Comparison of Neural Network Architectures for Art Genre Classification Using Image Metadata

Team Members:

HOON HAN (518370990019)
JOONHYOUNG LEE (519370990008)
STEPHEN YEAP WELLER (521370990028)

Course:

STAT4060J Computational Methods for Statistics and Data Science

Submission Date:

2024.12.08

UM-SJTU Joint Institute

1 Introduction

The classification of paintings by artistic genres is a compelling application of machine learning in art analysis and cultural heritage. This project aims to identify the genre of a painting based on visual features, focusing on stylistic patterns to deepen the understanding of genre-specific characteristics. To achieve the project objective, we implemented three genre prediction algorithms: a Simple Neural Network (SNN), a Boosted Neural Network (BNN) using an ensemble method, and an SNN with the Adam optimizer.

We aim for this work to contribute to practical applications in art education and digital curation. By focusing on genre classification, the project provides valuable tools for art enthusiasts and professionals. This report outlines the theoretical foundation, implementation, and evaluation of the models' performance.

2 Data Preprocessing

2.1 Overview of the Dataset

The preprocessed dataset consists of 8,355 entries, each representing a painting by one of 50 artists in the artists.csv file. The genre column associates a genre with each artist, applied to all their paintings, and is one-hot encoded for multi-label classification. Additionally, the features column contains 1D feature vectors used as input for model training and evaluation.

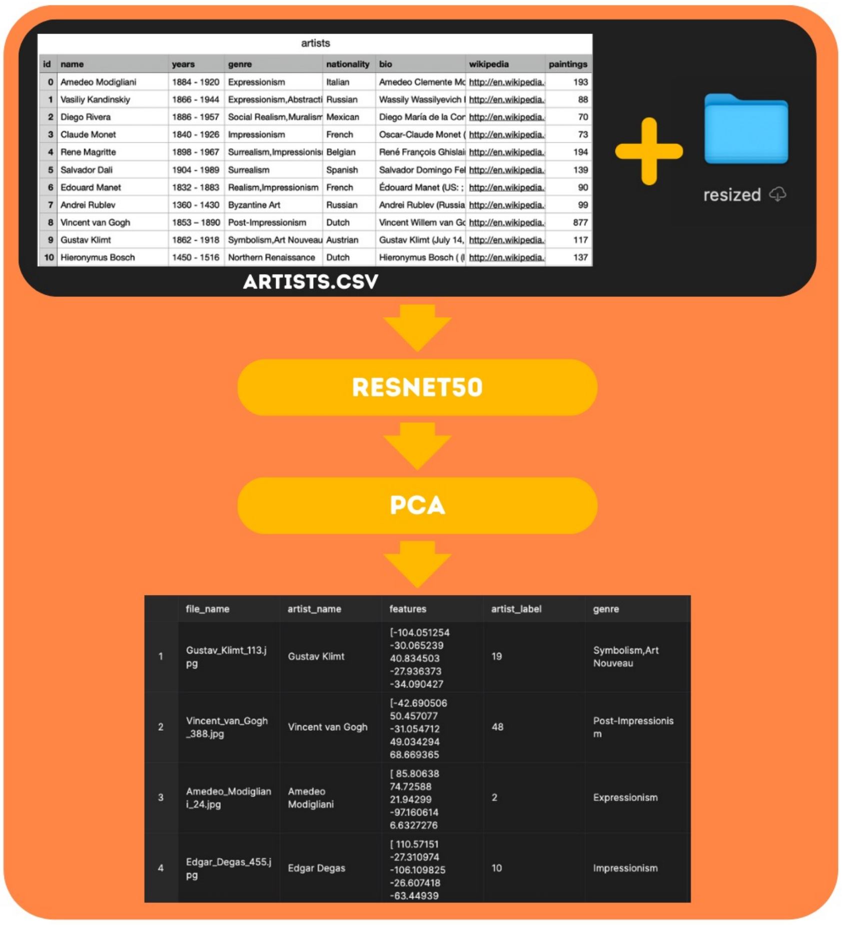
2.2 Preprocessing Steps

2.2.1 Step 1: Feature Extraction with Pretrained ResNet50

Images are resized to 224×224 pixels and preprocessed to align with ResNet50's input requirements. The ResNet50 model, pre-trained on ImageNet, extracts 100,352 features.

2.2.2 Step 2: Dimensionality Reduction with PCA

Principal Component Analysis (PCA) reduces the ResNet50 feature vectors to 100 principal components, minimizing redundancy and enhancing computational efficiency. This approach preserves essential patterns, mitigates overfitting, and accelerates model training and inference.



id	name	years	genre	nationality	bio	wikipedia	paintings
0	Amedeo Modigliani	1884 - 1920	Expressionism	Italian	Amedeo Clemente Mc	http://en.wikipedia.	193
1	Vasiliy Kandinskiy	1866 - 1944	Expressionism,Abstracti	Russian	Wassily Wassilyevich I	http://en.wikipedia.	88
2	Diego Rivera	1886 - 1957	Social Realism,Muralism	Mexican	Diego María de la Cor	http://en.wikipedia.	70
3	Claude Monet	1840 - 1926	Impressionism	French	Oscar-Claude Monet (	http://en.wikipedia.	73
4	Rene Magritte	1898 - 1967	Surrealism,Impressionis	Belgian	René François Ghislai	http://en.wikipedia.	194
5	Salvador Dali	1904 - 1989	Surrealism	Spanish	Salvador Domingo Fel	http://en.wikipedia.	139
6	Edouard Manet	1832 - 1883	Realism,Impressionism	French	Édouard Manet (US: ;	http://en.wikipedia.	90
7	Andrei Rublev	1360 - 1430	Byzantine Art	Russian	Andrei Rublev (Russia	http://en.wikipedia.	99
8	Vincent van Gogh	1853 – 1890	Post-Impressionism	Dutch	Vincent Willem van Gc	http://en.wikipedia.	877
9	Gustav Klimt	1862 - 1918	Symbolism,Art Nouveau	Austrian	Gustav Klimt (July 14,	http://en.wikipedia.	117
10	Hieronymus Bosch	1450 - 1516	Northern Renaissance	Dutch	Hieronymus Bosch ((I	http://en.wikipedia.	137

	file_name	artist_name	features	artist_label	genre
1	Gustav_Klimt_113.jpg	Gustav Klimt	[-104.051254 -30.065239 40.834503 -27.936373 -34.090427	19	Symbolism,Art Nouveau
2	Vincent_van_Gogh_388.jpg	Vincent van Gogh	[-42.690506 50.457077 -31.054712 49.034294 68.669365	48	Post-Impressionism
3	Amedeo_Modigliani_24.jpg	Amedeo Modigliani	[85.80638 74.72588 21.94299 -97.160614 6.6327276	2	Expressionism
4	Edgar_Degas_455.jpg	Edgar Degas	[110.57151 -27.310974 -106.109825 -26.607418 -63.44939	10	Impressionism

Figure 1: Pipeline of data preprocessing.

3 Prediction Algorithms

3.1 Model 1: Simple Neural Network

The simple neural network is a multi-layer perceptron (MLP) with two hidden layers. Each layer performs a linear transformation followed by a non-linear activation function to capture complex patterns.

3.1.1. Initialization of Parameters Weights and biases are initialized to ensure effective learning by breaking symmetry.

Numerical Implementation

```
self.weights = {
    "W1": np.random.randn(input_size, hidden_sizes[0]) * 0.01,
    "W2": np.random.randn(hidden_sizes[0], hidden_sizes[1]) * 0.01,
    "W3": np.random.randn(hidden_sizes[1], output_size) * 0.01,
}
self.biases = {
    "b1": np.zeros((1, hidden_sizes[0])),
    "b2": np.zeros((1, hidden_sizes[1])),
    "b3": np.zeros((1, output_size)),
}
```

3.1.2. Forward Propagation Forward propagation computes the output by applying linear transformations and activation functions.

