MACHINE LEARNING LAB

EXERCISE 9

Aim :

1.Implement a neural network from scratch. Take any dataset. Run minimum 200 iterations and get the result. Use the gradient descent optimization technique for weight optimization.

2.For the same dataset, build a neural network using keras library. Run the same number of epochs and compare the results obtained with your model vs the built-in keras mode.

Algorithm :

1. **Load and Preprocess the Dataset**:
   - Choose a dataset suitable for binary classification.
   - Preprocess the dataset by normalizing the features and encoding the labels if necessary.
2. **Initialize the Neural Network Class**:
   - Define a class `NeuralNetwork` with methods for initialization, forward propagation, backward propagation, training, prediction, and evaluation.
3. **Implement the Neural Network from Scratch**:
   - Define the architecture of the neural network including the number of layers, neurons in each layer, activation functions, loss function, and optimization technique (Gradient Descent).
   - Implement methods for weight initialization, activation functions (ReLU and Sigmoid), loss calculation (cross-entropy), and gradient descent optimization.
4. **Train the Neural Network from Scratch**:
   - Initialize an instance of the `NeuralNetwork` class.
   - Split the dataset into training and testing sets.
   - Train the neural network on the training data for a minimum of 200 iterations using gradient descent optimization.
   - Monitor the training loss and accuracy.
5. **Evaluate the Performance of the Scratch Model**:
   - Use the trained model to predict labels for the testing data.
   - Evaluate the accuracy and other performance metrics of the model.
6. **Build and Train a Neural Network using Keras**:
   - Use the Keras library to define a neural network with the same architecture as the one implemented from scratch.
   - Compile the model with appropriate loss function, optimizer (Gradient Descent), and metrics.

- Train the Keras model on the same dataset for the same number of epochs as the scratch model.

7. **Compare the Results**:

- Compare the accuracy and other metrics obtained from both models.

Code and Output Part 1 :

Importing required libraries

In [18]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder, StandardScaler, PolynomialFeatures,
from sklearn.metrics import confusion_matrix
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

In [19]:

```
df=pd.read_csv(r"C:\Users\TEJU\Downloads\osteoporosis.csv")
```

In [20]:

```
df.head()
```

Out[20]:

	Id	Age	Gender	Hormonal Changes	Family History	Race/Ethnicity	Body Weight	Calcium Intake	Vitamin D Intake
0	1734616	69	Female	Normal	Yes	Asian	Underweight	Low	Sufficien
1	1419098	32	Female	Normal	Yes	Asian	Underweight	Low	Sufficien
2	1797916	89	Female	Postmenopausal	No	Caucasian	Normal	Adequate	Sufficien
3	1805337	78	Female	Normal	No	Caucasian	Underweight	Adequate	Insufficien
4	1351334	38	Male	Postmenopausal	Yes	African American	Normal	Low	Sufficien

In [21]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1958 entries, 0 to 1957
Data columns (total 16 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   Id                  1958 non-null   int64 
 1   Age                 1958 non-null   int64 
 2   Gender              1958 non-null   object
 3   Hormonal Changes    1958 non-null   object
 4   Family History      1958 non-null   object
 5   Race/Ethnicity      1958 non-null   object
 6   Body Weight         1958 non-null   object
 7   Calcium Intake      1958 non-null   object
 8   Vitamin D Intake    1958 non-null   object
```

```
 9   Physical Activity    1958 non-null   object
 10  Smoking              1958 non-null   object
 11  Alcohol Consumption  1958 non-null   object
 12  Medical Conditions   1958 non-null   object
 13  Medications          1958 non-null   object
 14  Prior Fractures      1958 non-null   object
 15  Osteoporosis         1958 non-null   int64
dtypes: int64(3), object(13)
memory usage: 244.9+ KB
```

In [22]:

```
df.drop(columns = ['Id'], inplace = True)
```

In [23]:

```
encoder = LabelEncoder()
for col in df.columns[1:-1]:
    df[col] = encoder.fit_transform(df[col].values)

df.head()
```

Out[23]:

	Age	Gender	Hormonal Changes	Family History	Race/Ethnicity	Body Weight	Calcium Intake	Vitamin D Intake	Physical Activity	Smoking
0	69	0	0	1	1	1	1	1	1	1
1	32	0	0	1	1	1	1	1	1	0
2	89	0	1	0	2	0	0	1	0	0
3	78	0	0	0	2	1	0	0	1	1
4	38	1	1	1	0	0	1	1	0	1

In [24]:

```
X=df.drop('Osteoporosis',axis=1)
```

In [25]:

```
y=df['Osteoporosis'].values.reshape(X.shape[0], 1)
```

Splitting into training and testing dataset

In [26]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
```

Standardising

In [27]:

```
sc = StandardScaler()
sc.fit(X_train)
X_train = sc.transform(X_train)
X_test = sc.transform(X_test)
```

Neural Network from scratch

In [28]:

```
class NeuralNetwork():
    '''
    A two layer neural network having
```

```
    - input layer ( 14 nodes)
    - hidden layer (8 nodes)
    - output layer (1 node) (binary classification)
    '''

    def __init__(self, layers=[14,8,1], lr=0.005, epochs=200):
        self.parameters = {}
        self.lr = lr
        self.epochs = epochs
        self.losses = []
        self.sample_size = None
        self.layers = layers
        self.X = None
        self.y = None

    def parameter_init(self):
        '''
        Initialize the weights randomly using numpy
        W1- weights of the nodes in input layer (8,5)
        b1- biases of nodes in hidden layer
        W2- weights of the nodes in hidden layer (5,1)
        b2- biases of nodes in output layer
        '''
        np.random.seed(42) # Seed the random number generator
        self.parameters["W1"] = np.random.randn(self.layers[0], self.layers[1])
        self.parameters['b1']  =np.random.randn(self.layers[1],)
        self.parameters['W2'] = np.random.randn(self.layers[1],self.layers[2])
        self.parameters['b2'] = np.random.randn(self.layers[2],)

    def relu(self,Z):
        '''
        ReLU (Rectified Linear Unit)
        It will return the value passed to it if it is greater than zero;
        otherwise, it returns zero.
        The weighter sum and bias term from the input layer is passed to this activa
        '''
        return np.maximum(0,Z)

    def relu_der(self, x):
        x[x<=0] = 0
        x[x>0] = 1
        return x

    def eta(self, x):
        """
        When our neural network gives 0 value to log, this results in infinity which
        To avoid this, if our value is 0, then it is replaced with an extremely smal
        ETA = 0.00000001
        return np.maximum(x, ETA)


    def sigmoid(self,Z):
        '''
        Sigmoid function
        Take a real number and squashes it to value between 0 and 1.
        '''
        return 1/(1+np.exp(-Z))

    def entropy_loss(self,y, yhat):

        nsample = len(y)
        yhat_inv = 1.0 - yhat
        y_inv = 1.0 - y
        yhat = self.eta(yhat) ## clips value to avoid NaNs in log
```

```
        yhat_inv = self.eta(yhat_inv)
        loss = -1/nsample * (np.sum(np.multiply(np.log(yhat), y) + np.multiply((y_in
        return loss

    def forward_propagation(self):
        #Performs the forward propagation

        Z1 = self.X.dot(self.parameters['W1']) + self.parameters['b1']
        A1 = self.relu(Z1)
        Z2 = A1.dot(self.parameters['W2']) + self.parameters['b2']
        yhat = self.sigmoid(Z2)
        loss = self.entropy_loss(self.y,yhat)

        # save calculated parameters
        self.parameters['Z1'] = Z1
        self.parameters['Z2'] = Z2
        self.parameters['A1'] = A1

        return yhat,loss

    def back_propagation(self,yhat):
        # Computes the derivatives and update weights and bias according.
        y_inv = 1 - self.y
        yhat_inv = 1 - yhat

        dl_wrt_yhat = np.divide(y_inv, self.eta(yhat_inv)) - np.divide(self.y, self.
        dl_wrt_sig = yhat * (yhat_inv)
        dl_wrt_z2 = dl_wrt_yhat * dl_wrt_sig

        dl_wrt_A1 = dl_wrt_z2.dot(self.parameters['W2'].T)
        dl_wrt_w2 = self.parameters['A1'].T.dot(dl_wrt_z2)
        dl_wrt_b2 = np.sum(dl_wrt_z2, axis=0, keepdims=True)

        dl_wrt_z1 = dl_wrt_A1 * self.relu_der(self.parameters['Z1'])
        dl_wrt_w1 = self.X.T.dot(dl_wrt_z1)
        dl_wrt_b1 = np.sum(dl_wrt_z1, axis=0, keepdims=True)

        #gradient descent weight optimisation
        self.parameters['W1'] = self.parameters['W1'] - self.lr * dl_wrt_w1
        self.parameters['W2'] = self.parameters['W2'] - self.lr * dl_wrt_w2
        self.parameters['b1'] = self.parameters['b1'] - self.lr * dl_wrt_b1
        self.parameters['b2'] = self.parameters['b2'] - self.lr * dl_wrt_b2

    def fit(self, X, y):
        '''
        Trains the neural network using the specified data and labels
        '''
        self.X = X
        self.y = y
        self.parameter_init()

        for i in range(self.epochs):
            yhat, loss = self.forward_propagation()
            self.back_propagation(yhat)
            self.losses.append(loss)

    def predict(self, X):
        '''
        Predicts on a test data
        '''
        Z1 = X.dot(self.parameters['W1']) + self.parameters['b1']
        A1 = self.relu(Z1)
        Z2 = A1.dot(self.parameters['W2']) + self.parameters['b2']
        pred = self.sigmoid(Z2)
```

