Deep Learning Course-Gradient Descent

Farid Afzali, Ph.D., P.Eng.

Gradient Descent

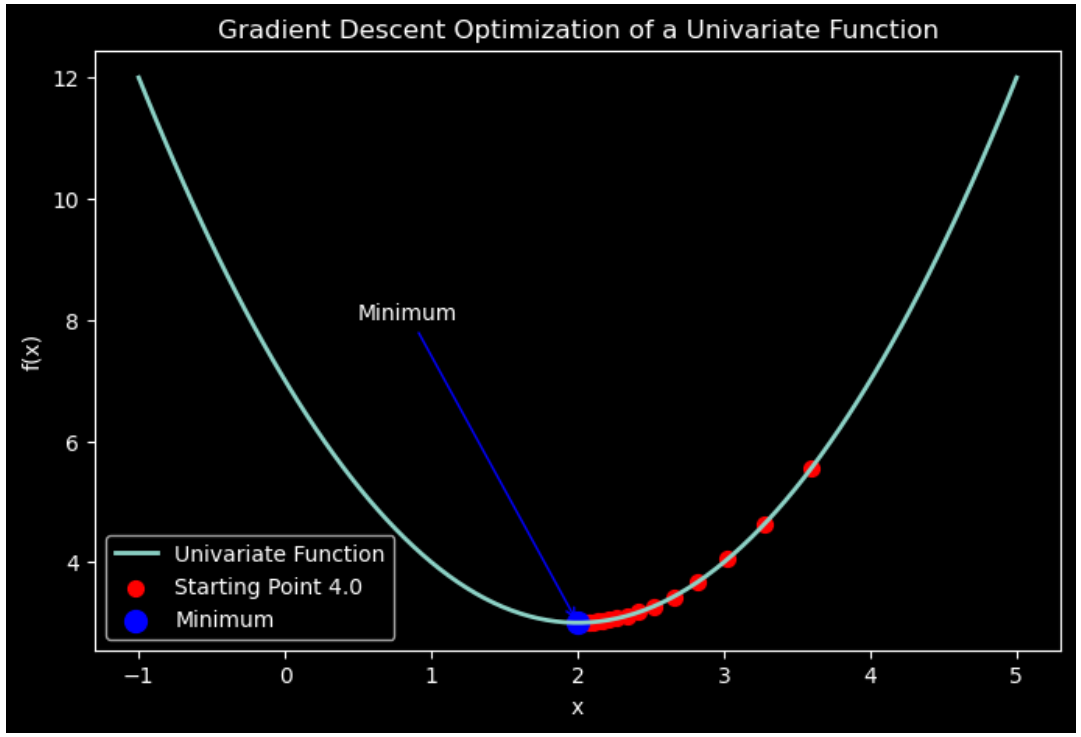
- **Motivation for Gradient Descent**: Gradient descent is used to optimize models by minimizing errors or mistakes in predictions.
- **Three Steps of Learning in Deep Learning**:
  - Step 1: Guess a solution (initialize model parameters).
  - Step 2: Compute the error (difference between predicted and actual values).
  - Step 3: Learn from mistakes by modifying model parameters to reduce errors in future predictions.
- **Mathematical Framework**: To enable the model to learn from its mistakes, we need to represent errors as a mathematical function and find its minimum.

What is gradient descent?

- **Role of Derivatives**: Calculus, particularly derivatives, plays a crucial role. The gradient descent algorithm follows the derivative (or gradient) of the error function to reach the minimum.
- **Gradient Descent Algorithm**:
  - Initialize model parameters randomly.
  - Repeat for a specified number of iterations:
    - ❑Compute the derivative of the error with respect to the parameters at the current guess.
    - ❑Update the guess by subtracting the derivative scaled by a learning rate.

