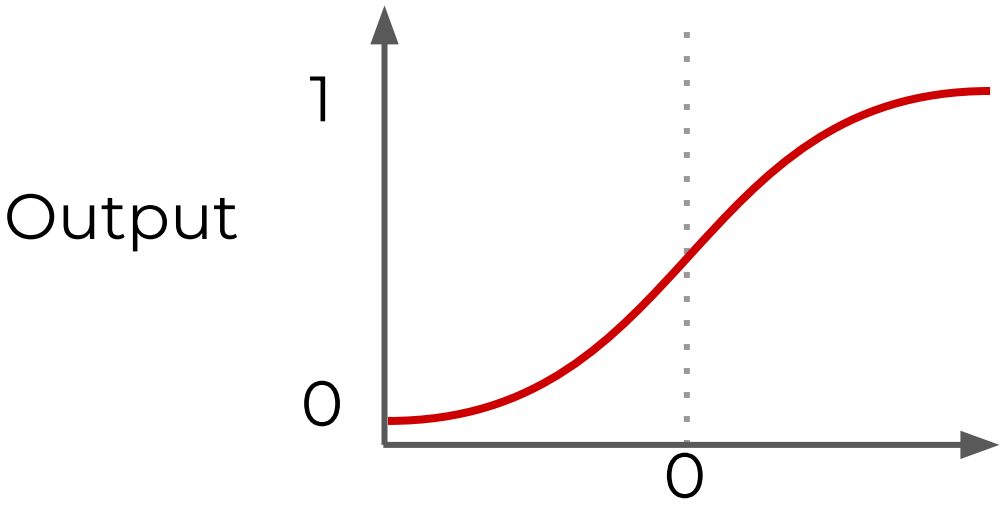
- As you go back to the “lower” layers, gradients often get smaller, eventually causing weights to never change at lower levels.
- The opposite can also occur, gradients explode on the way back, causing issues.

Deep Learning

- Let’s discuss why this might occur and how we can fix it.
- Then in the next lecture we’ll discuss how these issues specifically affect RNN and how to use LSTM and GRU to fix them.

Deep Learning

- Why does this happen?



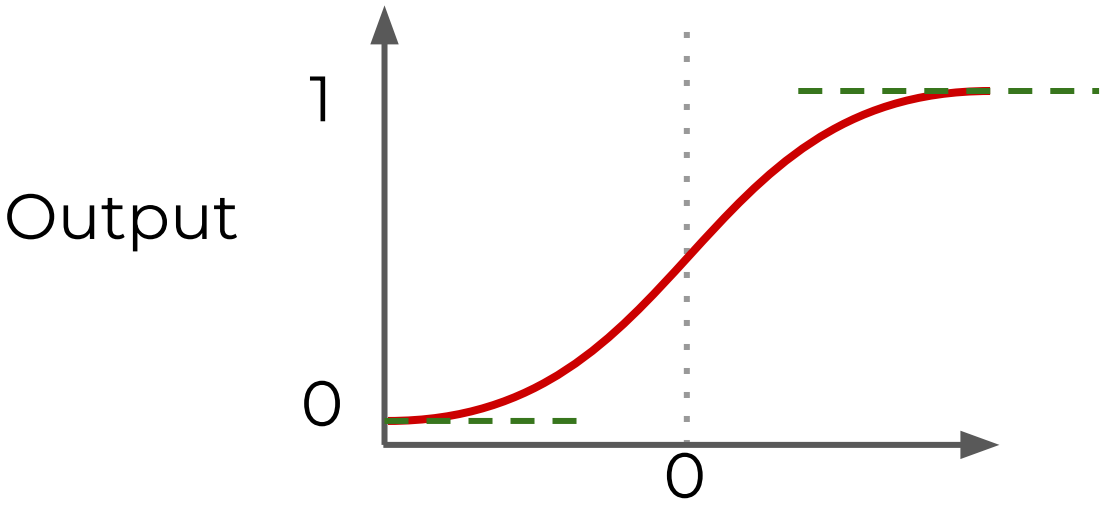
$$f(x) = \frac{1}{1 + e^{-(x)}}$$

z = wx + b

PIERIAN DATA

Deep Learning

- Why does this happen?



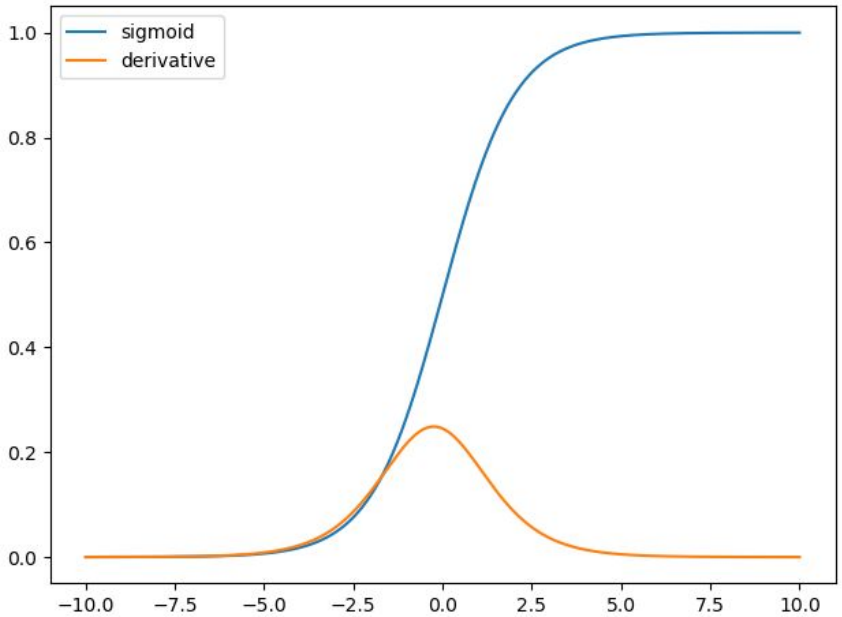
$$f(x) = \frac{1}{1 + e^{-(x)}}$$

z = wx + b

PIERIAN DATA

Deep Learning

- The derivative can be much smaller!

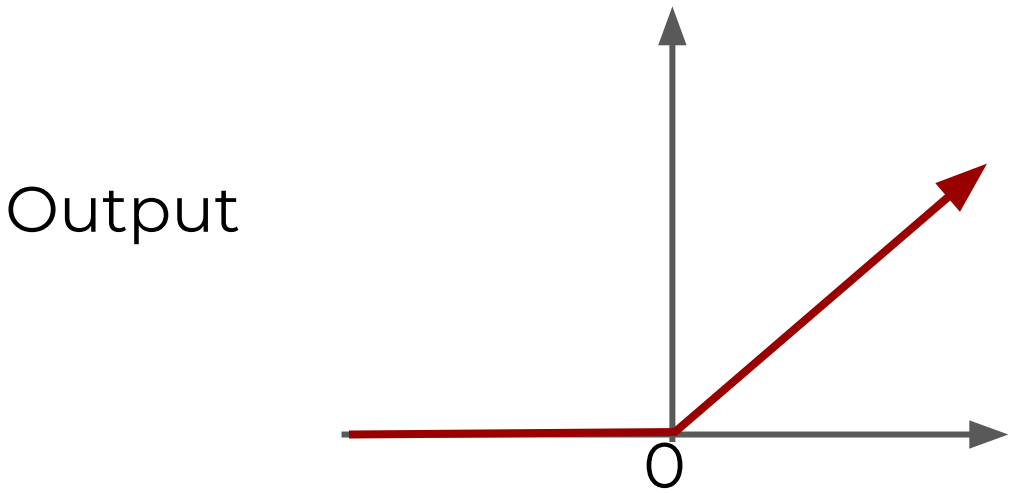


Deep Learning

- When **n** hidden layers use an activation like the sigmoid function, **n** small derivatives are multiplied together.
- The gradient could decrease exponentially as we propagate down to the initial layers.

Deep Learning

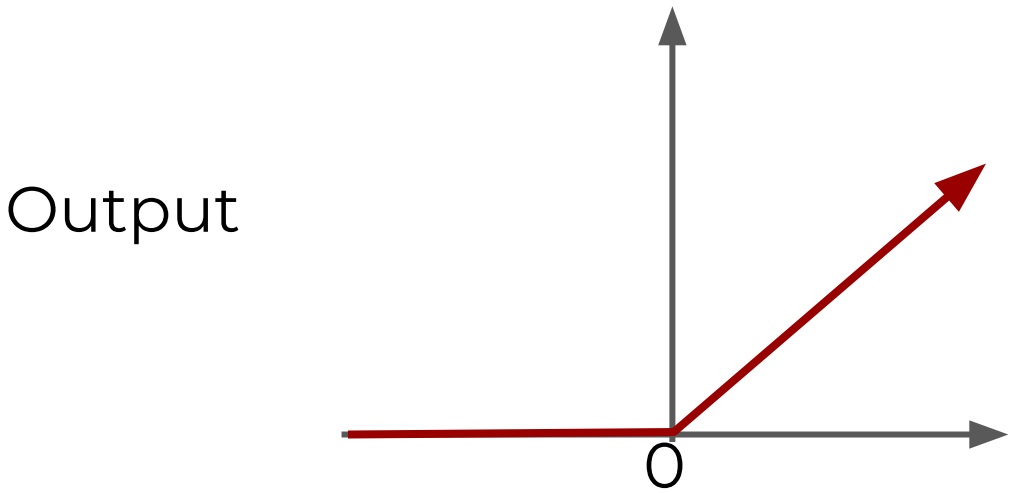
- Using Different Activation Functions



$$z = wx + b$$

Deep Learning

- The ReLu doesn't saturate positive values.