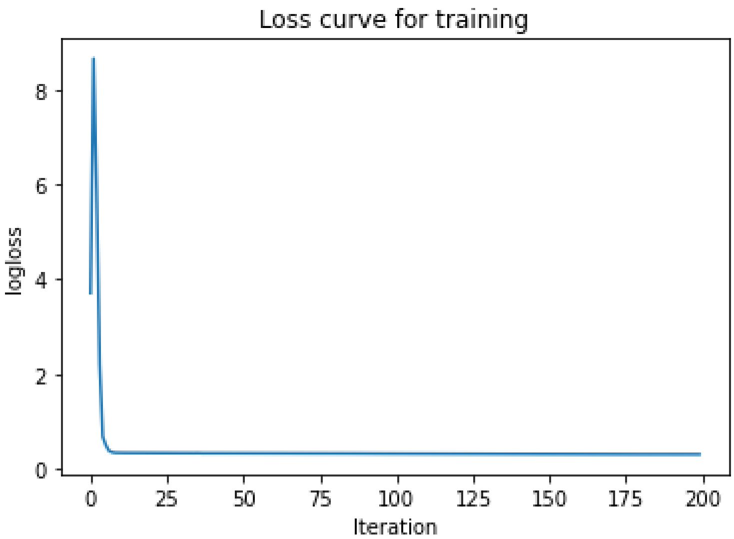
```
        return np.round(pred)

    def acc(self, y, yhat):
        '''
        Calculates the accutacy between the predicted valuea and the truth labels
        '''
        acc = int(sum(y == yhat) / len(y) * 100)
        return acc


    def plot_loss(self):
        '''
        Plots the loss curve
        '''
        plt.plot(self.losses)
        plt.xlabel("Iteration")
        plt.ylabel("logloss")
        plt.title("Loss curve for training")
        plt.show()
```

In [29]:

```
nn=NeuralNetwork()
nn.fit(X_train, y_train)
```

In [30]:

```
nn.plot_loss()
```



In [31]:

```
train_pred = nn.predict(X_train)
test_pred = nn.predict(X_test)

print("Train accuracy is {}".format(nn.acc(y_train, train_pred)))
print("Test accuracy is {}".format(nn.acc(y_test, test_pred)))

conf_matrix = confusion_matrix(y_test ,test_pred)
print(conf_matrix)
```

```
Train accuracy is 87
Test accuracy is 83
[[195  10]
 [ 54 133]]
```

```
C:\Users\TEJU\AppData\Local\Temp/ipykernel_13592/3831387372.py:138: DeprecationWarnin
g: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in
future. Ensure you extract a single element from your array before performing this op
```

```
eration. (Deprecated NumPy 1.25.)
  acc = int(sum(y == yhat) / len(y) * 100)
```

Code and Output Part 2 :

In [1]:
```
import tensorflow as tf
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder, StandardScaler, PolynomialFeatures,
from sklearn.metrics import confusion_matrix
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

```
C:\Users\TEJU\anaconda3\lib\site-packages\scipy\__init__.py:146: UserWarning: A NumPy
version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected version
1.26.4
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