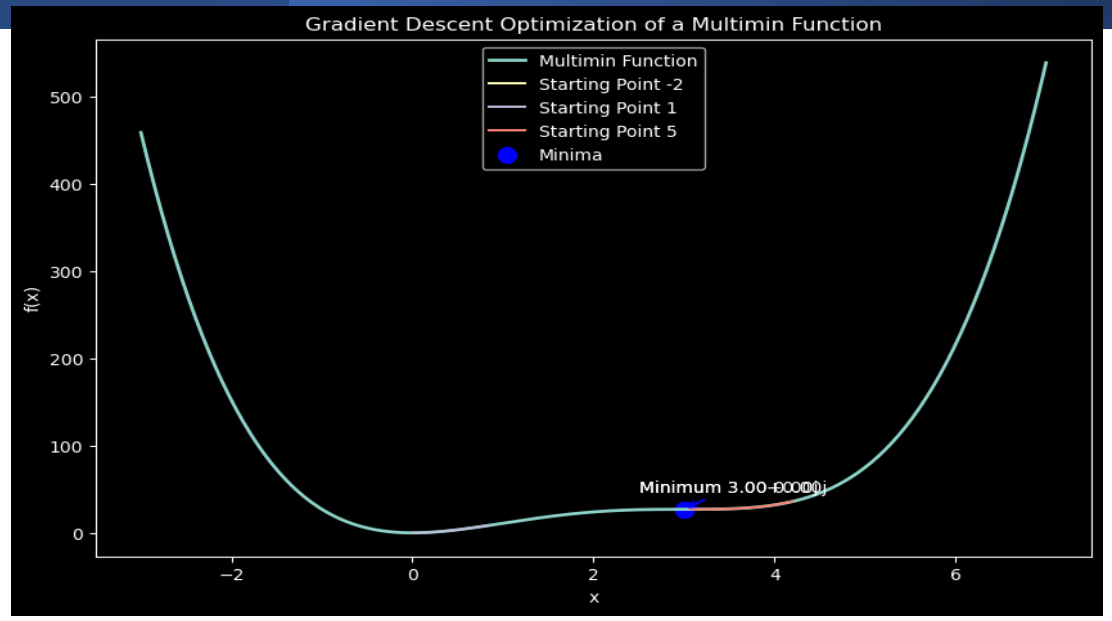
Gradient Descent

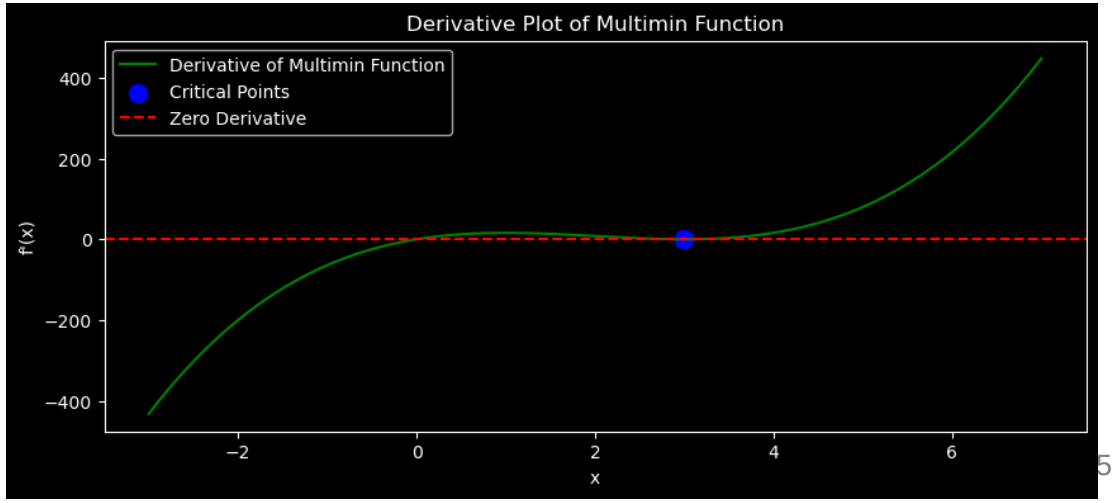
- **Visualizing Gradient Descent**: In 1D and 2D cases, you can visualize the process as moving down a hill toward the minimum. In higher dimensions, it's abstract but follows the same principles.
- **Example**: A numerical example with a simple polynomial function demonstrates how gradient descent updates the guess iteratively to find a local minimum. The true minimum is where the derivative is zero.



Challenges in Gradient Descent

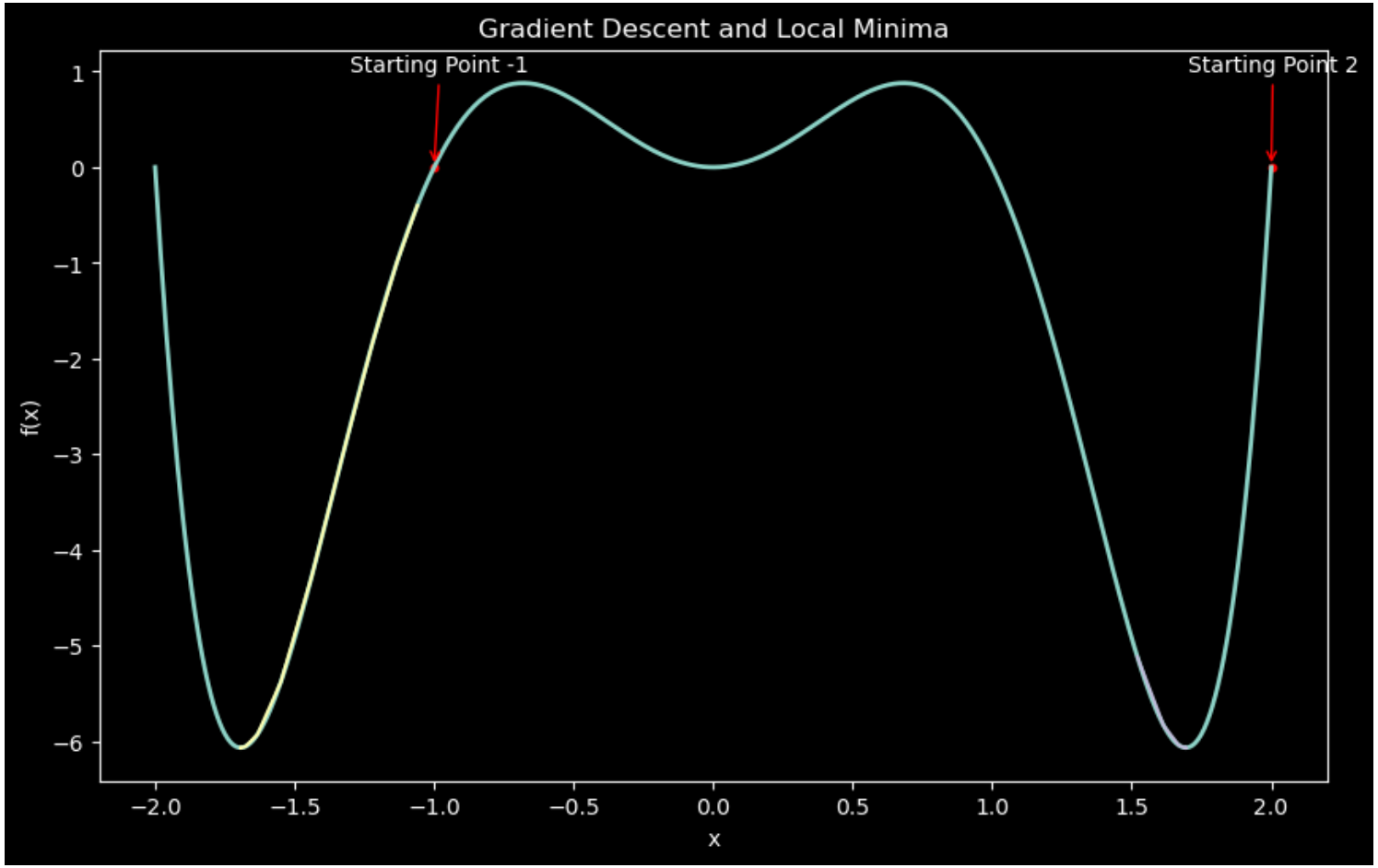
- **Challenges**:
- Gradient descent may not guarantee the exact or global minimum.
- It can get stuck in local minima, suffer from issues like vanishing/exploding gradients, and its effectiveness depends on **parameter choices**.