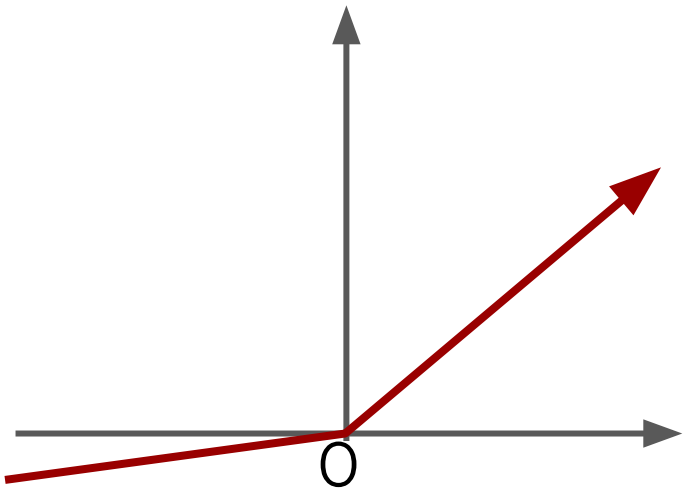


$$z = wx + b$$

Deep Learning

- “Leaky” ReLU

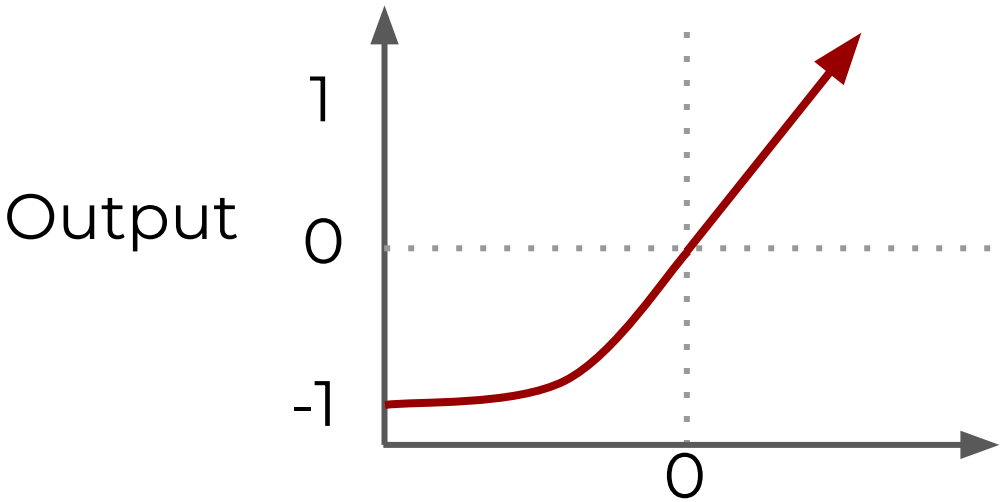
Output



$$z = wx + b$$

Deep Learning

- Exponential Linear Unit (ELU)



$$z = wx + b$$

Deep Learning

- Another solution is to perform batch normalization, where your model will normalize each batch using the batch mean and standard deviation.

Deep Learning

- Choosing different initialization of weights can also help alleviate these issues (Xavier Initialization).

Deep Learning

- Apart from batch normalization, researchers have also used “gradient clipping”, where gradients are cut off before reaching a predetermined limit (e.g. cut off gradients to be between -1 and 1)

Deep Learning

- RNN for Time Series present their own gradient challenges, let’s explore special LSTM (Long Short Term Memory) neuron units that help fix these issues!

LSTM and GRU Units

Deep Learning

- Many of the solutions previously presented for vanishing gradients can also apply to RNN: different activation functions, batch normalizations, etc...
- However because of the length of time series input, these could slow down training

Deep Learning

- A possible solution would be to just shorten the time steps used for prediction, but this makes the model worse at predicting longer trends.

Deep Learning

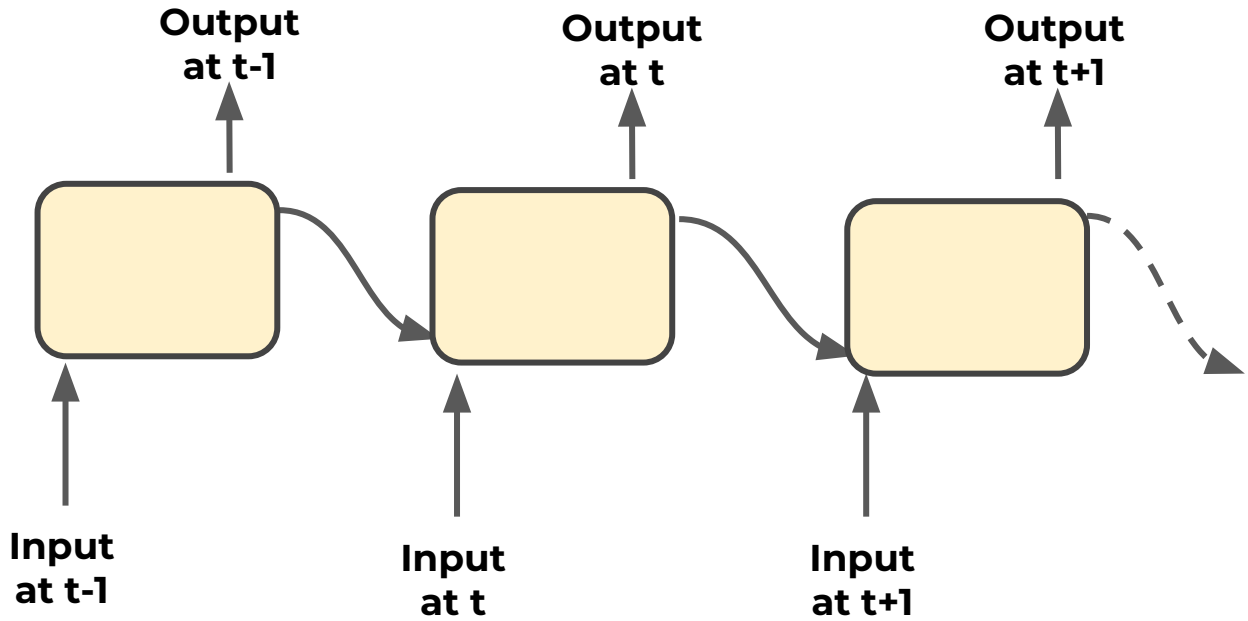
- Another issue RNN face is that after awhile the network will begin to “forget” the first inputs, as information is lost at each step going through the RNN.
- We need some sort of “long-term memory” for our networks.

Deep Learning

- The LSTM (Long Short-Term Memory) cell was created to help address these RNN issues.
- Let’s go through how an LSTM cell works!

