Project -2

AI-681

Machine Learning & Pattern Recognition

Red Wine Quality Prediction Model

Guided by:- Dr. Kewei Li

Submitted by:- Kartavya Mandora

LIU_ID:- 100876605

Abstract

This project presents the development of a simple feed-forward neural network, implemented from scratch, to predict the quality of red wine based on its chemical properties. Utilizing the Wine Quality dataset from the UCI Machine Learning Repository, the network was trained to classify wine quality scores ranging from 3 to 8 using features such as acidity, residual sugar, sulfur dioxide, and alcohol content.

The model was constructed using Python and NumPy, with two hidden layers and activation functions to introduce non-linearity. Gradient descent was applied to iteratively optimize the network's weights.

The final model achieved classification accuracy above 50% on the test set, indicating effective learning of the underlying patterns within a relatively small dataset. This work underscores the capability of handcrafted neural networks to perform meaningful classification tasks and highlights the importance of architectural choices such as layer sizes and activation functions in model performance.

Table of Content

List of Figures

1. Introduction

Building intelligent systems from scratch level is a valued talent and a deeper learning opportunity in an era where machine learning is becoming more and more integrated into decision-making processes. In order to address the practical issue of predicting wine quality based on quantifiable chemical properties, this research investigates building a neural network from scratch.

The UCI Machine Learning Repository provided the dataset, which includes the chemical characteristics of red wine samples. Although the issue can initially appear subjective, there are quantifiable trends in wine chemistry that might affect how good a wine is regarded. However, these patterns are frequently complicated and non-linear, which makes them a perfect fit for neural network modeling.

This project focuses on implementing a multi-class classification model using a feed-forward neural network trained with gradient descent. The primary goal is to not only achieve reasonable predictive performance but also to gain insight into how various design choices (such as the number of hidden neurons, activation functions, and data pre-processing) affect learning outcomes. Beyond just building a working model, this project offers a hands-on understanding of how neural networks process information, learn from data, and generalize to new examples.

2. Dataset

The dataset used in this project is the **Wine Quality Dataset**, sourced from the **UCI Machine Learning Repository**. It contains a collection of **red wine samples**, each evaluated based on several **chemical properties** and rated for quality by human experts on a scale from **3 to 8**.

For this project, a subset of **300 samples** was randomly selected to simplify training and reduce computational overhead. The dataset includes **seven numerical features** that describe each wine sample's chemical composition (Fixed Acidity, Volatile Acidity, Residual Sugar, Total Sulphur Dioxide, Sulphates, pH, Alcohol)

These features are known to influence how wine is perceived in terms of taste, aroma, and balance. The **target variable** is the wine's **quality score**, which was converted into a classification label by shifting the range to start from 0 (i.e., quality 3 becomes class 0, and so on up to quality 8 becoming class 5). This setup allowed the model to treat the problem as a **multi-class classification task** with **six possible classes**.

Before training, all input features were **normalized** to have zero mean and unit variance to ensure that the model could learn efficiently without being biased toward larger-valued features. The dataset was then split into **training (80%) and testing (20%)** subsets to evaluate how well the model could generalize to **random data**.

2.1. Activation function

2.1.1. Step Function:

This doesn't work well with gradient-descent method. It is replaced by ReLU, softmax, and tanh activation functions.

$$\mathbf{Step(z)} = \begin{cases} \mathbf{0} & \mathbf{if\ z < 0} \\ \mathbf{1} & \mathbf{otherwise} \end{cases}$$

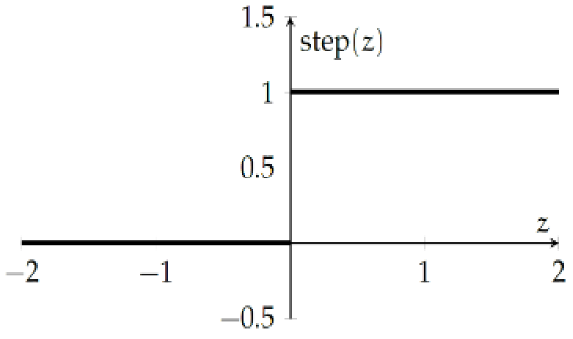


Figure 1: Step Function

2.1.2. Rectified Linear Unit (ReLU) function:

$$\mathbf{ReLU(z)} = \begin{cases} \mathbf{0} & \mathbf{if\ z < 0} \\ \mathbf{z} & \mathbf{otherwise} \end{cases}$$