Gradient Descent with different starting points

$$x^6 - 5x^4 + 4x^2$$

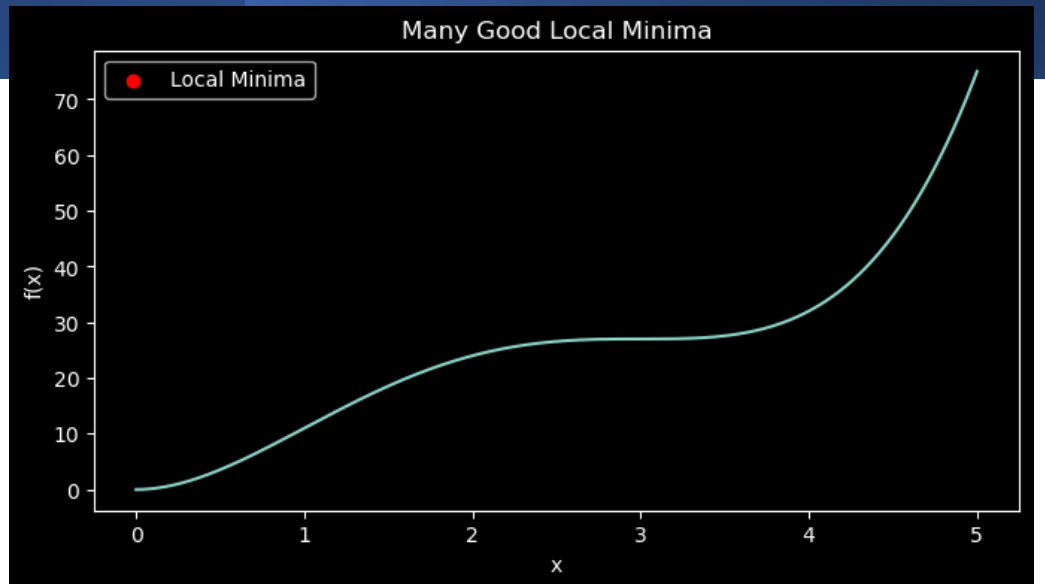


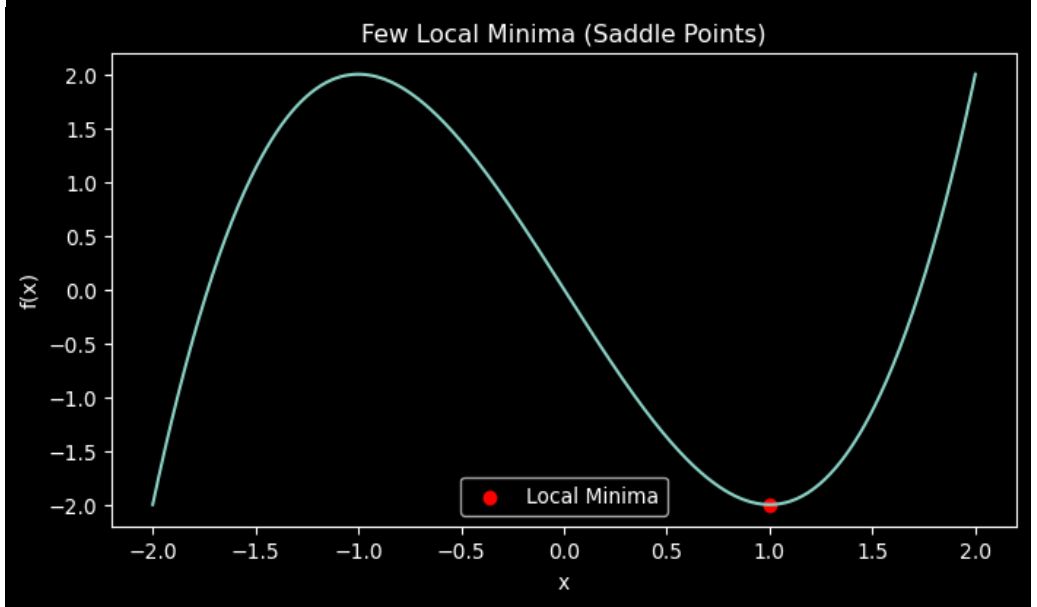
Local Minima

- **Local Minima**: Local minima are points in the error landscape where the loss function reaches a minimum value, but it's not necessarily the global minimum (the best possible solution).
- **Challenges in Deep Learning**: Gradient descent can get stuck in local minima if the model parameters are not set right or if it starts at an unfortunate initial point.
- **Visualization Limitations**: While local minima are visually apparent in one or two dimensions, deep learning models often have millions or even billions of parameters, making it impossible to visualize the full error landscape.

Gradient descent Scenarios

- **Two Possible Scenarios**:
  - **Many Good Local Minima**: In this scenario, there may be numerous local minima, some of which are equally good in terms of minimizing error. These local minima may be distinct solutions.
  - **Few Local Minima**: It's also possible that in high-dimensional spaces, there are actually very few true local minima. Saddle points, which have at least one dimension with a local minimum and another with a local maximum, may be more common.





Deep Learning and Challenges!!!

- **Deep Learning Success**: Despite the potential issue of local minima, deep learning has been remarkably successful, suggesting that local minima might not be a severe problem in practice.
- **Solutions**:
  - **Retraining with Different Initializations**: Train the model multiple times with different random initializations and choose the best-performing model.
  - **Increase Model Complexity**: Increasing the number of model parameters, which increases the dimensionality of the error landscape, may reduce the likelihood of encountering local minima.
- **Uncertainty**: It's often challenging to determine whether the model is trapped in a local minimum, and the choice of solutions depends on the specific problem and its characteristics.

Gradient Descent Algorithm

- **Gradient Descent Algorithm**: You learned how to implement the gradient descent algorithm manually in Python. Gradient descent is an iterative optimization algorithm used to find the minimum of a function.
- **Learning Rate**: The learning rate is a crucial hyperparameter in gradient descent. It determines the step size when updating the model parameters. Choosing an appropriate learning rate is important because a value that's too large can cause divergence, while one that's too small can lead to slow convergence.
- **Number of Training Epochs**: The number of training epochs represents how many iterations of gradient descent are performed. Increasing the number of epochs can help the model converge to a better solution, especially when the learning rate is small.

Python Practice-1 Conclusions

- **Visualization**: You visualized the progress of gradient descent by plotting the function, its derivative, and the estimated local minimum. This visualization helped you understand how the algorithm learns and approaches the minimum.
- **Effect of Learning Rate and Epochs**: You observed how changing the learning rate and the number of training epochs affected the convergence of gradient descent. Smaller learning rates required more epochs to converge, while larger learning rates led to instability.

Python Practice-2 Conclusions

Step 1: Gradient Descent with Random Initial Values

- ❑ You implemented the new function and its derivative in Python using NumPy.
- ❑ You visualized the function to see its landscape, identifying multiple local minima.
- ❑ You reused the gradient descent code from a previous video, modifying it to work with the new function.
- ❑ You ran the gradient descent algorithm multiple times with random initial values for x between -2 and 2. You observed that the algorithm sometimes converged to local minima that were not the global minimum.

