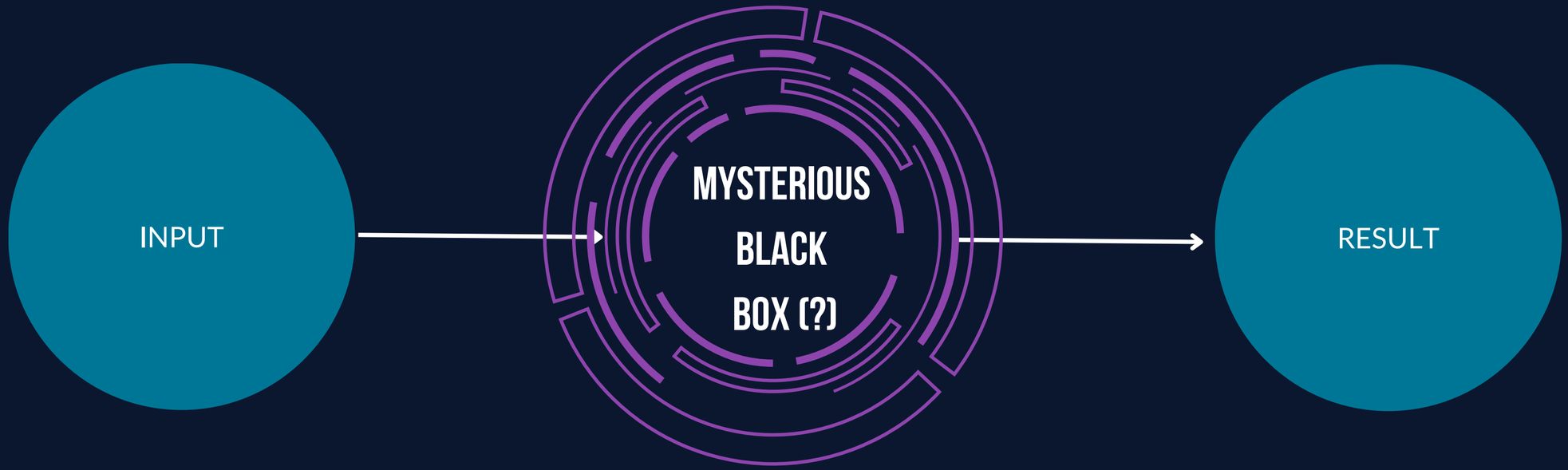
INTELLIGENT SYSTEMS

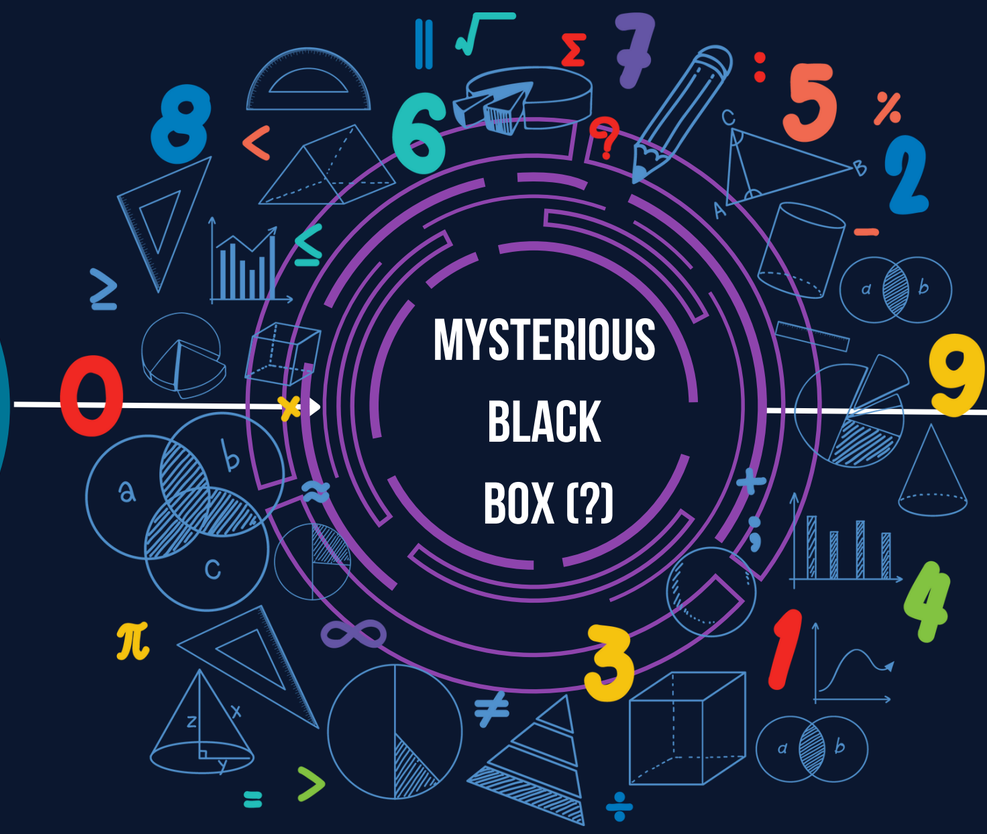
ARTIFICIAL INTELLIGENCE CONCEPTS

INTRODUCTION

- CONCEPTS OF ARTIFICIAL NEURAL NETWORKS
- INTRODUCTION TO THE ARCHITECTURE OF ANN
- INTRODUCTION TO THE DIFFERENT TYPES OF ANN


INPUT
MYSTERIOUS
BLACK
BOX (?)
RESULT

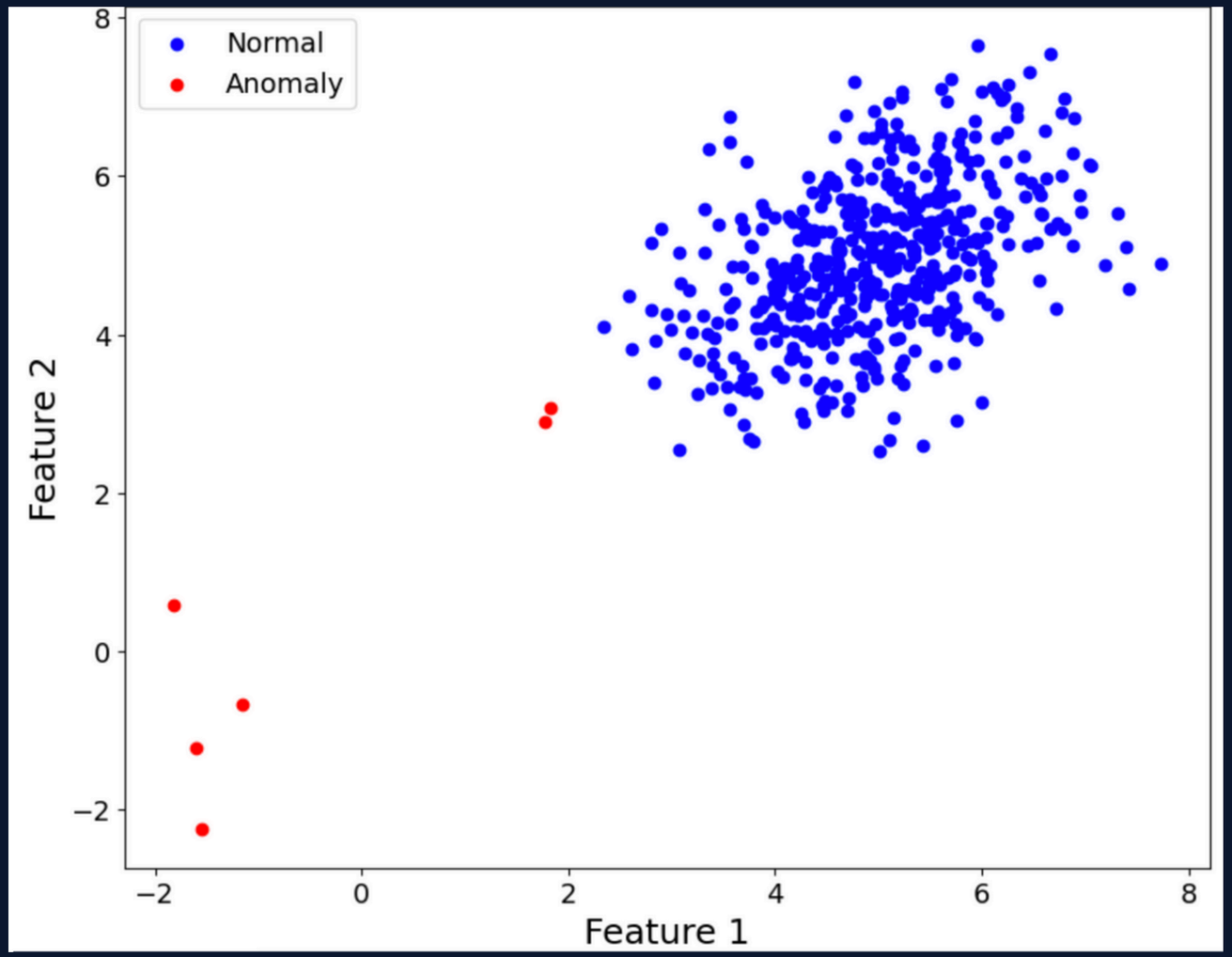
INPUT



RESULT

GOAL: Determine if Anomaly or Normal

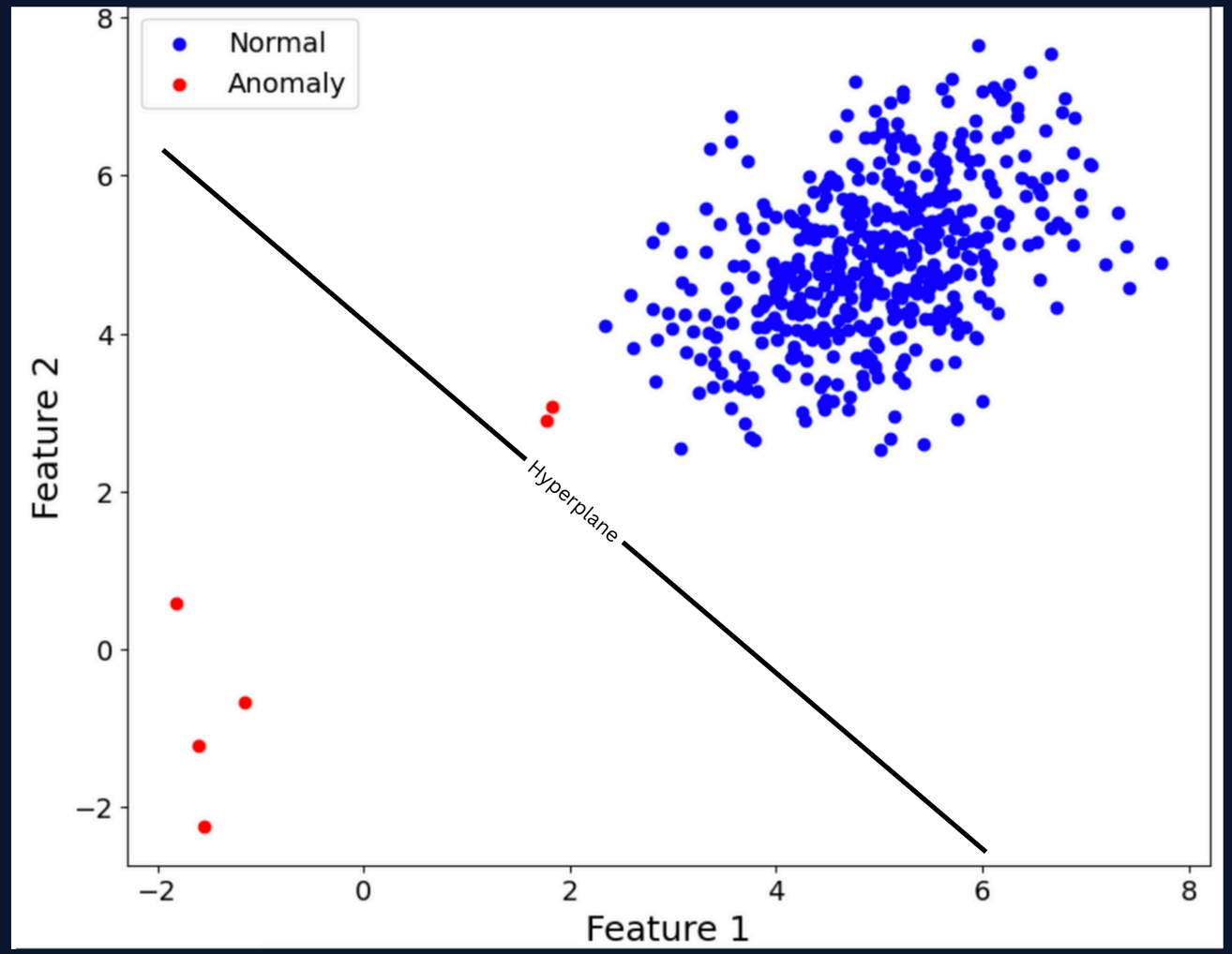
FEATURE 1	FEATURE 2	Y
X11	X21	0
X12	X22	1
X1N	X2N	0



GOAL: Determine if Anomaly or Normal

FEATURE 1	FEATURE 2	Y
X11	X21	0
X12	X22	1
X1N	X2N	0

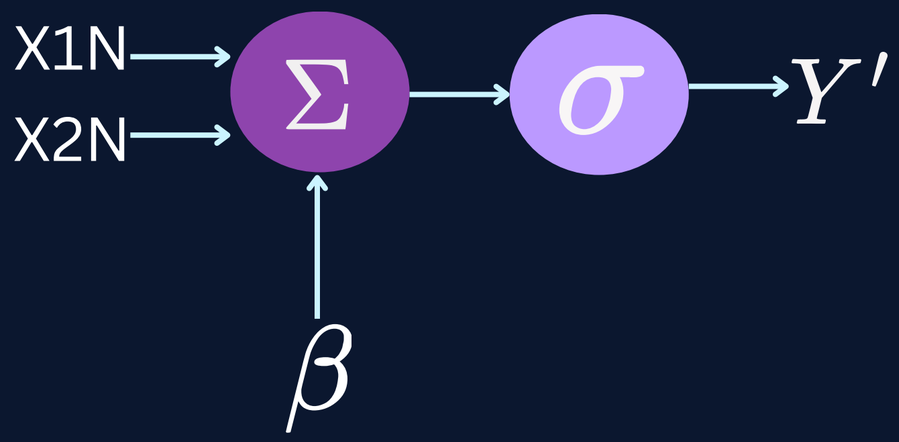
Solve using classical linear models

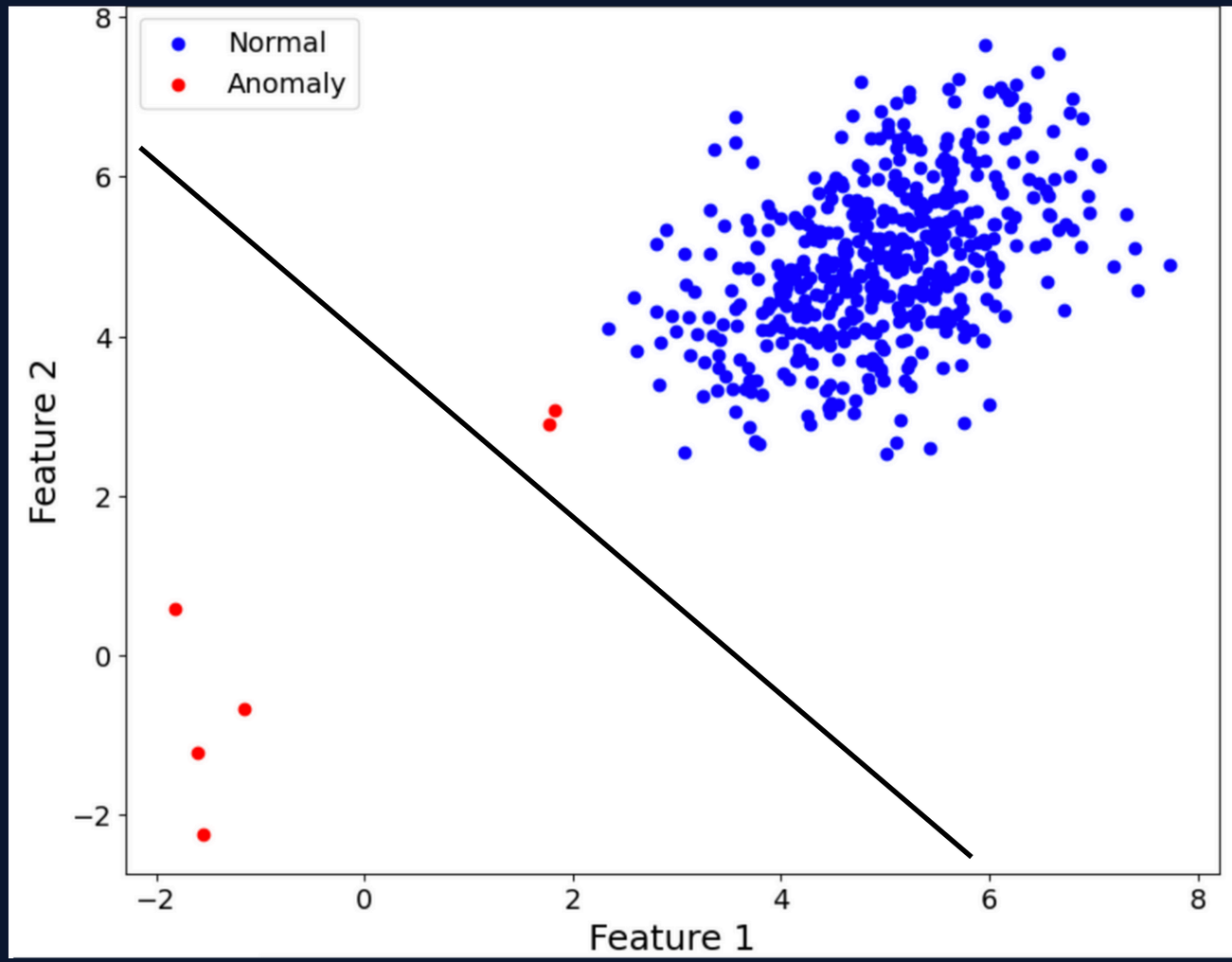


GOAL: Determine if Anomaly or Normal

FEATURE 1	FEATURE 2	Y
X11	X21	0
X12	X22	1
X1N	X2N	0

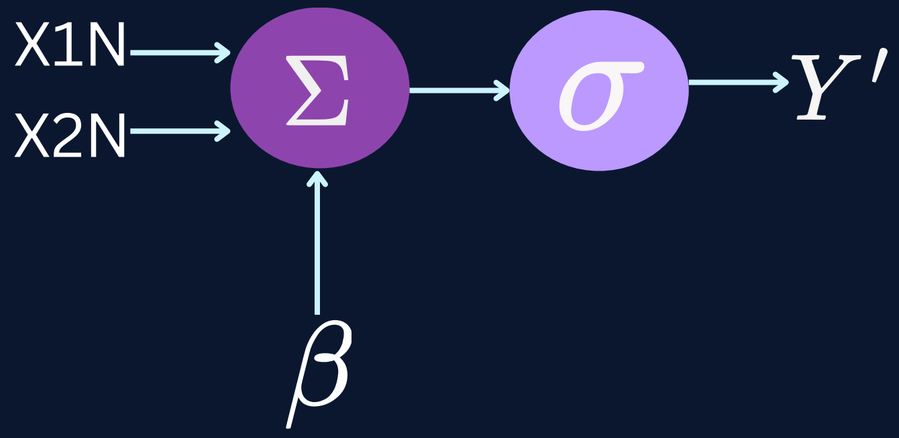
What if nonlinear?