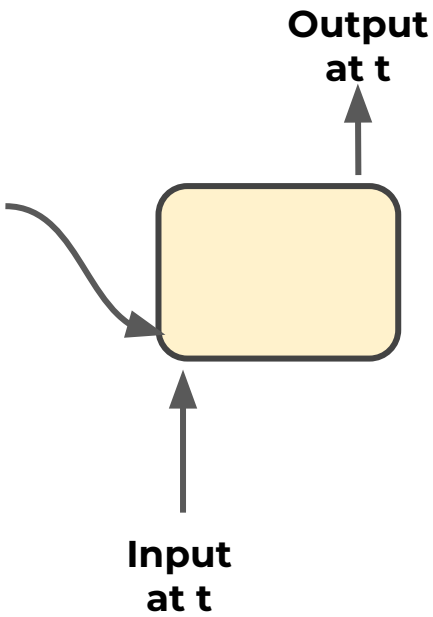
Deep Learning

- A typical RNN



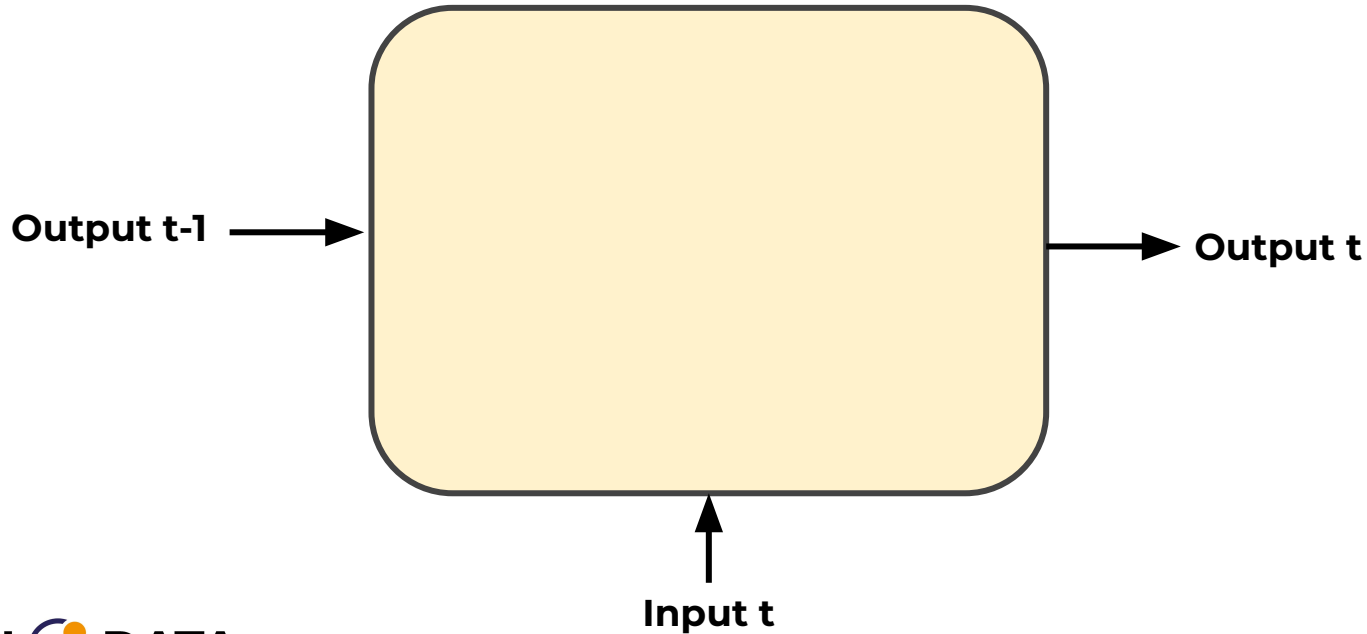
Deep Learning

- RNN



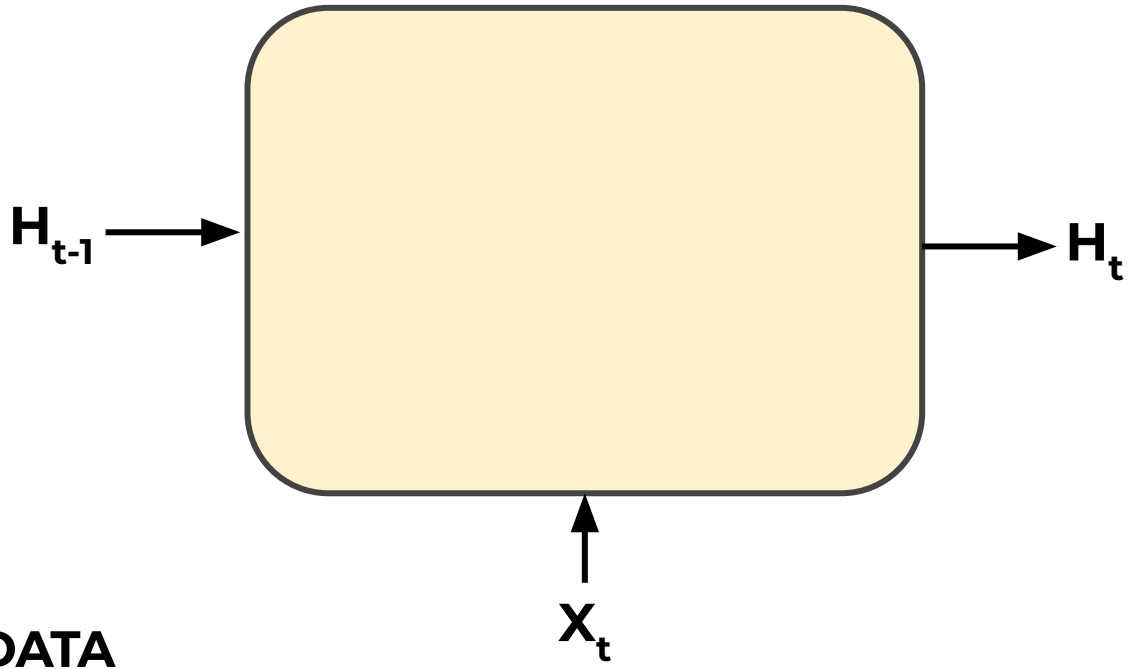
Deep Learning

- RNN



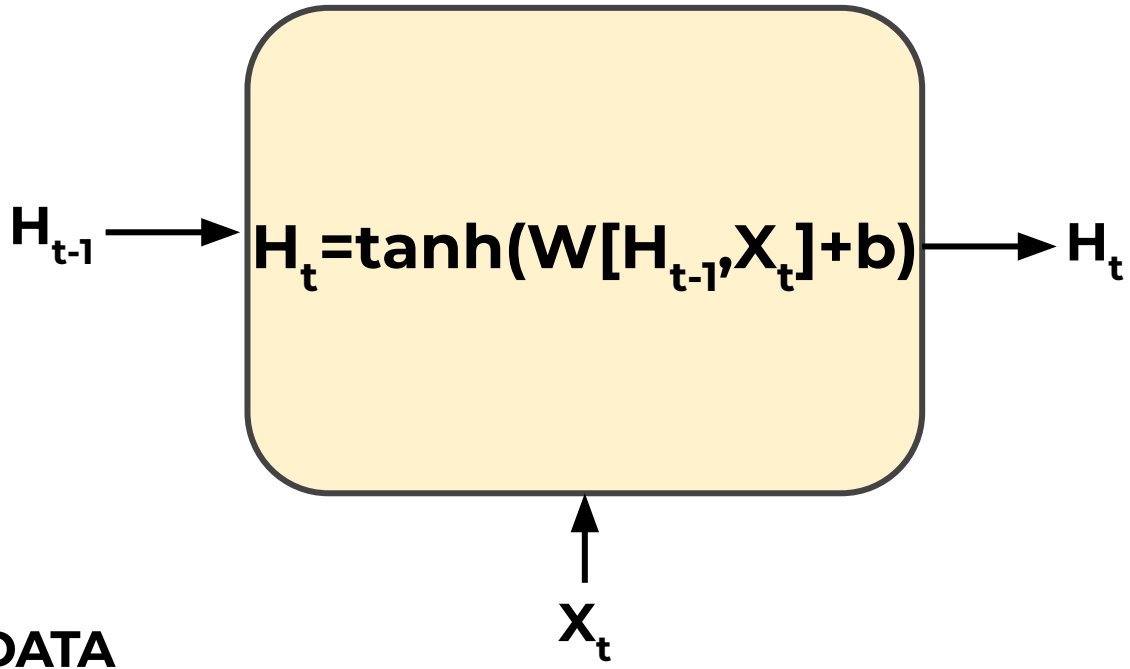
Deep Learning

- RNN



Deep Learning