Step 2: Fixed Initial Value of x = 0

- ❑ You modified the code to start with x = 0 as the initial value.
- ❑ You ran the gradient descent algorithm with this fixed initial value and observed that it consistently converged to the local maximum at x = 0.
- ❑ You explained the behavior by highlighting that the derivative of the function is zero at x = 0, causing gradient descent to get stuck.

Gradient Descent in Two Dimensions: A Mathematical Perspective

- This discussion delves into the mathematical intricacies of gradient descent, extending our comprehension from one-dimensional functions to their two-dimensional counterparts.
- We explore the consistency of gradient descent principles, highlighting its applicability in navigating multidimensional optimization landscapes.

Derivatives and Gradients: Bridging the Gap

- In two-dimensional spaces, derivatives assume a more expansive role. We encounter derivatives along each axis, allowing for the assessment of the function's behavior concerning each dimension.
- Partial derivatives, a fundamental concept, enable us to focus solely on one dimension while disregarding the other.
- The culmination of these partial derivatives is the gradient—a vector encapsulating information about the function's rate of change across all dimensions.

Notations and Terminology

$$\frac{d}{dx} f(x) = \frac{d}{d\xi_1} f(\xi_1)\Big|_{\xi_1 = x}$$

Diverse notations for derivatives, rooted in historical mathematical development, converge to represent a shared concept.

The '∂' symbol signifies partial derivatives, while the '**nabla**' symbol, an inverted triangle, represents the gradient—a comprehensive collection of multidimensional partial derivatives.

$$\frac{\partial}{\partial x} f(x, y, z) = \frac{\partial}{\partial \xi_1} f(\xi_1, y, z)\Big|_{\xi_1 = x}$$

$$\frac{\partial}{\partial x} f(x, y, z) = \frac{\partial}{\partial \xi_1} f(\xi_1, y, z)\Big|_{\xi_1 = x}$$

$$\frac{\partial}{\partial y} f(x, y, z) = \frac{\partial}{\partial \xi_2} f(x, \xi_2, z)\Big|_{\xi_2 = y}$$

$$\frac{\partial}{\partial z} f(x, y, z) = \frac{\partial}{\partial \xi_3} f(x, y, \xi_3)\Big|_{\xi_3 = z}$$

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}, \dots \right)$$

Python Practice-3 Navigating a Complex Landscape: The Peaks Function

- To illustrate these mathematical concepts, we employ the "Peaks" function, a two-dimensional function dependent on 'X' and 'Y.'
- Our objective is to utilize gradient descent to pinpoint critical features within this complex landscape.
- This task encompasses identifying global and local minima and maxima while navigating through the intricate terrain.
- The selection of initial starting points profoundly influences the algorithm's trajectory, a factor we meticulously scrutinize.

2D gradient descent

- **Problem Overview:** The challenge involves taking the code from the previous work, which was focused on 2D gradient descent, and adapting it to perform gradient ascent. Two distinct solutions are encouraged.
- **Solution 1: Inverting the Gradient:** One approach centers on reversing the direction of the gradient descent. Instead of moving in the negative gradient direction (which leads to local minima), we flip the sign of the gradient. This can be accomplished by multiplying the gradient by -1 or directly changing the sign in the code. The core idea is to make the algorithm move in the direction of the positive gradient, ultimately seeking local maxima.
- **Solution 2: Flipping the Function:** A second method involves flipping the function itself while preserving the gradient descent algorithm. This is achieved by computing the derivative of the negation of the function. It effectively mirrors the function across the z-axis (function axis), leading to an inverse gradient direction. However, it's essential to ensure this flip only affects the function's evaluation, not the plot's visualization.

Python Practice Objectives

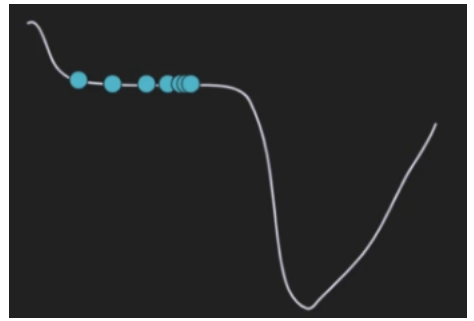
- **Objective 1: Explore Gradient Descent and Meta-Parameters:** The primary objective is to investigate gradient descent and its meta-parameters. These parameters include the initial starting value, the learning rate, and the number of training iterations (epochs), all of which significantly influence gradient descent performance.
- **Objective 2: Demonstrate Parametric Experiments:** The second objective is to illustrate the process of conducting parametric experiments. This is essential for understanding deep learning and comprehending how different parameters impact the training of models. Parametric experiments entail systematically adjusting specific parameters while keeping others constant, providing valuable insights.

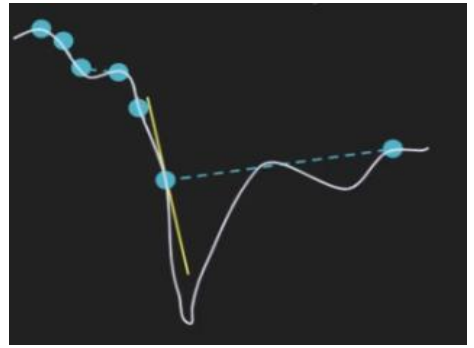
Python Practice Milestones

- **Experiment 1: Varying the Starting Location:** In this experiment, the initial starting point for the gradient descent algorithm is systematically altered. The objective is to observe how different initial guesses influence whether the algorithm converges to a local or global minimum.
- **Experiment 2: Varying the Learning Rate:** Experiment two involves altering the learning rate while maintaining other variables constant. It demonstrates how the magnitude of the learning rate affects the algorithm's convergence. The results emphasize that excessively small or large learning rates can lead to suboptimal or unstable convergence.
- **Experiment 3: Interaction between Learning Rate and Training Epochs:** Experiment three explores the interaction between the learning rate and the number of training epochs. By systematically changing both parameters, it reveals how they jointly impact the model's convergence. The outcomes show that the combination of learning rate and training epochs plays a crucial role in whether the algorithm converges to the correct minimum.

