Make Sure to Include all libaries being used. Since this is doign it from scracth, we mainly use numpy for easy mathmatical manipulation and pandas for data

```
import pandas as pd
import numpy as np
```

First We Prove The Loss Function for A logistical Regression

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \left[y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i) \right].$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{w}}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \tilde{\mathbf{w}}}}$$

$$\begin{aligned}
\frac{d\sigma}{dz} &= \frac{(1+e^{-z}) \cdot 0 - 1 \cdot (-e^{-z})}{(1+e^{-z})^2} \\
&= \frac{0 + e^{-z}}{(1+e^{-z})^2} \\
&= \frac{e^{-z}}{(1+e^{-z})^2}
\end{aligned}$$

Multiply numerator and denominator by e^z:

$$\begin{aligned}
&= \frac{e^{-z} \cdot e^z}{(1+e^{-z})^2 \cdot e^z} = \frac{1}{(1+e^{-z})(1+e^{-z}) \cdot e^z} \\
&= \frac{1}{(1+e^{-z})} \cdot \frac{1}{(1+e^{-z}) \cdot e^z} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{1}{e^z + 1} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{1}{1+e^z}
\end{aligned}$$

Note that $\frac{1}{1+e^z} = \frac{e^{-z}}{e^{-z}(1+e^z)} = \frac{e^{-z}}{e^{-z}+1} = 1 - \frac{1}{1+e^{-z}}$:

$$\begin{aligned}
&= \underbrace{\frac{1}{1+e^{-z}}}_{\sigma(z)} \cdot \left(1 - \underbrace{\frac{1}{1+e^{-z}}}_{\sigma(z)} \right) \\
&= \sigma(z)[1 - \sigma(z)]
\end{aligned}$$

$$\begin{aligned}
\frac{dL_i}{dz_i} &= \frac{dL_i}{d\hat{y}_i} \cdot \frac{d\hat{y}_i}{dz_i} \\
&= \left(-\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i} \right) \cdot \hat{y}_i(1-\hat{y}_i) \\
&= -y_i(1-\hat{y}_i) + (1-y_i)\hat{y}_i \\
&= \hat{y}_i - y_i.
\end{aligned}$$

Hence the familiar scalar gradient: $\frac{dL_i}{dz_i} = \hat{y}_i - y_i$.

$$\frac{\partial \mathcal{L}}{\partial \tilde{\mathbf{w}}} = \left(\frac{\partial \mathbf{z}}{\partial \tilde{\mathbf{w}}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{z}}$$

$$= A^T \cdot (\hat{\mathbf{y}} - \mathbf{y})$$

$$\nabla_{\mathbf{w}} \mathcal{L} = A^T (\hat{\mathbf{y}} - \mathbf{y})$$

First We define the sigmoid function and the loss function we are differentiating. The sigmoid function makes an array with the sigmoid function evaluation for each input. Then we define the Loss fucntion given the Binary corss entropy loss shown earlier. This loss is looking at how "suprised" the model is in contrast to the actual prediction.

```
def sigmoid(t):
    def evaluate(x):
        e = np.exp(-x)
        return 1 / (1 + e)
    arra_to_return=[]
    for i in t:
        arra_to_return.append(evaluate(i))
    return np.array(arra_to_return, dtype=np.float64)

def loss(y, yhat):
  # mean loss function for y and yhat = sigmoid(z)
  np_y=np.array(y)
  np_yhat=np.array(yhat)
  return -np.mean(np_y * np.log(np_yhat) + (1 - np_y) * np.log(1 - np_yhat))
```

We want to define the gradient of the loss function in matrix from. For this we have to look at the matrix layout of the gradient. Looking into the Matirx A we see the first column is all ones, then the data points are given

$$A = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1d} \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} \\ 1 & x_{31} & x_{32} & \cdots & x_{3d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & x_{N2} & \cdots & x_{Nd} \end{bmatrix} \in \mathbb{R}^{N \times (d+1)}$$