- RNN

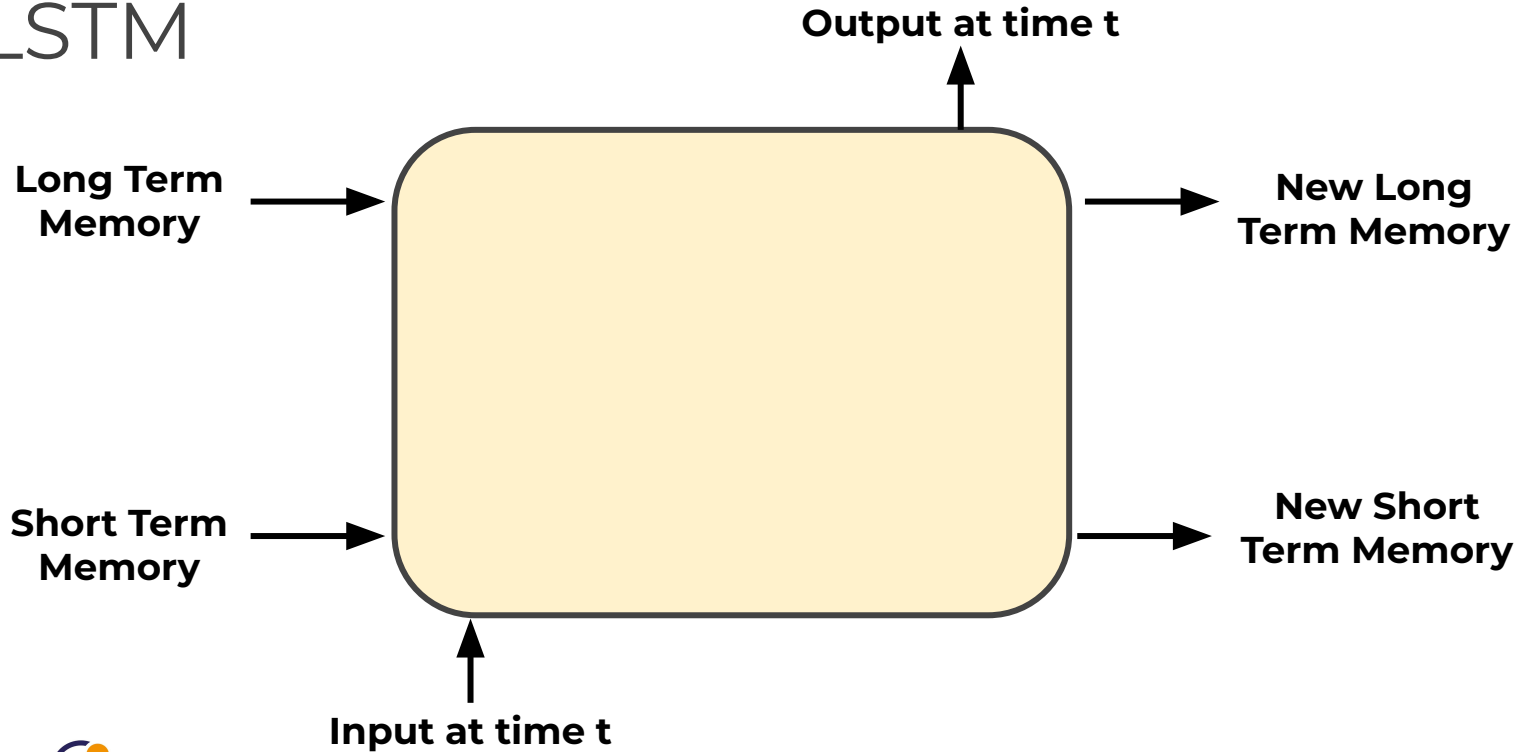


Deep Learning

- LSTM

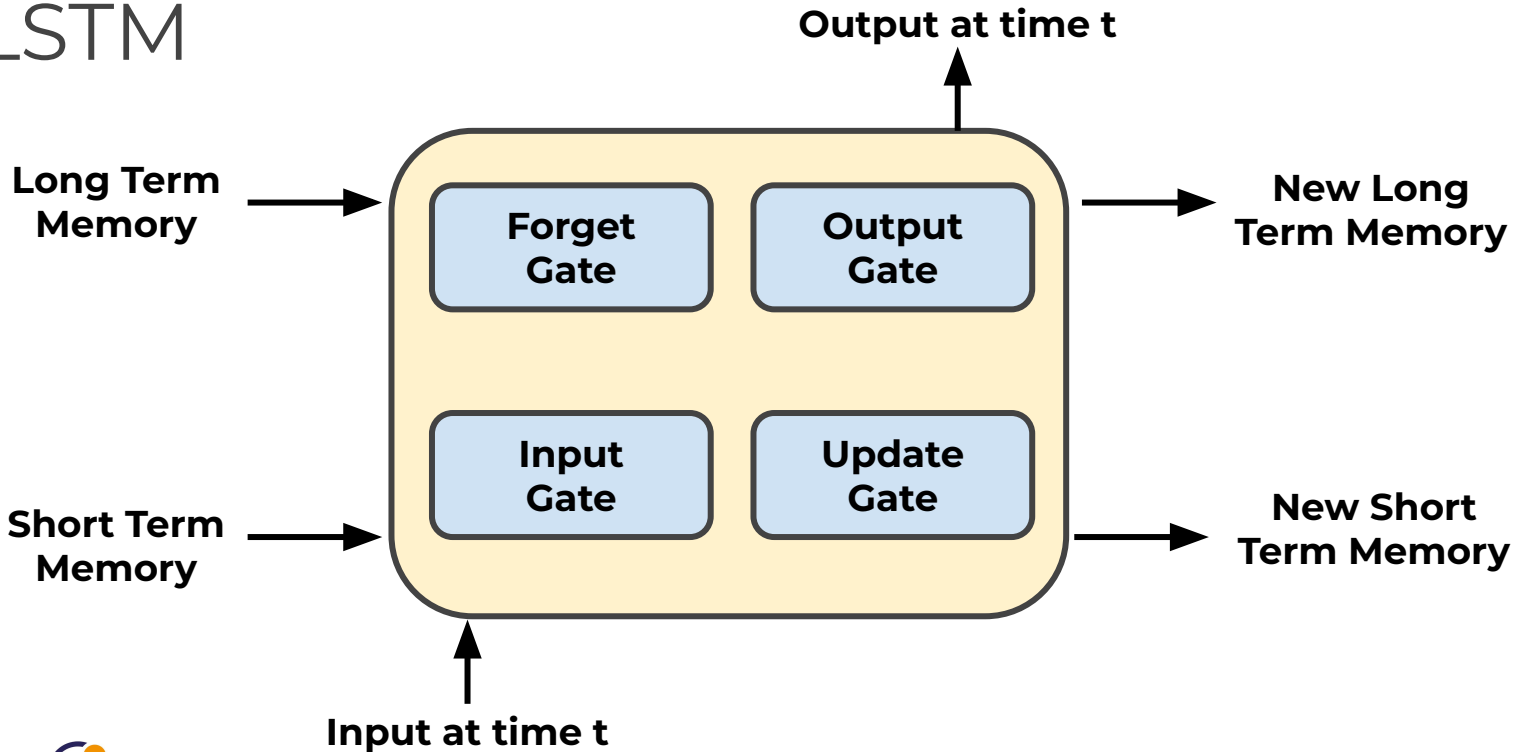
Deep Learning

- LSTM



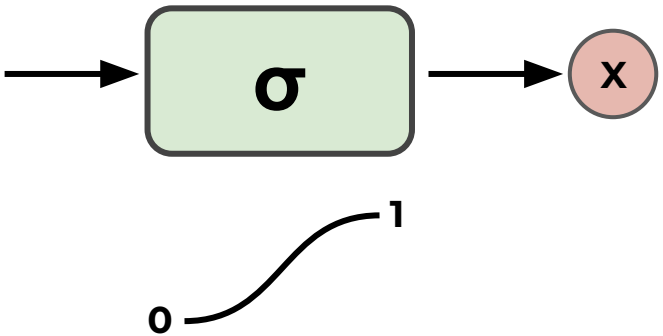
Deep Learning

- LSTM



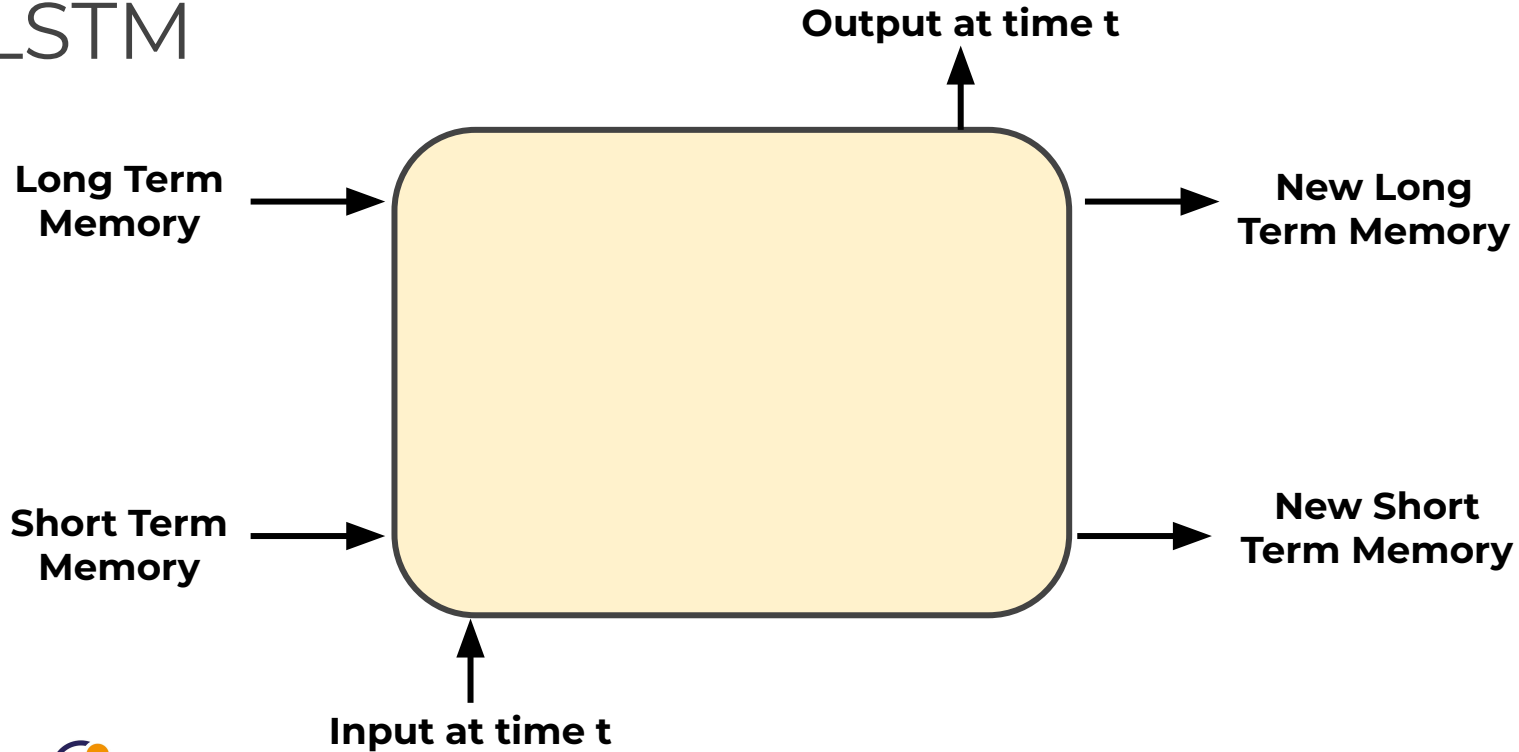
Deep Learning