Theory

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}, \quad A^{[l]} = \mathrm{ReLU}(Z^{[l]}), \quad A^{[L]} = \sigma(Z^{[L]})$$

Numerical Implementation

```
def forward(self, X):
    self.Z1 = np.dot(X, self.weights["W1"]) + self.biases["b1"]
    self.A1 = self.relu(self.Z1)

    self.Z2 = np.dot(self.A1, self.weights["W2"]) + self.biases["b2"]
    self.A2 = self.relu(self.Z2)

    self.Z3 = np.dot(self.A2, self.weights["W3"]) + self.biases["b3"]
    self.A3 = self.sigmoid(self.Z3)

    return self.A3
```

3.1.3. Loss Computation The Binary Cross-Entropy Loss measures the difference between predicted probabilities and true labels.

Numerical Implementation

```
def binary_cross_entropy_loss(self, y_pred, y_true):
    n_samples = y_true.shape[0]
    return -np.sum(y_true * np.log(y_pred + 1e-10) +
                   (1 - y_true) * np.log(1 - y_pred + 1e-10)) / n_samples
```

3.1.4. Backward Propagation Backward propagation computes gradients using the chain rule to adjust weights and biases.

Numerical Implementation

```
def backward(self, X, y_true, y_pred):
    n_samples = y_true.shape[0]
    dZ3 = y_pred - y_true
    dW3 = np.dot(self.A2.T, dZ3) / n_samples
    db3 = np.sum(dZ3, axis=0, keepdims=True) / n_samples

    dA2 = np.dot(dZ3, self.weights["W3"].T)
    dZ2 = dA2 * self.relu_derivative(self.Z2)
    dW2 = np.dot(self.A1.T, dZ2) / n_samples
    db2 = np.sum(dZ2, axis=0, keepdims=True) / n_samples

    dA1 = np.dot(dZ2, self.weights["W2"].T)
    dZ1 = dA1 * self.relu_derivative(self.Z1)
    dW1 = np.dot(X.T, dZ1) / n_samples
    db1 = np.sum(dZ1, axis=0, keepdims=True) / n_samples
```

3.1.5. Parameter Updates Gradient descent updates the weights and biases to minimize the loss.

Numerical Implementation

```
self.weights["W1"] -= self.learning_rate * dW1
self.biases["b1"] -= self.learning_rate * db1
self.weights["W2"] -= self.learning_rate * dW2
self.biases["b2"] -= self.learning_rate * db2
self.weights["W3"] -= self.learning_rate * dW3
self.biases["b3"] -= self.learning_rate * db3
```

3.1.6. Training Loop The training loop iterates through forward propagation, loss computation, backward propagation, and parameter updates.

Numerical Implementation

```
def train(self, X, y, epochs=20, batch_size=32):
    n_samples = X.shape[0]
    for epoch in range(epochs):
        indices = np.arange(n_samples)
        np.random.shuffle(indices)
        X = X[indices]
        y = y[indices]

        for i in range(0, n_samples, batch_size):
            X_batch = X[i:i + batch_size]
            y_batch = y[i:i + batch_size]
            y_pred = self.forward(X_batch)
            self.backward(X_batch, y_batch, y_pred)
```

3.1.7. Prediction The model predicts a genre with the highest probability

Numerical Implementation

```
def predict(self, X):
    y_pred = self.forward(X)
    max_probs = np.max(y_pred, axis=1)
    max_labels = np.argmax(y_pred, axis=1)
    return max_probs, max_labels
```

3.2 Model 2: Boosted Neural Network (Ensemble Learning)

(Rambhatla, Jones, & Chellappa, 2021) The boosted neural network leverages ensemble learning techniques to sequentially train multiple models. Each model focuses on the residuals (errors) from the predictions of the previous model, thereby improving overall performance.

3.2.1. Initialization of Parameters The parameters for each base model in the ensemble are initialized similarly to Model 1 to enable effective learning and prevent symmetry issues.

3.2.2. Forward Propagation The forward propagation process is identical to Model 1, where each layer applies linear transformations and activation functions. Predictions are sequentially adjusted based on the residuals from earlier models.

3.2.3. Loss Computation The Binary Cross-Entropy Loss is identical to Model 1.

3.2.4. Backward Propagation Unlike Model 1, here gradients are clipped using a predefined gradient clipping value to stabilize training and prevent exploding gradients.

Gradient Clipping Value Gradient clipping ensures that gradients do not exceed a predefined threshold, avoiding unstable updates. For this model, the clipping value is set to 0.5.

```
def backward(self, X, y_true, y_pred, gradient_clip_value):
    n_samples = y_true.shape[0]

    dZ3 = y_pred - y_true
    dW3 = np.dot(self.A2.T, dZ3) / n_samples
    db3 = np.sum(dZ3, axis=0, keepdims=True) / n_samples

    dA2 = np.dot(dZ3, self.weights["W3"].T)
    dZ2 = dA2 * self.relu_derivative(self.Z2)
    dW2 = np.dot(self.A1.T, dZ2) / n_samples
    db2 = np.sum(dZ2, axis=0, keepdims=True) / n_samples

    dA1 = np.dot(dZ2, self.weights["W2"].T)
    dZ1 = dA1 * self.relu_derivative(self.Z1)
    dW1 = np.dot(X.T, dZ1) / n_samples
    db1 = np.sum(dZ1, axis=0, keepdims=True) / n_samples

    #Gradient clipping
    dW3 = np.clip(dW3, -gradient_clip_value, gradient_clip_value)
    dW2 = np.clip(dW2, -gradient_clip_value, gradient_clip_value)
    dW1 = np.clip(dW1, -gradient_clip_value, gradient_clip_value)

    self.weights["W1"] -= self.learning_rate * dW1
    self.biases["b1"] -= self.learning_rate * db1
    self.weights["W2"] -= self.learning_rate * dW2
    self.biases["b2"] -= self.learning_rate * db2
    self.weights["W3"] -= self.learning_rate * dW3
    self.biases["b3"] -= self.learning_rate * db3
```

3.2.5. (Ensemble) Training Residual Updates The training loop for each model follows the same iterative process as in Model 1, with additional steps to handle residual updates between models. Residuals are iteratively updated for subsequent models to focus on minimizing the errors from prior predictions. This step is unique to the boosted neural network approach.

Numerical Implementation

```
def train(self, X, y, epochs=10, batch_size=32, residual_threshold=1e-3):
    residuals = y.astype(np.float64)
    for model_idx, model in enumerate(self.models):
        print(f"Training Model {model_idx + 1}/{len(self.models)}")
        model.train(X, residuals, epochs, batch_size, self.gradient_clip_value)
        predictions = model.forward(X)
        residuals -= predictions
        residual_norm = np.linalg.norm(residuals, axis=0, keepdims=True)
        if np.all(residual_norm < residual_threshold):
            print("Residuals below threshold. Stopping further training.")
            break
        residuals /= residual_norm
```

3.2.6. Prediction The ensemble's prediction is the average of the outputs from all models.

Numerical Implementation

```
def predict(self, X):
    ensemble_predictions = np.zeros_like(self.models[0].forward(X))
    for model in self.models:
        ensemble_predictions += model.forward(X)
    ensemble_predictions /= len(self.models)  # Average predictions

    max_probs = np.max(ensemble_predictions, axis=1)
    predicted_genres_indices = np.argmax(ensemble_predictions, axis=1)
    return max_probs, predicted_genres_indices
```

3.3 Model 3: Neural Network with Adam Optimizer

(Kingma & Ba, 2017)

The neural network with Adam optimizer leverages an adaptive learning rate optimization algorithm that combines the benefits of Momentum Optimization and RMSProp. It uses exponential moving averages of the gradients and squared gradients to adaptively adjust the learning rate for each parameter.