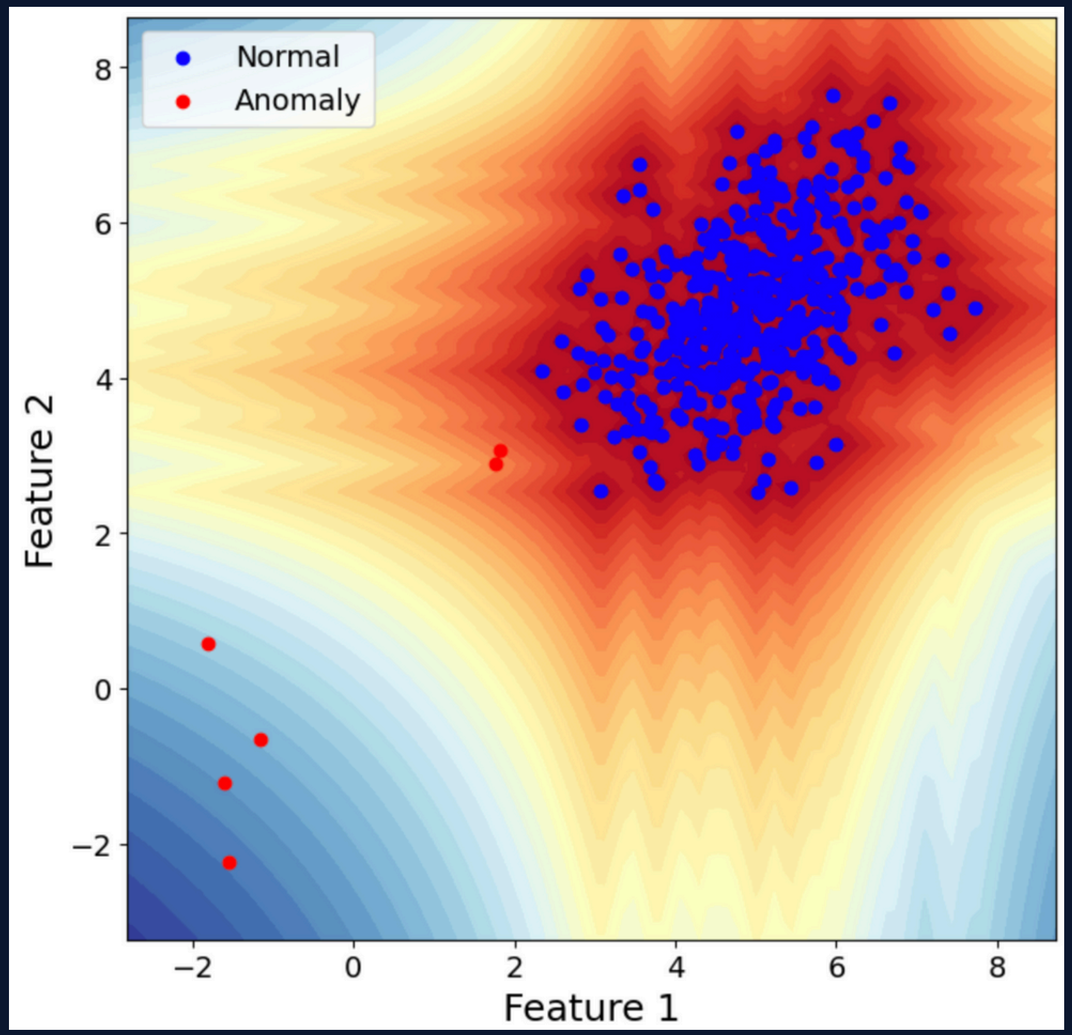




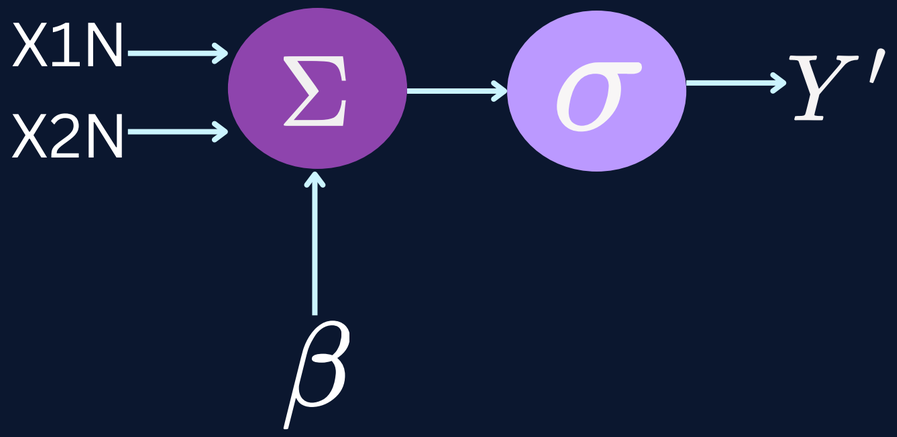
GOAL: Determine if Anomaly or Normal

FEATURE 1	FEATURE 2	Y
X11	X21	0
X12	X22	1
X1N	X2N	0

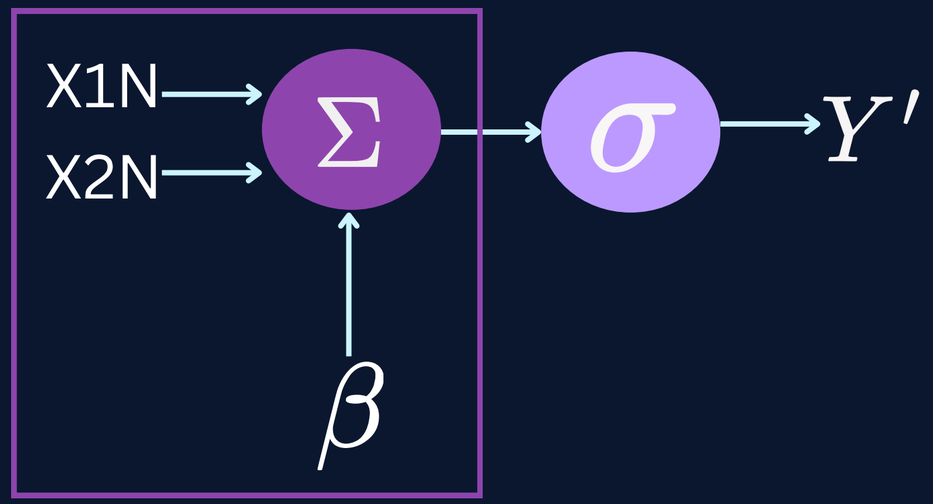


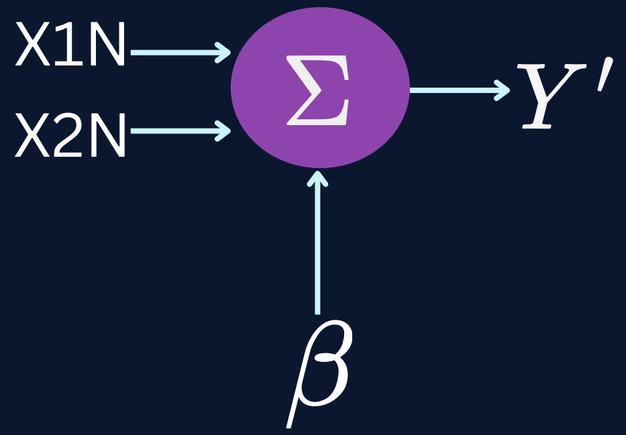


INTRODUCTION TO THE MATH BEHIND ANN



INTRODUCTION TO THE MATH BEHIND ANN





Data

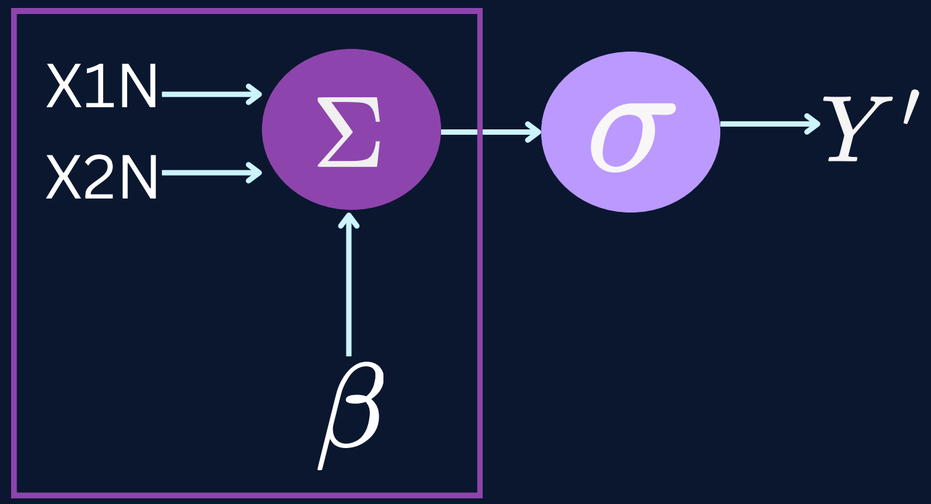
Prediction

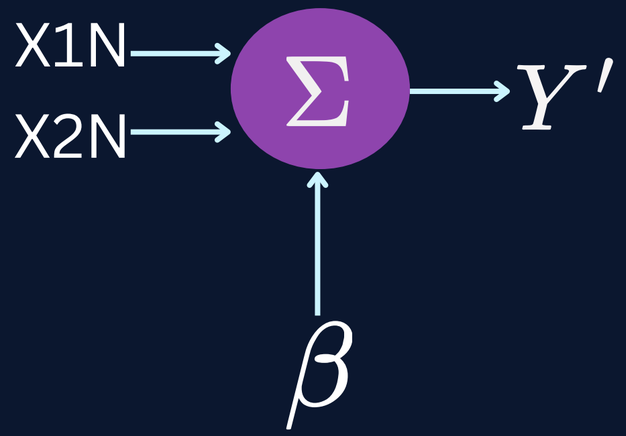
$$w_1(x_{1n}) + w_2(x_{2n}) + \beta = \hat{y}_n$$

Weights

Bias

INTRODUCTION TO THE MATH BEHIND ANN





Data

Prediction

$$w_1(x_{1n}) + w_2(x_{2n}) + \beta = \hat{y}_n$$

Weights

Bias

Question: What is the purpose of the bias?
(Hint: recall concepts in Linear Algebra)

Purpose of Bias:

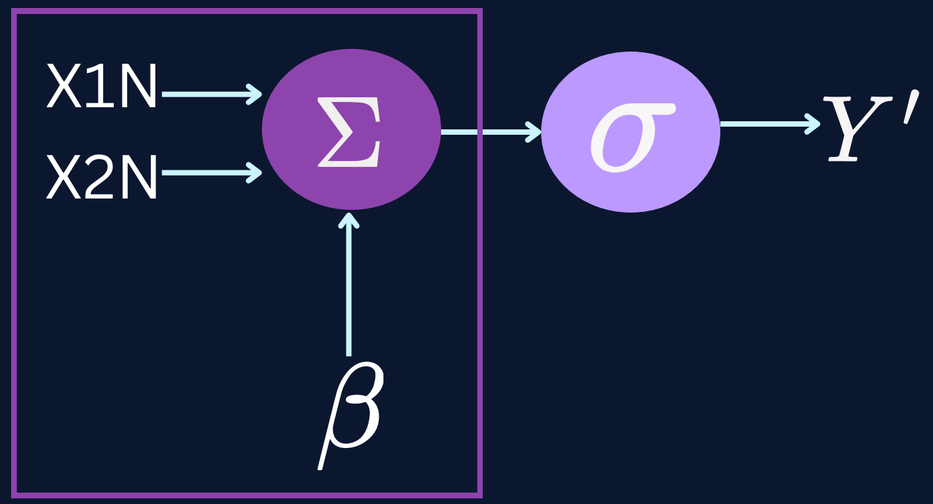
1. To provide each neuron a way to adapt to the output independent to the input.

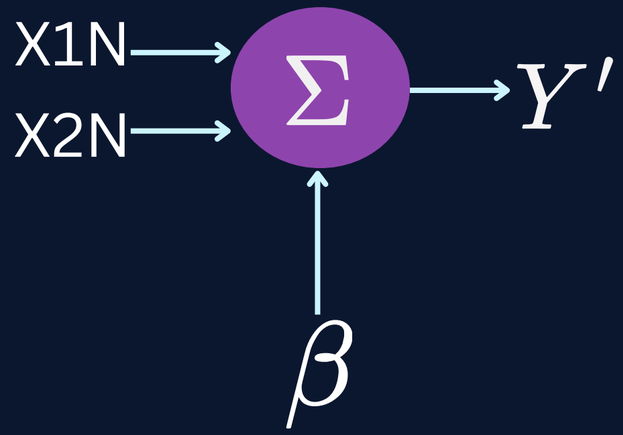
2. Bias allows each neuron to adjust the activation function threshold.

$$w_1(x_{1n}) + w_2(x_{2n}) + \beta = y_n \text{ vs. } w_1(x_{1n}) + w_2(x_{2n}) = y_n$$

3. Bias provides additional parameters allowing to learn more complicated tasks.

INTRODUCTION TO THE MATH BEHIND ANN





This entire equation is linear. What do we need to do to solve non-linear tasks?

Data

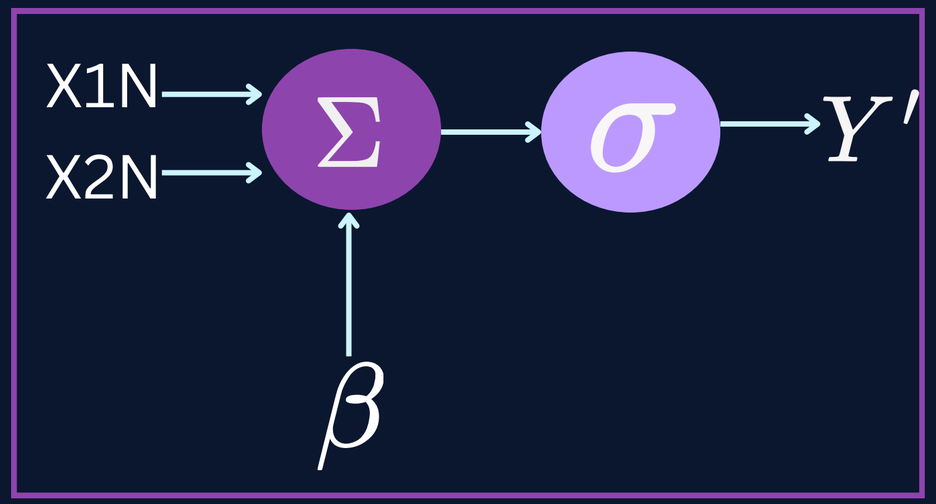
Prediction

$$w_1(x_{1n}) + w_2(x_{2n}) + \beta = \hat{y}_n$$

Weights

Bias

INTRODUCTION TO THE MATH BEHIND ANN



Data

Prediction

Non-linear function $\sigma\left(w_1(x_{1n}) + w_2\left(x_{2n}\right) + \beta\right) = \hat{y}_n$

Weights

Bias

INTRODUCTION TO THE MATH BEHIND ANN

$$\sigma_1 \left(w_{11}(x_{1n}) + w_{21} \left(x_{2n} \right) + \beta_1 \right) = y_1 \longrightarrow$$ 1st Neuron

$$\sigma_2 \left(w_{12}(x_{1n}) + w_{22} \left(x_{2n} \right) + \beta_2 \right) = y_2 \longrightarrow$$ 2nd Neuron

$$\sigma_3 \left(w_{13}(x_{1n}) + w_{23} \left(x_{2n} \right) + \beta_3 \right) = y_3 \longrightarrow$$ 3rd Neuron

.

.

.

$$\sigma_m \left(w_{1m}(x_{1n}) + w_{2m} \left(x_{2n} \right) + \beta_m \right) = y_m \longrightarrow$$ nth Neuron

Nomenclature:

$w_{km} \longrightarrow$ Weight of feature k going into node m

$x_{kn} \longrightarrow$ nth data of feature k

INTRODUCTION TO THE MATH BEHIND ANN

$$\sigma_1\left(w_{11}(x_{1n}) + w_{12}\left(x_{2n}\right) + \beta_1\right) = y_n \longrightarrow \sigma_1(x^T W + \beta_1)$$

$$\sigma_2\left(w_{12}(x_{1n}) + w_{22}\left(x_{2n}\right) + \beta_2\right) = y_2 \longrightarrow \sigma_2(x^T W + \beta_2)$$

$$\sigma_3\left(w_{31}(x_{1n}) + w_{32}\left(x_{2n}\right) + \beta_3\right) = y_n \longrightarrow \sigma_3(x^T W + \beta_3)$$

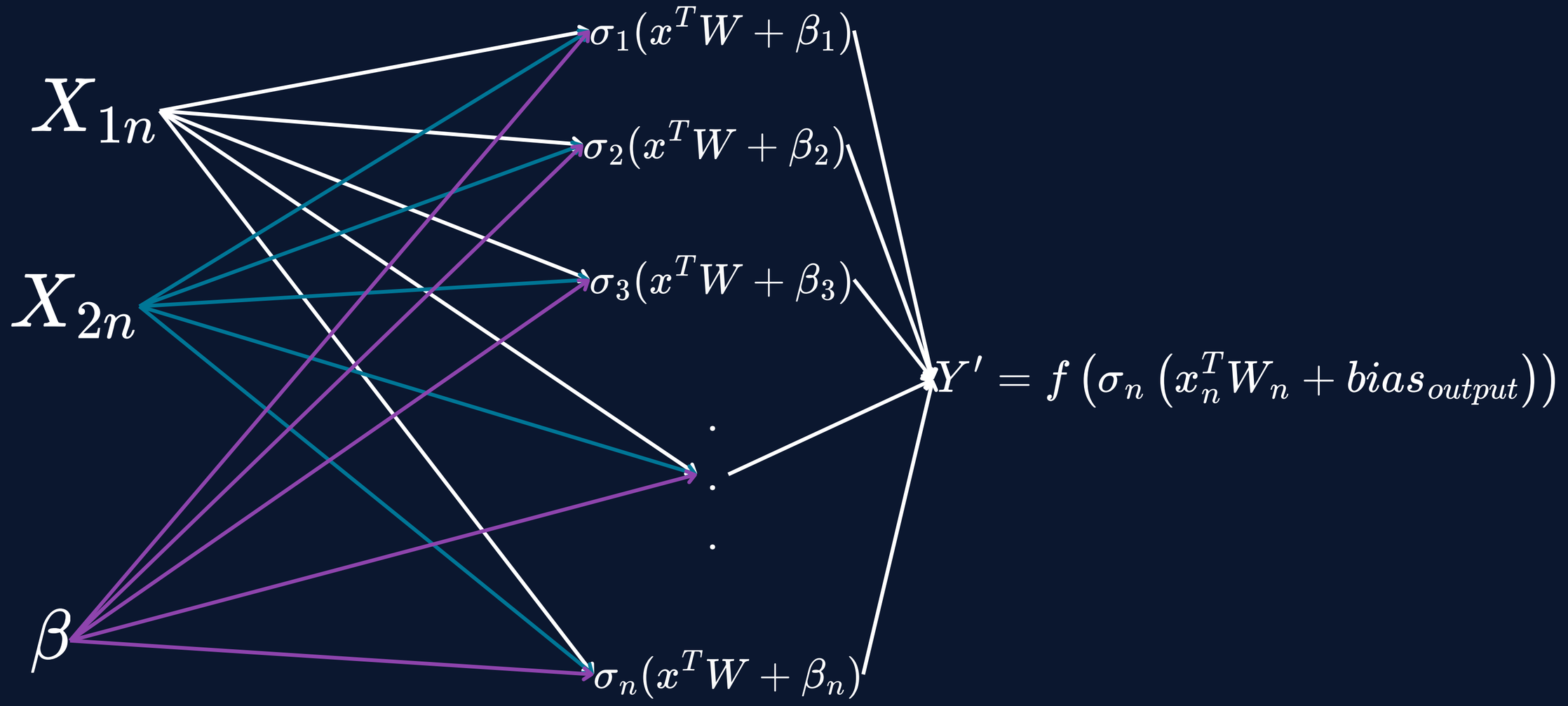
.