- Gates optionally let information through



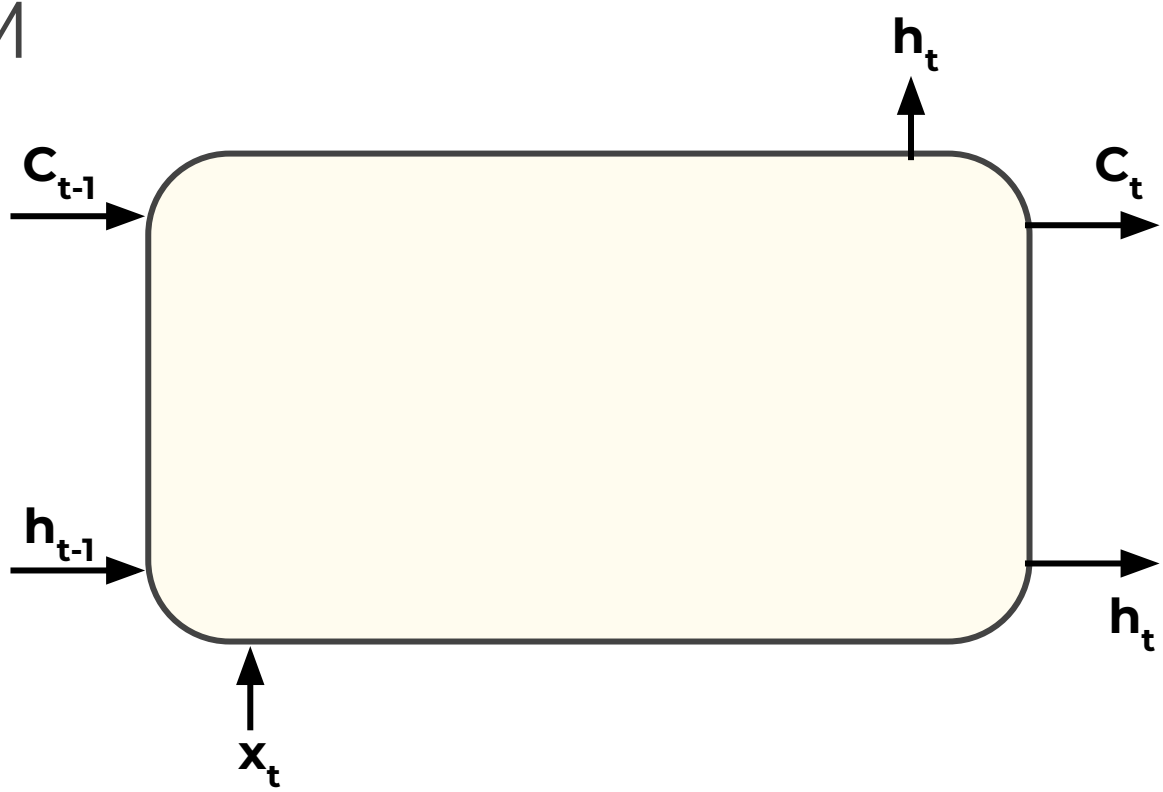
Deep Learning

- LSTM

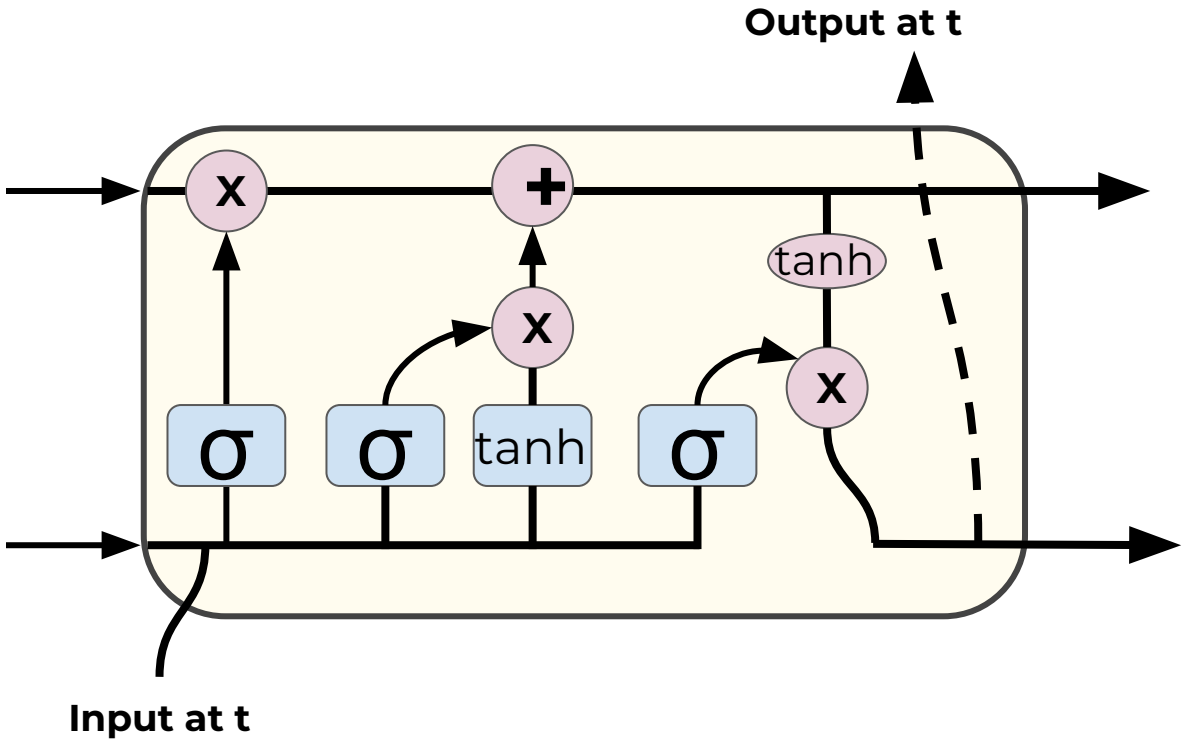


Deep Learning

- LSTM



An LSTM Cell

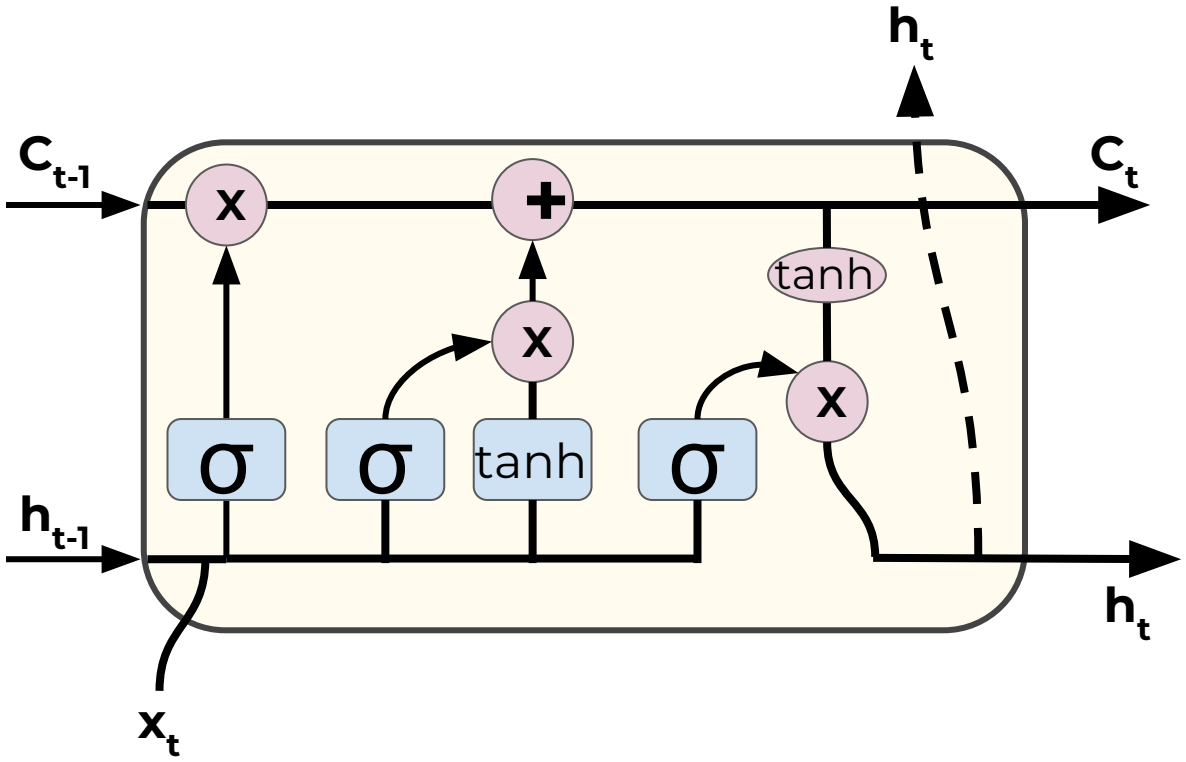


Here we can see the entire LSTM cell.

Let's go through the process!