In [2]:
```
df2=pd.read_csv(r"C:\Users\TEJU\Downloads\osteoporosis.csv")
```

In [3]:
```
df2.head()
```

Out[3]:

	Id	Age	Gender	Hormonal Changes	Family History	Race/Ethnicity	Body Weight	Calcium Intake	Vitamin D Intake
0	1734616	69	Female	Normal	Yes	Asian	Underweight	Low	Sufficien
1	1419098	32	Female	Normal	Yes	Asian	Underweight	Low	Sufficien
2	1797916	89	Female	Postmenopausal	No	Caucasian	Normal	Adequate	Sufficien
3	1805337	78	Female	Normal	No	Caucasian	Underweight	Adequate	Insufficien
4	1351334	38	Male	Postmenopausal	Yes	African American	Normal	Low	Sufficien

In [4]:
```
df2.drop(columns = ['Id'], inplace = True)
```

In [5]:
```
encoder = LabelEncoder()
for col in df2.columns[1:-1]:
    df2[col] = encoder.fit_transform(df2[col].values)

df2.head()
```

Out[5]:

	Age	Gender	Hormonal Changes	Family History	Race/Ethnicity	Body Weight	Calcium Intake	Vitamin D Intake	Physical Activity	Smoking
0	69	0	0	1	1	1	1	1	1	1
1	32	0	0	1	1	1	1	1	1	0
2	89	0	1	0	2	0	0	1	0	0

	Age	Gender	Hormonal Changes	Family History	Race/Ethnicity	Body Weight	Calcium Intake	Vitamin D Intake	Physical Activity	Smoking
3	78	0	0	0	2	1	0	0	1	1
4	38	1	1	1	0	0	1	1	0	1

In [6]:
```
X=df2.drop('Osteoporosis',axis=1)
```

In [7]:
```
y=df2['Osteoporosis'].values.reshape(X.shape[0], 1)
```

In [8]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
```

In [9]:
```
sc = StandardScaler()
sc.fit(X_train)
X_train = sc.transform(X_train)
X_test = sc.transform(X_test)
X_train.shape
```

Out[9]:
```
(1566, 14)
```

In [10]:
```
from keras.models import Sequential
from keras.layers import Dense
```

In [11]:
```
ann = Sequential()
ann.add(Dense(units=14, activation='relu',
 input_dim=14))
ann.add(Dense(units=8, activation='relu'))
ann.add(Dense(units=1, activation='sigmoid'))
```

```
C:\Users\TEJU\anaconda3\lib\site-packages\keras\src\layers\core\dense.py:88: UserWarn
ing: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequent
ial models, prefer using an `Input(shape)` object as the first layer in the model ins
tead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [12]:
```
ann.compile(optimizer='sgd', loss=
 'binary_crossentropy',
 metrics=['accuracy'])
```