Now we can define the gradient from what we know so far. Since we know that the

```
def gradients_basic(X, y, y_hat):
    """
    Compute gradients of the loss L(w, b) with respect to

        • w  — dw  (weights, column vector)
        • b  — db  (bias, scalar)

    Layout (row-standard):
        X.shape      == (m_examples × n_features)   # rows = examples
        y.shape      == (m_examples × 1)            # column vector
        y_hat.shape  == (m_examples × 1)

    Works for Binary  cross-entropy
    (logistic regression) where
        dw = (1/m) · Xᵀ(ŷ − y)   and   db = mean(ŷ − y).
    """

    # 1. m ← number of training examples (rows of X)
    m = X.shape[0]

    # 2. residuals r = ŷ − y
    #    Shape: (m × 1)  (column vector)
    residuals = y_hat - y

    # 3. dw = (1/m) · Xᵀ · r
    #    X.T : (n_features × m)
    #    r   : (m × 1)
    #    product -> (n_features × 1)  <- matches w’s shape
    dw = (1 / m) * np.dot(X.T, residuals)

    # 4. db = (1/m) Σ_i (ŷ_i − y_i)
    #    np.sum collapses the vector to a scalar (bias gradient)
    db = (1 / m) * np.sum(residuals)

    return dw, db
    # dw and db are trainable parameters since we need to use gradient decent to
    # figure out the best weights for the final product
```

$i=1$

Problem: The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ is a **transcendental function**. This means:

1. **No Polynomial Form:** Unlike linear regression where the loss is quadratic in $\mathbf{w}$, logistic regression involves exponentials and logarithms.
2. **No Closed-Form Solution:** Setting $\nabla_{\mathbf{w}}\mathcal{L} = 0$ gives us:

$$A^T(\hat{\mathbf{y}} - \mathbf{y}) = \mathbf{0} \tag{34}$$

   But $\hat{\mathbf{y}} = \sigma(A\mathbf{w})$ involves the sigmoid function! This creates a **transcendental equation** that cannot be solved algebraically.

3. **Implicit Dependence:** The predictions $\hat{\mathbf{y}}$ depend on $\mathbf{w}$ through the sigmoid, making the equation $A^T(\sigma(A\mathbf{w}) - \mathbf{y}) = \mathbf{0}$ impossible to solve directly for $\mathbf{w}$.

Enter Gradient Descent

Since we can't solve directly, we use **iterative optimization**:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(t)}) \tag{35}$$

where α is the **learning rate** and our gradient is:

$$\nabla_{\mathbf{w}}\mathcal{L} = A^T(\hat{\mathbf{y}} - \mathbf{y}) \tag{36}$$

10 The Role of the Learning Rate α

What α Controls

The learning rate α determines **how big steps** we take in the direction of the negative gradient:

- **Large α:** Fast convergence, but risk of overshooting the minimum
- **Small α:** Stable convergence, but slow progress
- **Just right α:** Efficient convergence to the global optimum

Why We Need α (Step Size Control)

1. **Gradient gives direction:** $\nabla_{\mathbf{w}}\mathcal{L}$ tells us which way to move
2. **But not how far:** The magnitude of the gradient depends on the scale of our problem
3. **α provides scale:** It converts the gradient direction into an appropriate step size

```
def train_model(X, y, learning_rate=0.01, num_iter=1000):


    # In each interation we want to make predcitons (y_hat), measure loss, L(w,b),
    # comute the gradient (dw = (1/m) · Xᵀ(ŷ − y)) , (db = (1/m) · Σ (ŷ − y))
    # and use Gradient Descent ( w ← w − α·dw ,  b ← b − α·db)
    # to minmize the loss for most accurate results.

    # Initialize parameters
    np.random.seed(42)
    w = np.random.randn(X.shape[1], 1) # weights at a random start
    b = 0.0
    loss_history = [] # loss histroy for loss curve (to see it go down with GD)

    for i in range(num_iter):
        # Forward propagation
        # z = Xw + b (Linear segment)
        # y_hat = σ(z)  (Trancendital segment)
        y_hat = sigmoid(forward_prop(w,b,X))

        # Compute loss in vector form
        loss = loss(y, y_hat)
        loss_history.append(loss)

        # Backward propagation
        dw, db = gradients_basic(X, y, y_hat)

        # Update parameters / Gradient Descent Update
        w -= learning_rate * dw
        b -= learning_rate * db

        # Print loss every 100 iterations
        if i % 100 == 0:
            print(f"Iteration {i}: Loss {loss:.4f}")
```

```
    # returns trained paramets and Learning Curve
    return w, b, loss_history
```

