Project Report: Titanic Survival Prediction

Objective

The goal of this project is to predict passenger survival in the Titanic disaster using machine learning and neural networks. The problem is framed as a binary classification task where survival (`Survived=1`) or death (`Survived=0`) is predicted based on passenger features such as class, age, sex, and fare.

Data Preprocessing

1. Data Cleaning

- **Dropped Columns**:
  - The `Cabin` column was removed due to excessive missing values.
- **Handling Missing Values**:
  - `Age`: Missing values were filled with the column mean.
  - `Embarked`: Missing values were replaced with the mode.
  - `Fare`: Missing values were filled with the mean (in the SQLite dataset).

2. Feature Encoding

- Categorical variables (`Sex`, `Embarked`, `Pclass`) were converted to numerical values using one-hot encoding and mapping.
  - **Sex**: Male = 0, Female = 1
  - **Embarked**: S = 0, C = 1, Q = 2
- Additional encoding was performed using one-hot coding for features like `Embarked` and `Sex` in the SQLite dataset.

3. Feature Selection

- Dropped irrelevant columns such as `PassengerId`, `Name`, `Ticket`, and other redundant columns.

4. Data Splitting

- The dataset was split into training and testing subsets in an 80:20 ratio.

Exploratory Data Analysis (EDA)

- **Missing Data Visualization**:
  - Bar charts highlighted the percentage of missing data across features.
- **Summary Statistics**:
  - Key descriptive statistics for each column, including mean, median, and standard deviation, were calculated.

Machine Learning Models

1. Logistic Regression

- A baseline model to evaluate performance with linear assumptions.
- Achieved reasonable performance metrics but was less flexible compared to advanced models.

2. Random Forest Classifier

- Utilized for comparative performance evaluation.
- Effective in capturing non-linear relationships and provided feature importance insights.

Deep Learning Model

Neural Network from Scratch Explanation

The "neural network from scratch" section implements a simple feedforward neural network with one hidden layer. Here's a step-by-step breakdown of its components:

---

1. Defining the Network Structure (`layer_sizes`)

This function calculates the sizes of:

- **Input Layer (`n_x`)**: Corresponds to the number of features in the dataset (e.g., `Pclass`, `Age`, etc.).
- **Hidden Layer (`n_h`)**: Defined with 4 neurons in this implementation.
- **Output Layer (`n_y`)**: Single neuron for binary classification (survived or not).

---

2. Parameter Initialization (`initialize_parameters`)

This function initializes the network parameters:

- **Weights (`W1` and `W2`)**:
  - `W1`: Matrix for weights between the input layer and the hidden layer.
  - `W2`: Matrix for weights between the hidden layer and the output layer.
  - Initialized randomly with small values to break symmetry and facilitate learning.
- **Biases (`b1` and `b2`)**:
  - Initialized to zeros.
- Small initial weights help the network converge faster during gradient descent.

---

3. Forward Propagation (`forward_propagation`)

Forward propagation computes the output of the network for given inputs:

- **Input to Hidden Layer**:
  - Z1=W1·X+b1Z1 = W1 \cdot X + b1Z1=W1·X+b1
  - A1=tanh⁡(Z1)A1 = \tanh(Z1)A1=tanh(Z1)
    - **Why `tanh`?**
      - It outputs values between -1 and 1, which helps center data around zero, improving convergence.
      - Useful for non-linear transformations, allowing the model to learn complex patterns.
- **Hidden to Output Layer**:
  - Z2=W2·A1+b2Z2 = W2 \cdot A1 + b2Z2=W2·A1+b2
  - A2=σ(Z2)A2 = \sigma(Z2)A2=σ(Z2)
    - **Why `sigmoid`?**
      - Sigmoid activation outputs values between 0 and 1, making it ideal for binary classification tasks.
      - It directly interprets as a probability (e.g., probability of survival).
- The outputs A2A2A2 represent the predicted probabilities.

---

4. Cost Calculation (`compute_cost`)

The cost function quantifies the difference between predictions and actual outcomes:

- Binary Cross-Entropy Loss: Cost=−1m∑[Y·log⁡(A2)+(1−Y)·log⁡(1−A2)]\text{Cost} = - \frac{1}{m} \sum \left[Y \cdot \log(A2) + (1 - Y) \cdot \log(1 - A2) \right]Cost=−m1 ∑[Y·log(A2)+(1−Y)·log(1−A2)]
  - This is ideal for binary classification as it penalizes incorrect predictions more heavily when the model is confident but wrong.

