Roj Deniz Aldemir 22102442
Eray Gündoğdu 22101962

EEE485 Term Project - Song Emotion Classifier

Problem Definition:

Music streaming companies benefit a lot from emotional classifiers since they increase their profit margins by providing playlists, songs or even genres all of which can be divided into emotional categories. A robust machine-learning algorithm can successfully classify songs and their corresponding emotions by using several parameters. This report presents an algorithm that simply takes several parameters about a song, which is provided by Spotify API, and makes an estimation of the major emotion of that specific song.

Dataset Description:

The dataset that was used during training the algorithm was taken from Kaggle [1]. It included approximately 278000 songs of various genres and artists. This datasets seeks to classify songs into 4 emotional categories: 0 is happy, 1 is sad, 2 is energetic, 3 is calm. It has 11 features for each song which are taken from Spotify's API. The features are acouticness, danceability, loudness, energy, instrumentalness, liveness, loudness, speechiness, valence and tempo.

Machine Learning Algorithms of the Project:

The three algorithms that we will use are Neural Networks, Support Vector Machines and XGBoost. Neural Network training is chosen because of various reasons such as a NN can learn the non-linear relationship between an input and an output, assign different weights to different features so that they can prioritize the contributions of the more important features to the output, and since classification is our task, we can assign probabilities to neural network output scores using a basic softmax function, so it becomes feasible to make estimations using argmax function of the output. We also chose SVM because it is a baseline when it comes to classification algorithms. It is more convenient to use when there is the risk of overfitting because SVM is generalizable unless there are vast amount of outliers. [2] XGBoost on the other hand was chosen because we wanted to use a tree-based algorithm so that we can handle the dataset efficiently [3]. Since it is tree-based, the algorithm will decide on the more important features by assigning them a score, and we can compare this scores with our previous feature selection methods and make a more informed judgement on feature selection. Also we can implement L1 or L2 regularization in XGBoost so that we can prevent overfitting.

Both group members will simultaneously participate in each step of the implementations and all the codes will be written together instead of partitioning the workload.
The very first challenge that comes to mind is our infamiliarity with the XGboost learning algorithm. Neither member is familiar with XGboost but we wanted to learn and implement a tree-based algorithm and writing the code from the scratch while learning the algorithm will be a major challenge. Another challenge will likely be the problem of overfitting. Our algorithm might start to memorize the data instead of learning it at a certain point so will use cross-validation methods (mostly k-folds) to measure the generalizability of the algorithm.

Preprocess:

As it is indicated previously this dataset consists of 11 columns and 278.000 rows where one of the columns represent the song labels. For a high efficient machine learning algorithm data preprocessing is crucial hence, we aim to alternate our dataset appropriately in the following steps.

1)Downsampling

After observing the number of samples of each class it is observed that label samples from label 1 dominate over other labels in terms of number of samples. The excessive amount of samples may adversely influence the efficiency of the implemented machine learning algorithm which is as a result of oversampling bias. Moreover, downsampling benefits the model generalization by providing equal distribution of samples. In this case, the number of samples of each label is downsampled to the number of samples of class 3 which has 42.386 labels (Figure 1)..

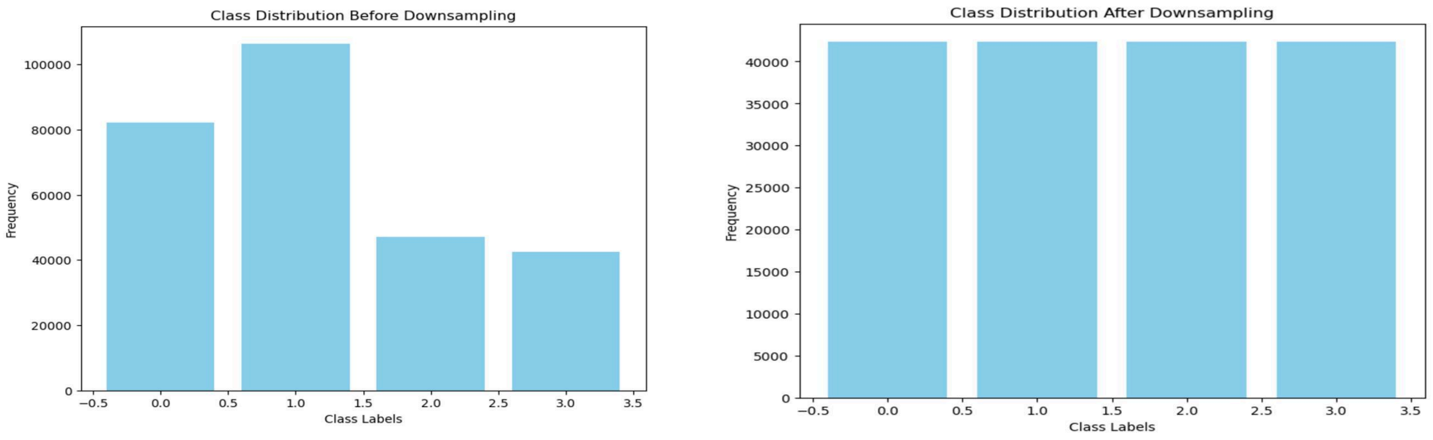
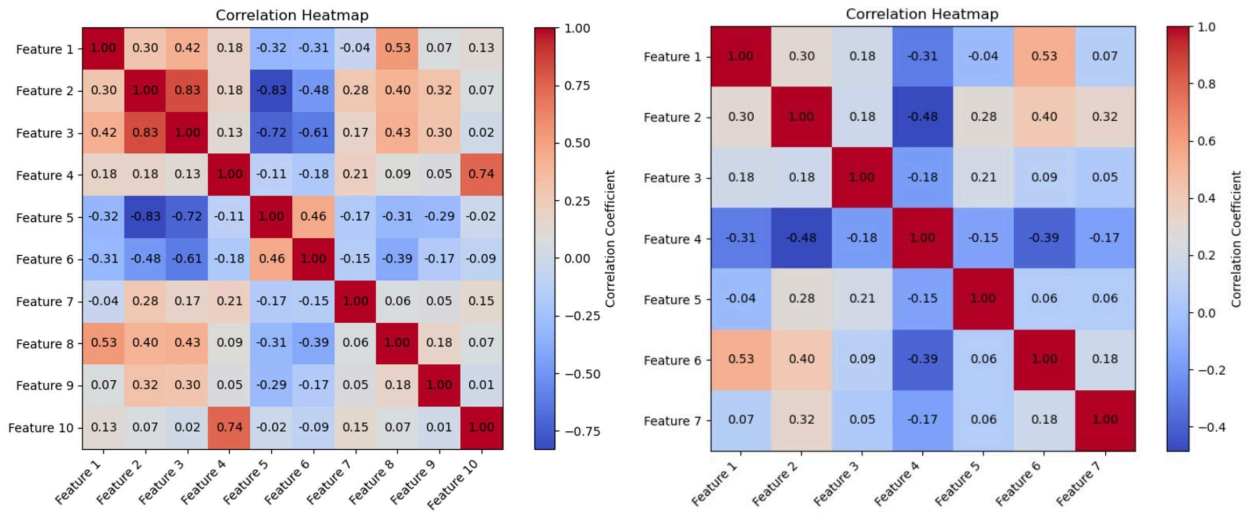