.

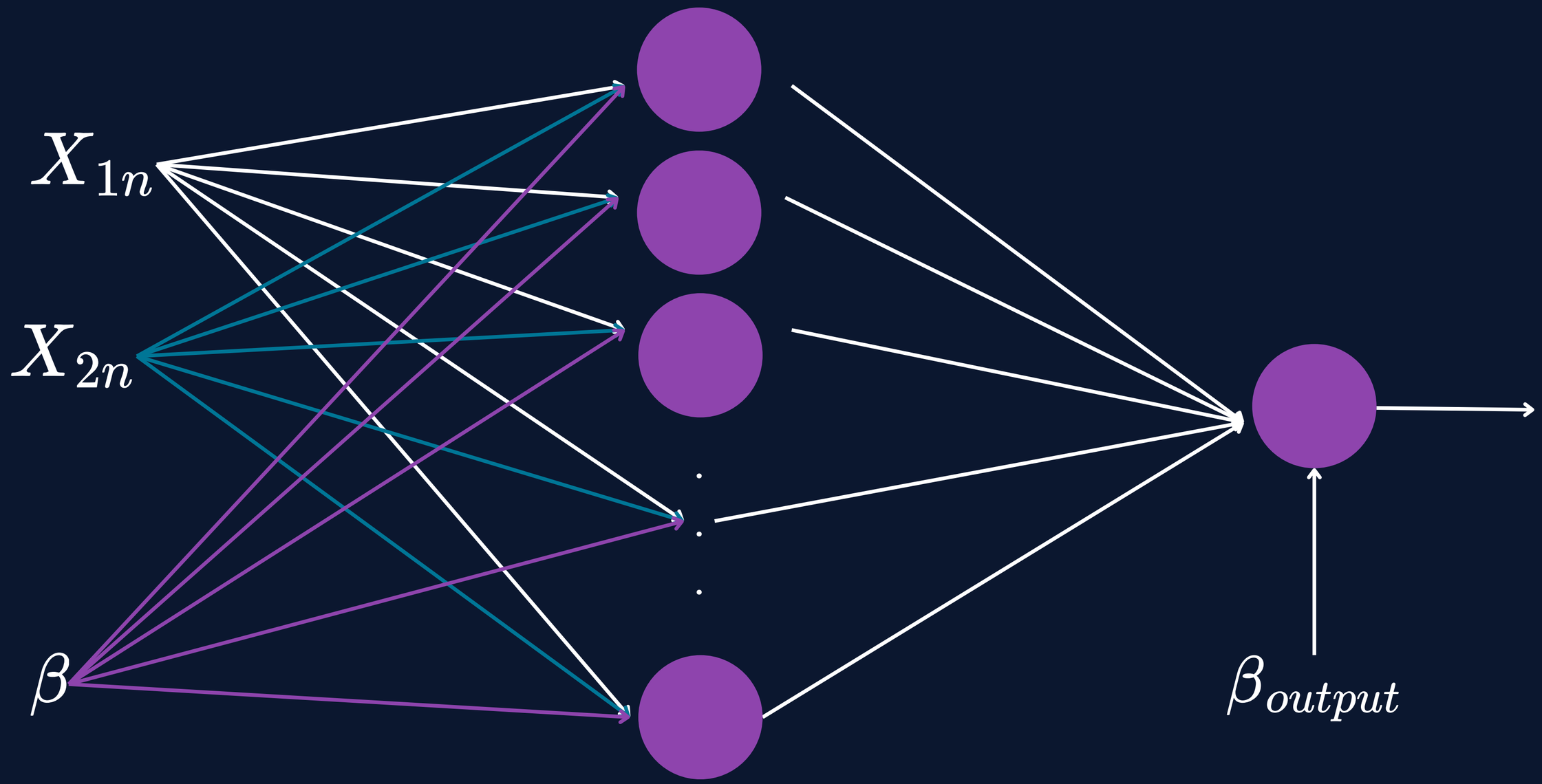
.

$$\sigma_m\left(w_{1m}(x_{1n}) + w_{2m}\left(x_{2n}\right) + \beta_m\right) = y_m \longrightarrow \sigma_3(x^T W + \beta_3)$$

INTRODUCTION TO THE MATH BEHIND ANN



INTRODUCTION TO THE MATH BEHIND ANN

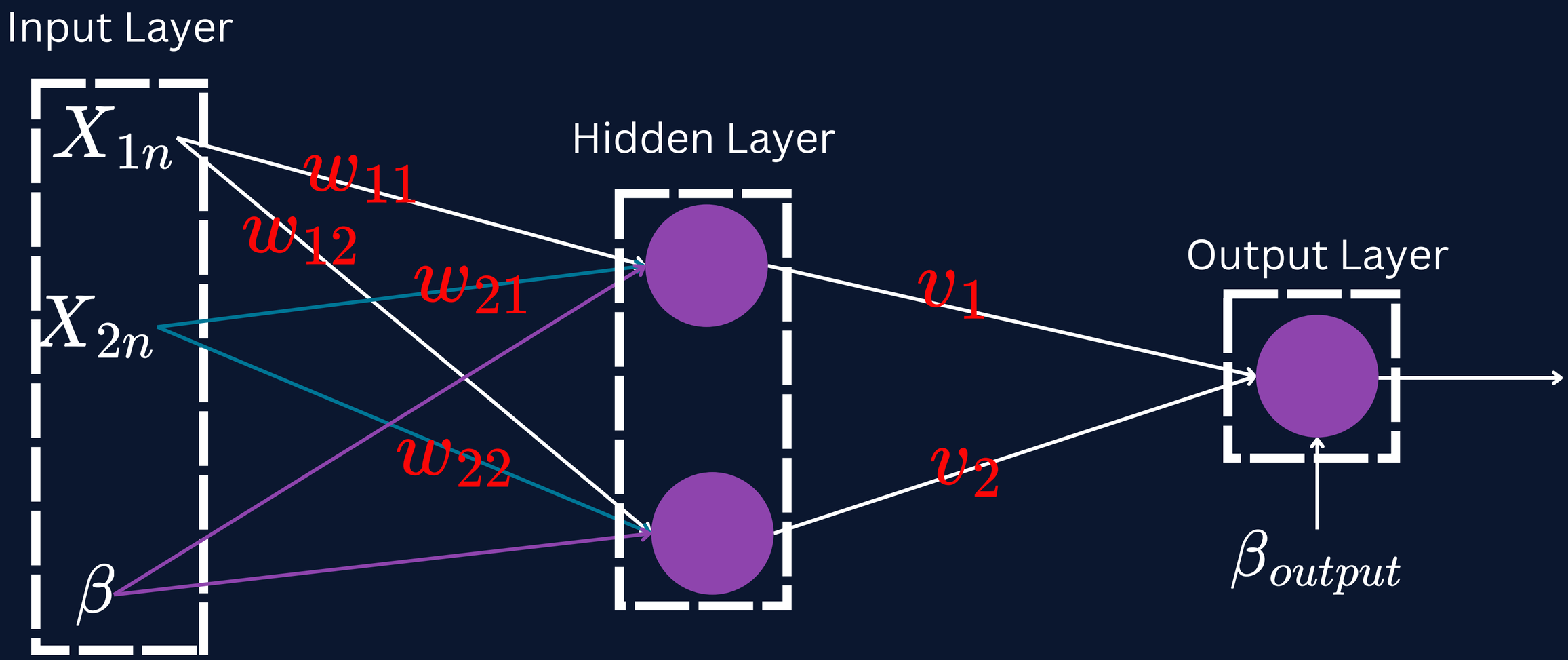


Data	Method
Tabular Regression	ANN
Tabular Classification	ANN
Image Classification and Detection	CNN, U-NET
Time Series	RNN, GRU, LSTM
Languages	RNN, GRU,LSTM
Image Generation	GAN
Molecule Generation	GNN

FORWARD PROPOGATION

The process of passing input data through the layers of a neural network to generate an output.

FORWARD PROPAGATION



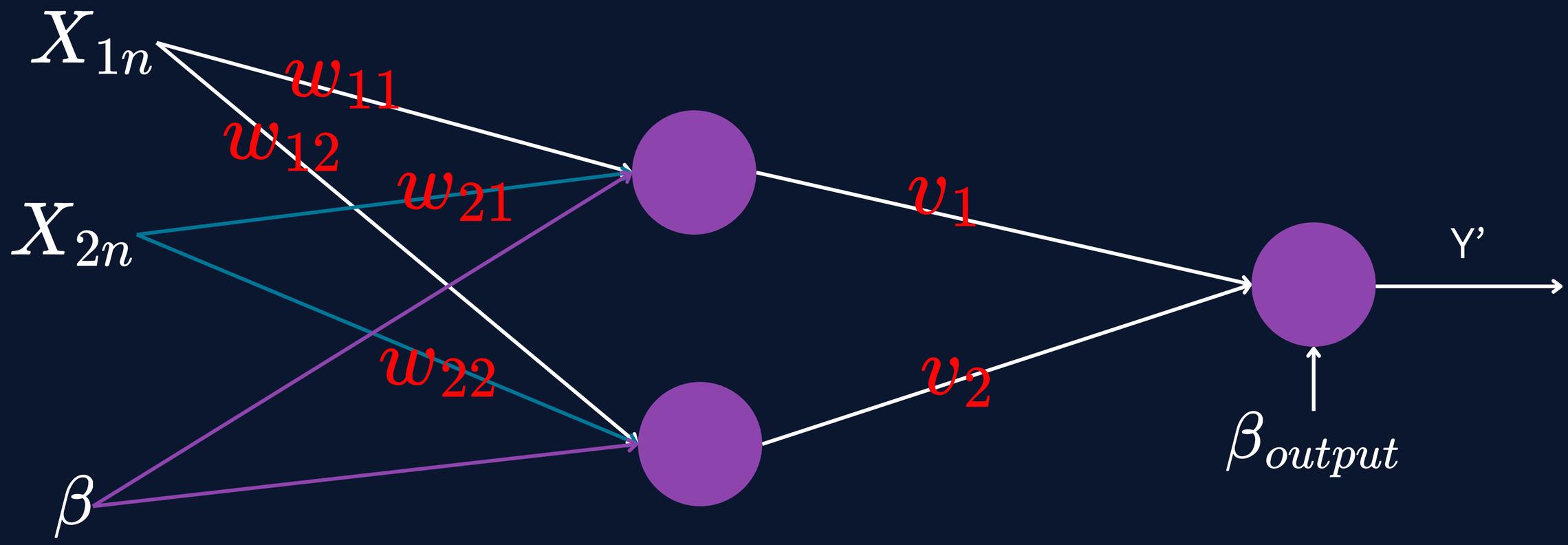
INTRODUCTION TO THE MATH BEHIND ANN

$$\sigma_1\left(w_{11}(x_{1n}) + w_{12}\left(x_{2n}\right) + \beta_1\right) = y_1 \longrightarrow$$ 1st Neuron

$$\sigma_2\left(w_{12}(x_{1n}) + w_{22}\left(x_{2n}\right) + \beta_2\right) = y_2 \longrightarrow$$ 2nd Neuron

$$Y' = \sigma_{output}\left(v_1 . y_1 + v_2 . y_2 + \beta_{output}\right)$$

FORWARD PROPAGATION



BACKWARD PROPOGATION

Backward propagation is the process of adjusting weights and biases to minimize the error between the predicted and actual outputs.

Step 1: Choose an appropriate loss function, L

$$L = \frac{1}{2}\left(Y - Y'\right)^2$$

Step 2: Optimize the loss function, L, with respect to the predicted output, Y’

$$\frac{\delta L}{\delta Y'} = \frac{\delta\left(\frac{1}{2}(Y - Y')^2\right)}{\delta Y'} = \frac{2}{2}\left(Y - Y'\right)^{2-1}\frac{\delta Y'}{\delta Y'} = Y - Y'$$

Step 3: Find the derivative of the activation function

$$d\left(\sigma_{output}\right) = \sigma'_{output}$$

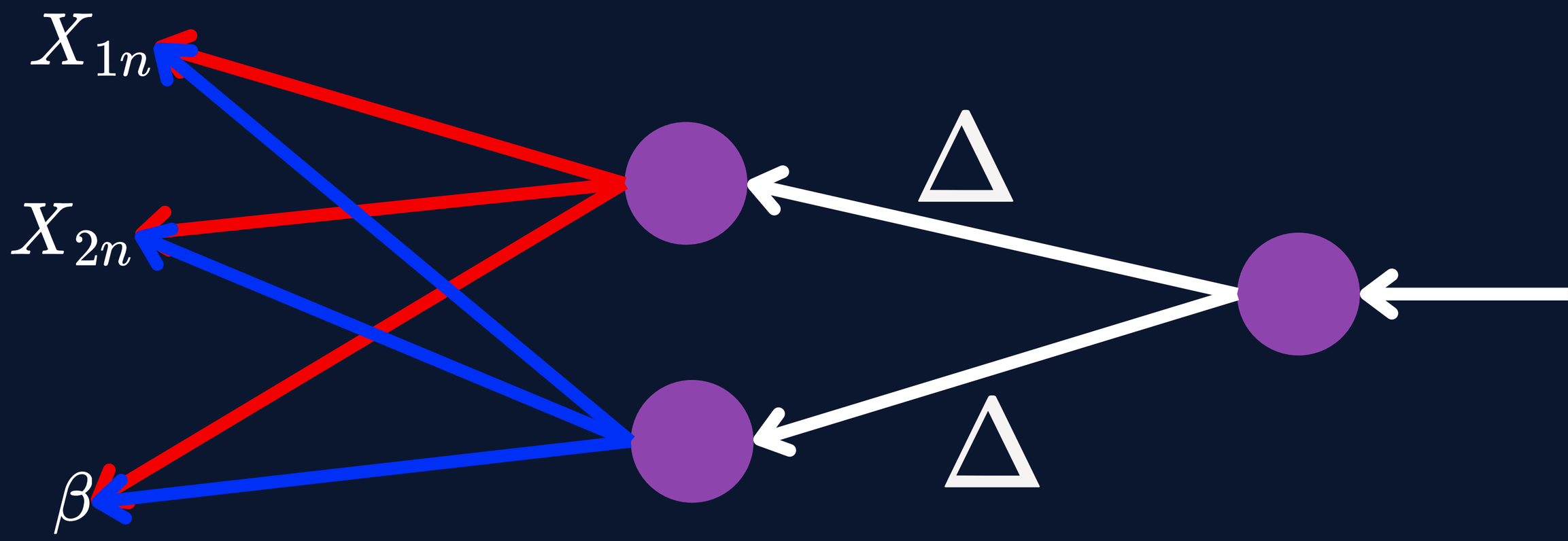
Step 4: Find the derivative of the value of the activation function given output.

$$\sigma'_{output}\left(Y'\right)$$

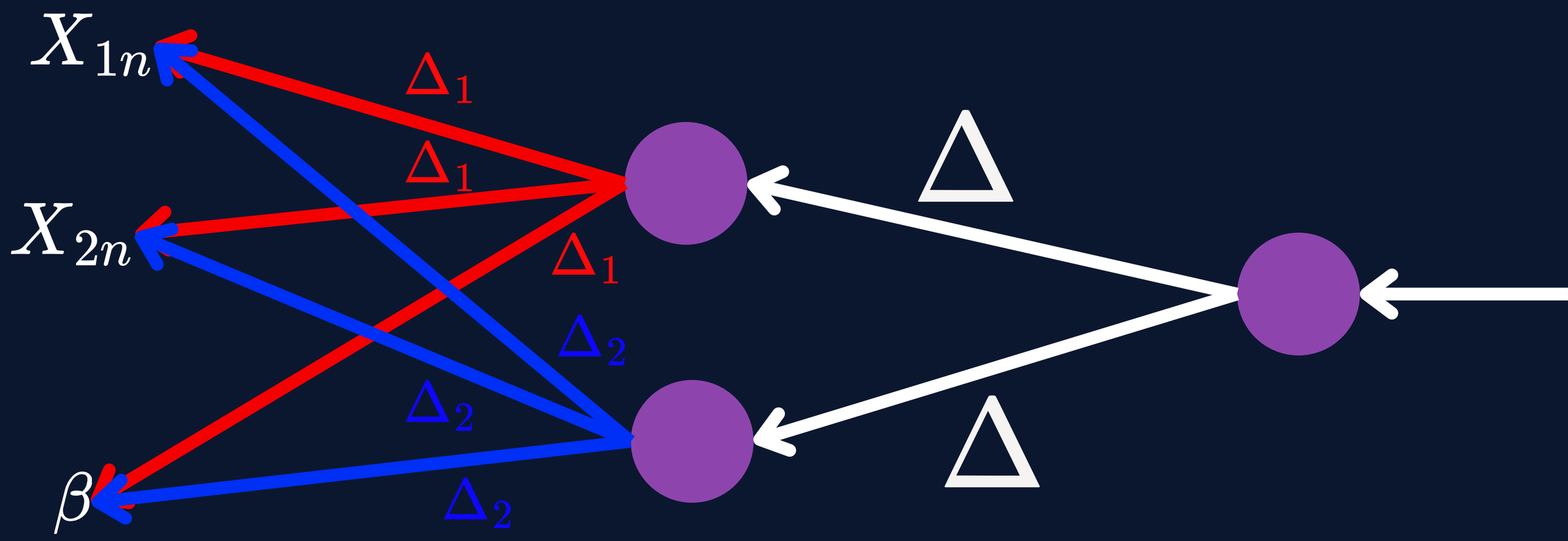
Step 5: Calculate the Gradient Loss

$$\Delta = \frac{\delta L}{\delta Y'}\sigma'_{output}\left(Y'\right)$$

BACK PROPAGATION



BACK PROPAGATION



Step 6: Propagate the Gradient Loss back to the hidden layers

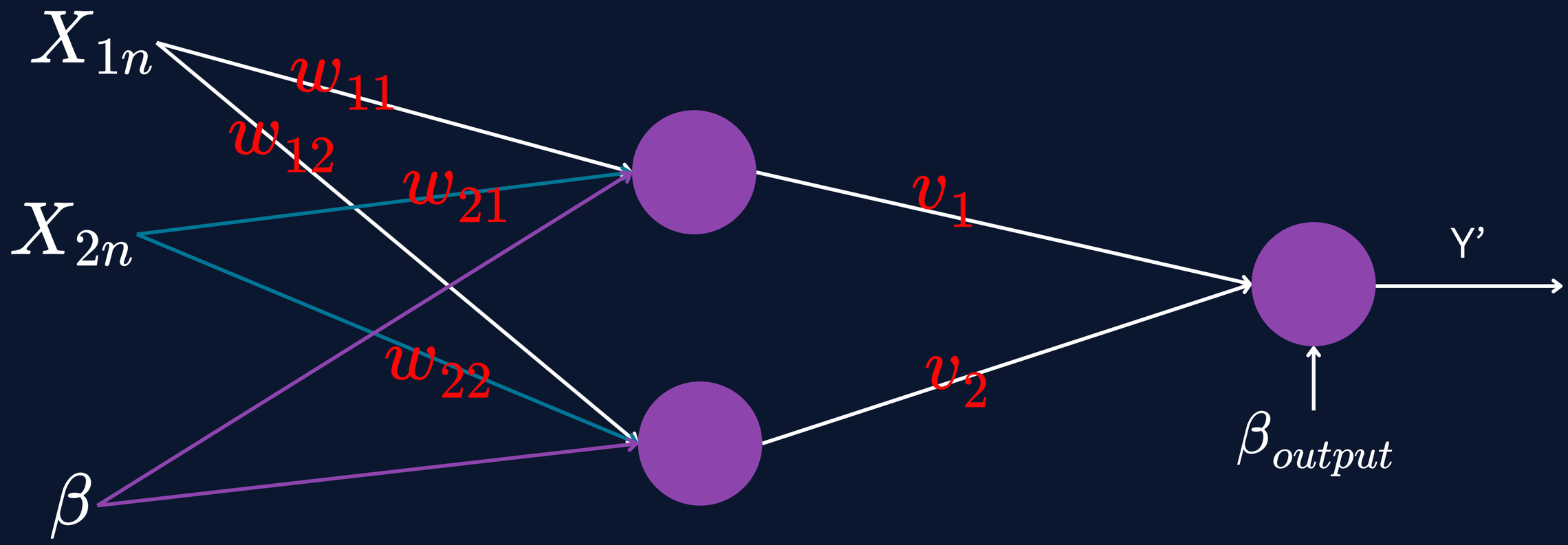
$$\Delta_1 = \Delta\,(w_{11} + w_{21}).\,\sigma_1'\,(w_{11}(x_{1n}) + w_{21}\,(x_{2n}) + \beta_1)$$
$$\Delta_2 = \Delta\,(w_{12} + w_{22}).\,\sigma_2'\,(w_{12}(x_{1n}) + w_{22}\,(x_{2n}) + \beta_2)$$

Step 7: Use the learning rate, LR , to update the weights and biases.