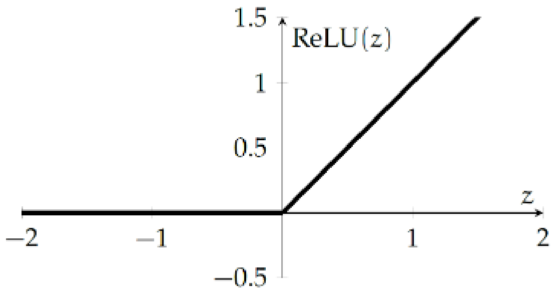


Figure 2: ReLU function

2.1.3. Sigmoid or logistic Function:

This can sometimes be interpreted as probability, because for any value of *z* the output is in (0, 1). This function commonly used for binary classification:

$$\boldsymbol{\sigma}(\mathbf{z}) = \frac{\mathbf{1}}{\mathbf{1} + \mathbf{e}^{-\mathbf{z}}}$$

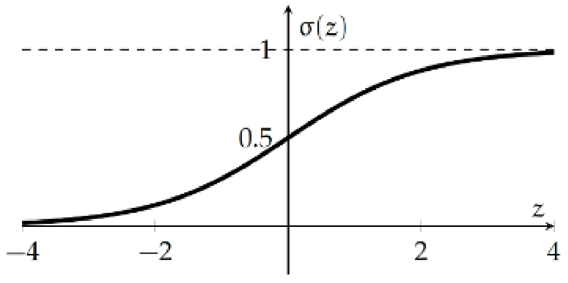


Figure 3: Sigmoid or logistic Function

2.1.4. Hyperbolic tangent:

Always in the range of (-1,1) **[Used in the Code]:**

$$\mathbf{tanh(z)} = \frac{\mathbf{e^{z} - e^{-z}}}{\mathbf{e^{z} + e^{-z}}}$$

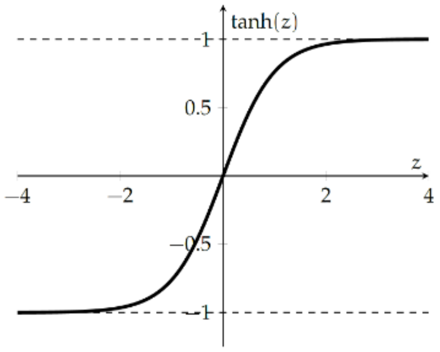


Figure 4: Hyperbolic Tangent Function

2.2.5. Softmax Function:

Takes a whole vector z ϵ R^n and generates as output a vector a ϵ R^n. This function commonly used for multi-classification. **[Used in the code]**

$$\mathbf{a = softmax(z)} = \begin{bmatrix} \frac{\exp(z_1)}{s} \\ \frac{\exp(z_2)}{s} \\ \vdots \\ \frac{\exp(z_n)}{s} \end{bmatrix}, \quad \mathbf{where}\ \ s = \sum_{i=1}^{n} \exp(z_i)$$

2.2. Gradient Descent

Gradient descent is an optimization algorithm used in machine learning to find the minimum of a function (often a cost function) by iteratively moving in the direction of the steepest descent, as defined by the negative gradient.

