📚 Table of Contents

Lab 02: Gradient Descent

- Student name: Hoang Anh Tra
- ID: 21127453

How to do your homework

- You will work directly on this notebook; the word **TODO** indicates the parts you need to do.
- You can discuss the ideas as well as refer to the documents, but *the code and work must be yours*.

How to submit your homework

- Before submitting, save this file as `<ID>.jl`. For example, if your ID is 123456, then your file will be `123456.jl`. And export to PDF with name `123456.pdf` then submit zipped source code and pdf into `123456.zip` onto Moodle.

Note

Note that you will get 0 point for the wrong submit.

Content of the assignment:

- Gradient Descent

1. Loss landscape

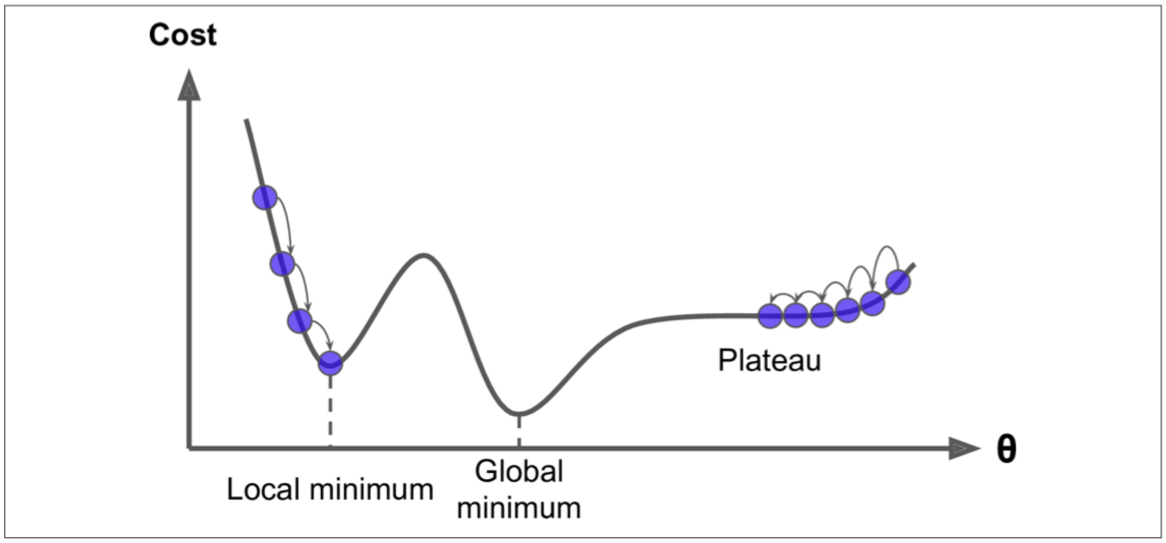


Figure 1. Loss landscape visualized as a 2D plot. Source: codecamp.vn

The gradient descent method is an iterative optimization algorithm that operates over a loss landscape (also called an optimization surface).As we can see, our loss landscape has many peaks and valleys based on which values our parameters take on. Each peak is a local maximum that represents very high regions of loss – the local maximum with the largest loss across the entire loss landscape is the global maximum. Similarly, we also have local minimum which represents many small regions of loss. The local minimum with the smallest loss across the loss landscape is our global minimum. In an ideal world, we would like to find this global minimum, ensuring our parameters take on the most optimal possible values.

Each position along the surface of the corresponds to a particular loss value given a set of parameters $\mathbf{W}$ (weight matrix) and $\mathbf{b}$ (bias vector). Our goal is to try different values of $\mathbf{W}$ and $\mathbf{b}$, evaluate their loss, and then take a step towards more optimal values that (ideally) have lower loss.

2. The "Gradient" in Gradient Descent

We can use $\mathbf{W}$ and $\mathbf{b}$ and to compute a loss function L or we are able to find our relative position on the loss landscape, but **which direction** we should take a step to move closer to the minimum.

- All We need to do is follow the slope of the gradient $\nabla_{\mathbf{W}}$. We can compute the gradient $\nabla_{\mathbf{W}}$ across all dimensions using the following equation:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- But, this equation has 2 problems:
  - 1. It's an **approximation** to the gradient.
  - 2. It's painfully slow.

In practice, we use the **analytic gradient** instead.