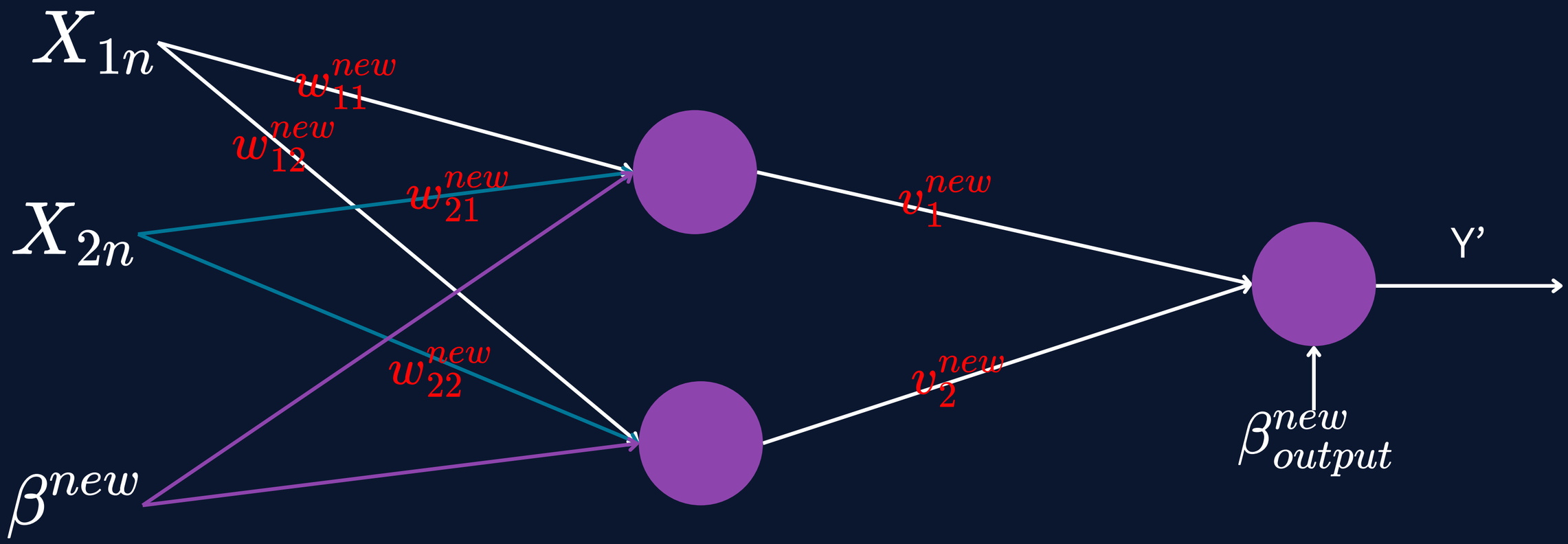
For weights and biases between the hidden and output layers.

$$v_1^{new} = v_1 - (LR \times \Delta \times y_1)$$
$$v_2^{new} = v_2 - (LR \times \Delta \times y_2)$$
$$\beta_{output}^{new} = \beta_{output} - (LR \times \Delta)$$

REACLL: FORWARD PROPAGATION



REACLL: FORWARD PROPAGATION



Step 8: Use the learning rate, LR , to update the weights and biases.

For weights and biases between the input and hidden layers.

$$w_{11}^{new} = w_{11} - (LR \times \Delta_1 \times x_{1n})$$

$$w_{21}^{new} = w_{21} - (LR \times \Delta_1 \times x_{2n})$$

$$w_{12}^{new} = w_{12} - (LR \times \Delta_2 \times x_{1n})$$

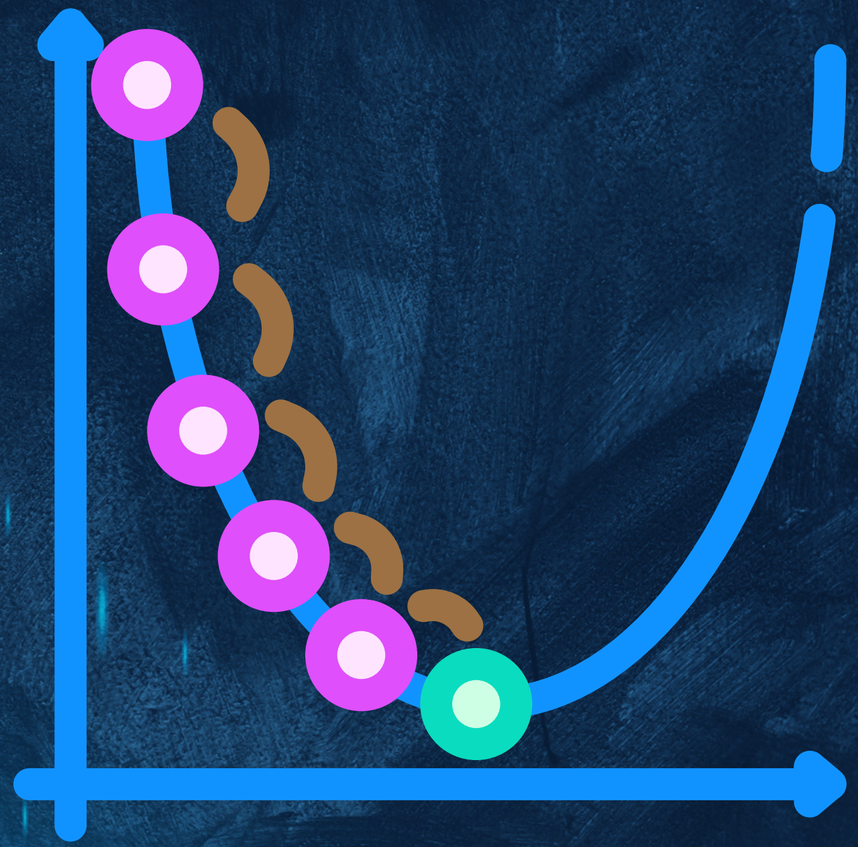
$$w_{22}^{new} = w_{22} - (LR \times \Delta_2 \times x_{2n})$$

$$\beta_1^{new} = \beta_1 - (LR \times \Delta_1)$$

$$\beta_2^{new} = \beta_2 - (LR \times \Delta_2)$$

GRADIENT DESCENT

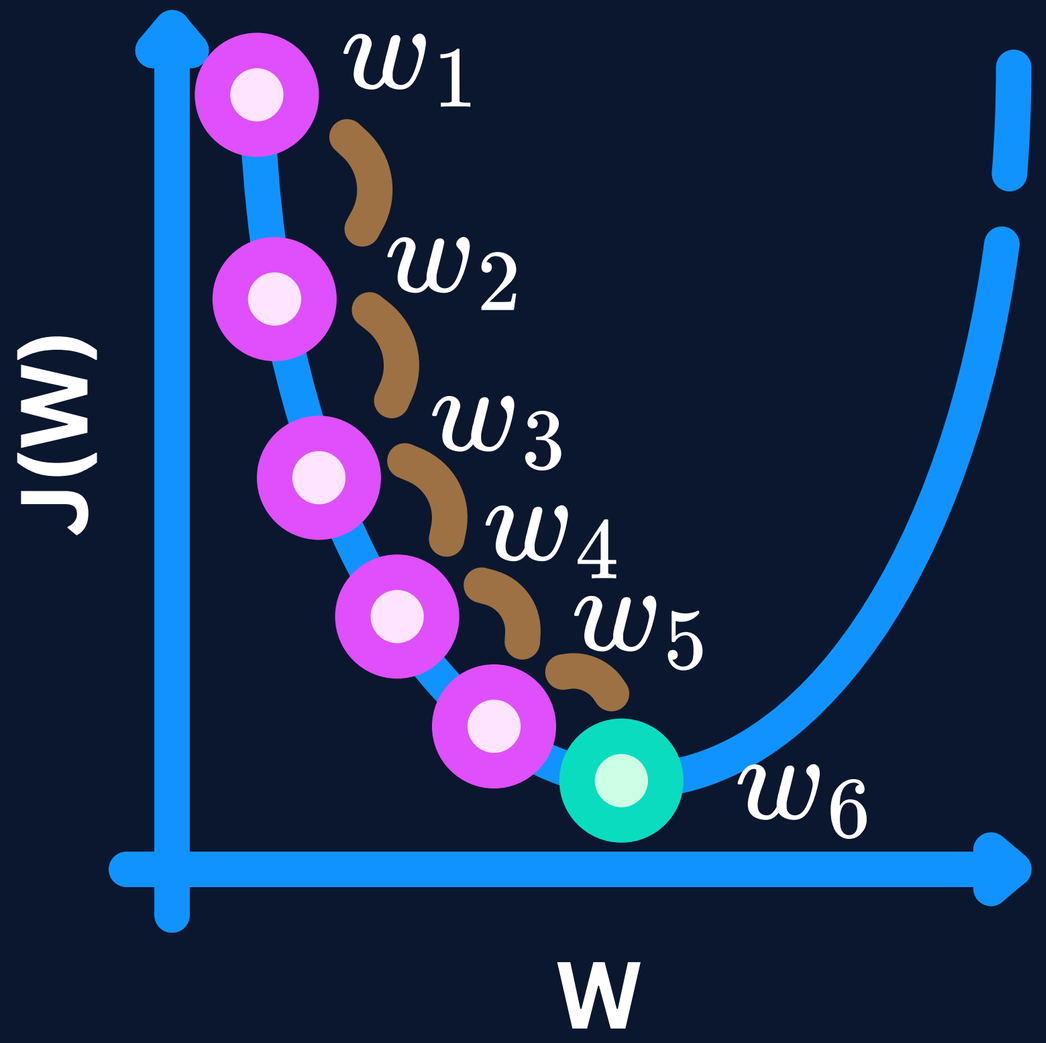
An optimization algorithm used to minimize the loss function in machine learning models. It works by iteratively adjusting the model parameters (weights and biases) to find the values that minimize the loss function. The basic idea is to take steps proportional to the negative of the gradient (partial derivatives) of the loss function with respect to the parameters.



How does Deep Learning Learn?

1. Create an initial value for weights and biases.
2. Using the initial values, the algorithm predicts the output.
3. The loss function is calculated.
4. Gradient loss is back propagated to adjust weights and biases.
5. Using the new values of the weights and biases the algorithm again tries to predict a new set of values for y.
6. Repeat step 3 to 5 until loss function is minimized.

Gradient Descent



JUPYTERHUB DISCUSSION

ISSUES OF LOCAL MINIMA

GRADIENT DESCENT

- Ensures to find the lowest values in the error space.

WRONG INITIALIZATION

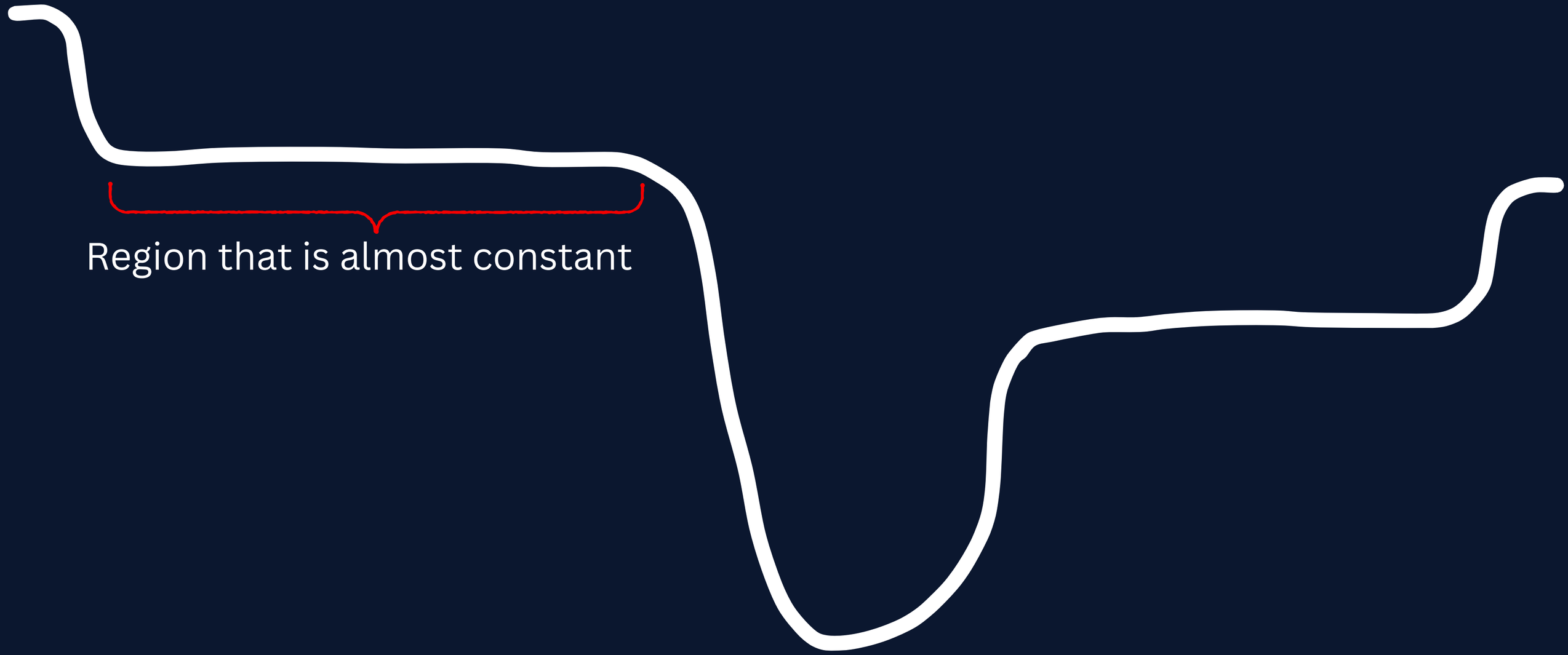
- A wrong initial parameter will not lead to the optimal solution.
- Stuck in the local minima.
- Gradient descent may not find the best optimum due to local minima.

TROUBLESHOOTING

- Training models many times with different random seeds.
- Increasing dimensionality (increasing complexity/parameters) to create few local minima.

Vanishing Gradient

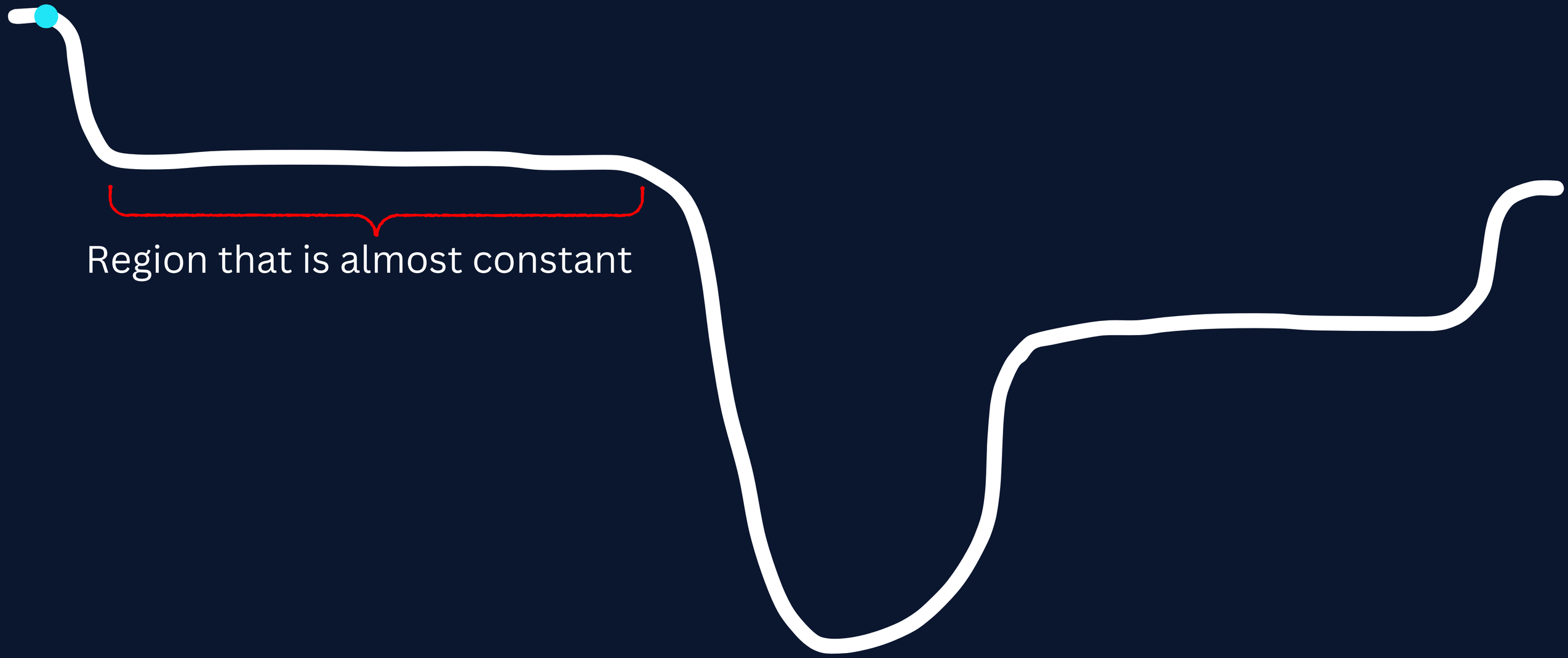
Example: Consider a function, f(x), with the following plot