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

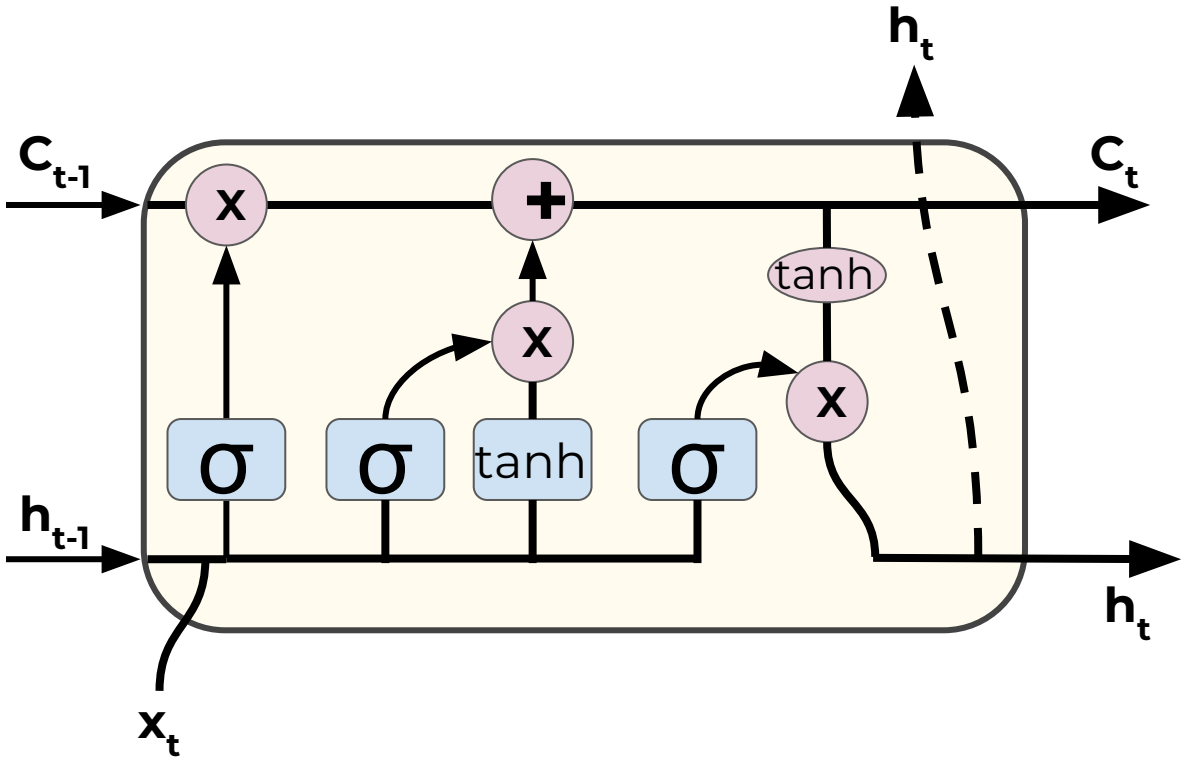
An LSTM Cell



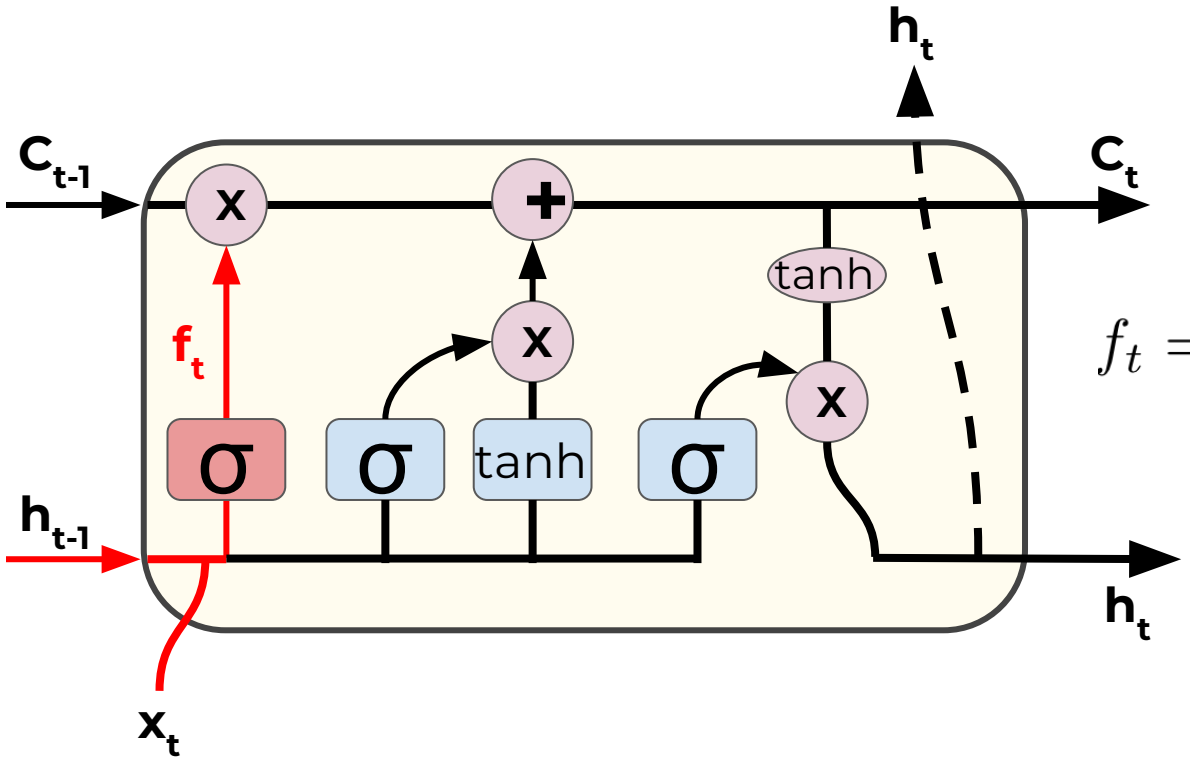
Here we can see the entire LSTM cell.

Let's go through the process!