Figure 1 Downsampling of the number of the samples for each class

2) Correlation Heatmap

It is important to observe the correlation between each feature which can be observed practically by inspecting the correlation heatmap. After dropping the columns, which do not have a role in determining the characteristics of a song such as 'uri, 'duration'', the correlation heatmap of 10 features is observed (Figure 2). As the conclusion of our observation, it is decided to eliminate the features which have positive and negative correlation and the number of features is reduced to 7. The eliminated features due to correlation are 'loudness', 'spec_rate', 'acousticness'. The final correlation heatmap can be observed in Figure 3.



	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Feature 7	Feature 8	Feature 9	Feature 10
Feature 1	1.00	0.30	0.42	0.18	-0.32	-0.31	-0.04	0.53	0.07	0.13
Feature 2	0.30	1.00	0.83	0.18	-0.83	-0.48	0.28	0.40	0.32	0.07
Feature 3	0.42	0.83	1.00	0.13	-0.72	-0.61	0.17	0.43	0.30	0.02
Feature 4	0.18	0.18	0.13	1.00	-0.11	-0.18	0.21	0.09	0.05	0.74
Feature 5	-0.32	-0.83	-0.72	-0.11	1.00	0.46	-0.17	-0.31	-0.29	-0.02
Feature 6	-0.31	-0.48	-0.61	-0.18	0.46	1.00	-0.15	-0.39	-0.17	-0.09
Feature 7	-0.04	0.28	0.17	0.21	-0.17	-0.15	1.00	0.06	0.05	0.15
Feature 8	0.53	0.40	0.43	0.09	-0.31	-0.39	0.06	1.00	0.18	0.07
Feature 9	0.07	0.32	0.30	0.05	-0.29	-0.17	0.05	0.18	1.00	0.01
Feature 10	0.13	0.07	0.02	0.74	-0.02	-0.09	0.15	0.07	0.01	1.00

	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Feature 7
Feature 1	1.00	0.30	0.18	-0.31	-0.04	0.53	0.07
Feature 2	0.30	1.00	0.18	-0.48	0.28	0.40	0.32
Feature 3	0.18	0.18	1.00	-0.18	0.21	0.09	0.05
Feature 4	-0.31	-0.48	-0.18	1.00	-0.15	-0.39	-0.17
Feature 5	-0.04	0.28	0.21	-0.15	1.00	0.06	0.06
Feature 6	0.53	0.40	0.09	-0.39	0.06	1.00	0.18
Feature 7	0.07	0.32	0.05	-0.17	0.06	0.18	1.00

Figure 2 Correlation Heatmap of 10 Features vs 7 Features

3) Normalization and Standardization
In the context of this project, min-max normalization is applied to the dataset after the downsampling and feature selection from correlation heatmap processes. The purpose of using min-max normalization is to fasten the convergence of the algorithm. In addition to min-max normalization standardization is also used by subtracting the mean of each feature from itself and dividing to its standard deviation. This technique is utilized in order to decrease the influence of the outliers in the dataset.

$$X_{normalized} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

$$X_{standardized} = \frac{X - \mu}{\sigma}$$

4) Test, Train Split
As the final step of the data pre-processing, we simply divided our dataset as training and test set. The training set will be divided into validation and training set later on in the cross-validation step. The training data size ratio to test data size is selected as 0.8/0.2. The ratio determination is 'Ad hoc' hence the reason behind choosing these ratios is to obtain a generalizable model and a relatively large test set to see model performance on unseen data.