3.3.1. Initialization of Parameters The weights and biases are initialized identically to Model 1. However, Adam introduces additional parameters for moment estimates, which are unique to this model.

Adam Initialization For each parameter, Adam maintains:

- First moment vector (m): the exponentially weighted average of the gradients.
- Second moment vector (v): the exponentially weighted average of the squared gradients.

Numerical Implementation

```
self.m_weights = {key: np.zeros_like(value) for key, value in self.weights.items()}
self.v_weights = {key: np.zeros_like(value) for key, value in self.weights.items()}
self.m_biases = {key: np.zeros_like(value) for key, value in self.biases.items()}
self.v_biases = {key: np.zeros_like(value) for key, value in self.biases.items()}
```

3.3.2. Forward Propagation The forward propagation process is identical to Model 1.

3.3.3. Loss Computation The Binary Cross-Entropy Loss is identical to Model 1.

3.3.4. Backward Propagation Backward propagation computes gradients similarly to Model 1, but the parameter updates are unique due to the use of the Adam optimization algorithm.

Adam Optimization Algorithm Adam computes parameter updates as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$$

Where:

- β_1, β_2: Exponential decay rates for the moment estimates.
- g_t: Gradient at time t.
- α: Learning rate.
- ϵ: Small value to prevent division by zero.

Numerical Implementation

```
def adam_update(self, gradients, params, m_params, v_params):
    self.t += 1
    updated_params = {}
    for key in params.keys():
         m_params[key] = self.beta1 * m_params[key]
         + (1 - self.beta1) * gradients[key]
        v_params[key] = self.beta2 * v_params[key]
        + (1 - self.beta2) * (gradients[key] ** 2)
        m_hat = m_params[key] / (1 - self.beta1 ** self.t)
        v_hat = v_params[key] / (1 - self.beta2 ** self.t)
        updated_params[key] = params[key] - self.learning_rate * m_hat /
        (np.sqrt(v_hat) + self.epsilon)
    return updated_params, m_params, v_params

def backward(self, X, y_true, y_pred):
    n_samples = y_true.shape[0]
    dZ3 = y_pred - y_true
    dW3 = np.dot(self.A2.T, dZ3) / n_samples
    db3 = np.sum(dZ3, axis=0, keepdims=True) / n_samples
    dA2 = np.dot(dZ3, self.weights["W3"].T)
    dZ2 = dA2 * self.relu_derivative(self.Z2)
    dW2 = np.dot(self.A1.T, dZ2) / n_samples
    db2 = np.sum(dZ2, axis=0, keepdims=True) / n_samples
    dA1 = np.dot(dZ2, self.weights["W2"].T)
    dZ1 = dA1 * self.relu_derivative(self.Z1)
    dW1 = np.dot(X.T, dZ1) / n_samples
    db1 = np.sum(dZ1, axis=0, keepdims=True) / n_samples

    gradients = {"W1": dW1, "W2": dW2, "W3": dW3, "b1": db1, "b2": db2, "b3": db3}
    self.weights, self.m_weights,
    self.v_weights = self.adam_update(
        {key: gradients[key] for key in self.weights.keys()},
        self.weights,
        self.m_weights, self.v_weights
    )
    self.biases, self.m_biases, self.v_biases = self.adam_update(
        {key: gradients[key] for key in self.biases.keys()}, self.biases,
        self.m_biases, self.v_biases
    )
```

3.3.5. Training Loop The training loop in this model incorporates the unique Adam update mechanism during the parameter update step, differentiating it from Model 1.

Numerical Implementation

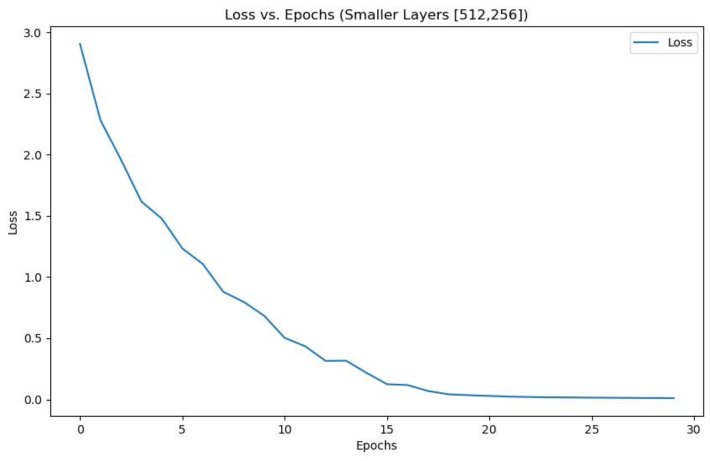
```
for epoch in range(epochs):
    indices = np.arange(n_samples)
    np.random.shuffle(indices)
    X = X[indices]
    y = y[indices]

    for i in range(0, n_samples, batch_size):
        X_batch = X[i:i + batch_size]
        y_batch = y[i:i + batch_size]
        y_pred = self.forward(X_batch)
        loss = self.binary_cross_entropy_loss(y_pred, y_batch)
        self.backward(X_batch, y_batch, y_pred) # Adam is applied here
```

3.3.6. Prediction The prediction process is identical to Model 1.

4 Evaluation and Results

4.1 Evaluation on Simple Neural Network (SNN)



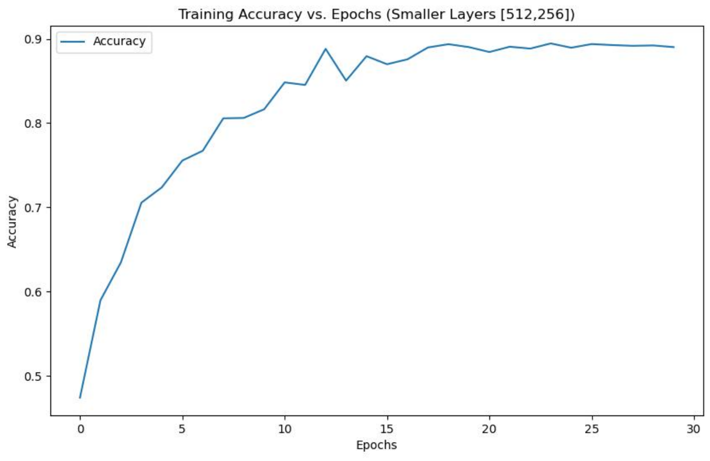


Figure 2: Loss vs. Epoch and Training Accuracy vs. Epoch for SNN.

4.1.1 Training Figure 2 shows the SNN's training performance with hidden layers [512, 256] over 30 epochs. Accuracy starts at 50%, rising to 90% by epoch 15 and stabilizing. Training loss decreases sharply from 3.0 to near 0.0 by epoch 25, indicating effective learning, convergence, and minimal overfitting.

4.1.2 Test The model evaluates multi-label classification by checking if predicted genres match any associated with the artist. For the architecture [512, 256] and a learning rate of 0.01, the SNN achieved a test accuracy of **73.13%**.