5. Backward Propagation (`backward_propagation`)

Backward propagation calculates the gradients of the cost with respect to the parameters:

- Gradients:
  - dZ2=A2−YdZ2 = A2 - YdZ2=A2−Y: Error at the output layer.
  - dW2,db2dW2, db2dW2,db2: Gradients for the second layer.
  - dZ1dZ1dZ1: Error at the hidden layer, using the derivative of tanh\tanhtanh.
  - dW1,db1dW1, db1dW1,db1: Gradients for the first layer.
- These gradients are used to update parameters.

---

6. Parameter Updates (`update_parameters`)

Parameters are updated using gradient descent:

- W=W−learning rate×dWW = W - \text{learning rate} \times dWW=W−learning rate×dW
- Biases are updated similarly.
- The learning rate controls the step size for updates to minimize the cost function.

---

7. Training the Model (`nn_model`)

- The model trains over several iterations:
  - Forward propagation computes predictions.
  - Backward propagation calculates gradients.
  - Parameters are updated.
- Every 1,000 iterations, the cost is printed to monitor progress.

---

8. Predictions (`predict`)

- The predictions are made using forward propagation:
  - If A2>0.5A2 > 0.5A2>0.5, the output is classified as 1 (survived); otherwise, 0.

Activation Functions Used and Their Role

1. **`tanh` Activation Function (Hidden Layer)**:
    - Formula: tanh⁡(z)=ez−e−zez+e−z\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}tanh(z)=ez+e−zez−e−z
    - **Range**: [-1, 1]
    - **Advantages**:
        - Zero-centered output accelerates convergence during gradient descent.
        - Suitable for non-linear transformations, allowing the network to capture complex relationships in the data.
    - **Why Not ReLU?**
        - While ReLU is popular, tanh⁡\tanhtanh was chosen here due to its bounded nature, which can prevent excessively large activations during training.
2. **Sigmoid Activation Function (Output Layer)**:
    - Formula: σ(z)=11+e−z\sigma(z) = \frac{1}{1 + e^{-z}}σ(z)=1+e−z1
    - **Range**: [0, 1]
    - **Advantages**:
        - Outputs probabilities, making it suitable for binary classification.
        - Intuitive interpretation: Values close to 1 indicate survival, and values close to 0 indicate non-survival.
    - **Drawbacks**:
        - Can suffer from vanishing gradients, but this is less of an issue in small networks with few layers.

Why we used These Functions

- The combination of `tanh` and `sigmoid` is effective for small networks and binary classification tasks.
- `tanh` enables learning non-linear features, while `sigmoid` translates the output into probabilities for binary decision-making. Together, they allow the model to handle the complexity of the Titanic dataset while ensuring interpretable outputs.

Further Explanation of Activation Functions

1. Why `tanh` in the Hidden Layer?

- **Properties**: The **tanh** (hyperbolic tangent) function squashes its output between -1 and 1. Unlike the **sigmoid** function, which outputs values between 0 and 1, the range of **tanh** is centered around zero. This is crucial for the following reasons:
    - **Centering the Output**: The fact that `tanh` produces both positive and negative values ensures that the network's activations are centered around zero. When activations are centered, the learning process typically converges faster, as the gradients are less likely to become excessively large or small. This helps with the stability of training.

- **Preventing Bias**: Since the values output by `tanh` are distributed around zero, it helps the network learn and update weights more effectively during backpropagation. This is especially useful in deep networks where the magnitude of weight updates can either explode (too large) or vanish (too small) as layers increase. A non-zero-centered activation function like **ReLU** can sometimes lead to slower convergence in certain scenarios, particularly with deeper models.
- **Gradient Flow**: In backpropagation, **tanh** has a derivative that ensures smooth gradient flow across layers, which helps maintain effective updates during training. However, this can still lead to the **vanishing gradient problem** for very large or small inputs, which is why **ReLU** is often preferred in deeper architectures. But, in shallow networks with one hidden layer, **tanh** can still perform quite well.