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

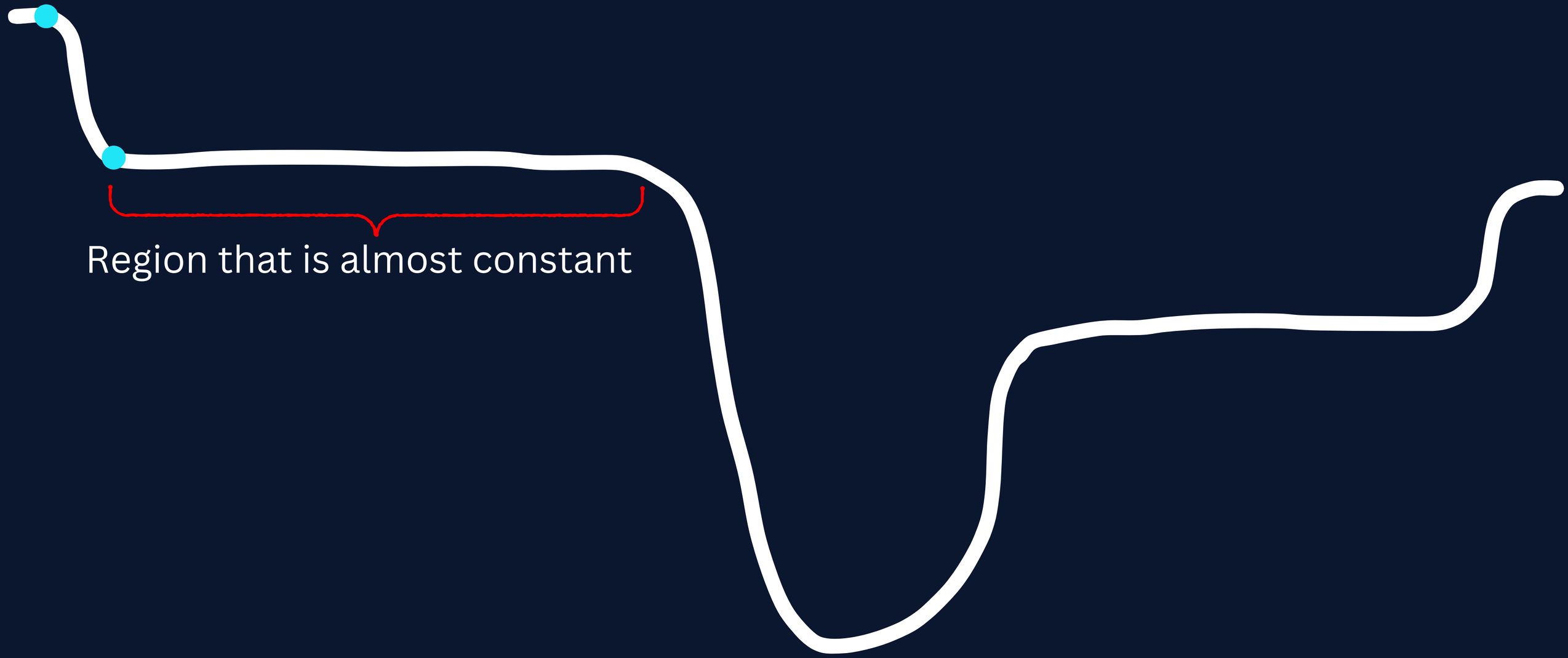
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

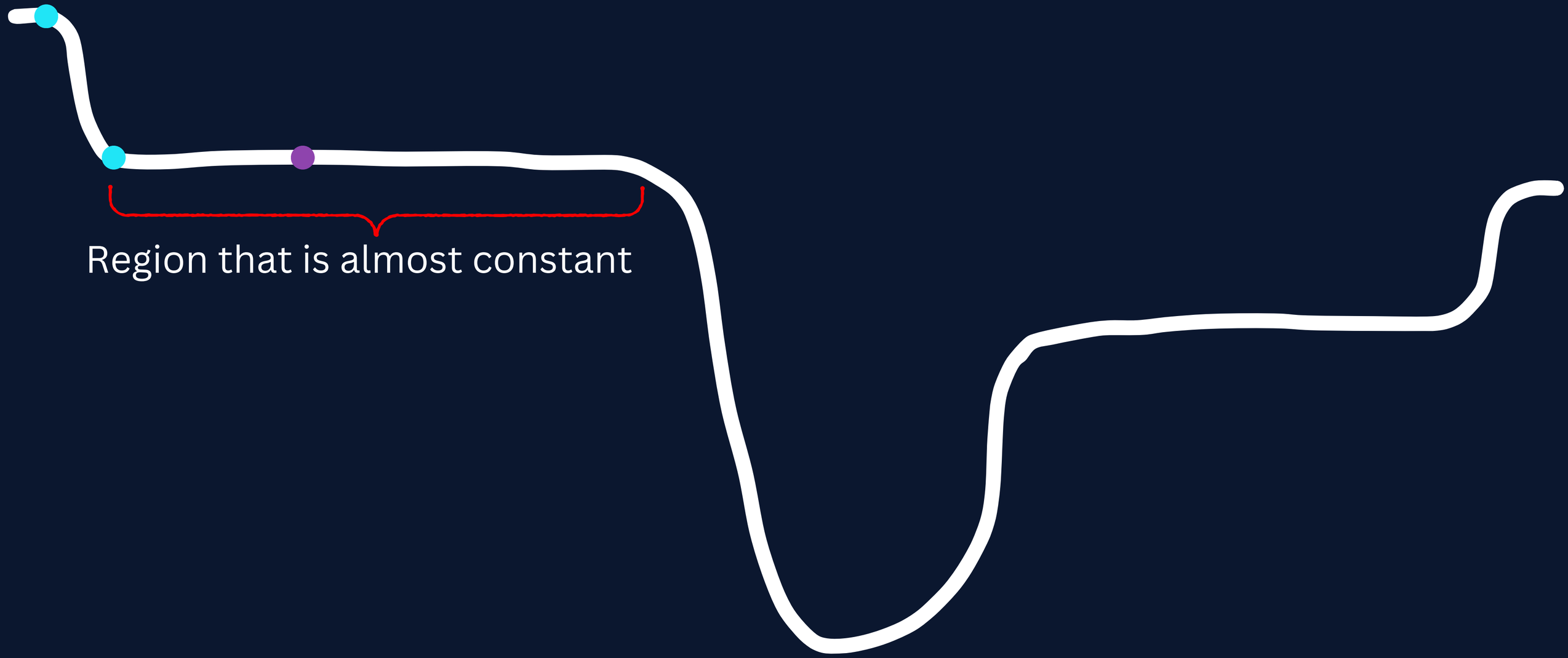
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

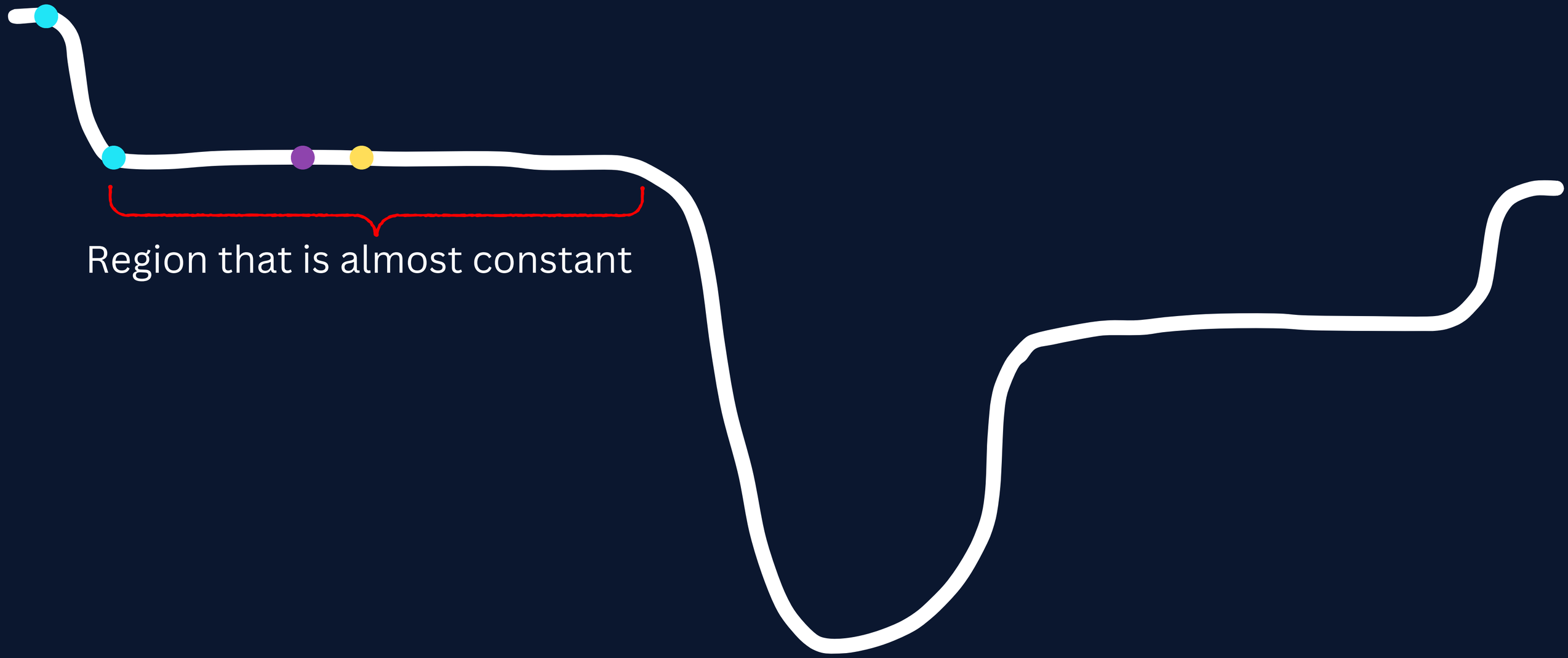
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

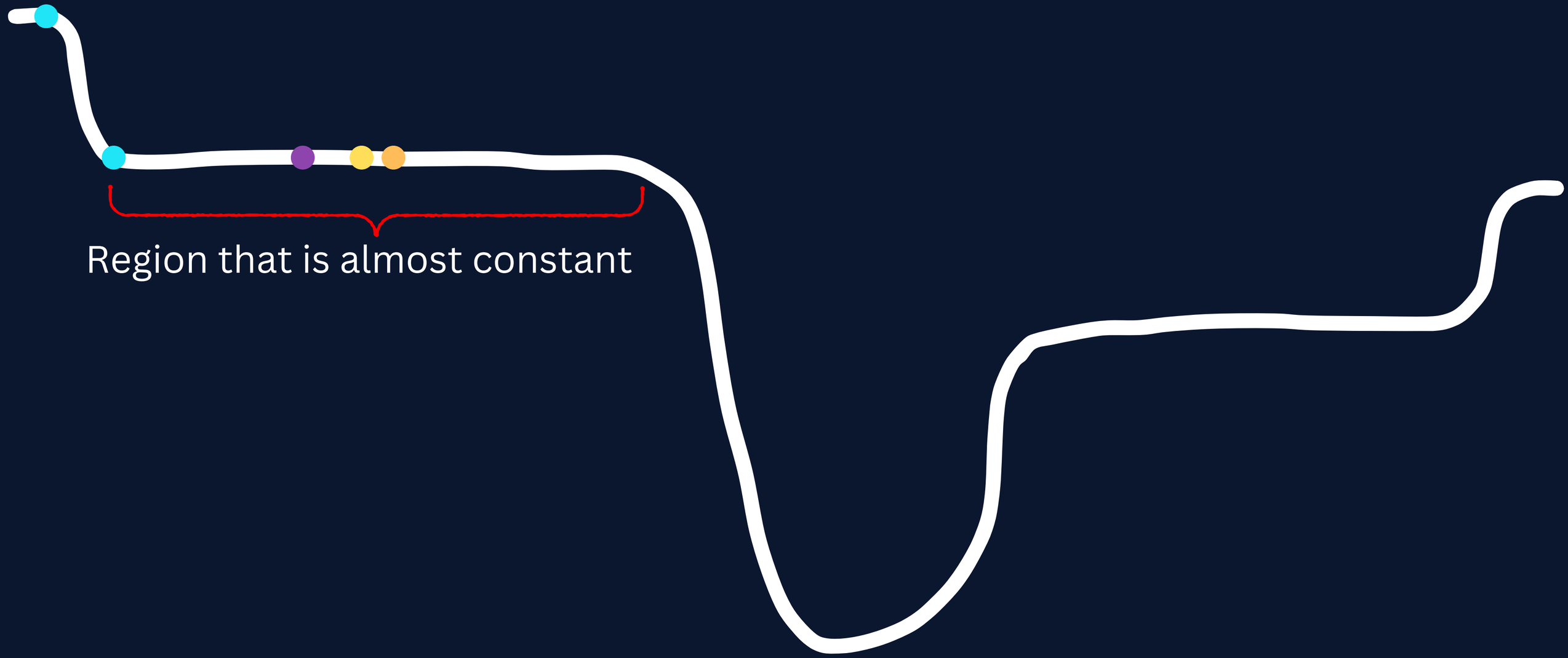
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

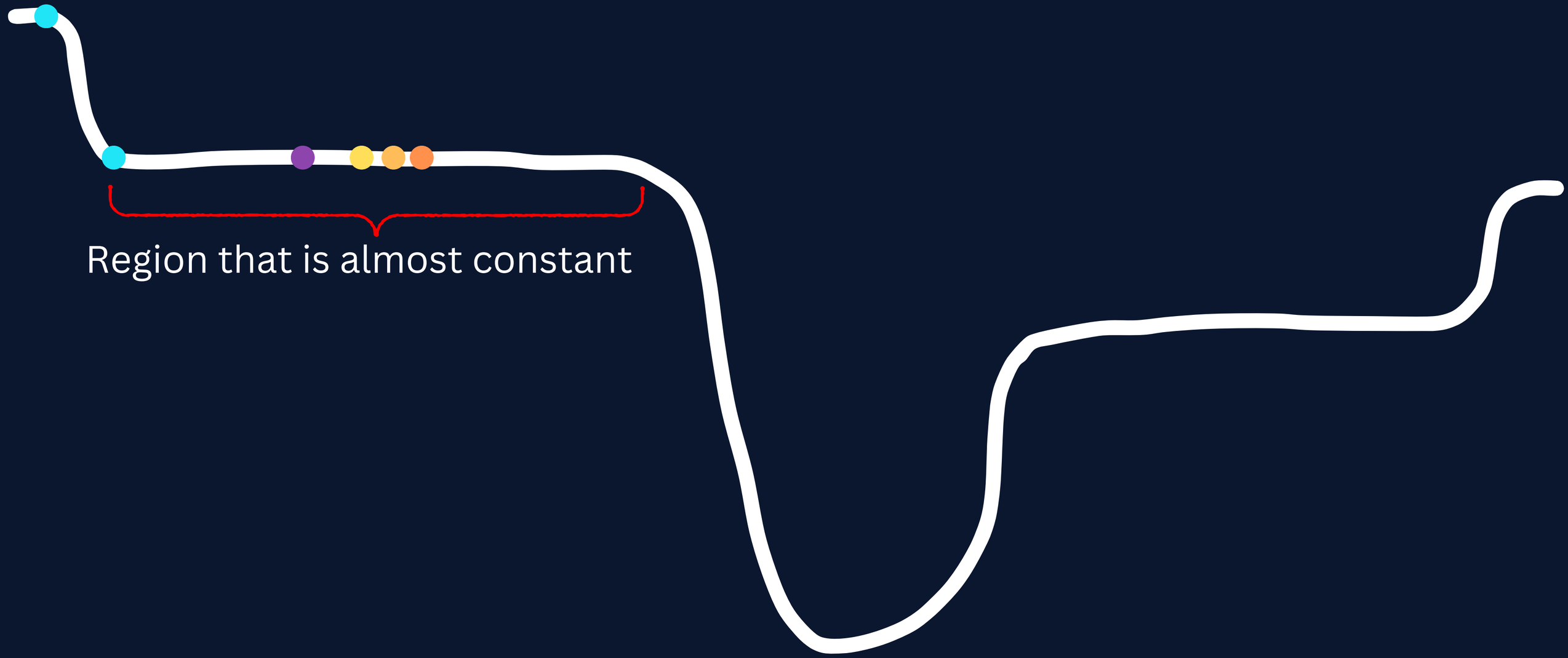
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

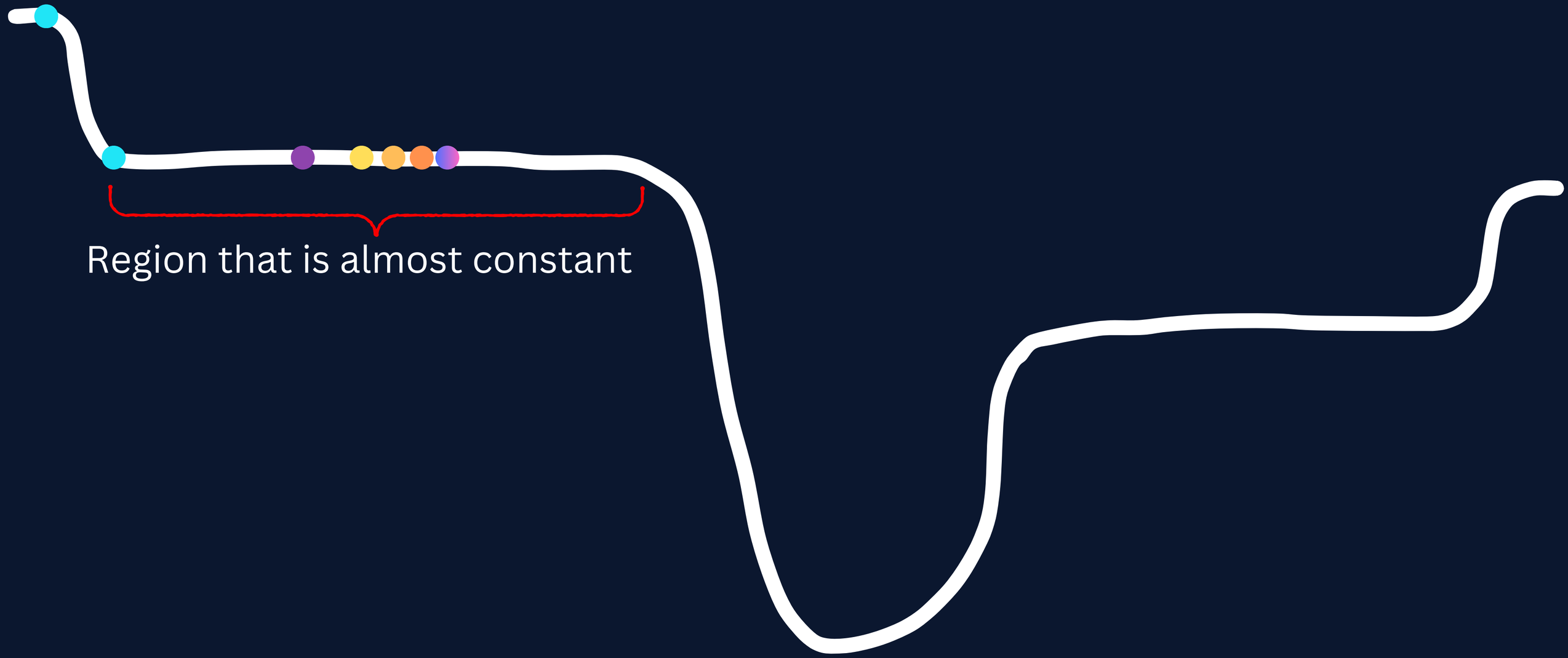
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

