To generate a linearly separable dataset using Gaussian distributions, we set the means and covariance matrices such that the distributions are distinct and can be separated by a linear boundary.

**Steps to Ensure Linearly Separable Data:**

1. **Means**: The means of the two classes should be sufficiently far apart. In this case, we chose means **[2, 2]** for Class 0 and **[-2, -2]** for Class 1.
2. **Covariance Matrix**: The covariance matrices should be identical and not too large to prevent overlap. We used the identity matrix **[[1, 0], [0, 1]]**, indicating that the features are uncorrelated and have unit variance.

**Data Generation:**

* **Class 0**: 5,000 samples from a Gaussian distribution with mean **[2, 2]** and covariance matrix **[[1, 0], [0, 1]]**.
* **Class 1**: 5,000 samples from a Gaussian distribution with mean **[-2, -2]** and covariance matrix **[[1, 0], [0, 1]]**.

The generated dataset consists of 10,000 data points, with 5,000 from each class. The plot above shows the data distribution, where blue points represent Class 0 and red points represent Class 1. The linear separability is visually evident due to the clear separation between the two classes.

### Perceptron Neural Unit

#### Structure and Initialization:

* **Input**: Each input sample has two features, so the input to the perceptron will be a vector [𝑥1,𝑥2][*x*1​,*x*2​].
* **Weights**: Initialize weights 𝑤1,𝑤2*w*1​,*w*2​ and a bias term 𝑏*b*. For simplicity, weights and bias can be initialized to zero or small random values.
* **Activation Function**: The perceptron uses a step function as its activation function, which outputs 1 if the weighted sum of inputs is greater than or equal to zero, and -1 otherwise.

#### Activity:

1. **Weighted Sum**: Compute the weighted sum of inputs:

𝑧=𝑤1𝑥1+𝑤2𝑥2+𝑏*z*=*w*1​*x*1​+*w*2​*x*2​+*b*

1. **Activation**: Apply the step function:

𝑦={1if 𝑧≥0−1if 𝑧<0*y*={1−1​if *z*≥0if *z*<0​

1. **Learning Rule**: Adjust weights and bias based on the error 𝑒=𝑦true−𝑦*e*=*y*true​−*y*:

𝑤𝑖←𝑤𝑖+𝜂𝑒𝑥𝑖for each 𝑖*wi*​←*wi*​+*ηexi*​for each *i*

𝑏←𝑏+𝜂𝑒*b*←*b*+*ηe*

Here, 𝜂*η* is the learning rate.

### Adaline (Adaptive Linear Neuron)

#### Structure and Initialization:

* **Input**: Similar to the perceptron, each input sample has two features [𝑥1,𝑥2][*x*1​,*x*2​].
* **Weights**: Initialize weights 𝑤1,𝑤2*w*1​,*w*2​ and a bias term 𝑏*b*. They can also be initialized to zero or small random values.
* **Activation Function**: Adaline uses a linear activation function followed by a threshold function for classification.

#### Activity:

1. **Weighted Sum**: Compute the weighted sum of inputs:

𝑧=𝑤1𝑥1+𝑤2𝑥2+𝑏*z*=*w*1​*x*1​+*w*2​*x*2​+*b*

1. **Activation**: For classification, apply a threshold function (sign function):

𝑦={1if 𝑧≥0−1if 𝑧<0*y*={1−1​if *z*≥0if *z*<0​

1. **Learning Rule**: Adaline updates weights based on the linear activation value 𝑧*z* rather than the thresholded output 𝑦*y*:

𝑤𝑖←𝑤𝑖+𝜂(𝑦true−𝑧)𝑥𝑖for each 𝑖*wi*​←*wi*​+*η*(*y*true​−*z*)*xi*​for each *i*

𝑏←𝑏+𝜂(𝑦true−𝑧)*b*←*b*+*η*(*y*true​−*z*)

### Differences Between Perceptron and Adaline:

* **Error Calculation**:
  + **Perceptron**: Updates weights based on the error between the predicted class label 𝑦*y* and the true label 𝑦true*y*true​.
  + **Adaline**: Updates weights based on the error between the linear combination 𝑧*z* and the true label 𝑦true*y*true​.
* **Activation Function**:
  + **Perceptron**: Uses a step function for activation, which can make learning difficult when the data is not linearly separable due to discontinuous gradient.
  + **Adaline**: Uses a linear activation followed by a threshold function for classification, allowing for continuous error gradients and typically more stable learning.
* **Learning Stability**:
  + **Perceptron**: Can oscillate and fail to converge if the data is not linearly separable.
  + **Adaline**: Can still adjust weights even if the data is not perfectly separable, thanks to the continuous cost function (mean squared error).

Let's implement and visualize the perceptron and Adaline learning on the generated dataset.

The plots above show the decision boundaries of both the Perceptron and Adaline classifiers on the generated dataset.

**Perceptron**

* **Structure**: Two inputs [𝑥1,𝑥2][*x*1​,*x*2​], two weights, and a bias term.
* **Weights Initialization**: Weights initialized to zero.
* **Learning Rule**: Adjust weights using the step function's error, updating only when the predicted class label is incorrect.
* **Decision Boundary**: The perceptron creates a linear decision boundary that separates the two classes.

**Adaline**

* **Structure**: Similar to the perceptron with two inputs [𝑥1,𝑥2][*x*1​,*x*2​], two weights, and a bias term.
* **Weights Initialization**: Weights initialized to zero.
* **Learning Rule**: Adjust weights based on the difference between the actual output 𝑧*z* and the true label, minimizing the mean squared error.
* **Decision Boundary**: Adaline creates a linear decision boundary, but its weights update continuously, making it potentially more stable in convergence.

**Differences:**

1. **Error Calculation**:
   * **Perceptron**: Uses binary class labels for error calculation.
   * **Adaline**: Uses the actual output for error calculation, leading to a continuous error function.
2. **Convergence and Stability**:
   * **Perceptron**: May not converge if the data is not perfectly linearly separable.
   * **Adaline**: Can adjust weights more smoothly, often leading to better convergence properties even if the data is not perfectly separable.

Both models effectively classify the dataset, with their linear decision boundaries clearly separating the two classes.