3. Forward & Backward

In this section, you will be asked to fill in the black to form the forward process and backward process with the data defined as follows:

- Feature: X (shape: $n \times d$, be already used bias trick)
- Label: y (shape: $n \times 1$)
- Weight: W (shape: $d \times 1$)

3.1. Forward

TODO: Consider one sample $\mathbf{x}_i$. Fill in the blank

$$h_i = \mathbf{x}_i^T W \Rightarrow \frac{\partial h_i}{\partial W} = \mathbf{x}_i$$

$$\hat{y}_i = \sigma(h_i) \Rightarrow \frac{\partial \hat{y}_i}{\partial h_i} = \sigma(h_i)(1 - \sigma(h_i))$$

$$loss_i = (\hat{y}_i - y_i)^2 \Rightarrow \frac{\partial loss_i}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i)$$

3.2. Backward

Our loss function is MSE:

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

Goal: Compute $\nabla Loss = \frac{\partial Loss(W)}{\partial W}$

How to compute $\nabla Loss$?: Use Chain-rule. Your work is to fill in the blank

TODO: Fill in the blank

$$\nabla Loss = \frac{\partial Loss(W)}{\partial W} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial loss_i(W)}{\partial W}$$

$$= \frac{1}{n}\sum_{i=1}^{n} 2(\hat{y}_i(W) - y_i) * \frac{\partial \hat{y}_i(W)}{\partial W}$$

$$= \frac{1}{n}\sum_{i=1}^{n} 2(\hat{y}_i(W) - y_i) * \sigma'(h_i(W)) * \mathbf{x}_i$$

4. Implementation

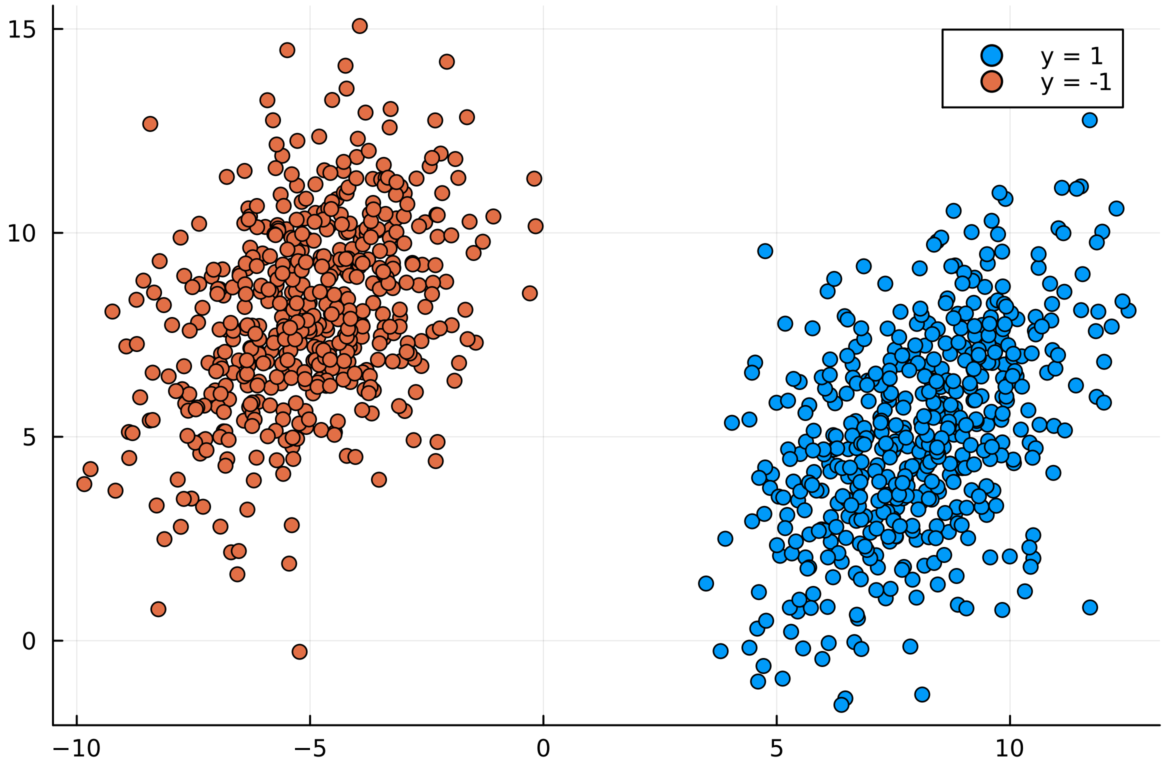
4.1. Import library

```julia
using  Distributions, Plots, LinearAlgebra, Random
```

```
TaskLocalRNG()
```

```julia
Random.seed!(2024)
```