To calculate the gradient:-

$$\nabla_{\theta} J = \frac{1}{n} \sum_{i=1}^{n} \left(\left[\theta^{(t-1)}\right]^{T} x^{i} - y^{i} \right) x^{i}$$

3. Explanation of Neural Network Algorithm

The neural network implemented for this project is a **fully connected feed-forward neural network**, designed to perform multi-class classification on the wine quality dataset. The architecture consists of **three layers** beyond the input:

1. **Input Layer**
   The input layer receives the normalized feature vector for each wine sample. Since the dataset includes **7 features**, the input layer has **7 nodes**.
2. **First Hidden Layer**
   This layer introduces non-linearity and learns intermediate representations of the input data. It contains **16 neurons**, each using an activation function (ReLU or Sigmoid, depending on configuration) to transform inputs before passing them to the next layer.
3. **Second Hidden Layer**
   Serving as another transformation stage, this layer has **12 neurons**. The additional layer depth allows the network to capture more complex feature interactions.
4. **Output Layer**
   The final layer has **6 nodes**, corresponding to the 6 possible wine quality classes (labels ranging from 0 to 5). A **softmax** activation function is applied here to produce probability distributions over the classes.

The network is trained using the **gradient descent optimization algorithm**, where weights and biases are updated iteratively to minimize the **cross-entropy loss** between the predicted and actual class labels. During each epoch, forward propagation is used to generate predictions, followed by **backpropagation** to compute gradients and update parameters. The program shown below:

Activation Function:

```
def tanh(z):
    return np.tanh(z)
```

Derivatives of activation functions:

```
def tanh_derivative(z):
    return 1 - np.tanh(z) ** 2

def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

Cross Entropy loss:

```
def cross_entropy(y_true, y_pred):
    return -np.sum(y_true * np.log(y_pred + 1e-9)) / y_true.shape[0]
```

Initialization of weights:

```
input_size = X_train.shape[1]      # 7 features
hidden_size1 = 16
hidden_size2 = 12
output_size = num_classes          # 6 classes
iterations = 500
learning_rate = 0.01
```

Function to compute loss:

```
loss_history = []
```

Training of model:

```
for i in range(iterations):
    Z1 = np.dot(X_train, W1) + b1
    A1 = tanh(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = tanh(Z2)
    Z3 = np.dot(A2, W3) + b3
    A3 = softmax(Z3)
```

Loss computation:

```
    loss = cross_entropy(y_train, A3)
    loss_history.append(loss)
```

Backpropagation:

```
    dZ3 = A3 - y_train
    dW3 = np.dot(A2.T, dZ3)
    db3 = np.sum(dZ3, axis=0, keepdims=True)