Now we want to Move from just logistic regression Updates, to Neural Networks. This requires us to Solve the gradients for Back Propigation of Nueral Networks

We start with an Input Layer then move to an intial

- **Linear transformation**:

$$a^{(1)} = \boldsymbol{W}^{(1)}\mathbf{x} + \boldsymbol{b}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$

Trainable parameters: $\boldsymbol{W}^{(1)} \in \mathbb{R}^{2\times 2}$ (4 parameters), $\boldsymbol{b}^{(1)} \in \mathbb{R}^{2\times 1}$ (2 parameters)

1

- **Hidden activation**: Using ReLU

$$\mathbf{h}^{(1)} = \mathrm{ReLU}(a^{(1)}) = \begin{bmatrix} \max(0, a_1^{(1)}) \\ \max(0, a_2^{(1)}) \end{bmatrix}$$

- **Output layer**:

$$z = \boldsymbol{w}^T\mathbf{h}^{(1)} + b = [w_1, w_2] \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \end{bmatrix} + b$$

Trainable parameters: $\boldsymbol{w} \in \mathbb{R}^{2\times 1}$ (2 parameters), $b \in \mathbb{R}$ (1 parameter)

- **Sigmoid prediction**: $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} \in (0,1)$.

We can Define H as being the Transformation after the NonLinear Activation Function ReLU

```
def ReLU(t):
  return np.maximum(t, 0)
# derivative of ReLu
def ReLUprime(t):
  return np.where(t > 0, 1, 0)
```

Now We can Define The model of the Nerual Network. We understand that the Model consists of aninput layer that has a linear Transformation subject to 2 trainable parameters (W1 and b1) . Then an acctivation Layer and finally another linear transformation which consits of antoher two trainable parameter variables (w, b). Finally we can output it using sigmoid to either a 0 or 1 class.

```
def model(w, b, W_one, b_one, X):
  # X --> H-by-n Input.
  # z --> model output.
  # w --> output layer weight.
  # b --> output layer bias.
  # W_one --> hidden layer weight.
  # b_one --> hidden layer bias.
    # 1. Hidden affine transform (result: h × N)
    a1 = W_one @ X.T + b_one[:, None]

    # 2. Hidden activation (ReLU keeps same (h, N) shape)
    h1 = ReLU(a1)

    # 3. Output layer (1 × N) then add scalar bias
    z = w @ h1 + b

    return z
```

We know that the gradient of the Loss in respect to Z is $\boxed{\frac{\partial L}{\partial z} = \hat{y} - y}$

Now we have to find the Gradient of each traniable parameter in the Neural Network

Since $z = \boldsymbol{w}^T\mathbf{h}^{(1)} + b$:

$$\frac{\partial z}{\partial b} = \frac{\partial}{\partial b}(\boldsymbol{w}^T\mathbf{h}^{(1)} + b) = 1 \tag{17}$$

Therefore:

$$\begin{aligned} \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b} & (18)\\ &= (\hat{y} - y) \cdot 1 & (19)\\ &= \hat{y} - y & (20) \end{aligned}$$

$$\boxed{\frac{\partial L}{\partial b} = \hat{y} - y} \quad \text{(Gradient for trainable parameter } b\text{)}$$

Since $z = \boldsymbol{w}^T\mathbf{h}^{(1)} + b = w_1h_1^{(1)} + w_2h_2^{(1)} + b$:

$$\begin{aligned} \frac{\partial z}{\partial w_i} &= h_i^{(1)} & (21)\\ \Rightarrow \frac{\partial z}{\partial \boldsymbol{w}} &= \mathbf{h}^{(1)} \quad \text{(as a column vector)} & (22) \end{aligned}$$

Step-by-step:

$$\begin{aligned} \frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_1} = (\hat{y} - y) \cdot h_1^{(1)} & (23)\\ \frac{\partial L}{\partial w_2} &= \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_2} = (\hat{y} - y) \cdot h_2^{(1)} & (24) \end{aligned}$$

In vector form:

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{w}} &= \left(\frac{\partial L}{\partial z}\right)^T \mathbf{h}^{(1)T} & (25)\\ &= (\hat{y} - y) \cdot \mathbf{h}^{(1)} & (26)\\ &= \begin{bmatrix} (\hat{y} - y) \cdot h_1^{(1)} \\ (\hat{y} - y) \cdot h_2^{(1)} \end{bmatrix} & (27) \end{aligned}$$

$$\boxed{\frac{\partial L}{\partial \boldsymbol{w}} = (\hat{y} - y) \cdot \mathbf{h}^{(1)}} \quad \text{(Gradient for trainable parameter } \boldsymbol{w}\text{)}$$

4.2.1 4. Hidden activation gradient (not a parameter, but needed for chain rule)

Since $z = \boldsymbol{w}^T\mathbf{h}^{(1)} + b$:

$$\begin{aligned} \frac{\partial z}{\partial h_i^{(1)}} &= w_i & (28)\\ \Rightarrow \frac{\partial z}{\partial \mathbf{h}^{(1)}} &= \boldsymbol{w} & (29) \end{aligned}$$

Therefore:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{h}^{(1)}} &= \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial \mathbf{h}^{(1)}} & (30)\\ &= (\hat{y} - y) \cdot \boldsymbol{w} & (31)\\ &= \begin{bmatrix} (\hat{y} - y) \cdot w_1 \\ (\hat{y} - y) \cdot w_2 \end{bmatrix} & (32) \end{aligned}$$

$$\boxed{\frac{\partial L}{\partial \mathbf{h}^{(1)}} = \frac{\partial L}{\partial z}\boldsymbol{w}}$$

For ReLU: $h_i^{(1)} = \max(0, a_i^{(1)})$, so:

$$\frac{\partial h_i^{(1)}}{\partial a_i^{(1)}} = g'(a_i^{(1)}) = \begin{cases} 1 & \text{if } a_i^{(1)} > 0 \\ 0 & \text{if } a_i^{(1)} \leq 0 \end{cases}$$

Using element-wise multiplication:

$$\frac{\partial L}{\partial a_1^{(1)}} = \frac{\partial L}{\partial h_1^{(1)}} \cdot g'(a_1^{(1)})$$
$$\frac{\partial L}{\partial a_2^{(1)}} = \frac{\partial L}{\partial h_2^{(1)}} \cdot g'(a_2^{(1)})$$

In vector form:

$$\boxed{\frac{\partial L}{\partial a^{(1)}} = \frac{\partial L}{\partial \mathbf{h}^{(1)}} \odot g'(a^{(1)})^T}$$

4.2.3 6. Gradient for trainable parameter $\boldsymbol{b}^{(1)}$ (first layer bias)

Since $a^{(1)} = \boldsymbol{W}^{(1)}\mathbf{x} + \boldsymbol{b}^{(1)}$:

$$\frac{\partial a_i^{(1)}}{\partial b_j^{(1)}} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

Therefore:

$$\frac{\partial L}{\partial b_1^{(1)}} = \frac{\partial L}{\partial a_1^{(1)}} \cdot 1 + \frac{\partial L}{\partial a_2^{(1)}} \cdot 0 = \frac{\partial L}{\partial a_1^{(1)}}$$
$$\frac{\partial L}{\partial b_2^{(1)}} = \frac{\partial L}{\partial a_1^{(1)}} \cdot 0 + \frac{\partial L}{\partial a_2^{(1)}} \cdot 1 = \frac{\partial L}{\partial a_2^{(1)}}$$

$$\boxed{\frac{\partial L}{\partial \boldsymbol{b}^{(1)}} = \frac{\partial L}{\partial a^{(1)}}} \quad \text{(Gradient for trainable parameter } \boldsymbol{b}^{(1)}\text{)}$$

4.2.4 7. Gradient for trainable parameter $\boldsymbol{W}^{(1)}$ (first layer weights)

Since $a^{(1)} = \boldsymbol{W}^{(1)}\mathbf{x} + \boldsymbol{b}^{(1)}$:

$$a_i^{(1)} = \sum_{k=1}^{2} W_{ik}^{(1)} x_k + b_i^{(1)}$$
$$\Rightarrow \frac{\partial a_i^{(1)}}{\partial W_{jk}^{(1)}} = \begin{cases} x_k & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