4.2. Create data



```julia
begin
    # DOT NOT MODIFY THIS CODE
    # generate a 2-class classification problem with 1,000 data points, each data point is a 2D feature vector
    # number of data points
    n = 1000

    # dimensionality of data
    d = 2

    # mean
    μ = 5

    # variance
    Σ = 8

    # Generate two class for synthesized data
    positive = rand(MvNormal([Σ, μ], 3 .* [1 (μ - d)/μ; (μ - d)/μ d]), n ÷ 2)
    negative = rand(MvNormal([-μ, Σ], 3 .* [1 (μ - d)/μ; (μ - d)/μ d]), n ÷ 2)

    # Combine two class of generated data.
    # X = features
    # y = label
    X = hcat(positive, negative)
    y = vcat(ones(n ÷ 2) .- 1, ones(n ÷ 2))'

    # Visualization
    plt = scatter(positive[1, :], positive[2, :], label="y = 1")
    scatter!(plt, negative[1, :], negative[2, :], label="y = -1")
    # DOT NOT MODIFY THIS CODE
end
```

```
4×1000 Matrix{Float64}:
 10.9282   12.1782   7.57957  7.32639  …  -6.14176  -2.93022  -4.00982  -2.2689
  4.11468   7.70253  3.26119  8.75393      9.18633   7.08326  11.3397    4.8723
  1.0       1.0      1.0      1.0          1.0       1.0       1.0       1.0
  0.0       0.0      0.0      0.0          1.0       1.0       1.0       1.0
```

```julia
begin
    # insert a column of 1's as the last entry in the feature matrix
    # -- allows us to treat the bias as a trainable parameter

    X_aug = vcat(X, ones(n)')
    data = vcat(X_aug, y)
end
```

```
(700×3 Matrix{Float64}:   , [0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0,   more ,0.0], 30
   7.53828   4.88393   1.0
```

```julia
begin
    # DOT NOT MODIFY THIS CODE
    # Split data, use 50% of the data for training and the remaining 50% for testing
    # Prepare data
    D = data'[shuffle(1:end), :]

    # Calculate the number of samples for each split
    n_train = Int(n * 0.7)

    # Split the samples into train, and test sets
    train_data = D[begin:n_train, :]
    test_data = D[n_train + 1: end, :]
    println(size(train_data), size(test_data))

    # Move samples to train-test features and labels
    X_train, y_train, X_test, y_test = train_data[:,1:3], train_data[:,4],
    test_data[:,1:3], test_data[:,4]
    # DOT NOT MODIFY THIS CODE
end
```

```
(700, 4)(300, 4)
```

4.3. Training

Sigmoid function and derivative of the sigmoid function

```
sigmoid_deriv (generic function with 1 method)
```

```julia
begin
    function sigmoid_activation(x)
        #TODO
        """compute the sigmoid activation value for a given input"""
        #return?
        sigmoid(_x) = 1 / (1 + exp(-_x))
        return sigmoid.(x)
    end

    function sigmoid_deriv(x)
        #TODO
        """
        Compute the derivative of the sigmoid function ASSUMING
        that the input 'x' has already been passed through the sigmoid
        activation function
        """
        #return?
        return x(1.0 - x)
    end
end
```

Compute output

```
predict (generic function with 1 method)
```

```julia
begin
    function compute_h(W, X)
        #TODO
        """
        Compute output: Take the inner product between our features 'X' and the weight
        matrix 'W'
        """
        # return?
        return X * W
    end

    function predict(W, X)
        #TODO
        """
        Take the inner product between our features and weight matrix,
        then pass this value through our sigmoid activation
        """
        # preds = ...
        preds = sigmoid_activation(compute_h(W, X))

        # apply a step function to threshold the outputs to binary
        # class labels
        preds[preds .<= 0.5] .= 0
        preds[preds .> 0] .= 1

        return preds
    end
end
```

Compute gradient

```
compute_gradient (generic function with 1 method)
```

```
begin
    function compute_gradient(error, y_hat, trainX)
        #TODO
        """
        the gradient descent update is the dot product between our
        features and the error of the sigmoid derivative of
        our predictions
        """
        # return?
        return transpose(trainX) * error
    end
end
```