2. Why `sigmoid` in the Output Layer?

- **Properties**: The **sigmoid** activation function is the most common choice for binary classification tasks because its output is constrained between 0 and 1, making it ideal for representing probabilities:
  - **Interpretability**: The output of the **sigmoid** function can be interpreted as a probability. For binary classification tasks like predicting survival on the Titanic, a **1** (survived) corresponds to a probability closer to 1, while a **0** (perished) corresponds to a probability closer to 0.
  - **Thresholding**: Typically, values above 0.5 are classified as "1" (survived), and those below 0.5 are classified as "0" (did not survive). This threshold is easy to set and makes the model output interpretable in terms of likelihood.
  - **Binary Cross-Entropy Loss**: The **sigmoid** function is well-suited for use with **binary cross-entropy loss**. This loss function is derived from the likelihood of the correct classification and is optimized when the model's output is probabilistic. Specifically, it penalizes the model more when it is highly confident but wrong, and less when it is less confident.

Training a Neural Network - The Backpropagation Process

When you train a neural network, the objective is to minimize the **cost function**, which represents how far the model's predictions are from the true labels. This process involves two main steps: **forward propagation** and **backward propagation**.

1. Forward Propagation

In **forward propagation**, data passes through the network layers:

- **Input Layer**: The raw features (e.g., age, gender, fare, etc.) are passed into the network.
- **Hidden Layer**: These inputs are transformed by the **tanh** activation function, introducing non-linearity.
- **Output Layer**: The final transformed result passes through the **sigmoid** function, producing a probability that a passenger survived.

2. Backward Propagation

In **backward propagation**, we compute how much each weight contributed to the error:

- **Error**: The difference between the predicted output A2A2A2 and the actual label YYY is the error. This error is propagated backward through the network to calculate the gradients (partial derivatives) of the cost function with respect to each weight and bias.
- **Gradient Computation**:
    - Gradients are computed for each weight and bias using the chain rule of calculus. The key idea is to determine how much each parameter (weight or bias) needs to change to reduce the overall error.
    - **Hidden Layer Gradients**: To calculate gradients for the hidden layer, we also use the derivative of the **tanh** function, which ensures smooth flow of information backward through the network.
- **Parameter Update**: Once gradients are computed, the parameters (weights and biases) are updated using gradient descent or its variants (e.g., stochastic gradient descent). The weights are updated in such a way that the loss function is minimized over time.

Neural Network Optimization and Activation Function Impact

Gradient Descent and Activation Functions

The optimization of the neural network is achieved through gradient descent, where we adjust the weights to minimize the loss function:

- **Learning Rate**: Determines how large the steps are during parameter updates.
- **Activation Functions Impact**: The choice of activation function affects the efficiency of gradient descent:
    - **tanh** is smooth and differentiable, which ensures stable updates for small networks.
    - **sigmoid** also allows for gradients, but it can suffer from the **vanishing gradient problem** if the outputs are near 0 or 1, especially when using deep networks. However, for a shallow network like the one in this case, **sigmoid** performs adequately.

Vanishing Gradient Problem

- **Cause**: When the input to an activation function (like **tanh** or **sigmoid**) is too large or too small, the gradient can become extremely small. This is particularly problematic for deeper networks, where gradients diminish as they propagate backward through the layers, making it harder to train.
- **Solution in Deep Networks**: In deeper networks, the **ReLU** activation function is often preferred due to its non-saturating nature (gradients are constant for positive inputs). However, this is not as problematic for shallow networks with just one hidden layer.

Summary of Key Concepts

- **tanh** in the hidden layer is used because it introduces non-linearity and centers data, which helps with faster convergence and stability during training.
- **sigmoid** in the output layer is used for binary classification, as it provides outputs in the range of 0 to 1, making it interpretable as probabilities.
- **Backpropagation** uses the chain rule to compute gradients and updates the parameters (weights and biases) to reduce the cost.
- The network architecture with one hidden layer and simple activations works well for this binary classification problem, although more complex networks might benefit from deeper architectures and more advanced activations like **ReLU** or **Leaky ReLU**.

This framework provides a solid foundation for learning and applying neural networks, especially in binary classification tasks like predicting Titanic survival!