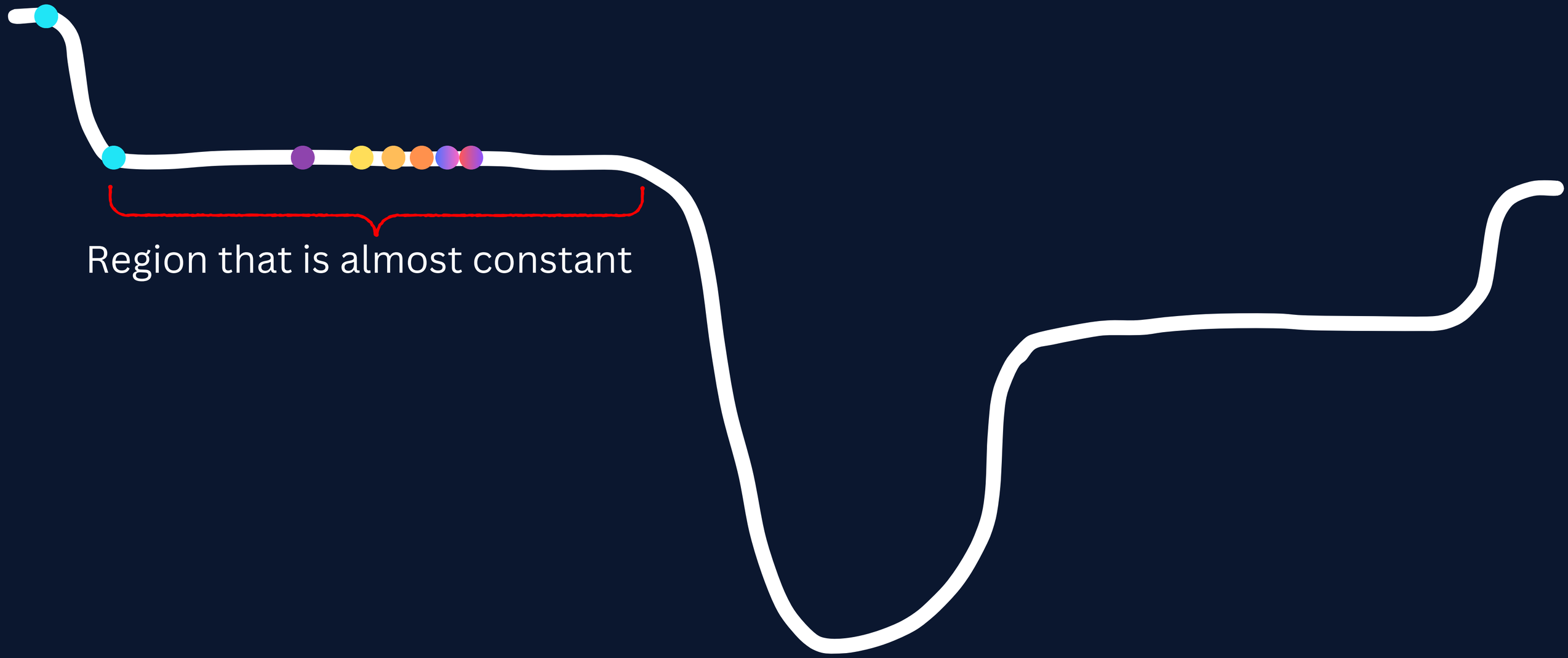
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

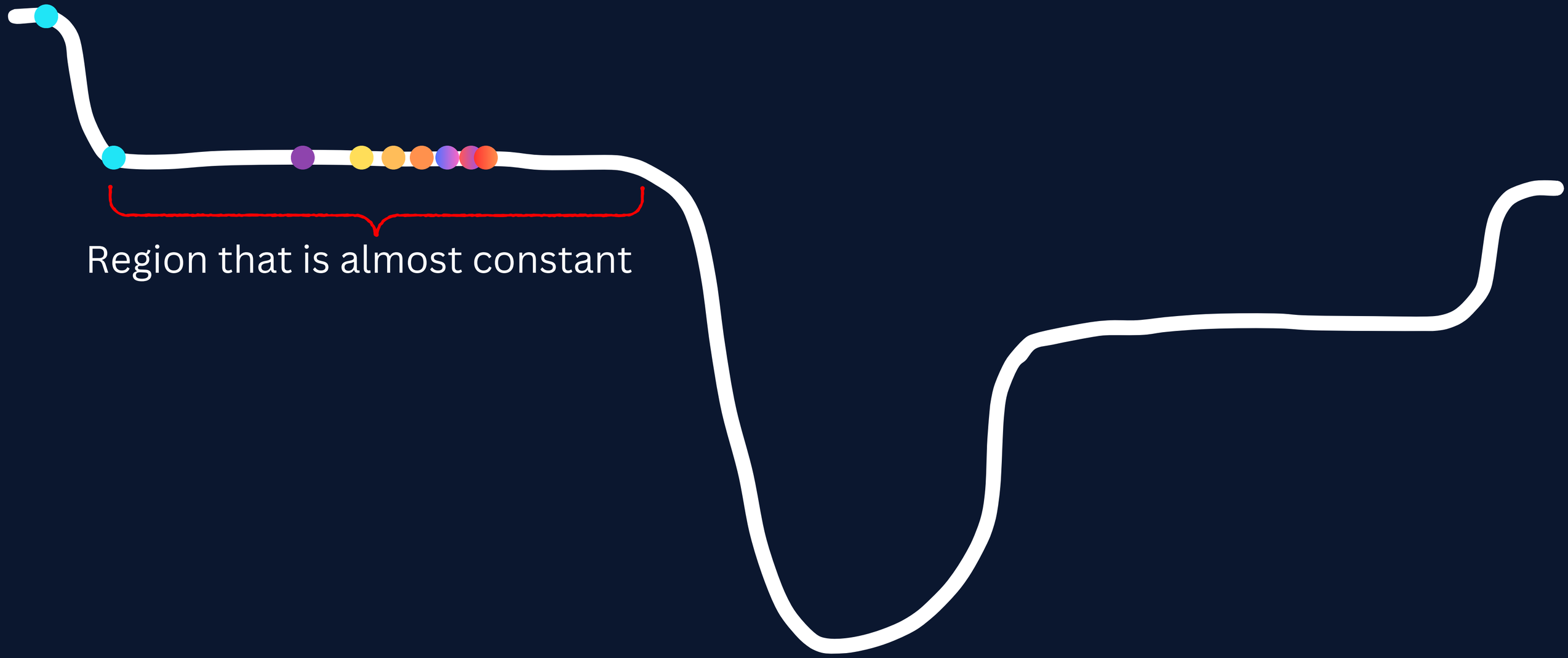
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