In [13]:
```
history=ann.fit(X_train,y_train, epochs=200,batch_size=32)
```

```
Epoch 1/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.5249 - loss: 0.7126
Epoch 2/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5610 - loss: 0.6914
Epoch 3/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5914 - loss: 0.6661
Epoch 4/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6142 - loss: 0.6481
```

```
Epoch 5/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6667 - loss: 0.6284
Epoch 6/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6660 - loss: 0.6178
Epoch 7/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6877 - loss: 0.6088
Epoch 8/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6903 - loss: 0.5947
Epoch 9/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7182 - loss: 0.5809
Epoch 10/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7395 - loss: 0.5589
Epoch 11/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7550 - loss: 0.5309
Epoch 12/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7482 - loss: 0.5360
Epoch 13/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7771 - loss: 0.5028
Epoch 14/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7773 - loss: 0.4994
Epoch 15/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8030 - loss: 0.4842
Epoch 16/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8013 - loss: 0.4720
Epoch 17/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7931 - loss: 0.4711
Epoch 18/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8089 - loss: 0.4447
Epoch 19/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8307 - loss: 0.4267
Epoch 20/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8092 - loss: 0.4421
Epoch 21/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8222 - loss: 0.4193
Epoch 22/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8231 - loss: 0.4119
Epoch 23/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8264 - loss: 0.4101
Epoch 24/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8367 - loss: 0.3805
Epoch 25/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8392 - loss: 0.3922
Epoch 26/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8435 - loss: 0.3820
Epoch 27/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8406 - loss: 0.3851
Epoch 28/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8287 - loss: 0.3764
Epoch 29/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8314 - loss: 0.3826
Epoch 30/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8343 - loss: 0.3873
Epoch 31/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8405 - loss: 0.3692
Epoch 32/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8256 - loss: 0.3884
Epoch 33/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8524 - loss: 0.3562
Epoch 34/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8442 - loss: 0.3598
Epoch 35/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8417 - loss: 0.3674
Epoch 36/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8211 - loss: 0.3987
```

```
Epoch 37/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8501 - loss: 0.3711
Epoch 38/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8423 - loss: 0.3674
Epoch 39/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8381 - loss: 0.3697
Epoch 40/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8339 - loss: 0.3577
Epoch 41/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8382 - loss: 0.3672
Epoch 42/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8625 - loss: 0.3250
Epoch 43/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.8436 - loss: 0.3622
Epoch 44/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8509 - loss: 0.3465
Epoch 45/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8455 - loss: 0.3514
Epoch 46/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8339 - loss: 0.3615
Epoch 47/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8488 - loss: 0.3500
Epoch 48/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8557 - loss: 0.3401
Epoch 49/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8474 - loss: 0.3518
Epoch 50/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8419 - loss: 0.3601
Epoch 51/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8516 - loss: 0.3503
Epoch 52/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8522 - loss: 0.3502
Epoch 53/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8326 - loss: 0.3867
Epoch 54/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8524 - loss: 0.3569
Epoch 55/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8474 - loss: 0.3465
Epoch 56/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8319 - loss: 0.3676
Epoch 57/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 991us/step - accuracy: 0.8497 - loss: 0.3552
Epoch 58/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8385 - loss: 0.3629
Epoch 59/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8446 - loss: 0.3462
Epoch 60/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8435 - loss: 0.3500
Epoch 61/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8394 - loss: 0.3681
Epoch 62/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8548 - loss: 0.3443
Epoch 63/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8499 - loss: 0.3458
Epoch 64/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8498 - loss: 0.3409
Epoch 65/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8535 - loss: 0.3362
Epoch 66/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8563 - loss: 0.3386
Epoch 67/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8368 - loss: 0.3595
Epoch 68/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 960us/step - accuracy: 0.8525 - loss: 0.3371
```

```
Epoch 69/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 958us/step - accuracy: 0.8569 - loss: 0.3292
Epoch 70/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8527 - loss: 0.3434
Epoch 71/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8418 - loss: 0.3468
Epoch 72/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8375 - loss: 0.3410
Epoch 73/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8577 - loss: 0.3233
Epoch 74/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8369 - loss: 0.3557
Epoch 75/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8532 - loss: 0.3466
Epoch 76/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8337 - loss: 0.3559
Epoch 77/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8486 - loss: 0.3381
Epoch 78/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8401 - loss: 0.3418
Epoch 79/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8548 - loss: 0.3249
Epoch 80/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8376 - loss: 0.3396
Epoch 81/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8571 - loss: 0.3352
Epoch 82/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8459 - loss: 0.3364
Epoch 83/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8470 - loss: 0.3297
Epoch 84/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8418 - loss: 0.3456
Epoch 85/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8386 - loss: 0.3414
Epoch 86/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8583 - loss: 0.3206
Epoch 87/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8504 - loss: 0.3364
Epoch 88/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8492 - loss: 0.3418
Epoch 89/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8407 - loss: 0.3447
Epoch 90/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8420 - loss: 0.3400
Epoch 91/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8472 - loss: 0.3417
Epoch 92/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8373 - loss: 0.3445
Epoch 93/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8414 - loss: 0.3333
Epoch 94/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8609 - loss: 0.3184
Epoch 95/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8516 - loss: 0.3292
Epoch 96/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8575 - loss: 0.3259
Epoch 97/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8511 - loss: 0.3282
Epoch 98/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8389 - loss: 0.3490
Epoch 99/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8478 - loss: 0.3349
Epoch 100/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8614 - loss: 0.3133
```

```
Epoch 101/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8629 - loss: 0.3130
Epoch 102/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8327 - loss: 0.3569
Epoch 103/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8473 - loss: 0.3266
Epoch 104/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8377 - loss: 0.3268
Epoch 105/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8503 - loss: 0.3294
Epoch 106/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8455 - loss: 0.3301
Epoch 107/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8482 - loss: 0.3415
Epoch 108/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8520 - loss: 0.3337
Epoch 109/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8365 - loss: 0.3495
Epoch 110/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8376 - loss: 0.3358
Epoch 111/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8421 - loss: 0.3382
Epoch 112/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8467 - loss: 0.3276
Epoch 113/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8277 - loss: 0.3609
Epoch 114/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8369 - loss: 0.3438
Epoch 115/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8549 - loss: 0.3124
Epoch 116/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8418 - loss: 0.3505
Epoch 117/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8562 - loss: 0.3172
Epoch 118/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8477 - loss: 0.3333
Epoch 119/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8420 - loss: 0.3436
Epoch 120/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8500 - loss: 0.3332
Epoch 121/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8438 - loss: 0.3425
Epoch 122/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8550 - loss: 0.3288
Epoch 123/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8584 - loss: 0.3241
Epoch 124/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8539 - loss: 0.3181
Epoch 125/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8460 - loss: 0.3312
Epoch 126/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8443 - loss: 0.3339
Epoch 127/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8503 - loss: 0.3351
Epoch 128/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8443 - loss: 0.3335
Epoch 129/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 998us/step - accuracy: 0.8315 - loss: 0.3534
Epoch 130/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8362 - loss: 0.3532
Epoch 131/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8171 - loss: 0.3723
Epoch 132/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8571 - loss: 0.3127
```

```
Epoch 133/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8358 - loss: 0.3416
Epoch 134/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8530 - loss: 0.3237
Epoch 135/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8496 - loss: 0.3331
Epoch 136/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8479 - loss: 0.3252
Epoch 137/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8505 - loss: 0.3224
Epoch 138/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8559 - loss: 0.3217
Epoch 139/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8568 - loss: 0.3148
Epoch 140/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8508 - loss: 0.3410
Epoch 141/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8532 - loss: 0.3133
Epoch 142/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8521 - loss: 0.3312
Epoch 143/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8362 - loss: 0.3405
Epoch 144/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8591 - loss: 0.3252
Epoch 145/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8526 - loss: 0.3215
Epoch 146/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8543 - loss: 0.3271
Epoch 147/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8459 - loss: 0.3382
Epoch 148/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8534 - loss: 0.3224
Epoch 149/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8468 - loss: 0.3374
Epoch 150/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8559 - loss: 0.3210
Epoch 151/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8600 - loss: 0.3180
Epoch 152/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8595 - loss: 0.3154
Epoch 153/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8501 - loss: 0.3293
Epoch 154/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8462 - loss: 0.3166
Epoch 155/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8621 - loss: 0.3244
Epoch 156/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8700 - loss: 0.3046
Epoch 157/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8512 - loss: 0.3315
Epoch 158/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8670 - loss: 0.3112
Epoch 159/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8604 - loss: 0.3142
Epoch 160/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8561 - loss: 0.3110
Epoch 161/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8690 - loss: 0.2958
Epoch 162/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8597 - loss: 0.3026
Epoch 163/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8485 - loss: 0.3241
Epoch 164/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8476 - loss: 0.3233
```

```
Epoch 165/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8535 - loss: 0.3202
Epoch 166/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8564 - loss: 0.3110
Epoch 167/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8492 - loss: 0.3217
Epoch 168/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8650 - loss: 0.3058
Epoch 169/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8687 - loss: 0.3032
Epoch 170/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8483 - loss: 0.3276
Epoch 171/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8600 - loss: 0.3203
Epoch 172/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8600 - loss: 0.3180
Epoch 173/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8607 - loss: 0.3132
Epoch 174/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8506 - loss: 0.3251
Epoch 175/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8618 - loss: 0.3137
Epoch 176/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8703 - loss: 0.3100
Epoch 177/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8622 - loss: 0.3133
Epoch 178/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8551 - loss: 0.3290
Epoch 179/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8692 - loss: 0.3011
Epoch 180/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8540 - loss: 0.3254
Epoch 181/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8479 - loss: 0.3269
Epoch 182/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8646 - loss: 0.3143
Epoch 183/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8635 - loss: 0.3046
Epoch 184/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8633 - loss: 0.3139
Epoch 185/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8807 - loss: 0.2946
Epoch 186/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8484 - loss: 0.3233
Epoch 187/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8628 - loss: 0.3107
Epoch 188/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8579 - loss: 0.3214
Epoch 189/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8617 - loss: 0.3110
Epoch 190/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8505 - loss: 0.3171
Epoch 191/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8712 - loss: 0.2954
Epoch 192/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8589 - loss: 0.3206
Epoch 193/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8559 - loss: 0.3172
Epoch 194/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8578 - loss: 0.3158
Epoch 195/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8496 - loss: 0.3260
Epoch 196/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8558 - loss: 0.3242
```