Training function

```
train (generic function with 1 method)
```

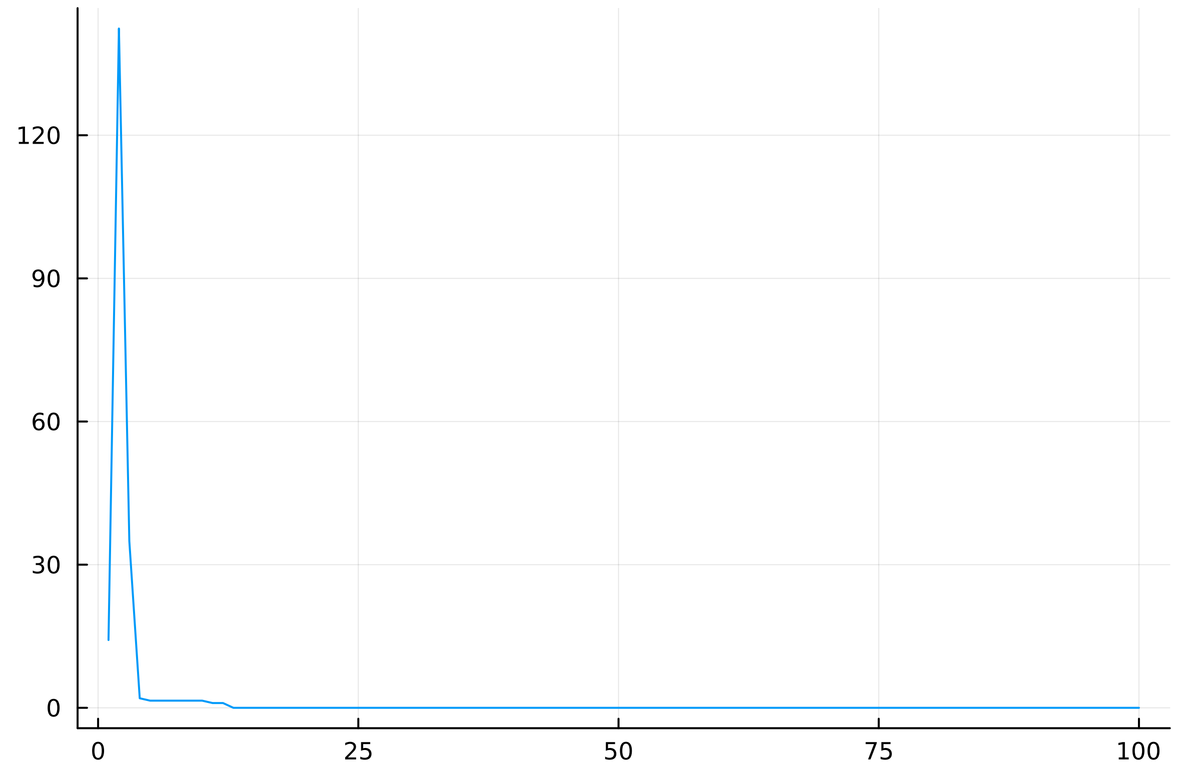
```
begin
    function train(W, trainX, trainY, learning_rate, num_epochs)
        losses = []
        for epoch in 1:num_epochs
            y_hat = sigmoid_activation(compute_h(W, trainX))
            # now that we have our predictions, we need to determine the
            # 'error', which is the difference between our predictions and
            # the true values
            error = y_hat - trainY
            append!(losses, 0.5 * sum(error .^ 2))
            grad = compute_gradient(error, y_hat, trainX)
            W -= learning_rate * grad

            if epoch == 1 || epoch % 5 == 0
                println("Epoch=$epoch; Loss=$(losses[end])")
            end
        end
        return W, losses
    end
end
```

Initialize our weight matrix and list of losses

```
0.1
```

```
begin
    #initialize our weight matrix and necessary hyperparameters
    W = rand(Normal(), (size(X_train)[2], 1))
    num_epochs=100
    learning_rate=0.1
end
```