An LSTM Cell

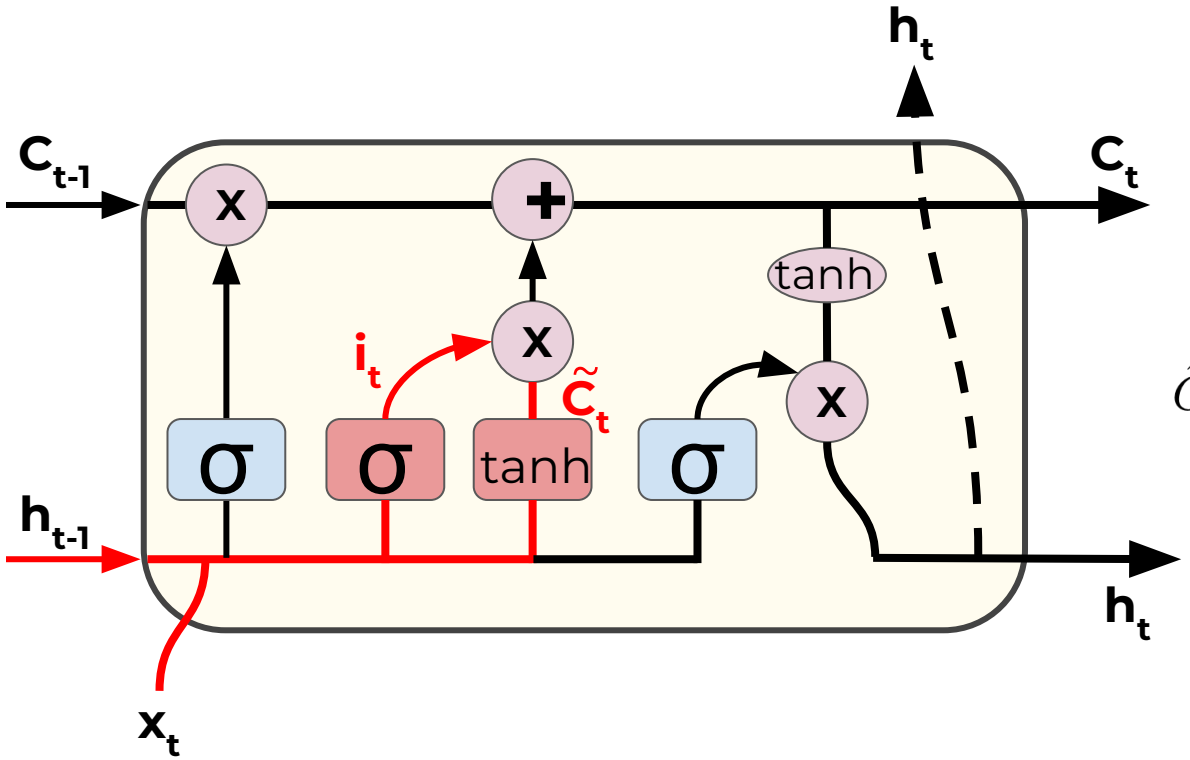


An LSTM Cell



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

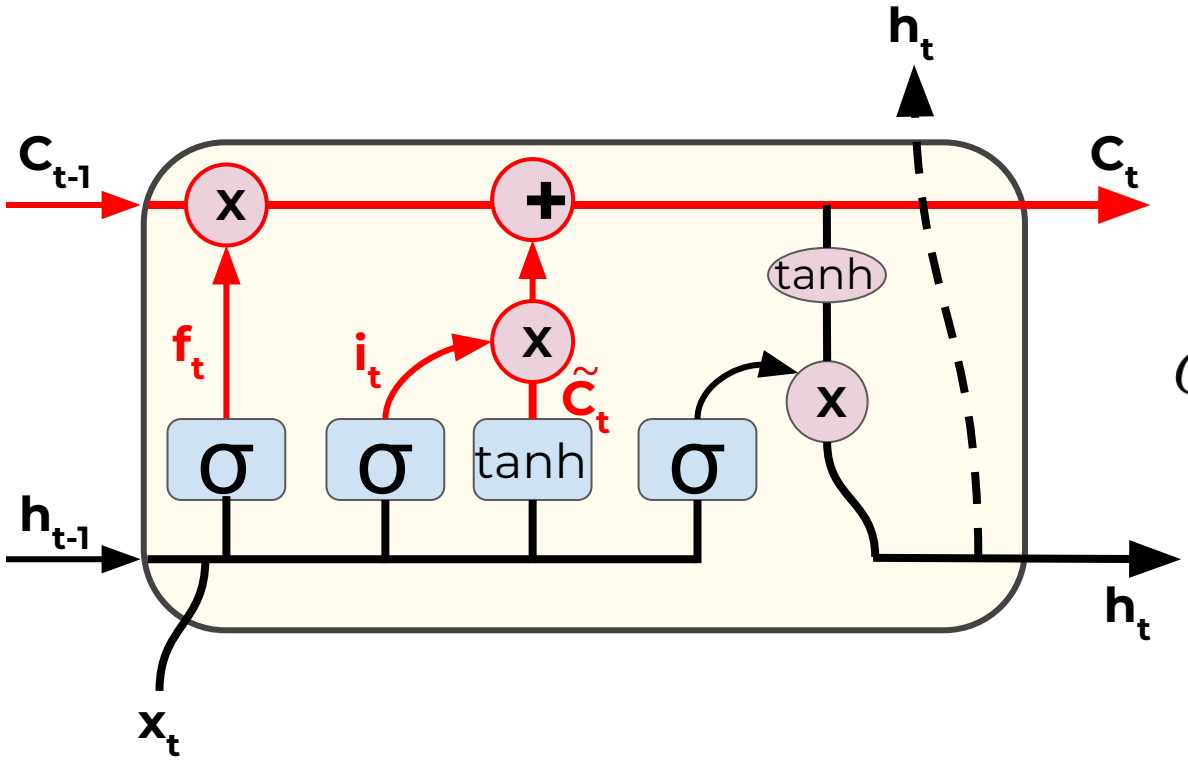
An LSTM Cell



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

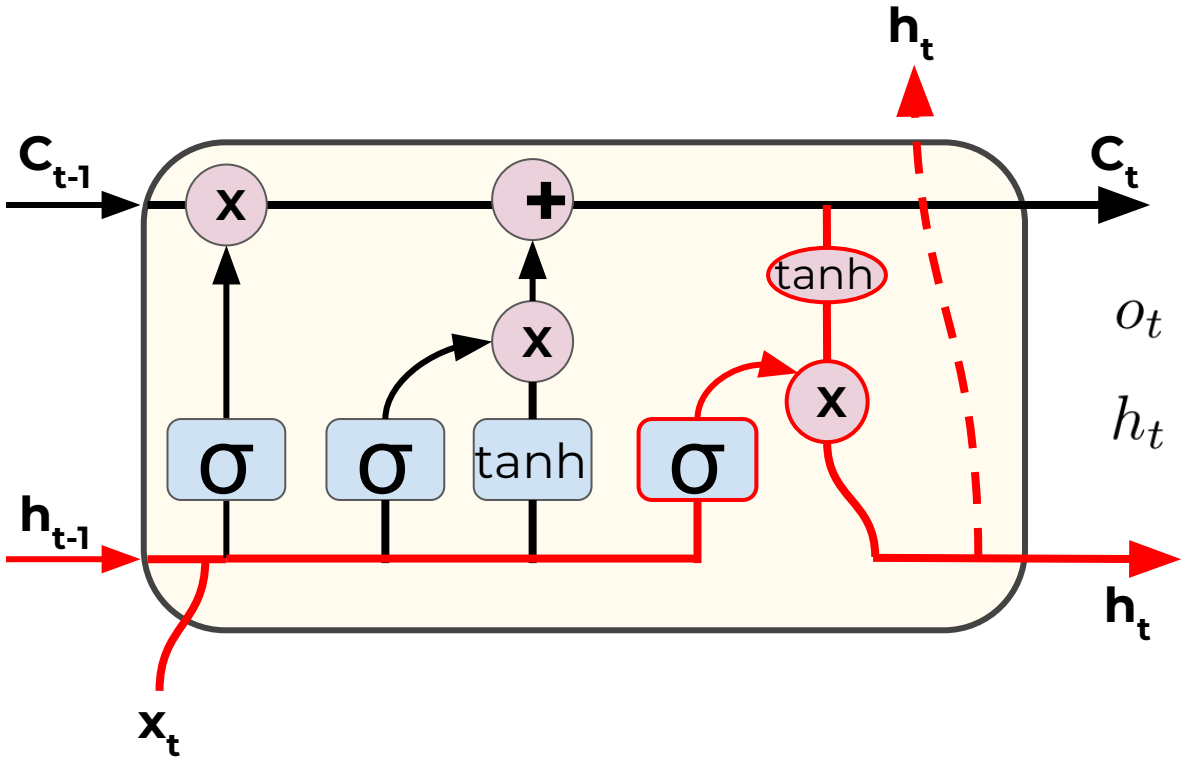
An LSTM Cell



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

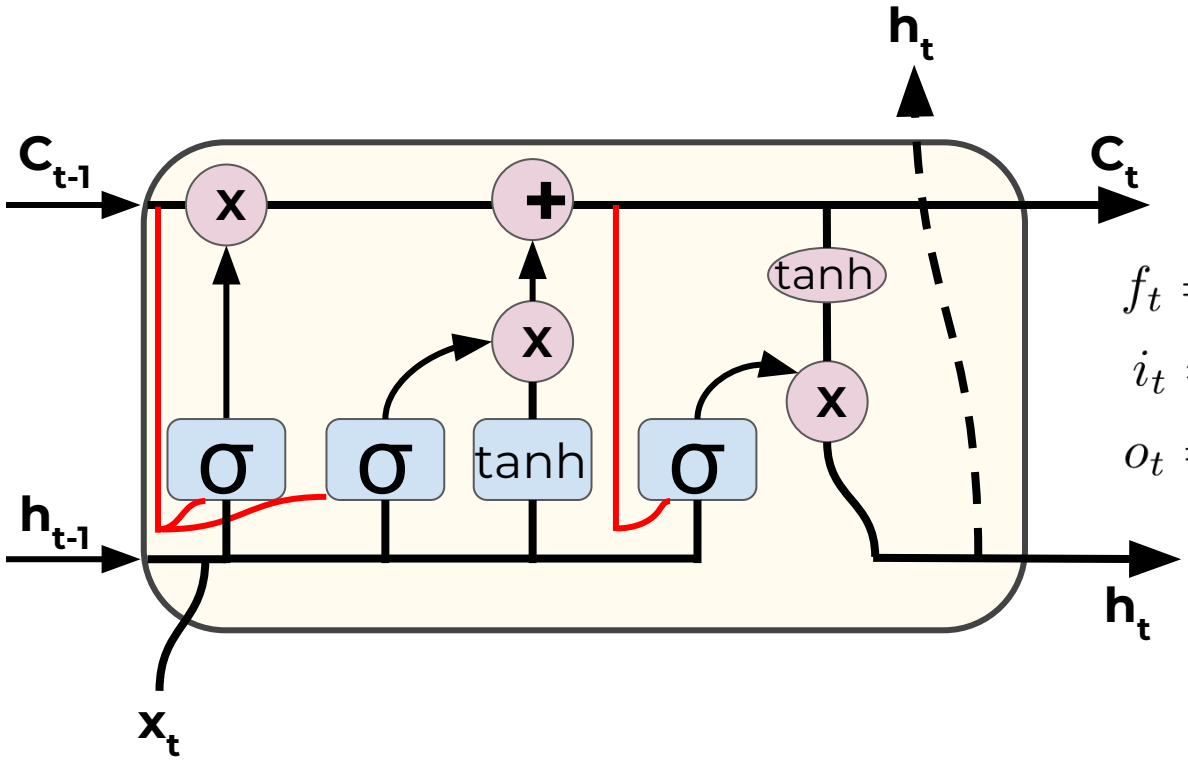
PIERIAN DATA

An LSTM Cell



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$



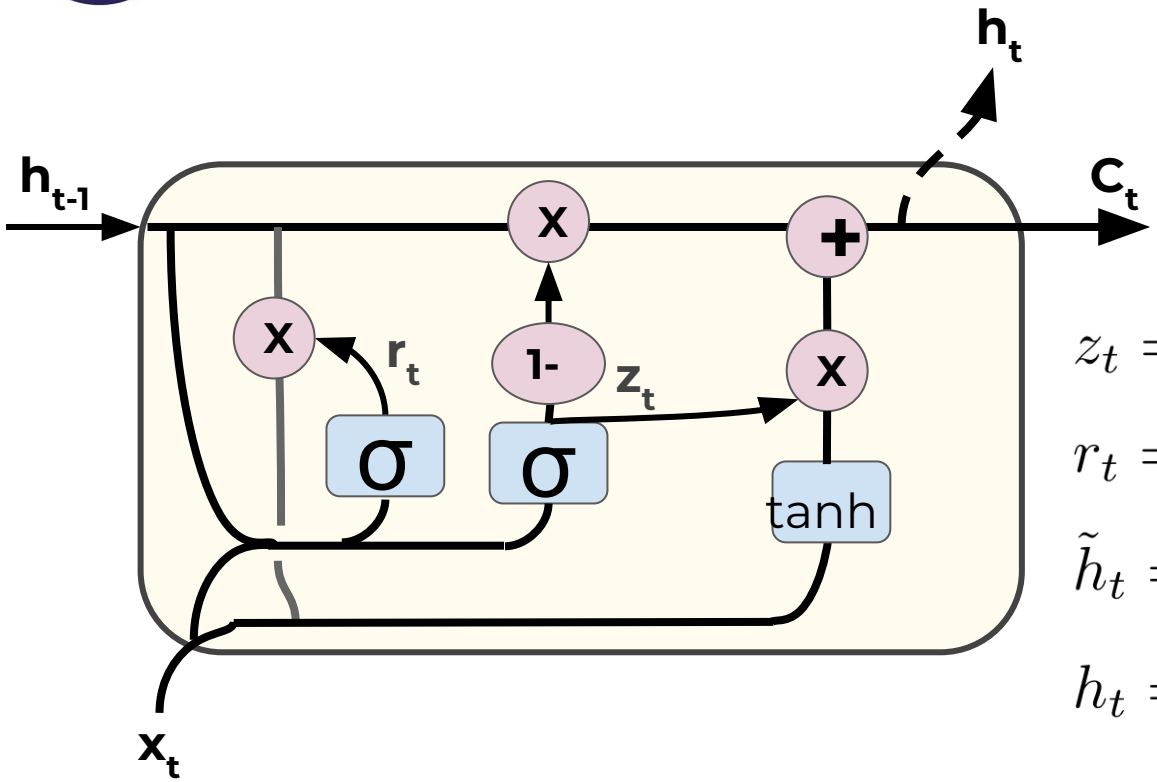
An LSTM Cell with “peepholes”

$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$

$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Gated Recurrent Unit (GRU)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Deep Learning

- Fortunately with our deep learning python library , we simply need to call the import for RNN or LSTM instead of needing to code all of this ourselves!
- Let’s explore how to use LSTMs with Python code!

Basic RNN

Deep Learning

- Let’s now explore how to use RNN on a basic time series, such as a sine wave.
- Before we jump to the notebook, let’s quickly discuss what RNN sequence batches look like.

Deep Learning

- Let’s imagine a simple time series:
  - **[0,1,2,3,4,5,6,7,8,9]**
  - We separate this into 2 parts:
    - **[0,1,2,3,4,5,6,7,8]** ⇨ **[9]**
  - Given **training sequence**, predict the **next sequence value**.

Deep Learning

- Keep in mind we can usually decide how long the training sequence and predicted label should be:
    - **[0,1,2,3,4] ⇨ [5,6,7,8,9]**

Deep Learning

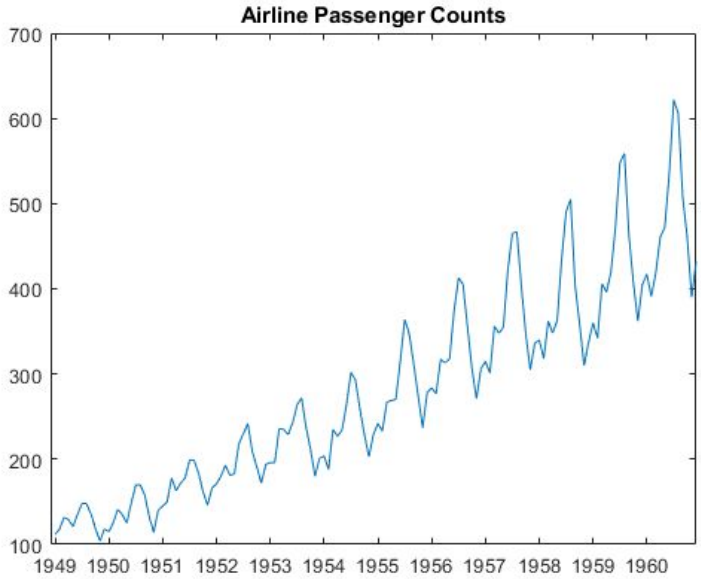
- We can also edit the size of the training point, as well as how many sequences to feed per batch:
    - **[0,1,2,3] ⇨ [4]**
    - **[1,2,3,4] ⇨ [5]**
    - **[2,3,4,5] ⇨ [6]**

Deep Learning

- So how do we decide how long the training sequence should be?
    - There is no definitive answer, but it should be at least long enough to capture any useful trend information.

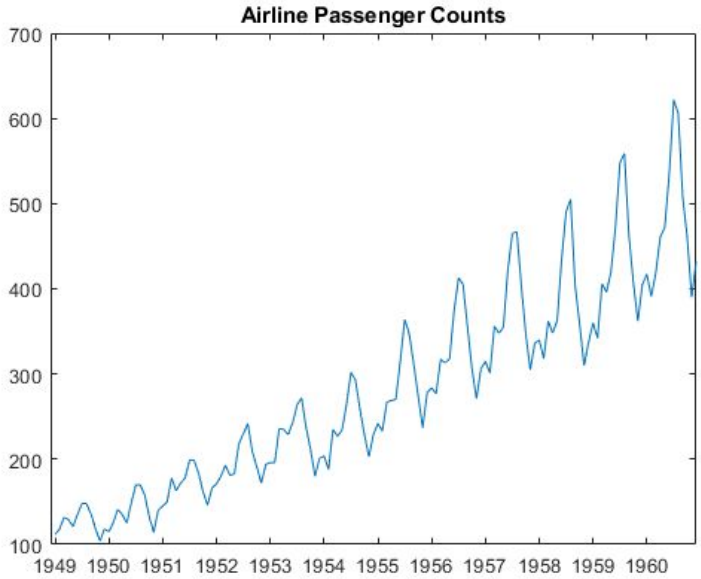
Deep Learning

- For example, if dealing with seasonal data:



Deep Learning