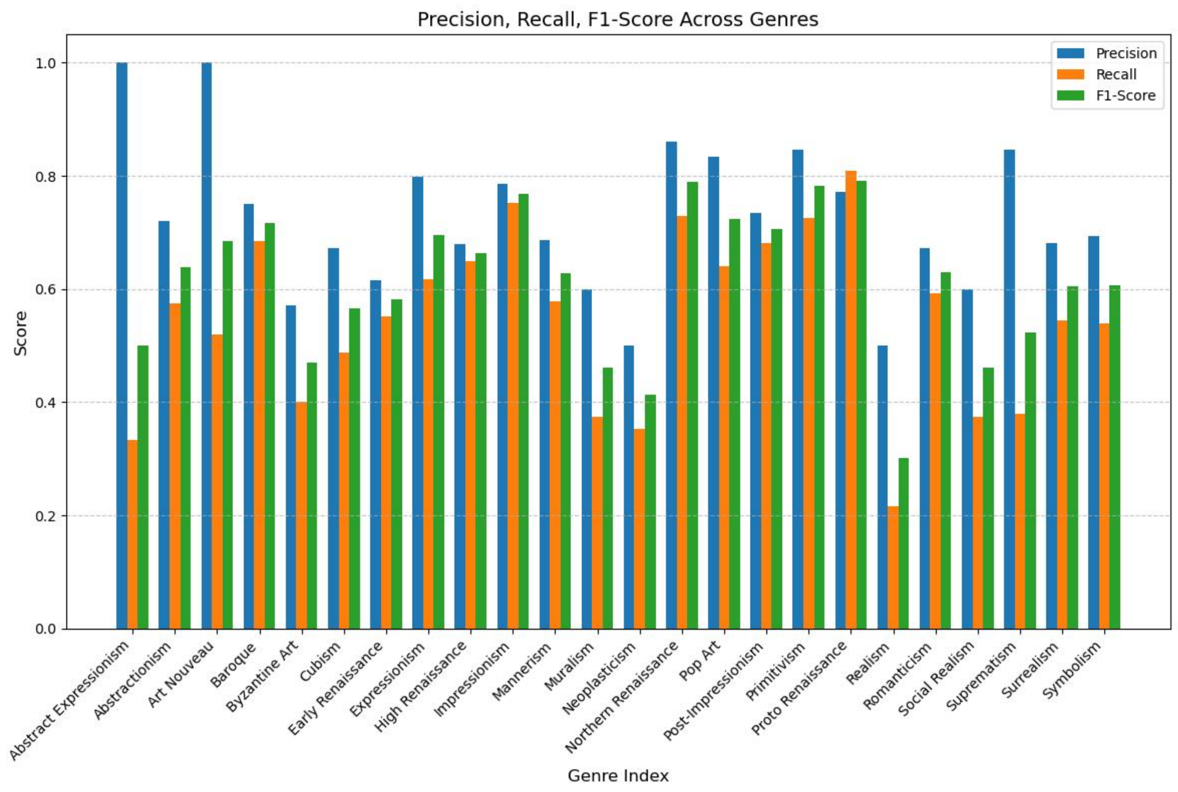


Figure 3: Precision, Recall, and F1-Score Across Genres (threshold: 0.5).

4.1.3 Precision, Recall, and F1-Score Figure 3 illustrates strong performance for well-represented genres (e.g., Impressionism and Symbolism) but lower scores for under-represented genres (e.g., Art Nouveau and Realism). This may reflect the impact of dataset imbalance on classification accuracy.

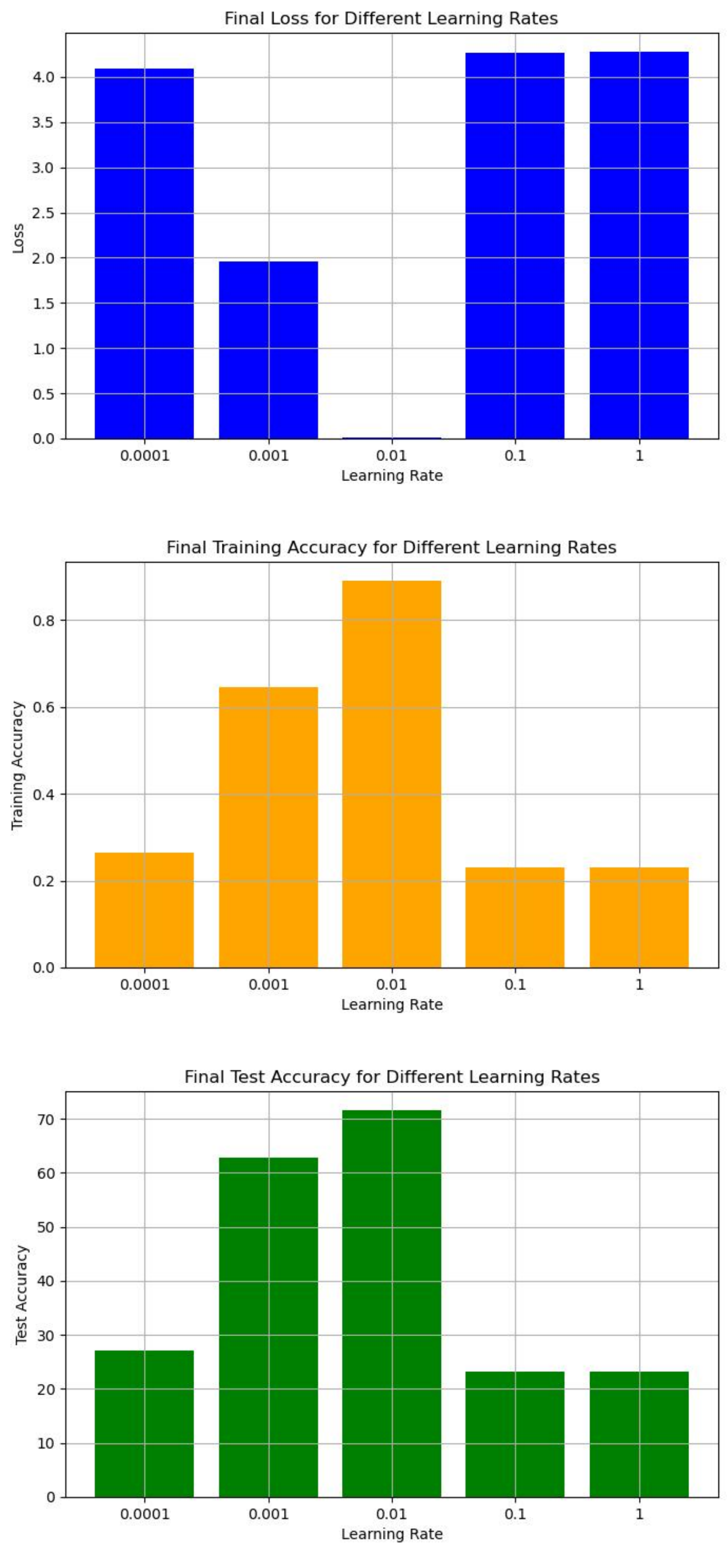


Figure 4: Impact of Different Learning Rates.

4.1.4 Impact of Learning Rate Figure 10 shows a learning rate of 0.01 achieves the best balance of loss, training accuracy, and test accuracy. Smaller rates (0.0001) cause may underfitting, while larger rates (0.1, 1) may lead to instability and poor performance.