Fixed vs Dynamic learning rate

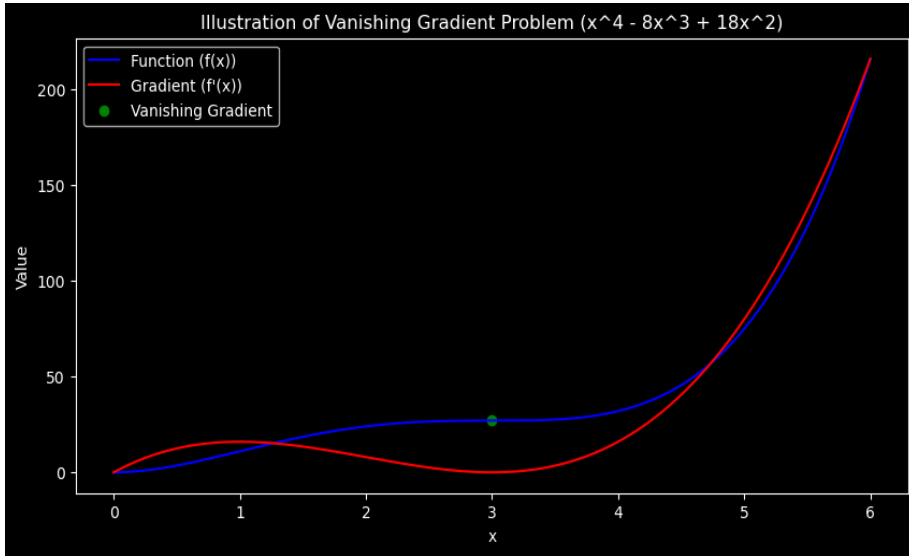
- **Fixed Learning Rate**: This is the standard gradient descent with a constant learning rate. The learning rate remains the same throughout the training process.
- **Gradient-Based Learning Rate**: In this approach, the learning rate is adjusted based on the magnitude of the gradient. If the gradient is large (indicating that the model is far from convergence), the learning rate is larger, and as the gradient decreases (as the model gets closer to convergence), the learning rate decreases.
- **Time-Based Learning Rate**: The learning rate decreases linearly over time. Initially, it's set to a relatively high value and gradually decreases as training progresses. This approach uses the number of training epochs to adjust the learning rate.

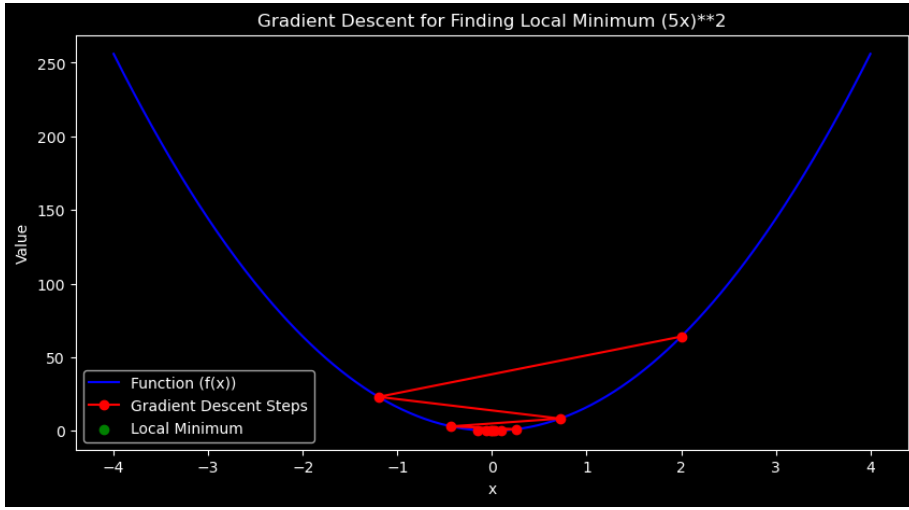




Vanishing and Exploding Gradients

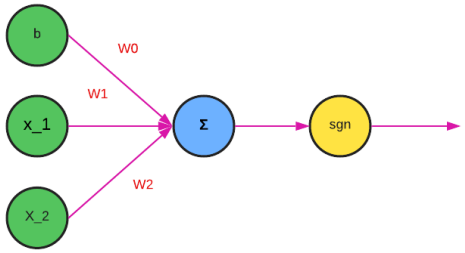
- **Vanishing Gradients**:
  - Imagine a one-dimensional function with a single minimum.
  - When using gradient descent, the derivative of the function guides the direction of movement.
  - In regions where the function changes very little (i.e., the gradient is close to zero), the steps taken in gradient descent become tiny.
  - As a result, the optimization process stagnates, and the model doesn't make significant progress.
  - This problem is referred to as the "vanishing gradient" problem because the gradients become so small that they effectively disappear, causing the learning process to halt prematurely.
- **Exploding Gradients**:
  - In another scenario, imagine a one-dimensional function with a single minimum but with steep slopes.
  - When using gradient descent, the steep gradient magnitudes lead to excessively large steps in the optimization process.
  - These large steps can cause the model to overshoot the minimum and keep oscillating, making it challenging for the optimization process to converge.
  - This problem is known as the "exploding gradient" problem because the gradients become extremely large, causing unstable optimization.





ANN Introduction

- Artificial Neural Networks (ANNs) are a powerful class of machine learning models designed to mimic the structure and functioning of the human brain.



- **Perceptron Architecture:** The perceptron is a fundamental unit of an ANN. It consists of three primary components: ***input nodes, weights, and an activation function.***

  - ✓ These components collectively allow the perceptron to process information and make decisions.

ANN Components

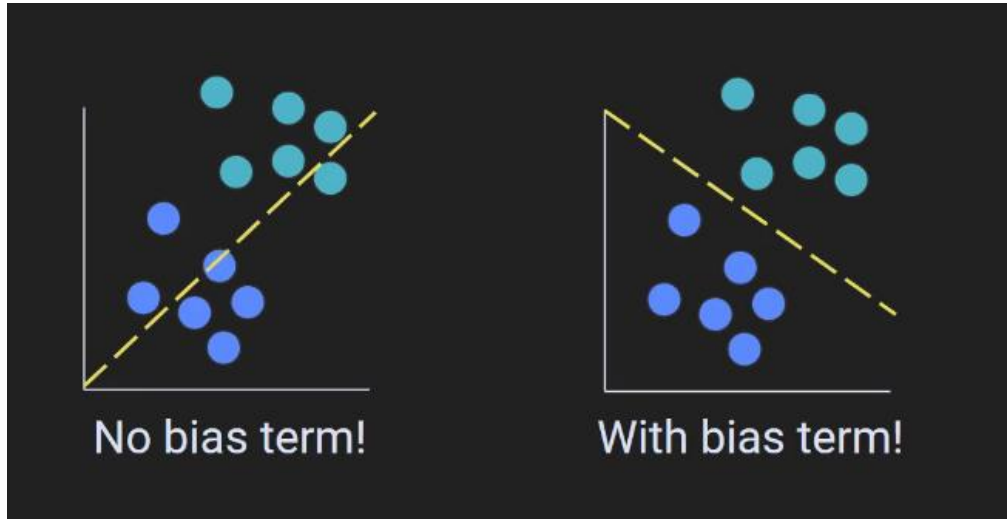
- **Weights**
  - ❑To enhance the perceptron's capabilities, weights are associated with each connection between input nodes and the central computation node.
  - ❑These weights determine the influence of each input on the final output. In the context of the perceptron, uniform weights are initially considered, but later modifications involve custom weights for each input.
- **Activation Function**
  - ❑The activation function plays a critical role in introducing nonlinearity into the perceptron's operations.
  - ❑The sigmoid function, in this case, serves as a basic example of a nonlinear function. It assigns an output of +1 for positive inputs and -1 for negative inputs.