- If this is monthly, we should include at least 12 months in the training sequence



Deep Learning

- This often takes domain knowledge and experience, as well as simply experimenting and using RMSE to measure error of forecasted predictions.
- Typically a good starting choice for the label is just one data point into the future.

Deep Learning

- How do we forecast with RNNs?
- Let's imagine all our data is:
    - **[0,1,2,3,4,5,6,7,8,9]**
    - And we trained on sequences such as:
        - **[0,1,2,3] ⇨ [4]**
        - **[1,2,3,4] ⇨ [5]**
        - **[2,3,4,5] ⇨ [6]**

Deep Learning

- Then our forecasting technique is to predict a time step ahead, and then incorporate our prediction into the next sequence we predict off of.
- Let’s walk through a quick example!

Deep Learning

- How do we forecast with RNNs?
- Let's imagine all our data is:
    - **[0,1,2,3,4,5,6,7,8,9]**
    - And we trained on sequences such as:
        - **[0,1,2,3] ⇨ [4]**
        - **[1,2,3,4] ⇨ [5]**
        - **[2,3,4,5] ⇨ [6]**

Deep Learning

- **[6,7,8,9]** ⇨ **[10]** Forecast prediction!

Deep Learning

- **[6,7,8,9]** ⇨ **[10]** Forecast prediction!
- Then to keep predicting further:

Deep Learning

- **[6,7,8,9] ⇨ [10]** Forecast prediction!
- Then to keep predicting further:
  - **[7,8,9,10] ⇨ [11.2]**

Deep Learning

- **[6,7,8,9] ⇨ [10]** Forecast prediction!
- Then to keep predicting further:
  - **[7,8,9,10] ⇨ [11.2]**
  - **[8,9,10,11.2] ⇨ [12.4]**

Deep Learning

- **[6,7,8,9] ⇨ [10]** Forecast prediction!
- Then to keep predicting further:
  - **[7,8,9,10] ⇨ [11.2]**
  - **[8,9,10,11.2] ⇨ [12.4]**
  - **[9,10,11.2,12.4] ⇨ [14]**

Deep Learning

- **[6,7,8,9]** ⇨ **[10]** Forecast prediction!
- Then to keep predicting further:
  - **[7,8,9,10]** ⇨ **[11.2]**
  - **[8,9,10,11.2]** ⇨ **[12.4]**
  - **[9,10,11.2,12.4]** ⇨ **[14]**
  - **[10,11.2,12.4,14]** ⇨ Completed Forecast

Deep Learning

- Let’s explore this further with Python!

Let’s get started!

Keras and RNN

CODE ALONG PROJECT
PART ONE

PIERIAN DATA

Keras and RNN

EXERCISE PROJECT

PIERIAN DATA

Keras and RNN

EXERCISE PROJECT - SOLUTIONS

PIERIAN DATA