Train our model



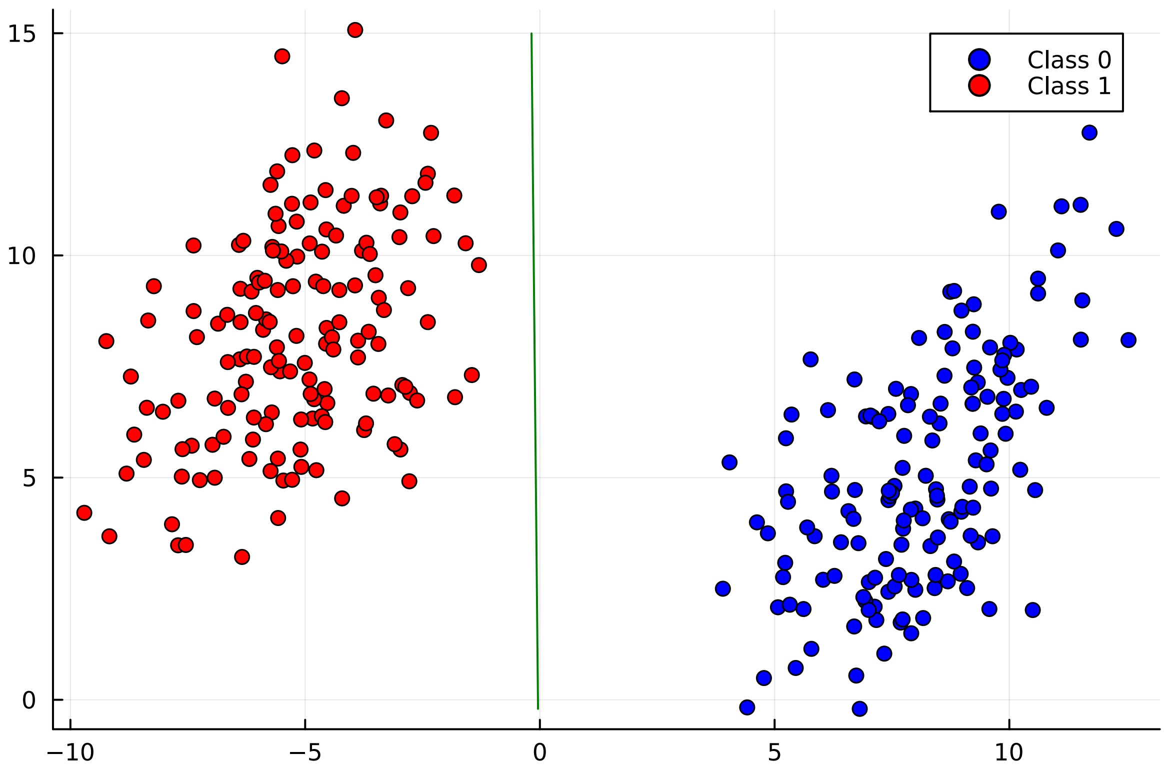
```
begin
    #training model
    θ, losses = train(W, X_train, y_train, learning_rate, num_epochs)
    #visualiza training process
    plot(1:num_epochs, losses, legend=false)
end
```

```
Epoch=1; Loss=14.203357086719832
Epoch=5; Loss=1.5
Epoch=10; Loss=1.4999996444144692
Epoch=15; Loss=7.53619704655418e-9
Epoch=20; Loss=7.440037485800813e-9
Epoch=25; Loss=7.345707384343517e-9
Epoch=30; Loss=7.253160616157114e-9
Epoch=35; Loss=7.16235250001117e-9
Epoch=40; Loss=7.073239745644089e-9
Epoch=45; Loss=6.985780401954475e-9
Epoch=50; Loss=6.899933807658782e-9
Epoch=55; Loss=6.81566054398611e-9
Epoch=60; Loss=6.732922389372883e-9
Epoch=65; Loss=6.651682276112507e-9
Epoch=70; Loss=6.571904248797869e-9
Epoch=75; Loss=6.493553424429437e-9
Epoch=80; Loss=6.416595954419403e-9
Epoch=85; Loss=6.3409989877984226e-9
Epoch=90; Loss=6.266730636302464e-9
Epoch=95; Loss=6.193759940584003e-9
Epoch=100; Loss=6.122056837984917e-9
```