4.2 Evaluation on Boosted Neural Network (Ensemble Method)

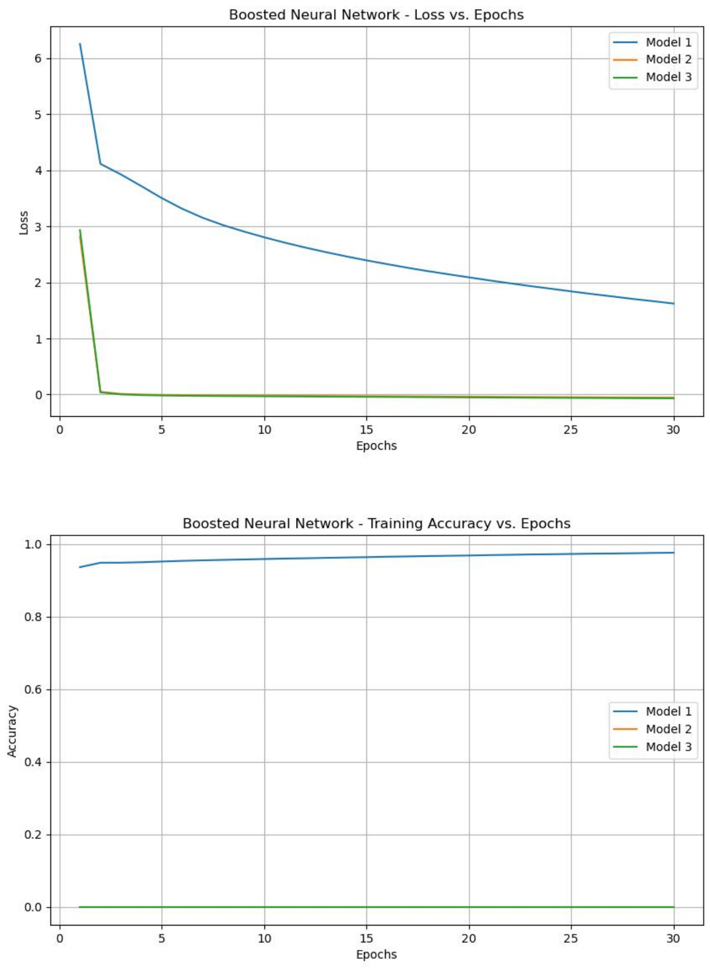


Figure 5: Loss vs. Epoch and Training Accuracy for Boosted NN.

4.2.1 Training Figure 5 shows Model 1 effectively reduces loss, forming a strong baseline, while Models 2 and 3 correct residual errors, though their contributions diminish, as indicated by the flattening accuracy curve at 0.

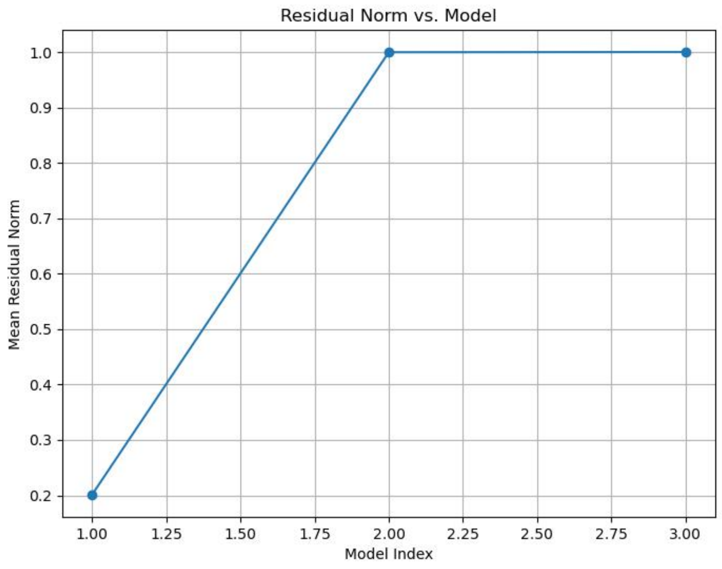


Figure 6: Residual Norm vs. Model for Boosted NN.

Figure 6 can seem to confirm that Model 1 captures most patterns, while subsequent models provide minimal residual reduction.

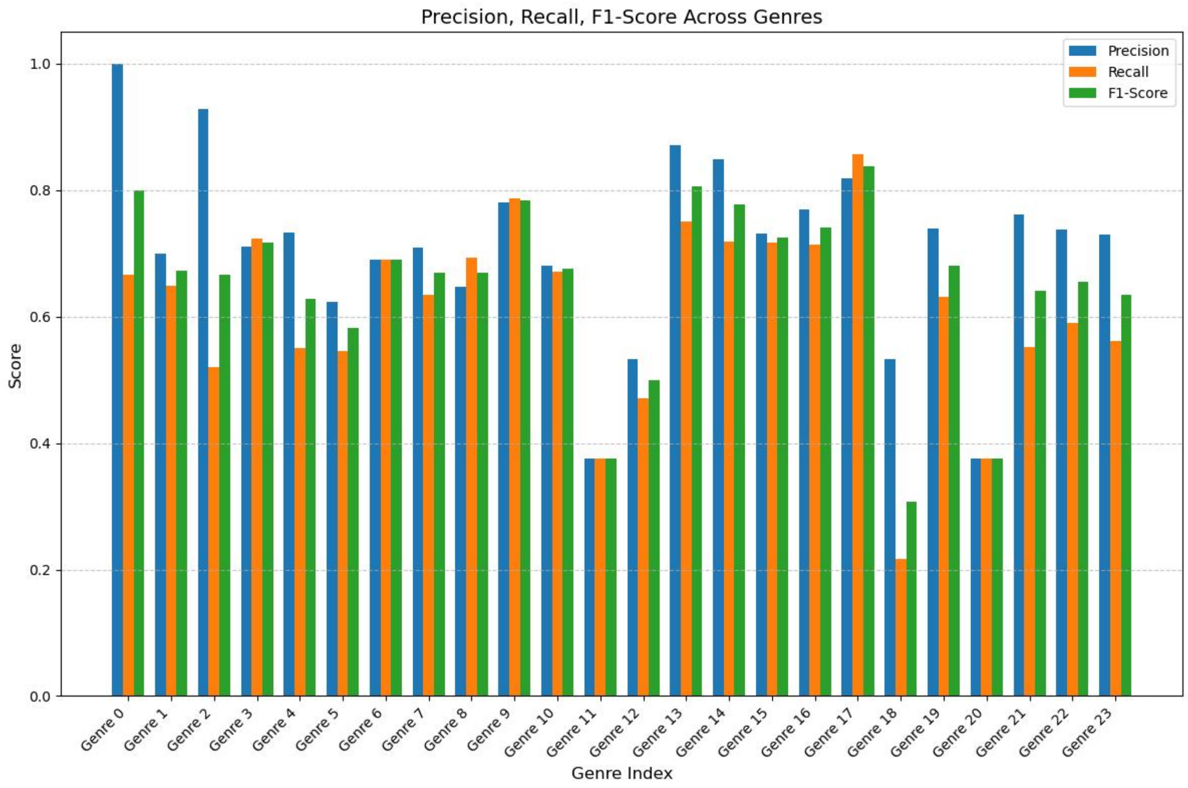


Figure 7: Precision, Recall, F1-Score Across Genres (threshold: 0.1).

4.2.2 Precision, Recall, and F1-Score Figure 7 shows BNN's ensemble nature can seem to improve under-represented genres by lowering the threshold to 0.1. This compensates for smoother predictions compared to SNN.