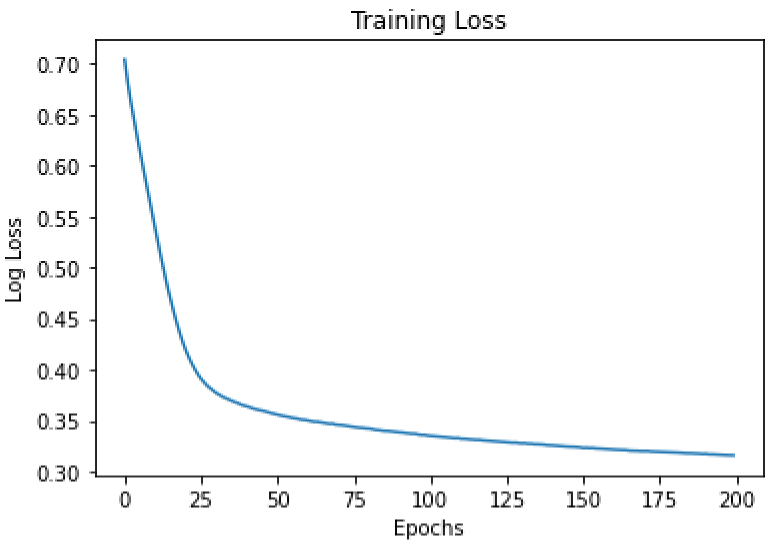
```
Epoch 197/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8640 - loss: 0.3089
Epoch 198/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8619 - loss: 0.3086
Epoch 199/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8643 - loss: 0.3085
Epoch 200/200
49/49 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8594 - loss: 0.3185
```