Activation Functions and Cross Entropy Loss
For the hidden layer, we chose to use the ReLU activation;

$$ReLU(x) = max(0, x), \frac{d}{dx}(ReLU) = 1, for\, x \geq 0$$

We do not want the derivative of the activation function to shrink at larger values during backpropagation, therefore ReLU is a good non-linear function to use as our hidden layer activation function.
For the output layer, we used the softmax activation function, which transforms the scores of each class into probabilities. Even though they are not exactly probabilities, this transformation helps us find the argmax of the NN scores, which gives us the class with the highest probability of being the actual value. Given N output values:

$$softmax(v_i) = \frac{e^{-v_i}}{\sum_{i=1}^{N} e^{-v_i}}$$

Cross Entropy Loss
Cross-entropy quantifies how well the predicted probability distribution matches the true class labels. To derive the cross entropy loss, think of an structure that follows as soft-max output , one hot encoding and, cross entropy loss. The loss for single data can be expressed as (some can derive the same output with KL divergence where y_i represents the one-hot encoded output and p_i represents the probability output of the soft-max function :

$$L = -\sum_{i=1}^{C} y_i log(p_i)$$

This expression is valid for only 1 sample from the dataset, for N samples the cross entropy loss can be generalized accordingly:

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{C} y_{i,n} log(p_{i,n})$$

Gradient Descent Optimizers
In our NN algorithm, after implementing regular gradient descent, we used two optimizers, SGD and momentum, then we observed the results to make a better evaluation of our algorithm.

SGD
Stochastic Gradient Descent is an iterative algorithm that updates the weight parameters of a ML algorithm with respect to random samples (mini-batches) of a training set. In a regular gradient descent algorithm, gradient is taken with respect to the entire data set at once and the weights are updated once after the whole dataset is iterated through (an epoch). Whereas in stochastic version, in each epoch, weights are updated as much as the number of data points in each epoch, resulting in faster weight updates and faster convergence, however more unstable and noisier updates.

$$W_{new} = W_{old} - \eta_{learning\ rate} * \frac{\partial L}{\partial W} \rightarrow Gradient\ Descent$$

$$W_{new} = W_{old} - \eta_{learning\ rate} * \frac{\partial L}{\partial W}(W; x_i, y_i) \rightarrow Stochastic\ Gradient\ Descent$$

Gradient Descent with Momentum
The momentum optimization method's fundamental purpose is to add a fraction of the previous update to the current update as it can be observable from the mathematical equations of the momentum given below. This helps the optimization process move faster in the relevant direction and dampens oscillations in directions that are not important due to the 'momentum'. Another remarkable benefit of the momentum optimizer is that it can create enough 'inertia' to escape from the local minimum which leads to optimal convergence as desired.

$$\Delta W(n) = -\eta\frac{\partial L}{\partial W} + \alpha\Delta W(n-1)$$

$$\Delta W(n) = -\eta\sum_{k=1}^{n}\alpha^{n-k}\frac{\partial L(k)}{\partial W}$$

It is important to highlight that Momentum method acts as a Low Pass Filter during updating the weights. The LPF characteristics can be observed with taking the z-Transform and simplifying the transform.

$$\Delta W(z) = \alpha\Delta W(z)z^{-1} + Z(-\eta\frac{\partial L}{\partial W})$$

$$\Delta W(z) = \frac{1}{1-\alpha z^{-1}}Z(-\eta\frac{\partial L}{\partial W})$$

Thus, one of the reasons to use the momentum method is to compensate the noisy update of SGD.

Neural Network Implementation:
NN Design
After determining the feature set to be used and processing the data, we began designing the Neural Network. We defined a class called Neural_Network, initialized the input and output layers, several hyperparameters such as hidden layer number and learning rate, randomly defined initial weights and made the initial biases zero. We defined the forward and backward propagation methods, backpropagation used the SGD equations to compute the appropriate weight updates and forward propagation to compute the current loss. Let L represent the output layer and l represent hidden layers.The forward propagation is modeled mathematically as :

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = ReLU(Z^{[l]})$$

$$A^{[L]} = softmax(Z^{[L]})$$

Let m represents the number of neurons. The backward propagation at the output layer can me modeled as:

$$dZ^{[L]} = A^{[L]} - Y$$

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T}$$

$$db^{[L]} = \frac{1}{m} \sum_{i=1}^{m} dZ_i^{[L]}$$

The backward propagation of hidden units are modeled as:

$$dA^{[l]} = W^{[l+1]T} dZ^{[l+1]}$$

$$dZ^{[L]} = dA^{[L]} \odot ReLU'(Z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{1}{m} \sum_{i=1}^{m} dZ_i^{[l]}$$

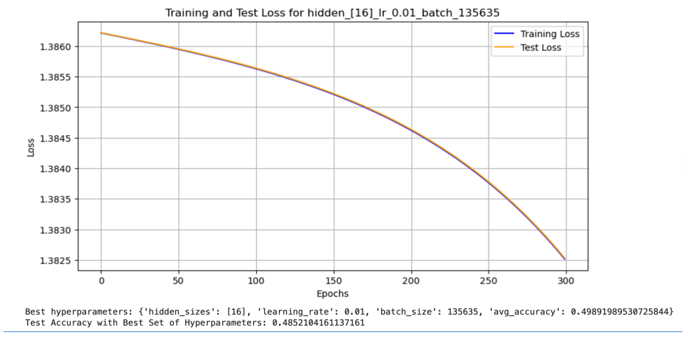
It is important to note that the weights are updated with the momentum method in addition to the gradients calculated. In the train method, we combined both of these methods and computed forward propagation and backpropagation for the given epoch iteration count and computed the loss for each epoch.

k-Fold Cross-Validation
If we used our entire dataset for training, the generalizability of the algorithm would be significantly lower because approximating the training error to 0 is not necessarily a good thing. The algorithm would memorize the entire data set and would overfit to that specific sample. In order to check for overfitting, we computed the 5-fold cross-validation of the dataset. We computed the validation accuracy of each fold and checked the average validation accuracy of the 5 folds. We also compared our model's output with the test data we created. We also checked the test accuracy every hundred epochs since the algorithm could start to overfit after a certain epoch.