4.2.3 Test BNN achieved a test accuracy of **73.25%** using the same evaluation method as SNN.

4.3 Evaluation on Adam Optimizer Method

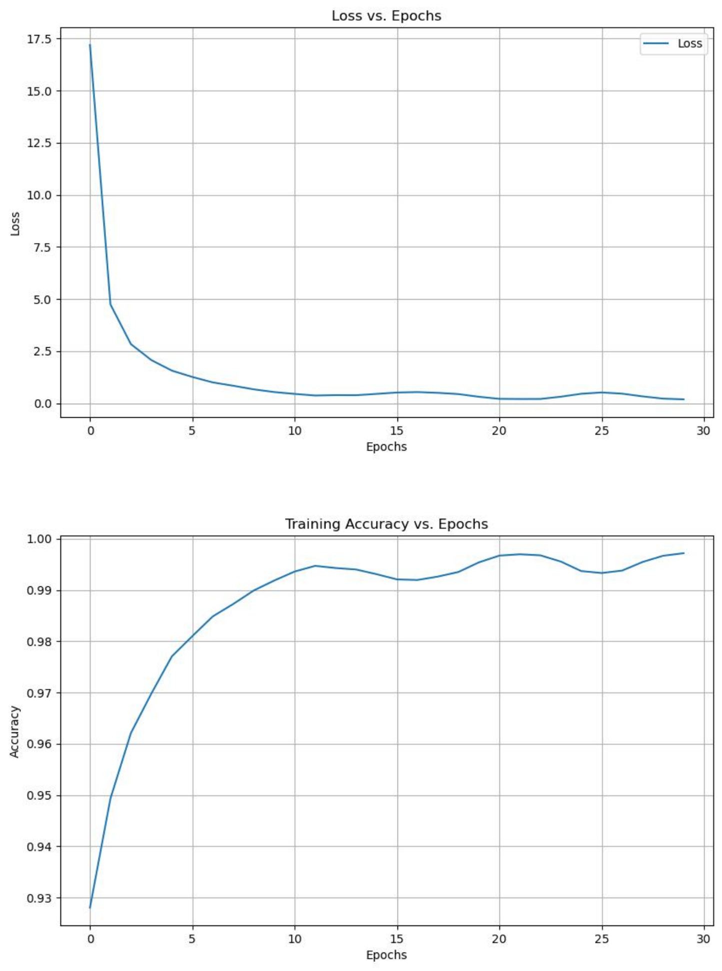


Figure 8: Loss vs. Epoch and Training Accuracy vs. Epoch for SNN.

4.1.1 Training Figure 8 shows the Adam Optimizer's training performance with hidden layers [512, 256] over 30 epochs. The **loss** decreases rapidly from 17.5 to 0.5 within the first 10 epochs, stabilizing around 0.2. **Training accuracy** starts at 93% and reaches over 99% by epoch 20, maintaining this level.

4.1.2 Test Adam Optimizer Method achieved a test accuracy of **66.55%** using the same evaluation method as SNN.

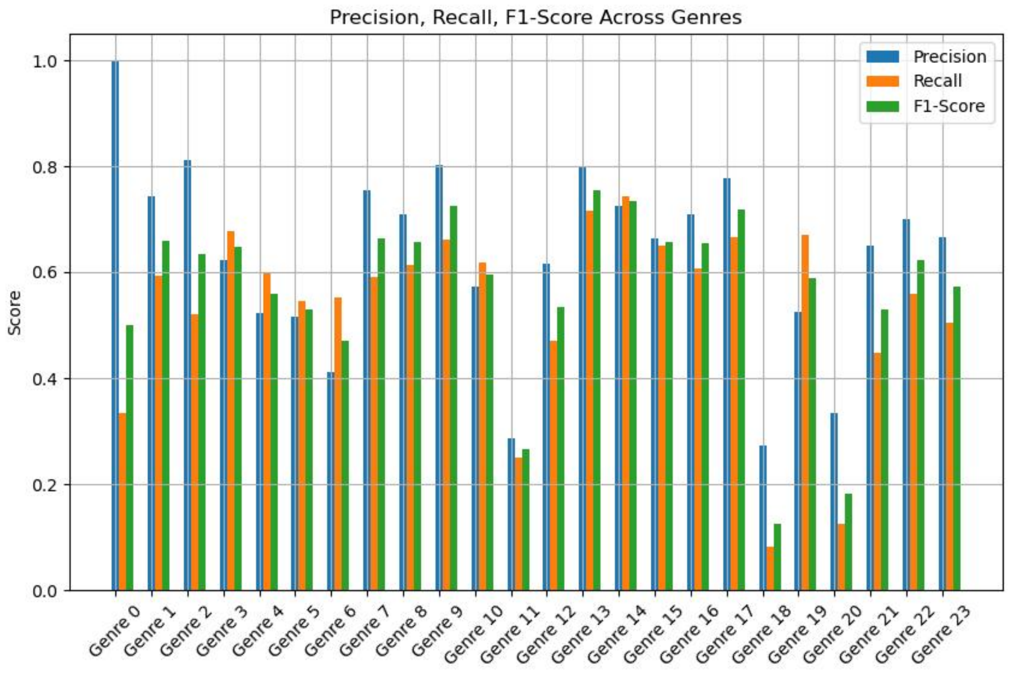


Figure 9: Precision, Recall, and F1-Score Across Genres (threshold: 0.5).

4.1.3 Precision, Recall, and F1-Score Figure 9 shows the **Precision, Recall, and F1-Score** across genres for the Adam Optimizer model. Compared to the SNN results in Figure 9, the Adam model achieves lower scores, particularly for under-represented genres. The Adam model appears to exhibits higher variance in precision and recall, perhaps indicating less consistent performance across genres. This seems to reflect the Adam model's tendency to converge faster but with reduced accuracy compared to the SNN.