ANN Components

- **Mathematical Representation:** The perceptron's operation can be expressed mathematically as the dot product between the input vector (X) and the weight vector (W), followed by the application of the activation function. This formula captures the essence of the perceptron: Y = X^T * W.
- A fundamental distinction in ANNs lies in the nature of operations emp**Linear vs. Nonlinear Operations:** loyed. Linear operations, involving addition and scalar multiplication, are contrasted with nonlinear operations, which encompass a broader range of mathematical transformations. Linear models excel at solving linearly separable problems, while nonlinear models are essential for tackling more complex, nonlinear problems.

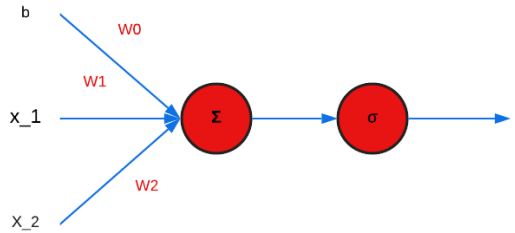
ANN Components: The Role of the Bias Term

- Introducing the bias term (intercept) is crucial in ANNs.
- It provides flexibility in positioning separating hyperplanes and allows models to fit data more effectively.
- The bias term ensures that models are not constrained to pass through the origin, enhancing their capacity to handle diverse datasets.



Perceptron Model

- The perceptron consists of two inputs, x1 and x2, each with corresponding weights, W1 and W2.
- A computational node calculates the weighted sum of the inputs, followed by a nonlinear function (activation function).
- The output of the model is represented as Y hat (ŷ), and there's a bias term (implicitly set to 1).

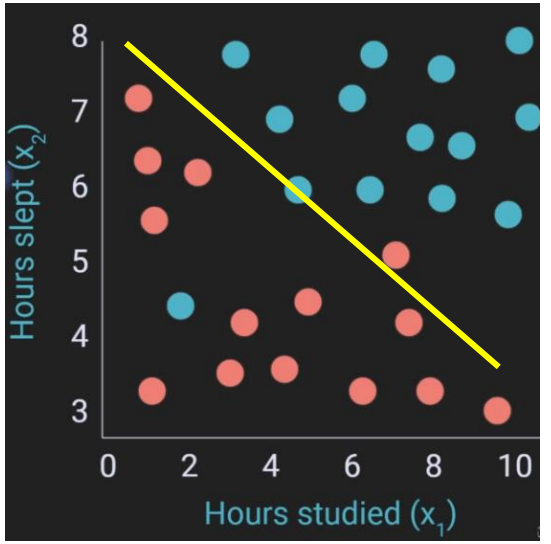


- **Example Problem:**
  - ❑ Predicting exam performance based on the number of hours studied and hours slept.
  - ❑ Data for students' study time, sleep time, and exam results are collected.
- **Feature Space:**
  - ❑ The feature space is a geometric representation where each feature corresponds to an axis, and each observation is a coordinate.
  - ❑ In this case, there are two features (hours studied and hours slept), so it's a two-dimensional feature space.

Separating Hyperplane

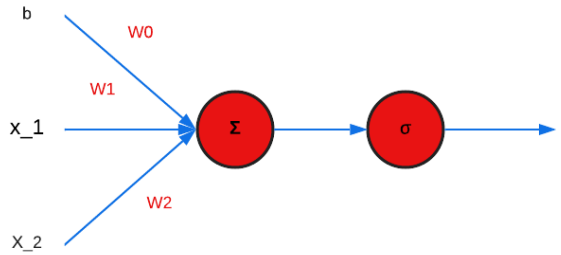
- The goal is to find a separating hyperplane (in two dimensions, it's a line) that separates students who pass from those who fail.
- The separating hyperplane serves as a decision boundary; any data point above predicts pass, and below predicts fail.
- **Different Types of Predictions:**
  - ❖ Discrete (categorical) and continuous (numeric) predictions.
  - ❖ Examples of discrete predictions include pass/fail, while continuous predictions can be exam scores.
- **Continuous Predictions:**
  - ❑ For continuous predictions, an additional dimension (exam score) is added to the feature space.
  - ❑ The model aims to produce continuous predictions that can be mapped to numerical exam scores.
- **Understanding the Output Type:**
- The type of output needed (discrete or continuous) affects the mathematical approaches used in deep learning.

ID#	X_1 Studied	X_2 Slept	Y Results
1	5	6	Pass
2	10	7	Pass
N	7	5	Fail



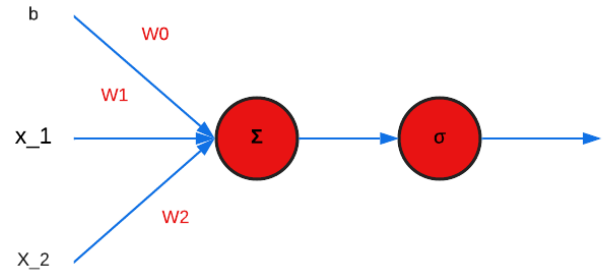
Forward propagation: Simplification of the Perceptron Equation

- The output of the perceptron (y hat) is calculated as the dot product of input features (x) and their corresponding weights (w).
- The bias term (b) is typically set to one and can be absorbed into the dot product.
- **Linear Part of the Model:**
- The linear part of the perceptron model computes a weighted sum of inputs.
- The instructor uses a simplified example with identity activation (no non-linearity) to explain the linear part's operation.