Hyperparameter-Tuning
We conducted experiments with different hyperparameter sets on our model. We entered 0.005, 0.01, 0.06 for possible learning rates. For hidden layers, we used 4 different combinations with 1 and 2 eight-neuron, 1 and 2 sixteen-neuron hidden layers and we adjusted the batch size to be either 32 or 64 samples. This left us with a total of 24 hyperparameter set combinations and we executed cross-validation for each combination using SGD with momentum gradient descent (see Appendix A). This generated us the best hyperparameter set.

Results and Observations:

We implemented a regular gradient descent algorithm, and used SGD and momentum optimizers afterwards. Without any optimizers, the gradient descent algorithm took almost an hour to execute 1000 epochs with 3-fold cross-validation. In order to save time, we executed a 300 epoch GD and it generated only 48% test accuracy, which is very low compared to the optimizers with the same number of epochs.



```
Best hyperparameters: {'hidden_sizes': [16], 'learning_rate': 0.01, 'batch_size': 135635, 'avg_accuracy': 0.49891989530725844}
Test Accuracy with Best Set of Hyperparameters: 0.4852104161137161
```

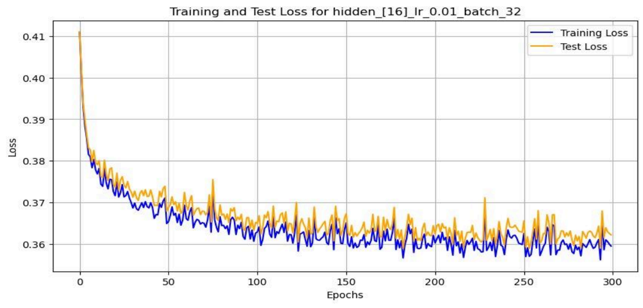
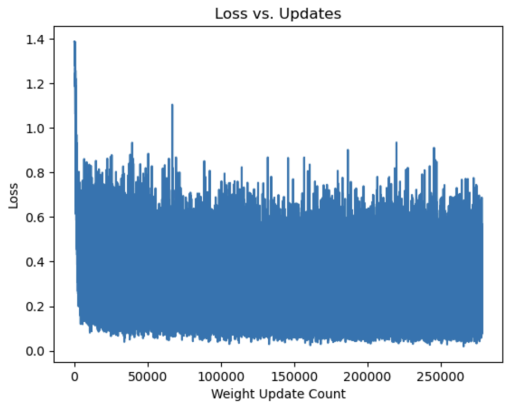


Figure 3 Regular gradient descent

Figure 4 SGD with momentum

After observing the figures, it is clear to see that SGD with momentum method fastens the convergence process compared to regular GD as expected. Even though regular gradient descent might be a more stable and direct approach to optimize the weights, it takes a lot of time for GD to converge. However, in the case of SGD, the model converges much more rapidly than the GD because the amount of updates in a single epoch is significantly larger in SGD. This fact results in less stability and noisier updates but the computational advantage is remarkably higher.



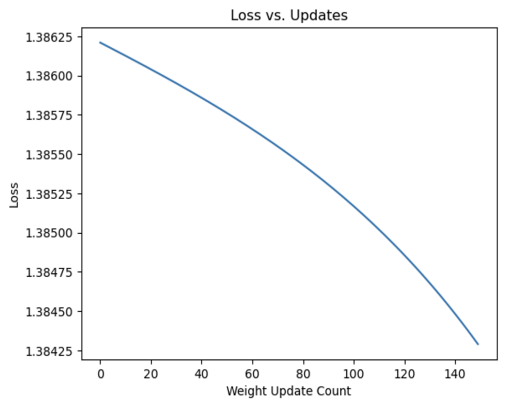
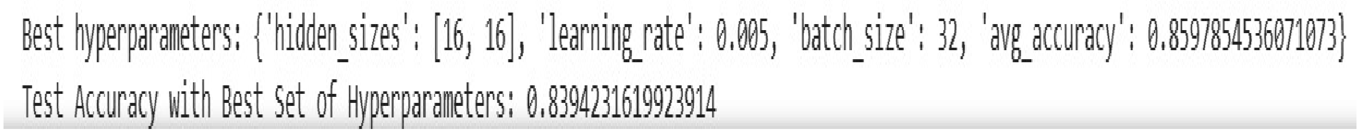


Figure 5 Loss/weight update in SGD

Figure 6 Loss/weight update in GD

The model with the graph given in Figure 4 is executed with the best hyperparameter set to be found. After training the algorithm for different sets for 1 hour, the resulting hyperparameters were as given in Figure 7.



```
Best hyperparameters: {'hidden_sizes': [16, 16], 'learning_rate': 0.005, 'batch_size': 32, 'avg_accuracy': 0.8597854536071073}
Test Accuracy with Best Set of Hyperparameters: 0.8394231619923914
```

Figure 7 Best hyperparameter set

During learning rate tuning three different learning rates are applied to all of the hidden layer combinations. Before commenting on the results it is beneficial to observe the gradient descent convergence on equal error contours.