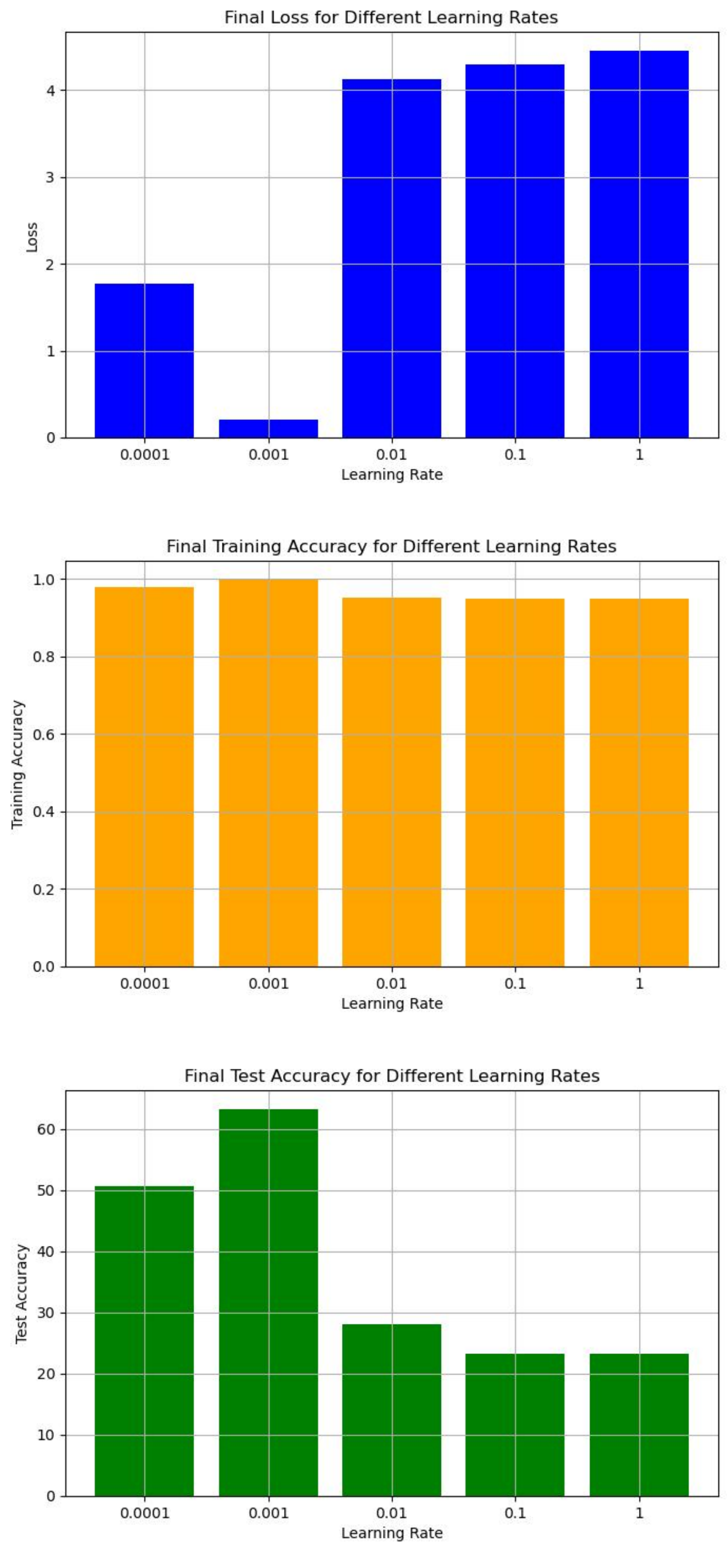


Figure 10: Impact of Different Learning Rates.

4.1.4 Impact of Learning Rate Figure 10 shows that a learning rate of **0.001** most likely achieves the best balance of loss, training accuracy, and test accuracy. The final loss is minimized, and both training and test accuracies seem to remain high.

Smaller learning rates, such as **0.0001**, possibly lead to underfitting, as indicated by higher loss and lower test accuracy. Larger learning rates, such as **0.01**, **0.1**, and **1**, could result in instability, with higher loss values and significantly lower test accuracy.

5 Discussion

5.1 Comparison Between Different Size of Layers (SNN)

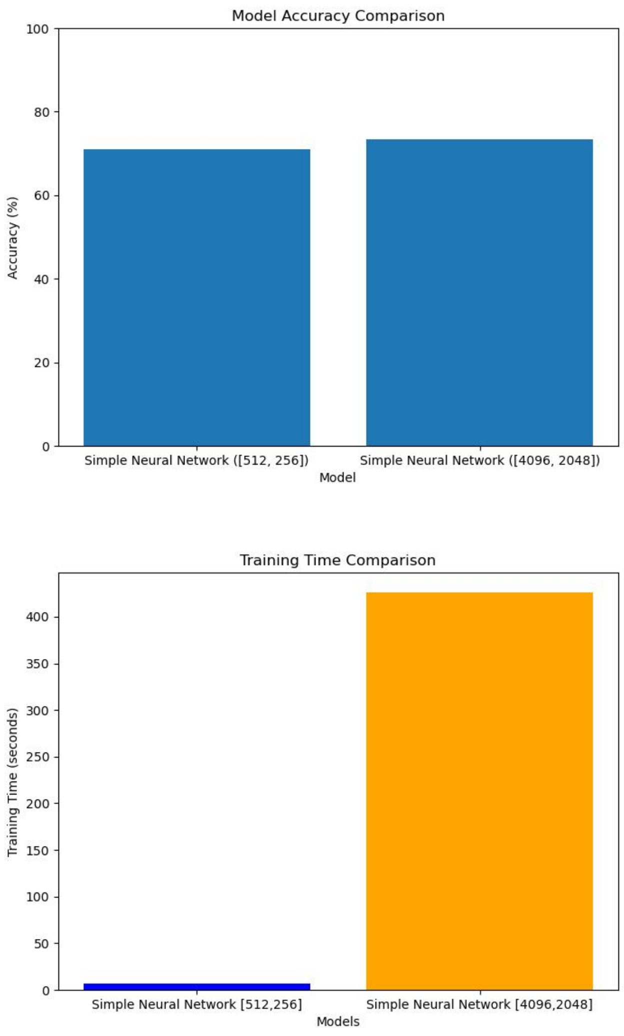


Figure 11: Comparison of Test Accuracy and Training Time Between Different Layer Size.

Figure 11 shows that although the SNN with larger hidden layers ([4096, 2048]) showed slightly better performance than the smaller configuration ([512, 256]), its training time was significantly higher, taking over 40 times longer to train.

5.2 Comparison Among All Three Models

5.2.1 Test Accuracy

The models' test accuracies possibly highlight their generalization capabilities. The Simple Neural Network (SNN) achieved **71.04%**, effectively capturing dataset patterns with a straightforward architecture. The Boosted Neural Network (BNN) had the highest accuracy at **73.43%**, possibly through its ability to leverage ensemble learning to correct misclassifications. The Adam Optimizer model reached **66.01%**; although it converged faster, it was less accurate than SNN and BNN.

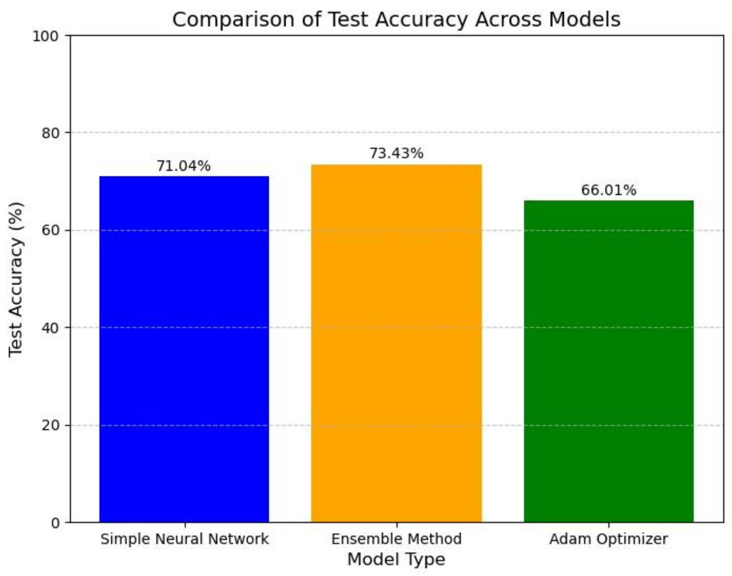


Figure 12: Comparison of Test Accuracy Across Models.

5.2.2 Training Efficiency

Training efficiency varies significantly. The SNN completed training in **8 seconds**, demonstrating quick convergence due to its simplicity. BNN required **44 seconds** because of the sequential training of multiple networks, offering higher accuracy but at a computational cost. The Adam Optimizer model trained in **22 seconds**, possibly benefiting from adaptive learning rates but still slower than SNN, seemingly due to additional computations.

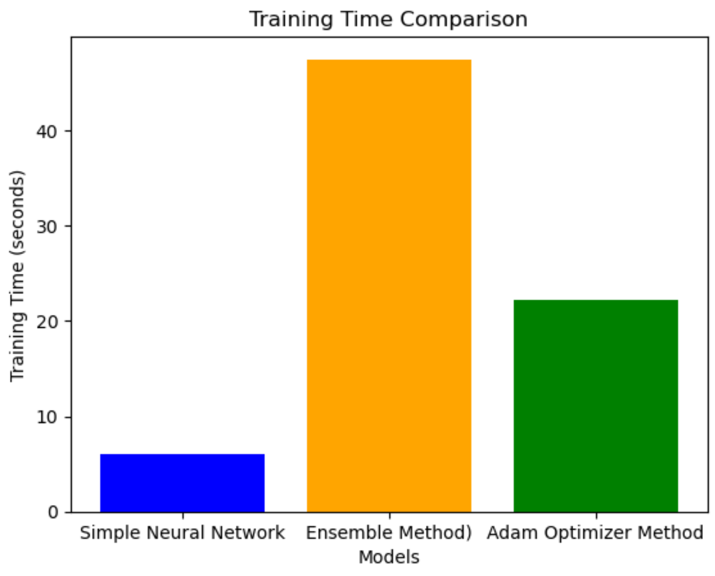


Figure 13: Training Time Comparison.