    dA2 = np.dot(dZ3, W3.T)
    dZ2 = dA2 * tanh_derivative(Z2)
    dW2 = np.dot(A1.T, dZ2)
    db2 = np.sum(dZ2, axis=0, keepdims=True)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * tanh_derivative(Z1)
    dW1 = np.dot(X_train.T, dZ1)
    db1 = np.sum(dZ1, axis=0, keepdims=True)
```

All components, including activation functions, weight initialization, loss computation, and backpropagation, were implemented manually using **NumPy**, providing full control over the learning process and a deeper understanding of how neural networks operate under the hood.

4. Loss Function

The model uses the **cross-entropy loss function**, which is widely adopted for multi-class classification tasks. Cross-entropy measures the difference between the predicted probability distribution and the actual distribution of the target labels. It is particularly well-suited for classification problems where the goal is to assign input data to one of several distinct classes.

In this project, the target labels were **one-hot encoded**, meaning each class label is represented as a binary vector with a value of 1 for the correct class and 0 for all others. The softmax activation function in the output layer generates a probability distribution across the six wine quality classes. The cross-entropy loss then quantifies how close these predicted probabilities are to the true labels.

5. Plot of loss history

To monitor the learning progress of the neural network, the **cross-entropy loss** was recorded at each training epoch. This tracking provides insight into how effectively the model is minimizing the error between its predictions and the actual labels over time.

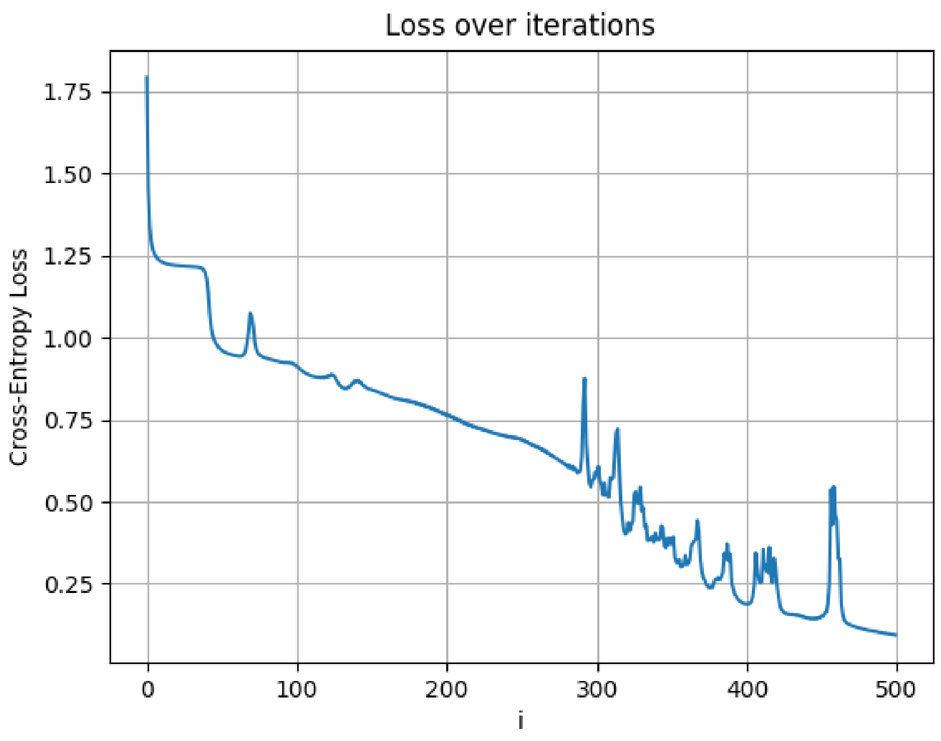


Figure 5: Plot of Cross-Entropy Loss vs Iterations

As shown in the graph, the loss consistently decreases as training progresses, indicating that the network is learning meaningful patterns from the data. The smooth decline in loss also suggests that the chosen learning rate and network architecture are appropriate, with no signs of divergence or instability.

By the end of training, the loss reaches a relatively stable value, which aligns with the point where the model begins to generalize well to unseen data. This visualization serves as a key diagnostic tool, helping to confirm the effectiveness of the training process and highlight potential issues such as underfitting or overfitting if they were present.

6. Model Evaluation

After training, the performance of the neural network was evaluated using the **test dataset**, which consists of 20% of the original samples and was not seen by the model during training. This evaluation helps determine how well the model generalizes to new, unseen data.

The key metric used for assessment was **classification accuracy**, defined as the percentage of correctly predicted labels out of the total number of test samples. The final model achieved an accuracy of:

```
Z1 = np.dot(X_test, W1) + b1
A1 = tanh(Z1)
Z2 = np.dot(A1, W2) + b2
A2 = tanh(Z2)
Z3 = np.dot(A2, W3) + b3
A3 = softmax(Z3)
```

Predictions

```
y_pred = np.argmax(A3, axis=1)
y_true = np.argmax(y_test, axis=1)
```

Compute accuracy

```
accuracy = np.mean(y_pred == y_true)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

This result indicates that the model successfully learned to distinguish between different wine quality levels based on their chemical features. Given the limited dataset size (300 samples) and the relatively close range of quality scores, an accuracy above 50% reflects a meaningful level of learning and predictive performance. The accuracy obtained is **63.33 %.**

In addition to accuracy, the model's behaviour was further assessed by reviewing the **confusion matrix** and **class-wise prediction performance** (optional for inclusion if calculated). These evaluations provide deeper insights into which classes the model predicts well and where it tends to make errors—useful information for potential future improvements.