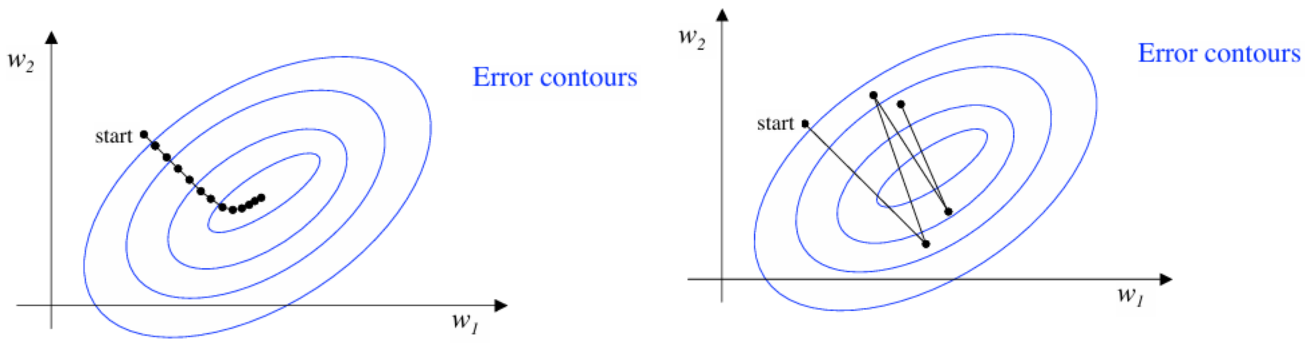


Figure 8 Normal learning rate vs large learning rate convergence on equal error contours [4]

The large learning rate causes distortions in the convergence. Parallel to this statement, in our implementation we observed that as we increase the learning rate the change in loss becomes noisy. Check Appendix A for all figures for different combinations of learning rates, neurons and, hidden layer sizes.

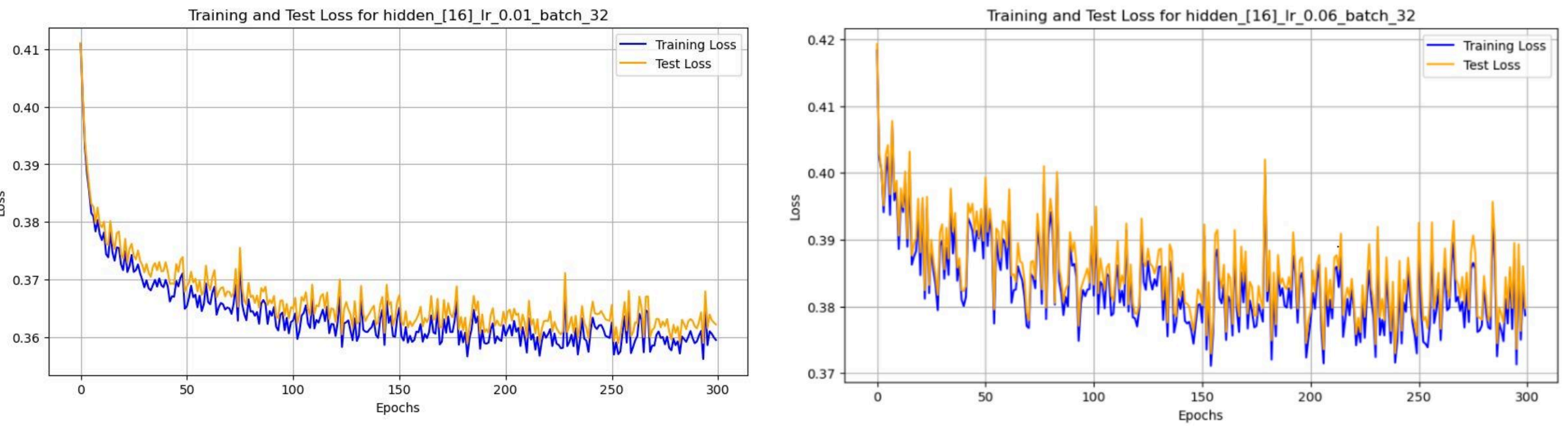
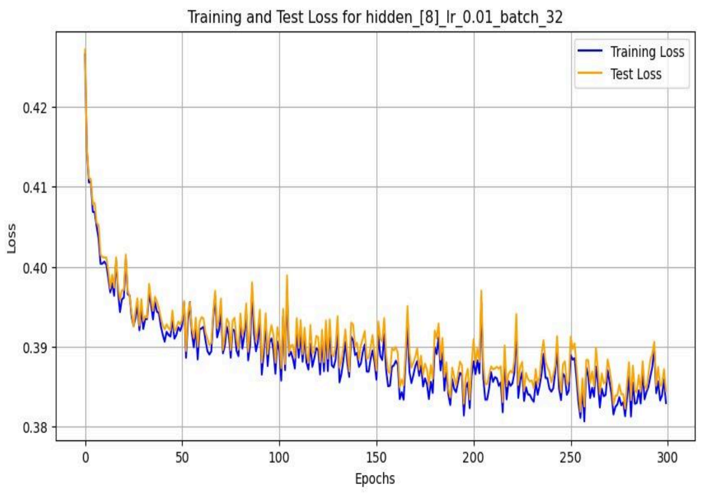


Figure 9 Demonstration of the noise as learning rate increases

It is also possible to observe the effect of adding more layers to the neural network. A single hidden-layer network with 8 neurons and a two-hidden layer network with both hidden layers having 8 neurons are trained during the hyperparameter tuning phase as well. When more layers are added to the network, the loss plot falls faster in the beginning since more layers mean more parameters and this allows the model to represent more complex functions. This enables the model to approximate the function faster initially but there is a trade-off since more parameters also makes optimization more challenging so the convergence becomes slower compared to the less-layered model in towards the later epochs.