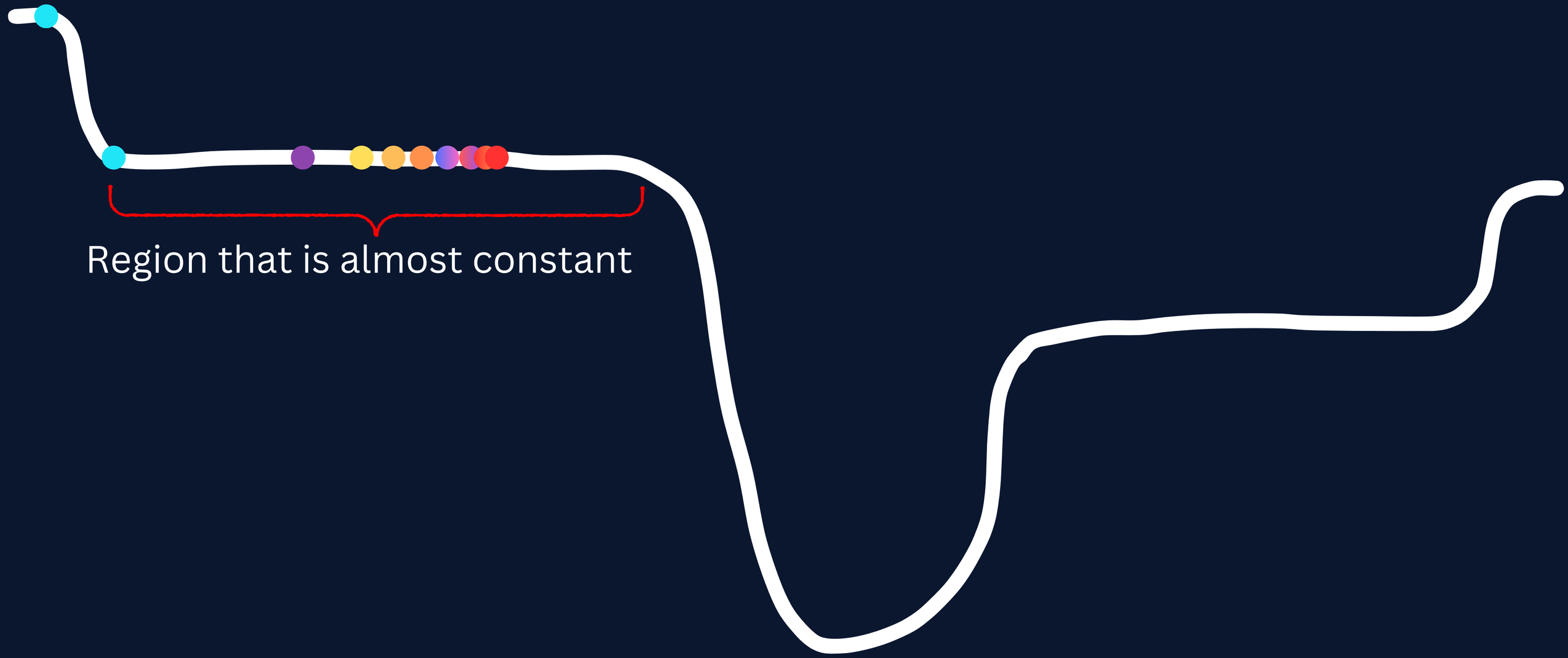
Initial value



Vanishing Gradient

Example: Consider a function, f(x), with the following plot

Initial value



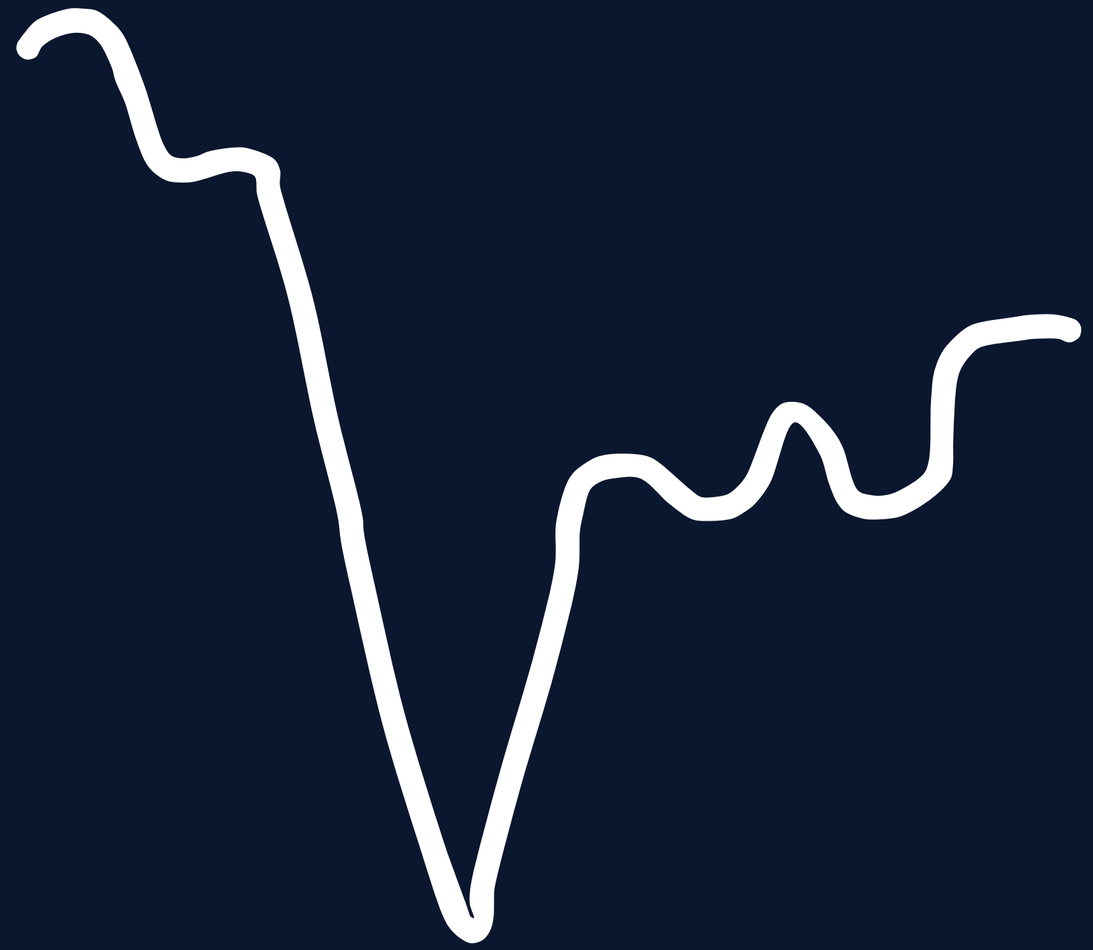
Vanishing Gradient

Example: Consider a function, f(x), with the following plot

Initial value

Exploding Gradient

Example: Consider a function, f(x), with the following plot

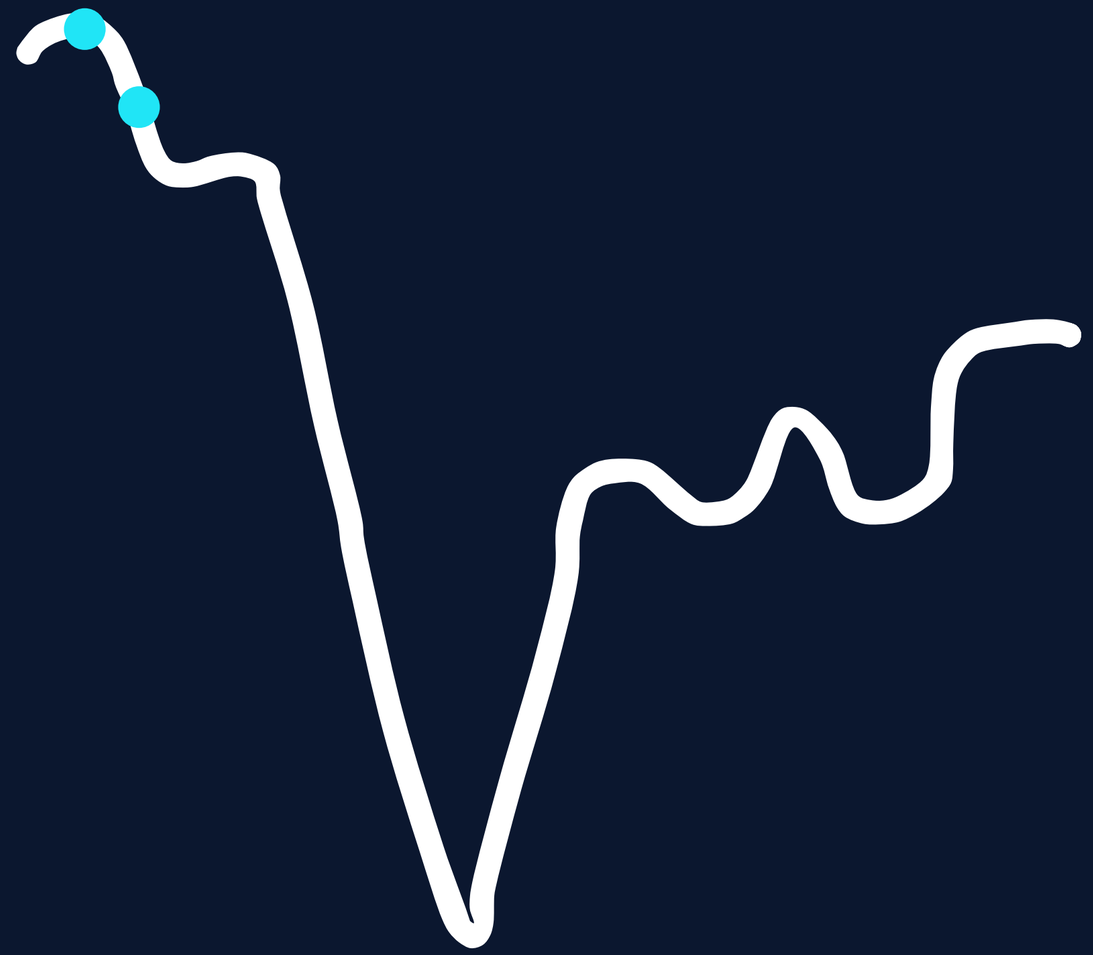


Exploding Gradient

Example: Consider a function, f(x), with the following plot

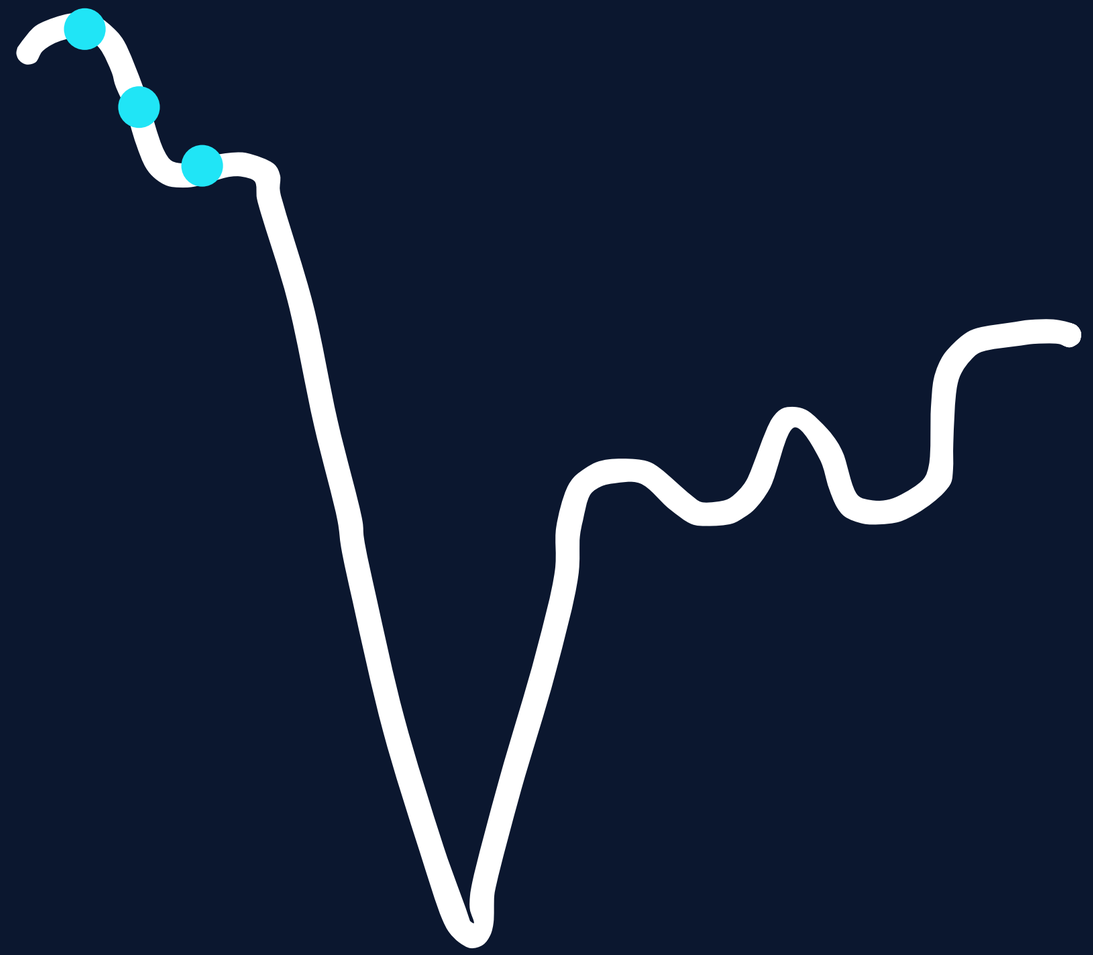
Exploding Gradient

Example: Consider a function, f(x), with the following plot



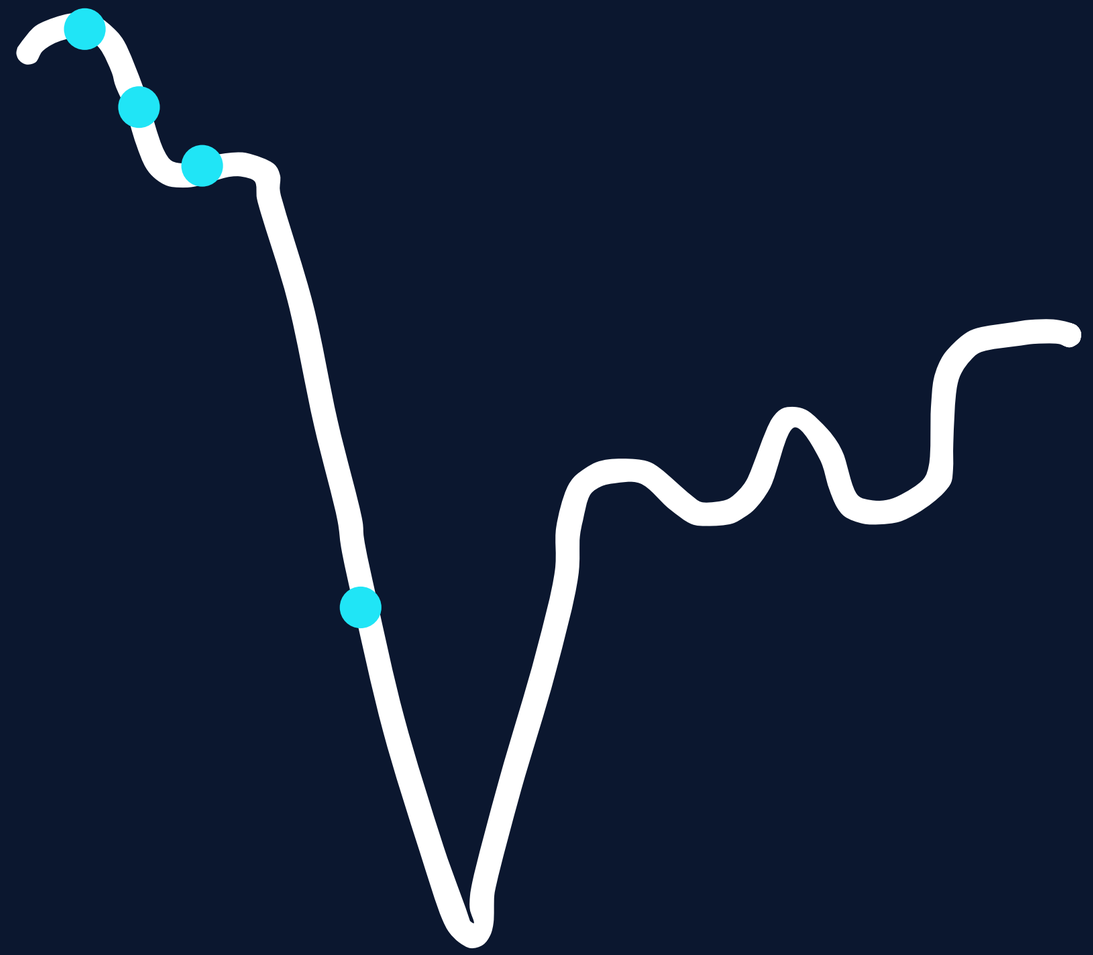
Exploding Gradient

Example: Consider a function, f(x), with the following plot



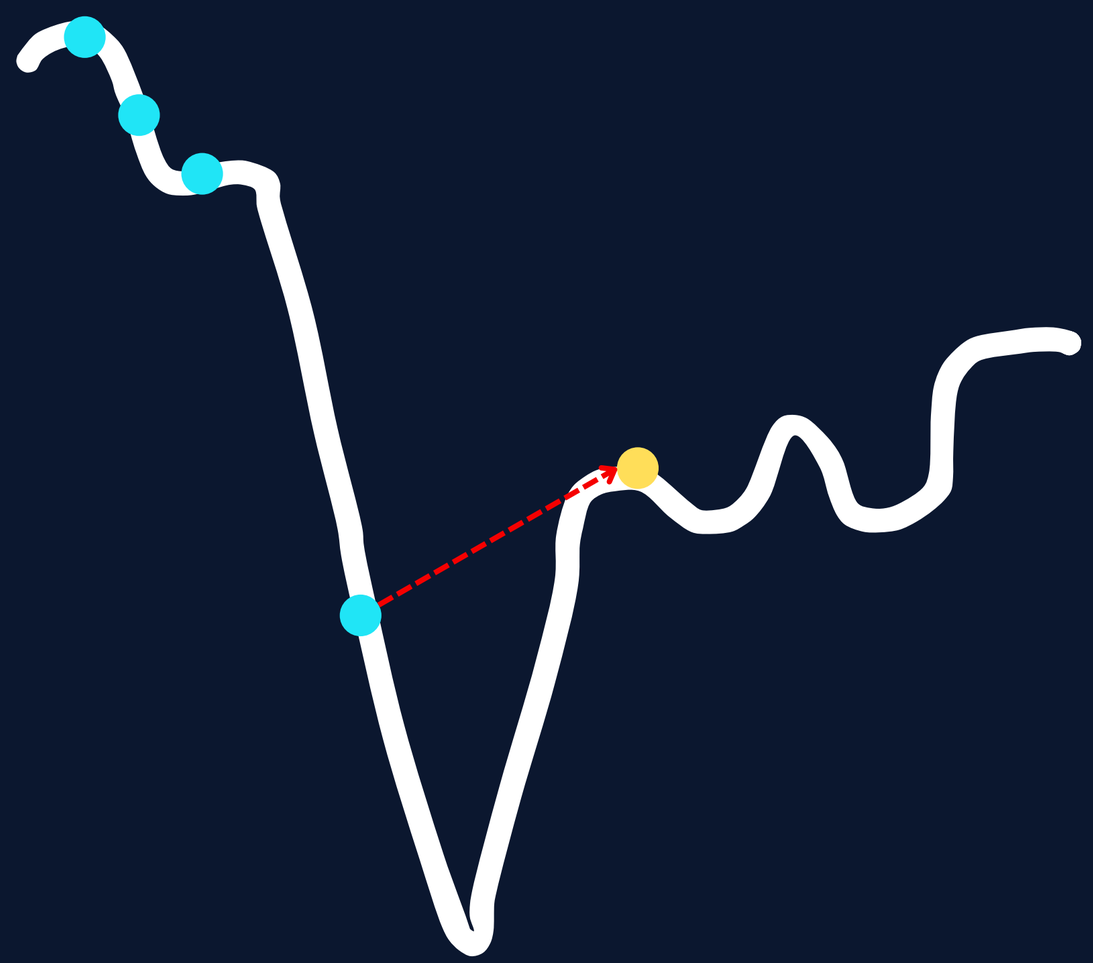
Exploding Gradient

Example: Consider a function, f(x), with the following plot



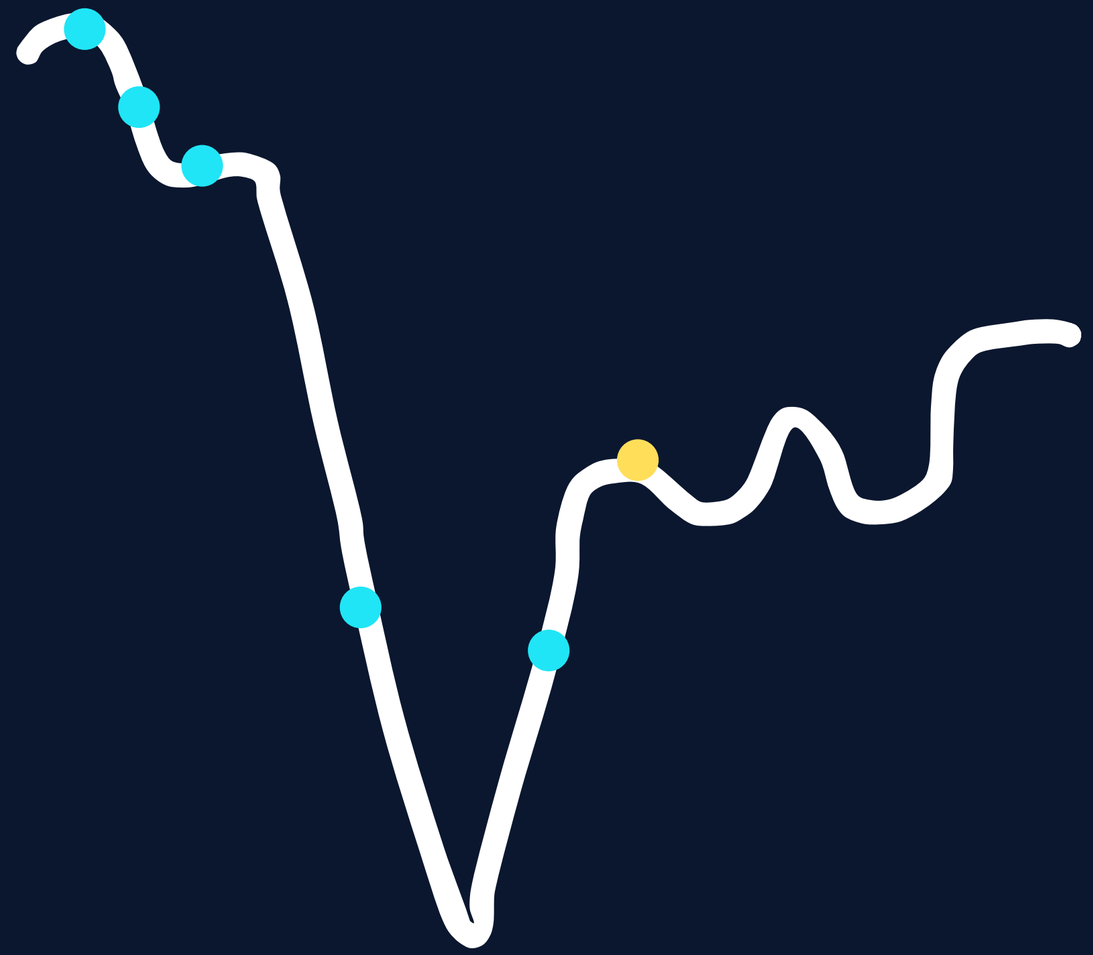
Exploding Gradient

Example: Consider a function, f(x), with the following plot



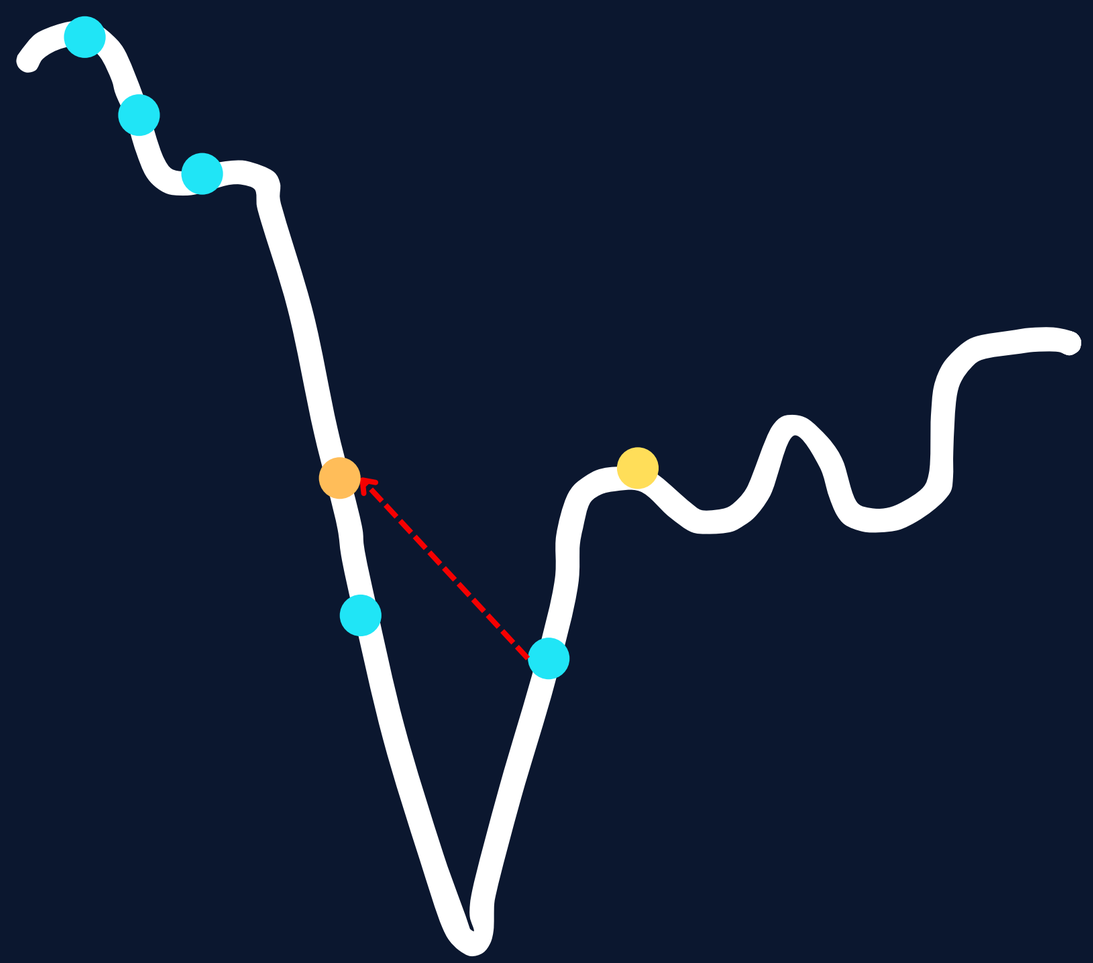
Exploding Gradient

Example: Consider a function, f(x), with the following plot



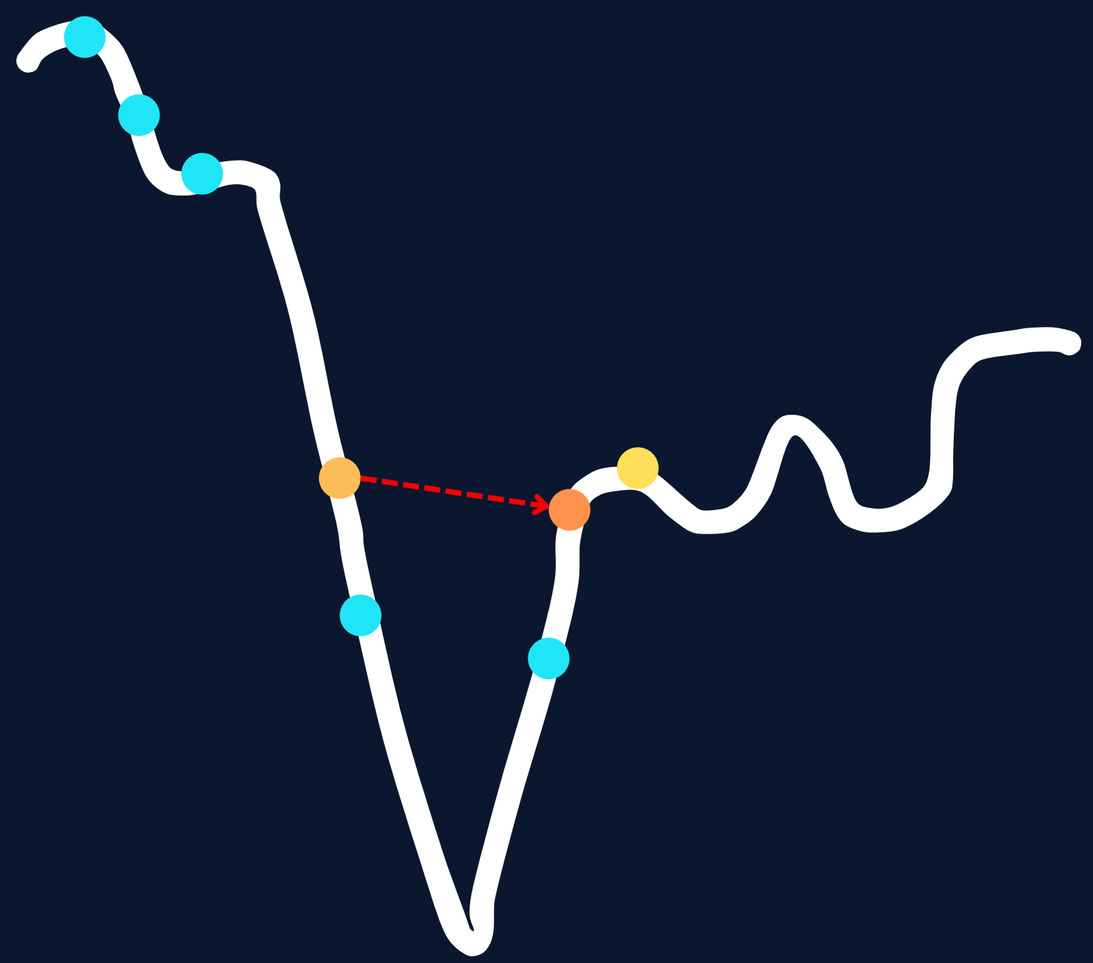
Exploding Gradient

Example: Consider a function, f(x), with the following plot



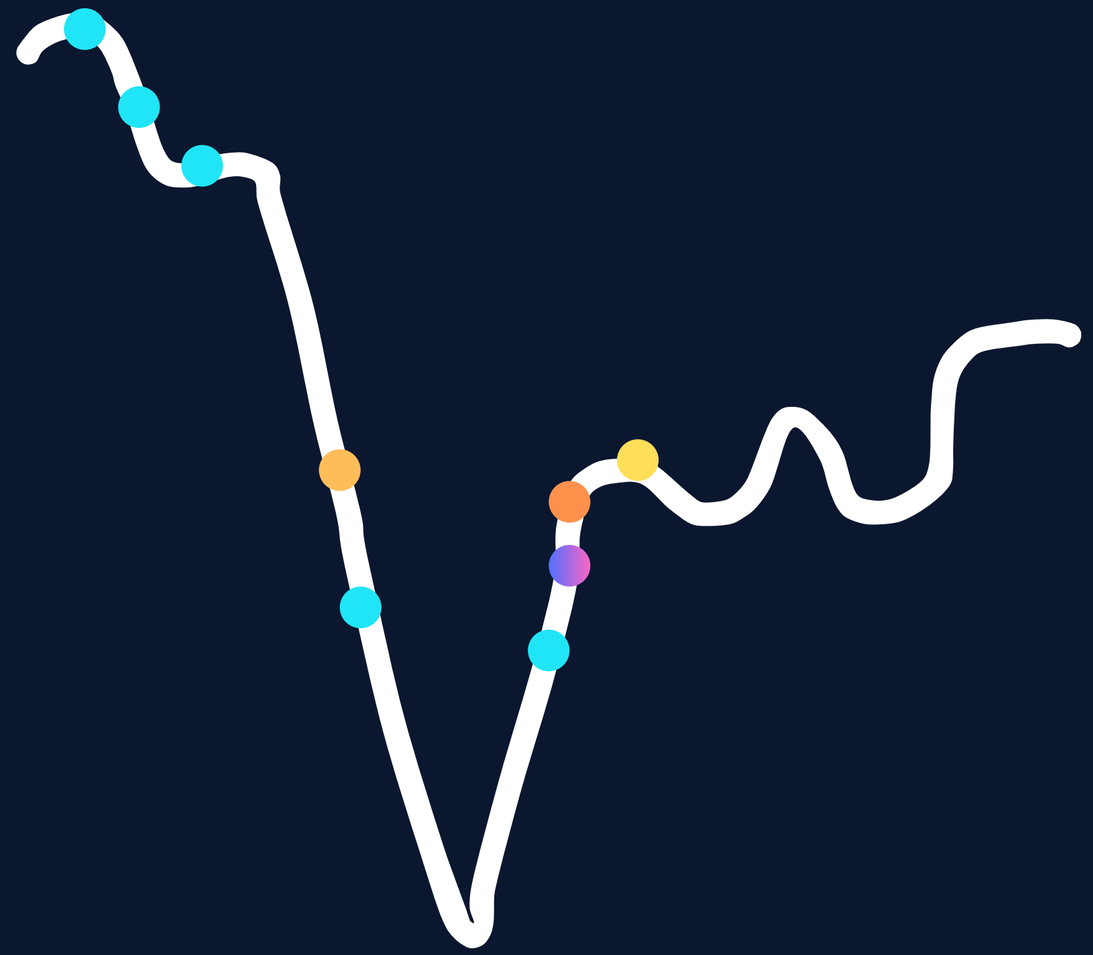
Exploding Gradient

Example: Consider a function, f(x), with the following plot



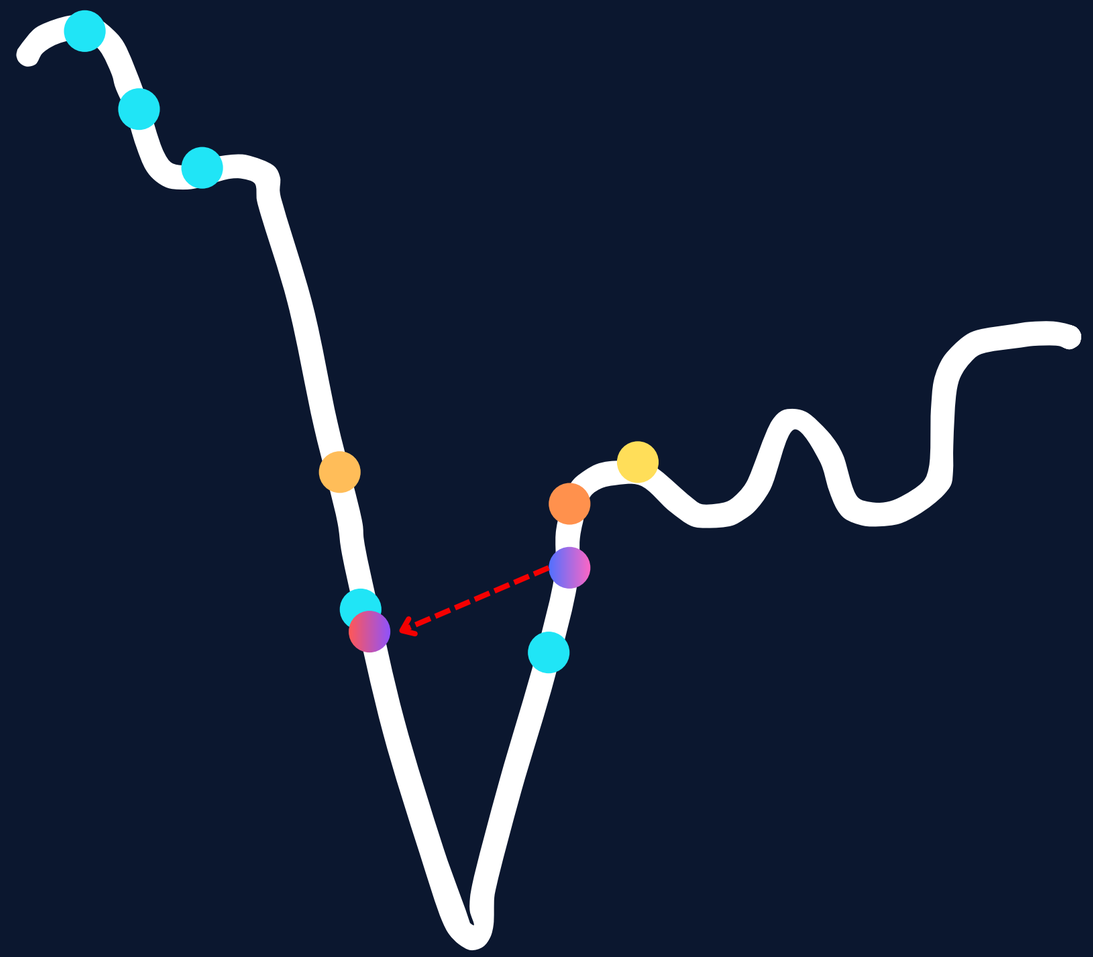
Exploding Gradient

Example: Consider a function, f(x), with the following plot



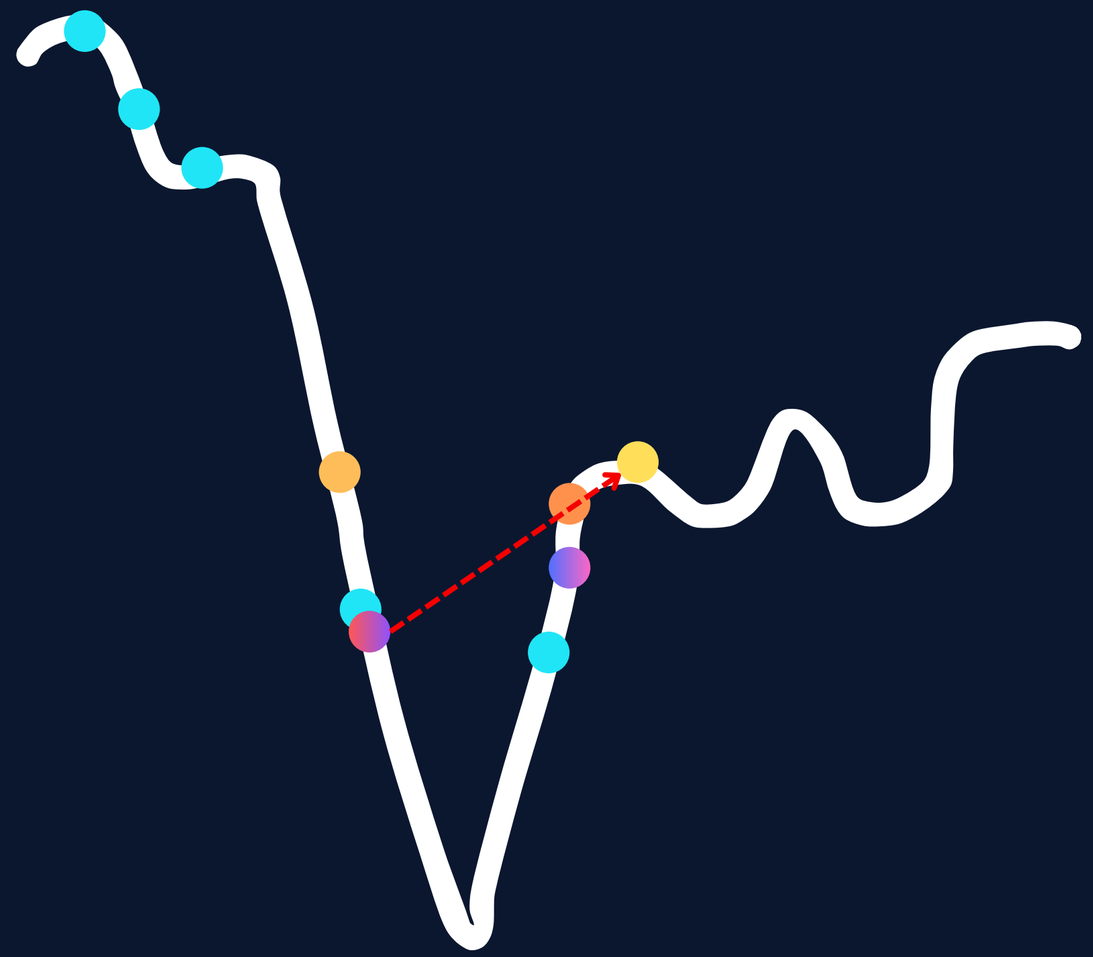
Exploding Gradient

Example: Consider a function, f(x), with the following plot



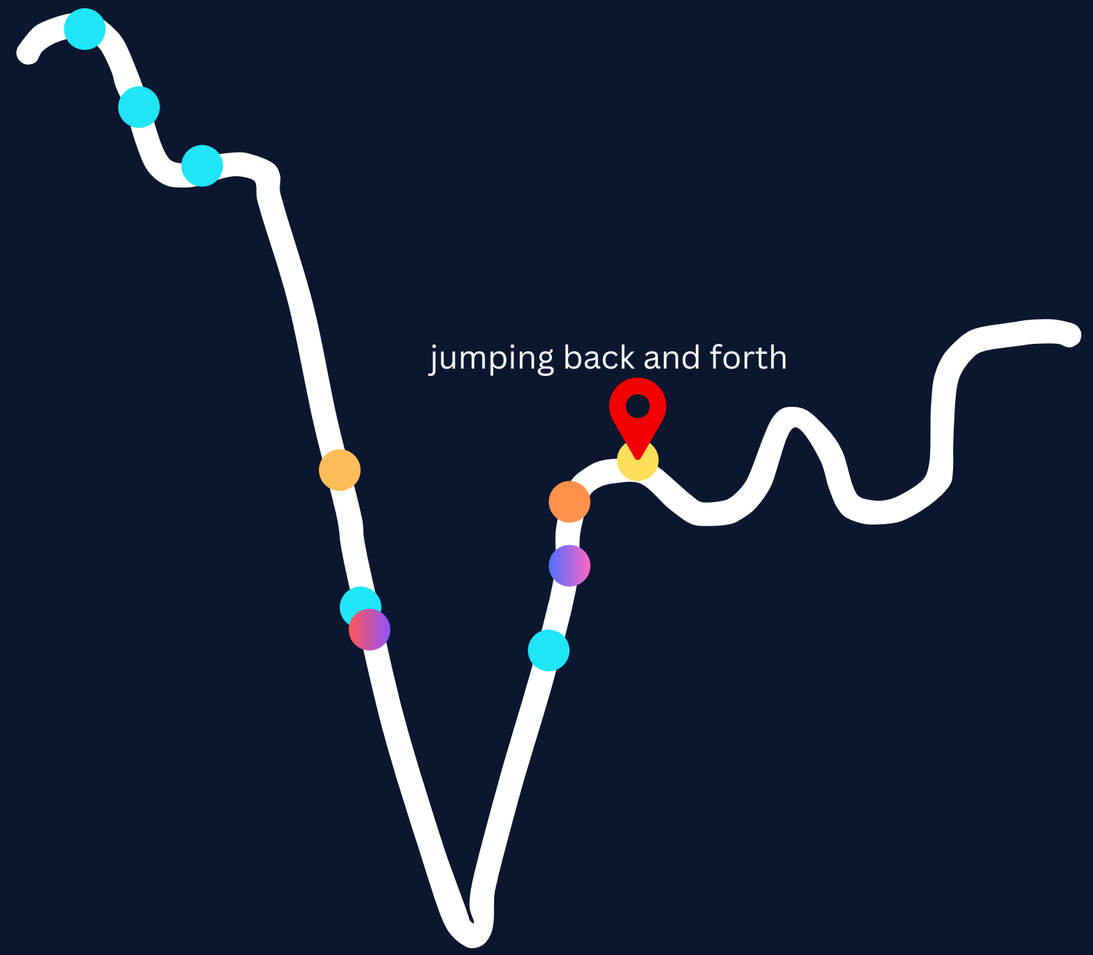
Exploding Gradient

Example: Consider a function, f(x), with the following plot



Exploding Gradient

Example: Consider a function, f(x), with the following plot



Troubleshooting: Use Smaller Hidden Layers

Gradient Stability:

- In smaller architectures (i.e., networks with fewer layers), the gradient is less likely to undergo extreme transformations as it backpropagates through the network. This reduces the risk of gradients either vanishing (becoming too small) or exploding (becoming too large), both of which can hinder effective learning.

Shorter Backpropagation Paths:

- In deeper networks, the gradient must pass through many layers, each of which can modify the gradient through multiplication by the weights and activation function derivatives. If many of these multiplications reduce the gradient (which often happens with saturating activation functions like sigmoid), the gradient can become very small by the time it reaches the earlier layers. Conversely, if many of these multiplications amplify the gradient, it can become excessively large. Smaller architectures have shorter backpropagation paths, making it easier to maintain a stable gradient.

Simpler Optimization Landscape:

- Smaller architectures generally have a simpler optimization landscape, with fewer parameters and potential interactions between layers. This can make it easier for the optimization algorithm (like stochastic gradient descent) to find a good solution without encountering the extreme gradients that can occur in very deep networks.

Reduced Risk of Overfitting:

- Larger architectures have more parameters and can model more complex functions, which increases the risk of overfitting, especially when the amount of training data is limited. Overfitting can exacerbate issues with gradient stability, as the network may focus on minute details in the data, leading to very large or very small gradients in certain regions.

Efficiency in Training:

- Smaller architectures are generally more computationally efficient to train. With fewer layers, there are fewer parameters to update, which can help avoid numerical instability that can arise in deeper networks, particularly when dealing with the gradients.

Troubleshooting: Use Smaller Hidden Layers

TRADE OFFs:

- Smaller architectures can help mitigate the vanishing or exploding gradient problems, they may also be limited in their ability to model complex patterns in the data.

Troubleshooting: Carefully Choose Activation Functions

Non-saturating activation functions refer to activation functions in neural networks that do not have a tendency to "saturate" or flatten out as the input values become very large or very small. Saturation occurs when the output of the activation function reaches a plateau, making gradients very small (or even zero), which can slow down or completely stop the learning process during backpropagation.

Common Saturating Activation Functions:

- **Sigmoid: The sigmoid function saturates at 0 and 1, meaning that for very large or very small input values, the gradient (slope) becomes very small, leading to vanishing gradients.**

- **Tanh: The hyperbolic tangent function saturates at -1 and 1, which also leads to vanishing gradients for large positive or negative inputs.**

Non-Saturating Activation Functions:

Non-saturating activation functions are designed to avoid this problem by ensuring that the output continues to change significantly with changes in input, even for large input values. This helps maintain larger gradients and supports more effective learning.

Rectified Linear Unit (ReLU):

ReLU is non-saturating for positive input values because it continues to increase linearly as the input increases. However, it can "saturate" at 0 for negative inputs, which is why some variants like Leaky ReLU are used.

$$\sigma\left(x\right) = ReLU\left(x\right) = max\left(0, x\right)$$

Troubleshooting: Carefully Choose Activation Functions

Leaky Rectified Linear Unit (Leaky ReLU):

Leaky ReLU allows a small, non-zero gradient for negative inputs, reducing the risk of "dying ReLUs" where neurons might stop learning entirely if they fall into the saturated region of the ReLU.

$$\sigma\left(x\right) = LeakyReLU\left(x\right) = max\left(\alpha x, x\right), \alpha << 1$$

Exponential Linear Unit (ELU):

ELU can also be considered non-saturating for positive inputs and has a more gradual approach to negative inputs.

$$\sigma\left(x\right) = ELU\left(x\right) \begin{cases} \sigma\left(x\right) = ELU\left(x\right) = x, x > 0 \\ \sigma\left(x\right) = ELU\left(x\right) = \alpha\left(e^{x} - 1\right), x \leqslant 0, \alpha << 1 \end{cases}$$

Troubleshooting: Carefully Choose Activation Functions

Softplus:

The Softplus function is a smooth approximation of ReLU and is non-saturating since its output continues to grow with increasing input.