Forward propagation: Simplification of the Perceptron Equation

- **Activation Functions:**
  - Activation functions introduce non-linearity into the model.
  - Common activation functions are sigmoid, hyperbolic tangent (tanh), and Rectified Linear Unit (ReLU).
- **Output Interpretation with Activation Functions:**
  - Activation functions were shown to map linear outputs to different numerical ranges, with sigmoid resembling probabilities between 0 and 1.
- **Learning Weights:**
  - Allusions were made to the learning process, in which weights are learned from training data through techniques like backpropagation and gradient descent.
- **Scaling to Deep Learning:**
  - The idea that deep learning extends the single perceptron model by connecting multiple such models together was mentioned.



Errors and Loss Functions

- Deep learning models make predictions (y hat) about various real-world phenomena, such as house prices, disease probabilities, or image content.
- These predictions are not always perfectly accurate and can differ from the actual reality (Y).
- The difference between the model's prediction (y hat) and the true value (Y) is termed an error. Errors quantify the model's mistakes.
- Errors can be continuous or binarized based on their magnitude. Continuous errors are more sensitive to the size of the error, while binarized errors are easier to interpret.

sample	ŷ	y	error	bin.err
X_1	.9	1	-.1	0
X_2	.2	0	+.2	0
X_3	.1	1	-.9	1
X_4	.51	0	+.51	1

Errors and Loss Functions/Cost Functions

- Two commonly used loss functions are Mean Squared Error (MSE) for continuous data and Cross Entropy for categorical data.
- MSE is used when the model provides numerical predictions, while Cross Entropy is used when the model outputs probabilities.
- MSE squares the difference between predictions and targets, providing sensitivity to large errors and relating to concepts like regression and Euclidean distance.
- Cross Entropy measures the difference between predicted probabilities and actual categories, often with a negative sign for interpretation.

$$\text{Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n} p_i \log(q_i)$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

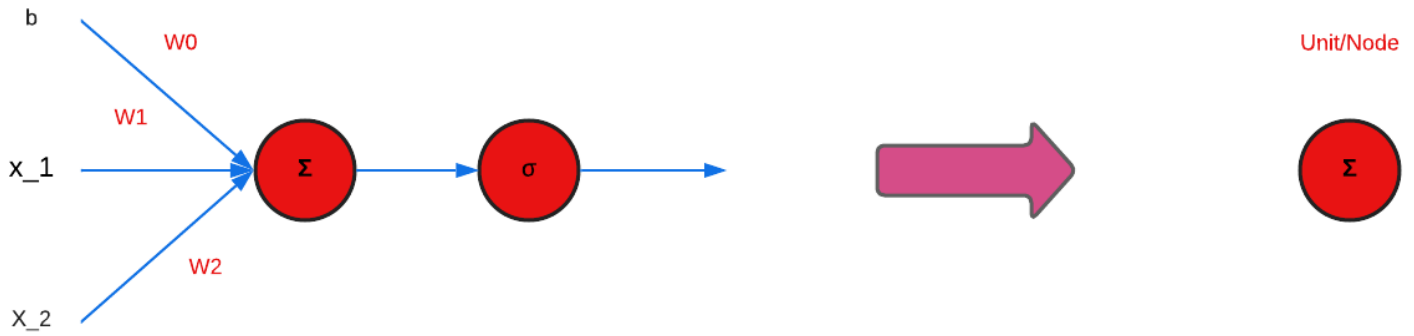
Cost Functions

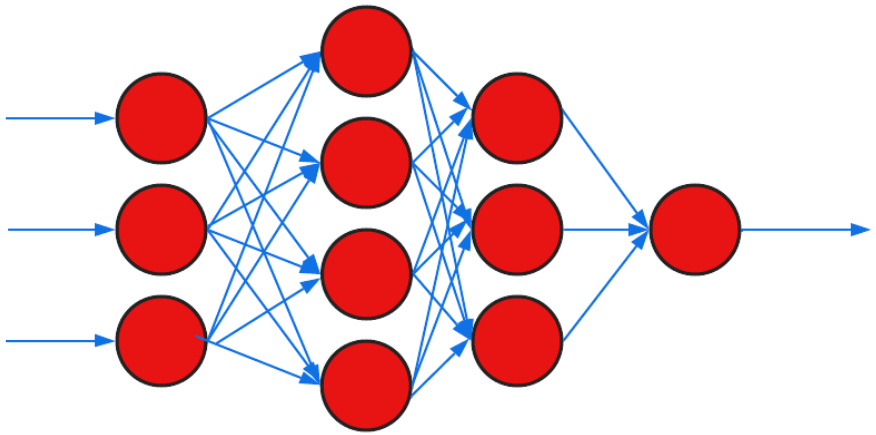
- The cost function, often denoted as J, is the average of the losses computed for multiple data samples. It represents the overall model performance across a dataset.
- Cost functions are used as optimization criteria during training to find the set of weights that minimizes the cost.
- Cost functions aggregate individual losses into a single value that the model aims to minimize.
- **In practice, models are often trained on batches of samples, not individual losses, for efficiency and to mitigate overfitting.**

Transition to Deep Networks

- Instead of representing each individual perceptron in a deep learning network separately, you can simplify the diagram with a single circle, which represents a unit or node.
- Each node performs a linear weighted sum and passes the result through a non-linear activation function.
- Nodes in earlier layers receive input data, while nodes in subsequent layers receive outputs from previous layers.
- *Each unit or node operates independently and doesn't have knowledge of the larger network structure.*

Perceptron Model





Backpropagation

- Backpropagation is the process of adjusting the weights of each node in the network to minimize the loss or cost function.
- Backpropagation is essentially an application of gradient descent, but in a higher-dimensional feature space.
- The key idea is to update the weights in the direction that reduces the loss or cost function.
- The gradient descent formula for updating weights is used:

$$\theta = \theta - \alpha \cdot \nabla J$$

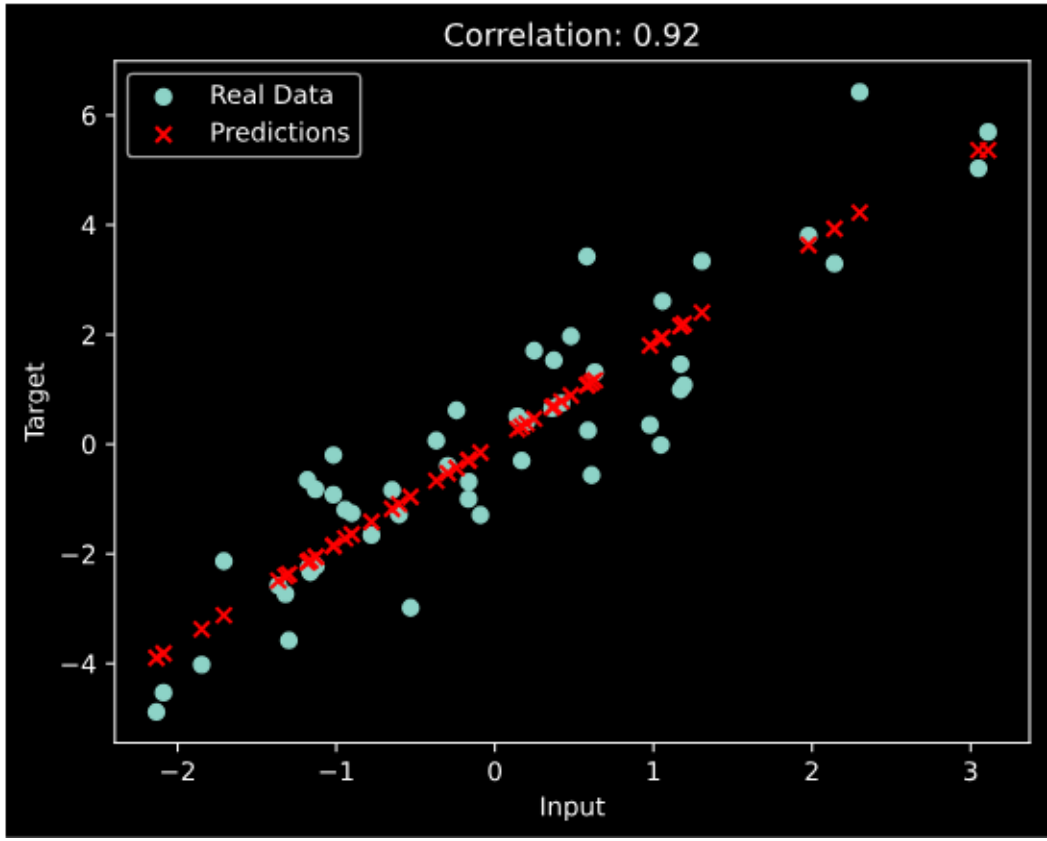
Derivatives and Chain Rule

- To compute the gradient of the loss function with respect to the weights, you need to use the chain rule, especially when dealing with compound functions.
- The exact form of the derivative depends on the choice of activation function within each node.

- **Complexity and Implementation:**
  - While you briefly went through the calculus involved in backpropagation, the exact details can become complex and depend on factors like the choice of activation functions.
  - In practice, deep learning frameworks like PyTorch have optimized implementations that address numerical instabilities and computational efficiency issues.
- **Cost Function:**
  - You can apply backpropagation to either the loss function or the cost function, depending on whether you want to train on individual samples or batches of samples.
- **Practical Application:**
  - Understanding backpropagation is essential for developing, training, and evaluating deep learning models.

ANN Regression

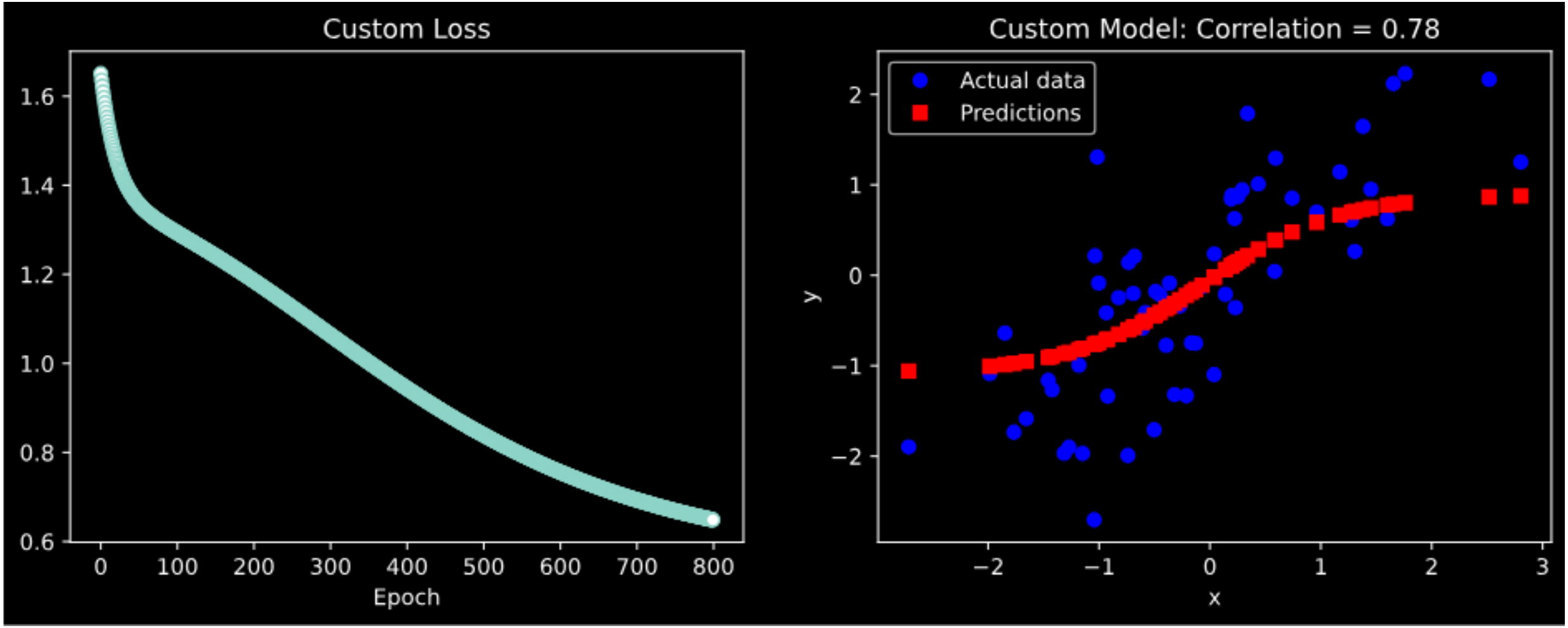
- Regression analysis, a statistical method for predicting data values, forms the basis of this note's exploration of using Artificial Neural Networks (ANNs) for regression tasks.
- **Simple Regression**: Simple regression involves predicting one continuous variable (dependent variable) from another (independent variable).



$$y = X \cdot w + b$$

$$y = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + b$$

ANN Regression



ANN Classification