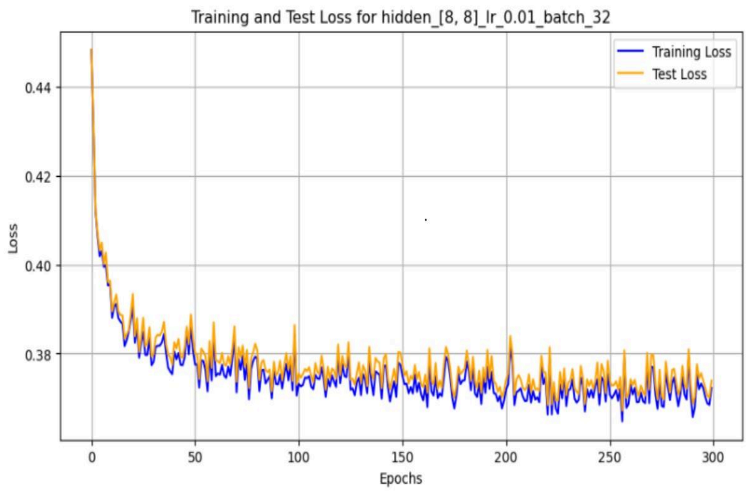


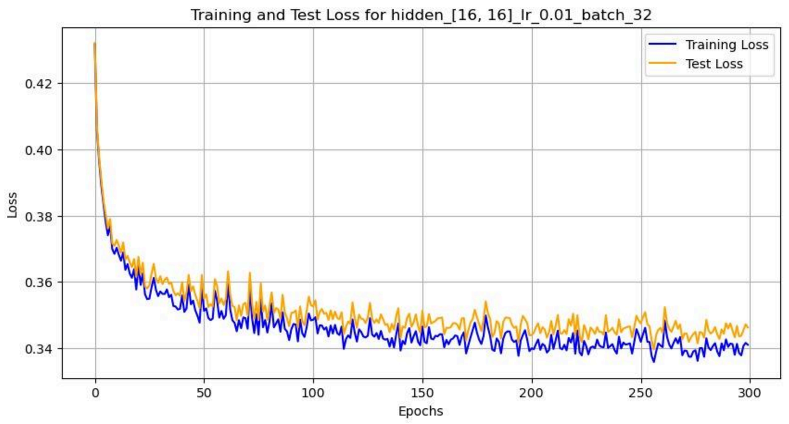
Figure 10 Demonstration of the faster drop in loss as with changing layer size

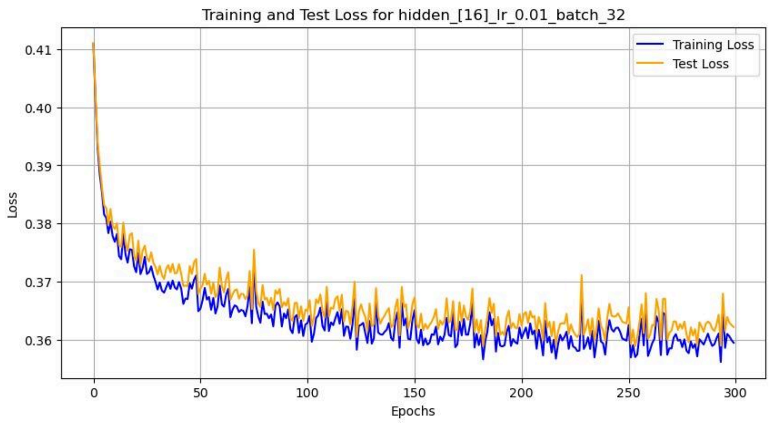
References

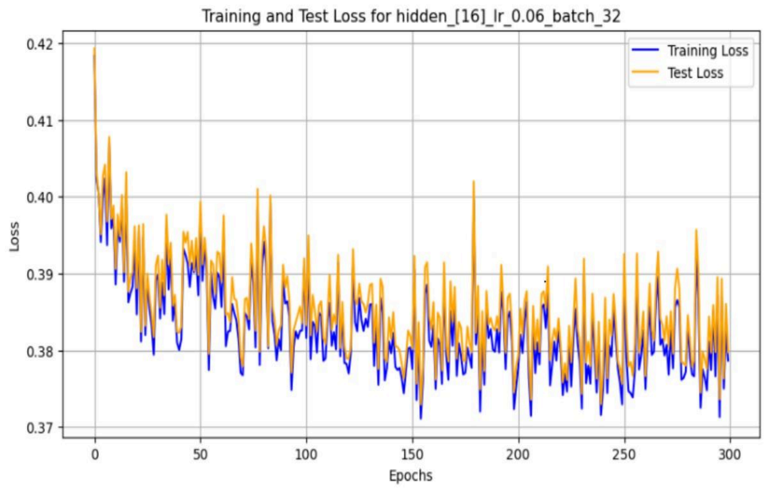
[1] Abdullah Orzan, “278k Emotion Labeled Spotify Songs,” *Kaggle.com*, 2023. https://www.kaggle.com/datasets/abdullahorzan/moodify-dataset (accessed Oct. 17, 2024).
[2] IBM, “What is support vector machine? | IBM,” *www.ibm.com*, Dec. 27, 2023. https://www.ibm.com/topics/support-vector-machine (accessed Nov. 16, 2024).
[3] Nvidia, “What is XGBoost?,” *NVIDIA Data Science Glossary*, 2024. https://www.nvidia.com/en-us/glossary/xgboost/ (accessed Nov. 16, 2024).
[4]I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, Massachusetts: The MIT Press, 2016. Available: https://www.deeplearningbook.org/