$$\sigma(x) = softplus(x) = log(1 + e^x)$$

Swish:

Swish is non-saturating and tends to perform better than ReLU in many cases.

$$\sigma(x) = softplus(x) = log(1 + e^x)$$

Troubleshooting: Weight Normalizations

Weight normalization is a technique used in training neural networks to improve the convergence of the training process and control the vanishing and exploding gradient problems. The key idea is to decouple the length (or magnitude) of the weights from their direction, allowing the optimization process to more effectively control the scale of the weights. Let **w** be the normalized weight vector, (**u**,**v**) as a vectors, ||**u.v**|| be the Euclidean norm, and **g**, a scalar parameter that controls the magnitude of the weight vector,**w**.

$$w = \frac{u.v}{||u.v||}.g$$

Troubleshooting: Weight Normalizations

Benefits

Improved Gradient Flow:

- By normalizing the weights, weight normalization helps maintain a more stable gradient flow through the network. The norm of the weights is kept constant during backpropagation, which reduces the risk of the gradients becoming too large (exploding gradients) or too small (vanishing gradients).

Faster Convergence:

- Weight normalization can lead to faster convergence during training because it stabilizes the learning process. The network can more effectively adjust the magnitude and direction of weights independently, which often results in more efficient and stable updates.

Better Control Over Weight Magnitude:

- By controlling the magnitude of the weights separately through the scalar, g, the network has better control over the scale of the weight updates. This reduces the likelihood of weights growing too large, which could lead to exploding gradients, or shrinking too small, which could contribute to vanishing gradients.

Decoupling Weight Magnitude and Direction:

- The separation of magnitude and direction allows the network to focus on learning the direction of the weights without being overly influenced by their scale. This can lead to more meaningful weight adjustments and better overall learning.

Troubleshooting: Weight Normalizations

Vanishing Gradients:

- Weight normalization ensures that the weight vectors do not shrink too much during training, which helps to prevent the gradients from vanishing. Since the norm of the weight vector is normalized, the gradients are less likely to become extremely small, which can otherwise hinder learning in deep networks.

Exploding Gradients:

- By controlling the scale of the weight vectors, weight normalization also prevents them from growing too large. This control reduces the chance of gradients becoming excessively large during backpropagation, thus preventing the exploding gradient problem.
-

Troubleshooting: Weight Normalizations

Relationship with Other Techniques

Batch Normalization:

- Weight normalization is conceptually similar to batch normalization in that it aims to stabilize the learning process. However, while batch normalization normalizes the outputs of each layer, weight normalization directly normalizes the weights themselves.

Layer Normalization:

- Similar to weight normalization, layer normalization focuses on stabilizing training by normalizing across layers, though it operates on the activations rather than the weights.

Troubleshooting: Regularization techniques

What is Dropout?

Dropout is a regularization technique used in neural networks to prevent overfitting. During training, dropout randomly "drops out" (sets to zero) a certain percentage of neurons in the network at each training step. This forces the network to learn redundant representations and makes it less likely to rely on any single neuron, thus improving generalization.

How Dropout Works:

- During each forward pass, neurons are randomly deactivated with a certain probability p.
- The remaining active neurons must still pass the correct information through the network.
- During testing or inference, all neurons are active, but their weights are scaled by 1−p to account for the dropout during training.

Troubleshooting: Regularization techniques

How does dropout and Vanishing/Exploding Gradients:

- Indirect Impact: Dropout does not directly address vanishing or exploding gradients but can help by making the network less sensitive to large or small gradients that affect specific neurons. By forcing the network to learn more robust features, it can prevent situations where certain neurons dominate the gradient flow, which might otherwise exacerbate vanishing or exploding gradients.

- Regularization Effect: Dropout helps by reducing the likelihood of co-adaptation of neurons, which can contribute to a more stable training process, potentially mitigating some of the effects of unstable gradients.

Troubleshooting: Regularization techniques

Weight decay:

- Also known as L2 regularization, is a technique used to prevent the weights in a neural network from growing too large. It does this by adding a penalty to the loss function proportional to the square of the magnitude of the weights. single neuron, thus improving generalization.

How Weight Decay Works:

- During the gradient update step, weight decay effectively reduces the weights slightly after each update. This prevents the weights from growing too large over time.

$$L_{new} = L_{original} + \alpha w_i^2$$

Troubleshooting: Regularization techniques

How Weight Decay Control Vanishing/Exploding Gradients:

- Exploding Gradients: Weight decay helps prevent the exploding gradient problem by controlling the magnitude of the weights. By keeping weights small, it reduces the risk that the gradients will become excessively large during backpropagation.
- Vanishing Gradients: While weight decay is more directly effective against exploding gradients, by preventing excessively large weights, it can also contribute to maintaining more stable gradients overall. However, weight decay does not directly address the vanishing gradient problem, which is more about the gradient becoming too small as it propagates backward through the network.
-

THANK YOU!

A presentation is a method of sharing information, ideas, or arguments with an audience using spoken words and visual aids. It typically involves a speaker conveying content to inform, persuade, or entertain the listeners.

www.linkedin.com/in/gerard-ompad

gerard.ompad@gmail.com

https://github.com/gerard-ompad