Summary The SNN appears to strike a balance between accuracy (**71.04%**) and efficiency (8 seconds), making it practical for most tasks. BNN achieves the highest accuracy (**73.43%**) but is maybe less feasible for time-constrained scenarios. The Adam Optimizer model (**66.01%**) offers a middle ground with moderate accuracy and training time (22 seconds).

5.3 Limitations and Potential Improvements

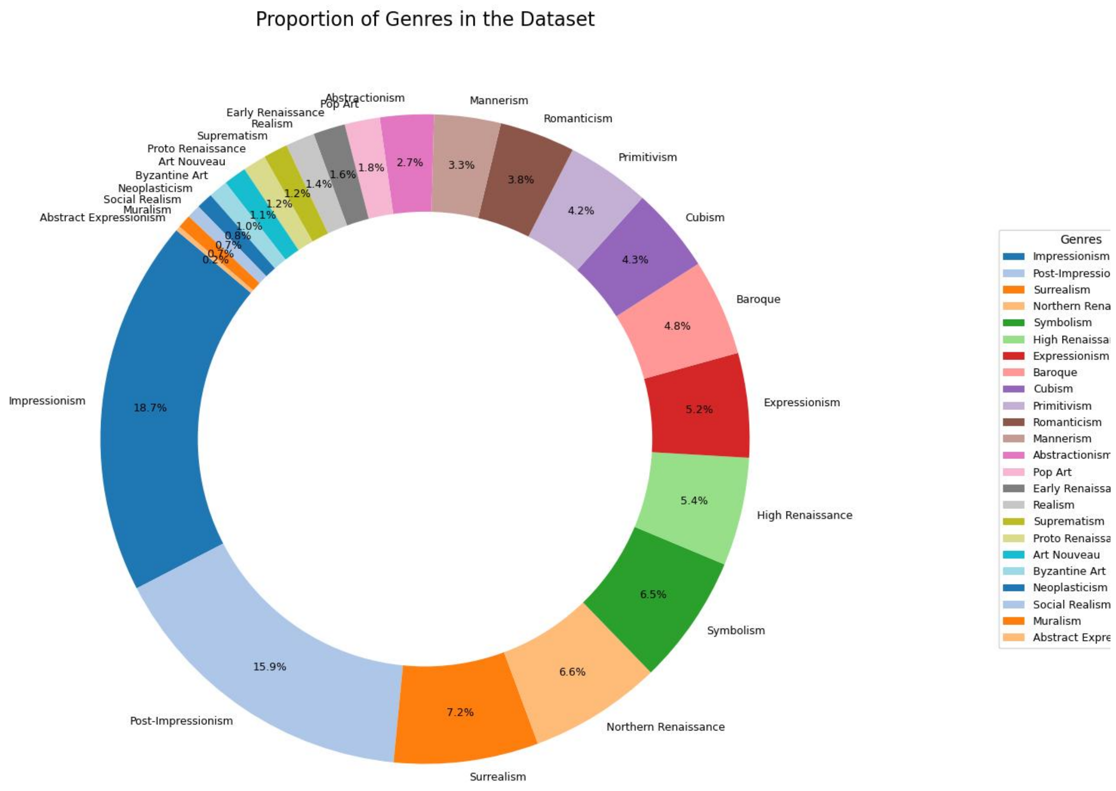


Figure 14: Genre Distribution in the Dataset.

5.3.1 Imbalanced Dataset Figure 14 shows the dataset imbalance, with genres like Impressionism dominating and others like Abstract Expressionism underrepresented. This imbalance causes lower Precision, Recall, and F1-Scores for minority genres. To improve performance, data augmentation or balanced sampling could possibly help the models generalize better across all genres.

5.3.2 Imperfect Feature Extraction Using a pretrained ResNet50, trained on ImageNet, may have limited performance as ImageNet focuses on general object recognition rather than the nuanced stylistic details required for art genre classification. This mismatch means the extracted features may not fully align with the task. Fine-tuning on domain-specific data or integrating advanced techniques like attention mechanisms could improve performance.

5.4 Literature Review on Art Classification Models

Recent advancements in art genre classification have leveraged sophisticated machine learning methods to enhance accuracy and robustness. Joshi et al. (2020) introduced a self-supervised ensemble learning approach, utilizing the Ensemble of Auto-Encoding Transformations (EnAET). This method excels in handling highly imbalanced datasets, such as the WikiArt collection, by employing spatial and non-spatial transformations to create robust feature representations. By leveraging both labeled and unlabeled data, the EnAET model achieves an impressive 82.6% accuracy, significantly surpassing traditional supervised methods like ResNet50, which only reached 55% with augmentation. This approach highlights the effectiveness of weakly supervised learning in scenarios with limited labeled data and high class imbalance (Joshi, Agrawal, & Nair, 2020).

Similarly, Menis-Mastromichalakis et al. (2024) presented a stacking ensemble methodology that combines outputs from diverse neural network architectures to enhance classification consistency. Using state-of-the-art models like DenseNet and Inception-ResNet, their approach achieved a notable 68.5% accuracy on the challenging WikiArt dataset. The stacking ensemble integrates outputs through a meta-classifier, effectively addressing data-dependent variability while exploiting unique perspectives of individual sub-models. This innovative technique ensures robust performance, particularly for datasets with intricate visual features and imbalanced class distributions (Menis-Mastromichalakis, Sofou, & Stamou, 2024).

6 Conclusion

This project explored the use of machine learning for art genre classification, implementing and evaluating three models: a Simple Neural Network (SNN), a Boosted Neural Network (BNN) using ensemble methods, and an SNN enhanced with the Adam optimizer. The primary goal was to classify paintings based on their genres using a pipeline that included feature extraction with ResNet50 and dimensionality reduction via PCA.

The evaluation highlighted that the BNN seemed to achieve the highest accuracy (73.43%), leveraging ensemble learning to correct misclassifications effectively. However, this came at the cost of increased computational time. The SNN, while slightly less accurate (71.04%), offered a more practical solution due to its simplicity and efficiency. The Adam optimizer model showed quicker convergence but achieved the lowest accuracy (66.01%), possibly indicating a trade-off between efficiency and generalization.

Key limitations include the dataset imbalance, which appeared to affect the performance for underrepresented genres, and the use of general-purpose feature extraction methods that may not fully capture the nuances of art genres. Addressing these issues through balanced sampling, data augmentation, or domain-specific fine-tuning could further enhance model performance. This study underscores the potential of machine learning in art analysis, with practical applications in education and digital curation, while also identifying avenues for improvement in future research.

References

Joshi, A., Agrawal, A., & Nair, S. (2020). Art style classification with self-trained ensemble of autoencoding transformations. *arXiv preprint arXiv:2012.03377*. Retrieved from `https://arxiv.org/abs/2012.03377`

Kingma, D. P., & Ba, J. (2017). *Adam: A method for stochastic optimization.* Retrieved from `https://arxiv.org/abs/1412.6980`

Menis-Mastromichalakis, O., Sofou, N., & Stamou, G. (2024). Deep ensemble art style recognition. *arXiv preprint arXiv:2405.11675*. Retrieved from `https://arxiv.org/abs/2405.11675`

Rambhatla, S. S., Jones, M., & Chellappa, R. (2021). *To boost or not to boost: On the limits of boosted neural networks.* Retrieved from `https://arxiv.org/abs/2107.13600`