In [14]:

```
loss = history.history['loss']
```

In [15]:

```
plt.plot(loss)
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Log Loss')
plt.show()
```



In [16]:

```
losstest, accuracy = ann.evaluate(X_test, y_test,verbose=0)
```

In [17]:

```
print("Test Loss:", losstest)
print("Test Accuracy:", accuracy)
```

```
Test Loss: 0.3878213167190552
Test Accuracy: 0.8214285969734192
```

Comparative Analysis :

Neural Networks implemented from scratch and using the keras library :

- Used the same number of nodes in input, hidden and output layer (14,8,1)
- Used the same gradient descent optimisation for weights

From scratch :

- Train set accuracy - 87
- Test set accuracy - 83

Using keras (tensorflow) library :

- Train set accuracy - 85.9

- Test set accuracy - 82.1

Based on the provided data comparing the neural network implemented from scratch and using the Keras library:

1. **Performance Consistency**:

    - The accuracy achieved by the scratch implementation on both the train and test sets (87% and 83% respectively) is slightly higher than that achieved by the Keras implementation (85.9% and 82.1% respectively).
    - This consistency in performance suggests that the scratch implementation is effective in learning the underlying patterns in the data and generalizing well to unseen samples, comparable to the performance of the Keras library.

2. **Implementation Complexity vs. Performance**:

    - The scratch implementation requires more effort in terms of coding and understanding the underlying algorithms, but it achieves similar performance to the Keras library.
    - While the Keras library offers convenience and abstraction, allowing for faster prototyping and implementation, the scratch implementation provides a deeper understanding of neural network concepts and greater flexibility in customization.
    - Therefore, the choice between the two approaches depends on factors such as the trade-off between implementation complexity and performance consistency, as well as the specific requirements of the project.

Conclusion: In this particular scenario, the neural network implemented from scratch yielded better results. But it is possible that on improving the parameters (like using adam instead of sgd for optimisation and so on) could make the keras library neural network yield better results.

Results :

Therefore, we were successfully able to implement a neural network using the tensorflow (keras) library as well as from scratch and draw insights from the same.