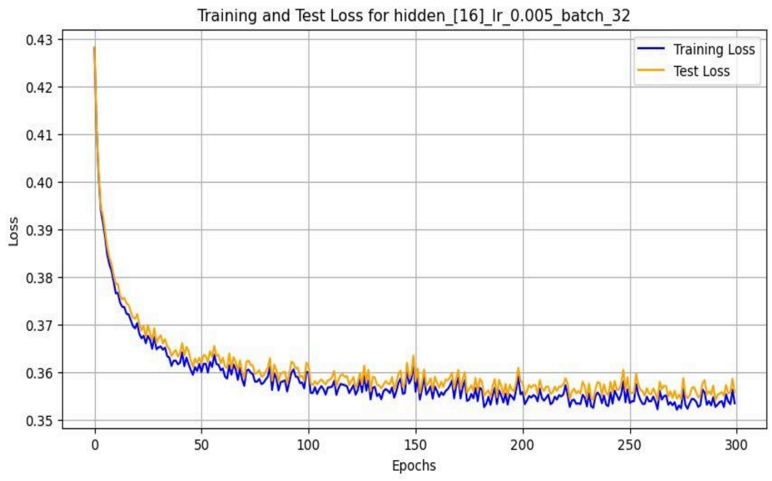
Appendices

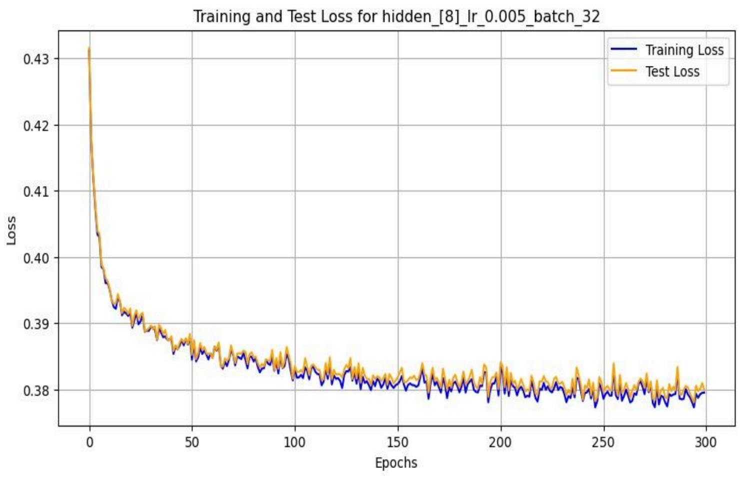
Appendix A: Hyperparameter-Tuning with SGD-momentum