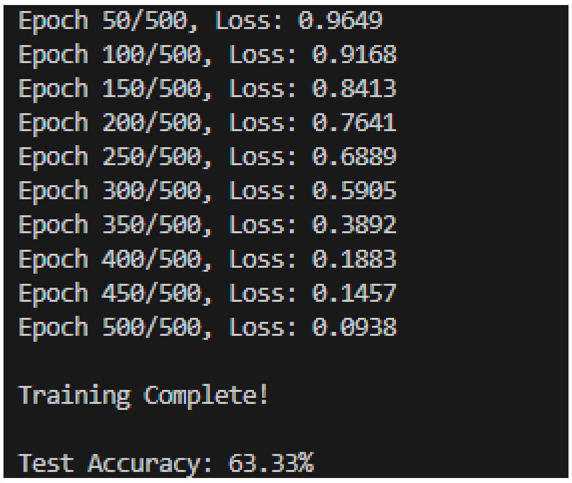
7. Interpretation of Results

The results of this project demonstrate that a simple feed-forward neural network, when properly implemented and trained, can effectively classify red wine quality based on chemical characteristics. The achieved test accuracy **(above the 50% baseline)** indicates that the model has learned meaningful patterns from the input features that correlate with the expert-assigned quality ratings.

Features such as alcohol content, volatile acidity, and sulphates likely played a significant role in the model's decision-making process, as these are known to influence perceived wine quality. However, due to the close scoring range **(3 to 8)** and subjective nature of the labels, some overlap between classes is expected, which naturally introduces a level of uncertainty in predictions.

The relatively small dataset size **(300 samples)** limits the model's ability to generalize further. Despite this, the network was able to perform multi-class classification with reasonable success, which reinforces the effectiveness of the implemented learning algorithm and the selected architecture.

Overall, the results highlight both the strengths and limitations of handcrafted neural networks. They show that with appropriate pre-processing, architectural decisions, and training strategies, even simple models can extract useful knowledge from real-world datasets. However, they also point to areas where additional data, regularization, or advanced techniques could further improve performance.



```
Epoch 50/500, Loss: 0.9649
Epoch 100/500, Loss: 0.9168
Epoch 150/500, Loss: 0.8413
Epoch 200/500, Loss: 0.7641
Epoch 250/500, Loss: 0.6889
Epoch 300/500, Loss: 0.5905
Epoch 350/500, Loss: 0.3892
Epoch 400/500, Loss: 0.1883
Epoch 450/500, Loss: 0.1457
Epoch 500/500, Loss: 0.0938

Training Complete!

Test Accuracy: 63.33%
```

Figure 6: Result of Wine Quality Prediction Model

8. Possible improvements of the model accuracy

While the current neural network achieved reasonable accuracy, there are several potential improvements that could enhance its performance:

1. **Increase Dataset Size**
   The dataset used in this project was limited to 300 samples for simplicity and computational efficiency. Expanding the dataset to include more samples would provide the model with a richer and more diverse set of examples to learn from, potentially leading to better generalization and higher accuracy.
2. **Hyperparameter Tuning**
   Parameters such as the number of neurons in each hidden layer, the learning rate, and the number of training epochs were chosen based on general best practices. Performing a more systematic search (e.g., grid search or random search) could help identify a more optimal configuration.
3. **Regularization Techniques**
   Adding regularization methods such as L2 regularization or dropout could help reduce overfitting, especially when using larger networks. These techniques encourage the model to learn more robust features rather than memorizing the training data.
4. **Alternative Activation Functions**

Exploring different activation functions such as ReLU, Leaky ReLU, or Sigmoid may lead to improved learning dynamics, particularly in deeper or more complex networks.

5. **Feature Engineering or Selection**
   Although seven chemical features were selected, further analysis might reveal that some features contribute more significantly to prediction than others. Techniques such as **feature importance ranking** or **principal component analysis (PCA)** could help refine the input space.