Step-by-step for each weight:

$$\frac{\partial L}{\partial W_{11}^{(1)}} = \frac{\partial L}{\partial a_1^{(1)}} \cdot x_1$$
$$\frac{\partial L}{\partial W_{12}^{(1)}} = \frac{\partial L}{\partial a_1^{(1)}} \cdot x_2$$
$$\frac{\partial L}{\partial W_{21}^{(1)}} = \frac{\partial L}{\partial a_2^{(1)}} \cdot x_1$$
$$\frac{\partial L}{\partial W_{22}^{(1)}} = \frac{\partial L}{\partial a_2^{(1)}} \cdot x_2$$

In matrix form:

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{W}^{(1)}} &= \left(\frac{\partial L}{\partial a^{(1)}}\right)^T \mathbf{x}^T \\ &= \begin{bmatrix} \frac{\partial L}{\partial a_1^{(1)}} \\ \frac{\partial L}{\partial a_2^{(1)}} \end{bmatrix} [x_1, x_2] \\ &= \begin{bmatrix} \frac{\partial L}{\partial a_1^{(1)}} \cdot x_1 & \frac{\partial L}{\partial a_1^{(1)}} \cdot x_2 \\ \frac{\partial L}{\partial a_2^{(1)}} \cdot x_1 & \frac{\partial L}{\partial a_2^{(1)}} \cdot x_2 \end{bmatrix}\end{aligned}$$

$$\boxed{\frac{\partial L}{\partial \boldsymbol{W}^{(1)}} = \left(\frac{\partial L}{\partial a^{(1)}}\right)^T \mathbf{x}^T} \quad \text{(Gradient for trainable parameter } \boldsymbol{W}^{(1)}\text{)}$$

```
def gradients(w,b,W_one,b_one,X,y):
  z,h1,a1 = model(w,b,W_one,b_one,X)
  y_hat = sigmoid(z)   #(1 X N)

  dl_dz = y_hat - y[None,:]   # 1 X N ŷ − y

  db = np.mean(dl_dz, axis =1)  # collumn wise / scaler
  dw = np.mean(dl_dz * h1, axis = 1, keepdims=True)   # 1 X h

  dh1 = w.T @ dl_dz

  da1 = dh1 * ReLUprime(a1)    # H X N

  db1 = np.mean(da1,axis=1)  # H X
  dW1 = (da1 @ X) / X.shape[0]  # H X N divide due to taking the average

  return z,dw,db.squeeze(),dW1,db1
```

For a parameter θ and loss L, vanilla GD updates are θ ← θ − η · ∂L/∂θ where • η (lr) is a small positive step size, • ∂L/∂θ is the gradient returned by `gradients(...)

```
def train(w: np.ndarray, b: float, W1: np.ndarray, b1: np.ndarray,
          X: np.ndarray, y: np.ndarray, num_iter: int, lr: float):


    w   = np.asarray(w,  dtype=np.float64)
    b   = np.asarray(b,  dtype=np.float64)        # scalar array
    W1  = np.asarray(W1, dtype=np.float64)
    b1  = np.asarray(b1, dtype=np.float64)

    losses = []

    for i in range(num_iter):
        z,dw,db,dW1,db1 = gradients(w,b,W1,b1,X,y)
        w = w - lr * dw
        b = b - lr * db
        W1 = W1 - lr * dW1
        b1 = b1 - lr * db1

        y_hat = sigmoid(z).squeeze()
        losses.append(loss(y,_hat))

    return w, b, W1, b1, losses
```

$\sigma(z)>0.5 \iff z>0$. Meaning We can simlify the output Its midpoint is at z=Its midpoint is at z = 0 and sigmoid(0) = .5

```
def predict(output):

  pred_class = (output > 0).astype(int)
  return np.squeeze(pred_class)
```

Finally We are Done with Our Prediction Model. In Summary : **A neural network learns to map inputs to outputs through a sequence of transformations—each layer takes the previous layer's output, applies weights and biases (linear transformation), then a non-linear activation function, progressively extracting more complex features until the final layer produces a prediction.**