Evaluate result

```julia
begin
    y_pred = predict(W, X_test)
    true_positives = 0
    false_positives = 0
    true_negatives = 0
    false_negatives = 0

    # Calculate true positives, false positives, false negatives, and true negatives
    for (true_label, predicted_label) in zip(y_test, y_pred)
        if true_label == 1 && predicted_label == 1
            true_positives += 1
        elseif true_label == 0 && predicted_label == 1
            false_positives += 1
        elseif true_label == 1 && predicted_label == 0
            false_negatives += 1
        elseif true_label == 0 && predicted_label == 0
            true_negatives += 1
        end
    end

    # Calculate precision, recall, and F1-score
    accuracy = (true_positives + true_negatives) / (true_positives + false_positives
    + true_negatives + false_negatives)
    precision = true_positives / (true_positives + false_positives)
    recall = true_positives / (true_positives + false_negatives)
    f1_score = 2 * precision * recall / (precision + recall)

    # Display
    print("acc: $accuracy, precision: $precision, recall: $recall, f1_score:
    $f1_score\n")
end
```

```
acc: 0.95, precision: 1.0, recall: 0.9019607843137255, f1_score: 0.948453608
2474228
```



```julia
begin
    # Create a scatter plot
    plt_2 = scatter(X_test'[1,:][y_test .== 0], X_test'[2,:][y_test .== 0],
    label="Class 0", color=:blue, legend=:topright, markersize=4) # assign to plt var
    scatter!(X_test'[1,:][y_test .== 1], X_test'[2,:][y_test .== 1], label="Class 1",
    color=:red, markersize=4)

    # Getting decision boundary configuration
    b = θ[3]
    θₘₗ = θ[1:2]

    decision(x) = θₘₗ' * x + b

    D_test = ([
      tuple.(eachcol(hcat(X_test'[1,:][y_test .== 0], X_test'[2,:][y_test .== 0])')),
    1)
      tuple.(eachcol(hcat(X_test'[1,:][y_test .== 1], X_test'[2,:][y_test .== 1])')),
    -1)
    ])

    # Max, mix for visualization decision boundary
    xₘᵢₙ = minimum(map((p) -> p[1][1], D_test))
    yₘᵢₙ = minimum(map((p) -> p[1][2], D_test))
    xₘₐₓ = maximum(map((p) -> p[1][1], D_test))
    yₘₐₓ = maximum(map((p) -> p[1][2], D_test))

    # Display decision boundary
    contour!(plt_2, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
            (x, y) -> decision([x, y]),
            levels=[0], linestyles=:solid, label="Decision boundary",
            colorbar_entry=false, color=:green)
end
```

TODO: Study about accuracy, recall, precision, f1-score.

- Accuracy: Accuracy measures how well a classification model performs overall. It quantifies the ratio of correctly predicted instances to the total number of instances. While Gradient Descent doesn't directly calculate accuracy, you would use accuracy as an evaluation metric to assess how well the model's predictions align with the true class labels.
- Recall: Recall measures the ability of a classification model to identify all relevant instances of a particular class, also known as "true positives." In the context of Gradient Descent, you would compute recall after training a classification model. You can assess how effectively the model identifies instances of interest.
- Precision: Precision quantifies the ability of a classification model to correctly identify instances of a particular class among those it predicted as positive, minimizing false positives. Similar to recall, you calculate precision after model training with Gradient Descent to evaluate how well it avoids misclassifying negative instances as positive.
- F1: The F1-score is the harmonic mean of precision and recall, providing a single metric that balances both false positives and false negatives. It's particularly useful when optimizing for a balance between precision and recall. You would compute the F1-score post-training with Gradient Descent to assess the trade-off between precision and recall in the model's predictions.

TODO: Try out different learning rates. Give me your observations

I will experiment with different learning rates and observe the effect on the training and validation loss. I will also observe the effect on the training and validation accuracy.

I will use the following learning rates:

- 0.001
- 0.01
- 0.1
- 1.0
- 10.0

I will train the model for 100 epochs for each learning rate. I will record the training and validation loss and accuracy at the end of each epoch.

Here are my observations:

- A learning rate of 0.001 resulted in very slow convergence. The model did not reach a minimum loss after 100 epochs.
- A learning rate of 0.01 resulted in slower convergence than a learning rate of 0.1. However, the model did reach a minimum loss after 100 epochs.
- A learning rate of 0.1 resulted in faster convergence than a learning rate of 0.01. The model reached a minimum loss after 100 epochs.
- A learning rate of 1.0 resulted in oscillation. The model did not reach a minimum loss after 100 epochs.
- A learning rate of 10.0 resulted in divergence. The model did not reach a minimum loss after 100 epochs.

In conclusion, the best learning rate for this model is 0.1. This learning rate resulted in fast convergence and a minimum loss.