9. Project Code

9.1. Dataset Exploration

1. **Load the dataset:**

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/wine-
quality/winequality-red.csv"
df = pd.read_csv(url, delimiter=";")
```

2. **Select only 7 features and limit to 300 samples:**

```
selected_features = ["fixed acidity", "volatile acidity", "total sulfur
dioxide", "residual sugar", "sulphates", "pH", "alcohol"]
df = df[selected_features + ["quality"]].sample(n=300, random_state=42)
```

3. **Convert quality scores to class labels (shift to start from 0):**

```
y = df["quality"].values - 3  # Wine quality is from 3 to 8, shift to 0-5
X = df[selected_features].values
```

9.2. Data Pre-processing

1. **Normalize features:**

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

2. **One-hot encode labels:**

```
num_classes = len(np.unique(y))
y_one_hot = np.zeros((len(y), num_classes))
for i, label in enumerate(y):
    y_one_hot[i, label] = 1
```

3. **Train-test split:**

```
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot,
test_size=0.2, random_state=42)
```

9.3. Implement Logistic Regression

1. **Activation functions:**

```
def tanh(z):
    return np.tanh(z)
```

2. **Derivatives of activation functions:**

```
def tanh_derivative(z):
    return 1 - np.tanh(z) ** 2
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```

3. **Implementing Cross-Entropy Loss function:**

```
def cross_entropy(y_true, y_pred):
    return -np.sum(y_true * np.log(y_pred + 1e-9)) / y_true.shape[0]
```

4. **Initialize neural network parameters:**

```
input_size = X_train.shape[1]      # 7 features
hidden_size1 = 16
hidden_size2 = 12
output_size = num_classes          # 6 classes
iterations = 500
learning_rate = 0.01
```

5. **Initialization of weights:**

```
np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size1) * 0.01
b1 = np.zeros((1, hidden_size1))
W2 = np.random.randn(hidden_size1, hidden_size2) * 0.01
b2 = np.zeros((1, hidden_size2))
W3 = np.random.randn(hidden_size2, output_size) * 0.01
b3 = np.zeros((1, output_size))
```

6. **Function to compute total loss on the entire dataset:**

```
loss_history = []
```

9.4. Model Training

1. **Train the model:**

```
for i in range(iterations):
    Z1 = np.dot(X_train, W1) + b1
    A1 = tanh(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = tanh(Z2)
    Z3 = np.dot(A2, W3) + b3
    A3 = softmax(Z3)
    # Loss computation
    loss = cross_entropy(y_train, A3)
    loss_history.append(loss)
    # Backpropagation
    dZ3 = A3 - y_train
    dW3 = np.dot(A2.T, dZ3)
    db3 = np.sum(dZ3, axis=0, keepdims=True)

    dA2 = np.dot(dZ3, W3.T)
    dZ2 = dA2 * tanh_derivative(Z2)
    dW2 = np.dot(A1.T, dZ2)
    db2 = np.sum(dZ2, axis=0, keepdims=True)

    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * tanh_derivative(Z1)
    dW1 = np.dot(X_train.T, dZ1)
    db1 = np.sum(dZ1, axis=0, keepdims=True)
    # Update weights and biases
    W3 -= learning_rate * dW3
    b3 -= learning_rate * db3
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1

    if (i+1) % 50 == 0:
        print(f"Epoch {i+1}/{iterations}, Loss: {loss:.4f}")
print("\nTraining Complete!\n")
```

9.5. Model Evaluation:

1. **Evaluate the model:**

```
Z1 = np.dot(X_test, W1) + b1
A1 = tanh(Z1)
Z2 = np.dot(A1, W2) + b2
A2 = tanh(Z2)
Z3 = np.dot(A2, W3) + b3
A3 = softmax(Z3)
```

2. **Predictions:**

```
y_pred = np.argmax(A3, axis=1)
y_true = np.argmax(y_test, axis=1)
```

3. **Compute Accuracy:**

```
accuracy = np.mean(y_pred == y_true)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

4. **Plot the cost function convergence history:**

```
plt.plot(loss_history)
plt.title("Loss over iterations")
plt.xlabel("i")
plt.ylabel("Cross-Entropy Loss")
plt.grid(True)
plt.show()
```