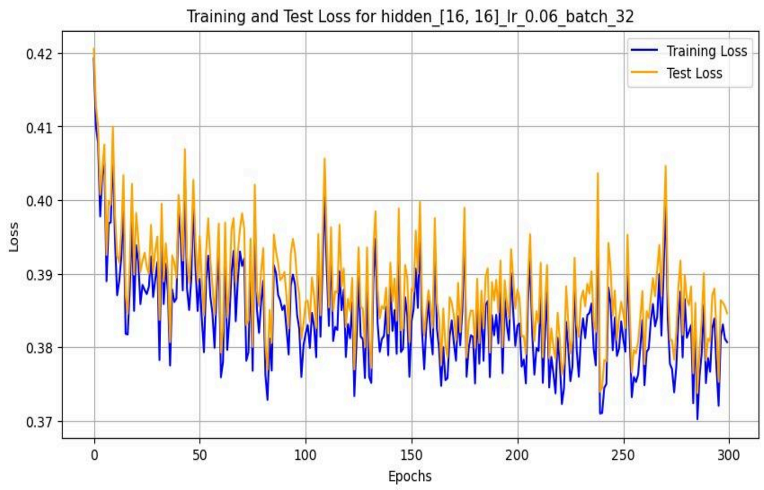


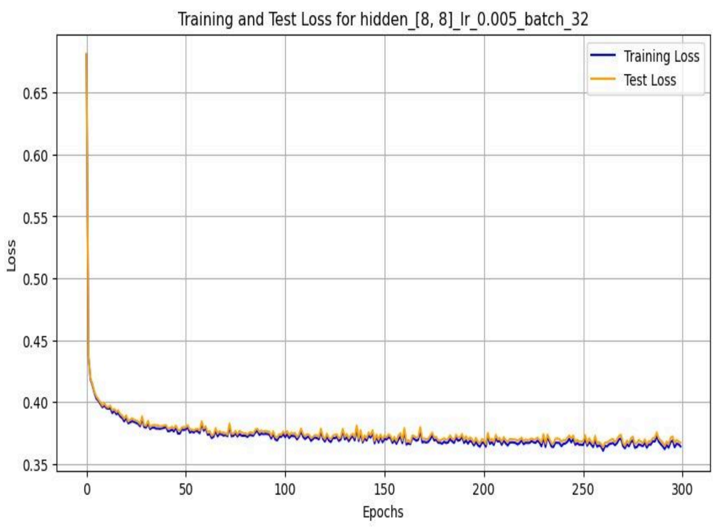


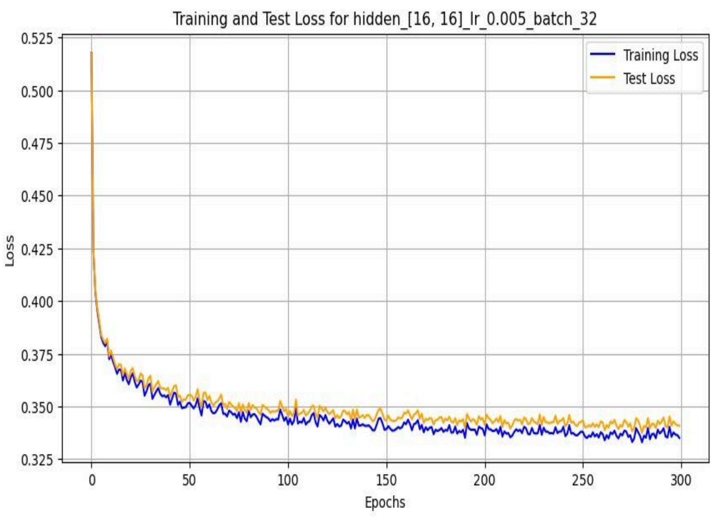












Appendix B: Python Codes

Python Code for Preprocessing

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

path = r"278k_labelled_uri.csv"
data = pd.read_csv(path)
data = data.drop(columns=['Unnamed: 0.1', 'Unnamed: 0', 'uri','duration
(ms)','loudness','spec_rate','acousticness'])
#missing_values = data.isnull().sum()
#print("Missing values:\n", missing_values)
#data = data.dropna()
#print(data.shape)
X = data.drop(columns=['labels']).values
print(X[0])
y = data['labels'].values
label_counts = data['labels'].value_counts()
min_class_count = label_counts.min()
balanced_data = data.groupby('labels', group_keys=False).apply(lambda x:
x.sample(min_class_count)).reset_index(drop=True)
balanced_counts = balanced_data['labels'].value_counts()
X=balanced_data.drop(columns=['labels']).values
y = balanced_data['labels'].values
print(X.shape)
plt.figure(figsize=(8, 6))
plt.bar(label_counts.index, label_counts.values, color='skyblue')
plt.xlabel("Class Labels")
plt.ylabel("Frequency")
plt.title("Class Distribution Before Downsampling")
plt.show()
plt.figure(figsize=(8, 6))
plt.bar(balanced_counts.index, balanced_counts.values, color='skyblue')
plt.xlabel("Class Labels")
plt.ylabel("Frequency")
plt.title("Class Distribution After Downsampling")
plt.show()
X_processed=np.copy(X)
for i in range(X.shape[1]):
   min_val = X[:, i].min()
   max_val = X[:, i].max()
   X_processed[:, i] = (X[:, i] - min_val) / (max_val - min_val)

for i in range(X_processed.shape[1]):
   mean = X_processed[:, i].mean()
   std_dev = X_processed[:, i].std()
   X_processed[:, i] = (X_processed[:, i] - mean) / std_dev

print(X_processed.shape)
def train_val_test_split(X, y, train_ratio=0.8, val_ratio=0.20):
   n = X.shape[0]
   train_end = int(train_ratio * n)
   val_end = int((train_ratio + val_ratio) * n)
   indices = np.random.permutation(n)
   X_train, y_train = X[indices[:train_end]], y[indices[:train_end]]
   X_val, y_val = X[indices[train_end:val_end]], y[indices[train_end:val_end]]
   X_test, y_test = X[indices[val_end:]], y[indices[val_end:]]
   return X_train, y_train, X_val, y_val, X_test, y_test


X_train, y_train, X_val1, y_val1, X_test, y_test = train_val_test_split(X_processed, y)
print(y_train_coded[1:3])
correlation_matrix = np.corrcoef(X_train, rowvar=False)
plt.figure(figsize=(8, 6))
plt.imshow(correlation_matrix, cmap='coolwarm', interpolation='nearest')
plt.colorbar(label="Correlation Coefficient")
num_features = X_train.shape[1]
feature_labels = [f'Feature {i+1}' for i in range(num_features)]
plt.xticks(range(num_features), feature_labels, rotation=45, ha="right")
plt.yticks(range(num_features), feature_labels)

for i in range(num_features):
   for j in range(num_features):
      plt.text(j, i, f"{correlation_matrix[i, j]:.2f}", ha='center', va='center', color='black')

plt.title("Correlation Heatmap")
plt.tight_layout()
plt.show()
```

Python Code for Neural network Implementation and Test:

```
def relu(x):
   return np.maximum(0, x)

def relu_derivative(x):
   return (x > 0).astype(float)

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

```
def cross_entropy_loss(predictions, labels):
    n_samples = labels.shape[0]
    log_p = - np.log(predictions[range(n_samples), labels])
    loss = np.sum(log_p) / n_samples
    return loss

def cross_entropy_loss_derivative(predictions, labels):
    n_samples = labels.shape[0]
    grad = predictions
    grad[range(n_samples), labels] -= 1
    grad = grad / n_samples
    return grad

class Neural_Network:
    def __init__(self, input_size, hidden_sizes, output_size, learning_rate=0.01, momentum=0.9):

        self.learning_rate = learning_rate
        self.momentum = momentum
        self.layers = len(hidden_sizes) + 1
        self.weights = []
        self.biases = []
        self.velocities_w = []
        self.velocities_b = []
        self.weights.append(np.random.randn(input_size, hidden_sizes[0]) * 0.01)
        self.biases.append(np.zeros((1, hidden_sizes[0])))
        self.velocities_w.append(np.zeros_like(self.weights[-1]))
        self.velocities_b.append(np.zeros_like(self.biases[-1]))


        for i in range(1, len(hidden_sizes)):
            self.weights.append(np.random.randn(hidden_sizes[i-1], hidden_sizes[i]) * 0.01)
            self.biases.append(np.zeros((1, hidden_sizes[i])))
            self.velocities_w.append(np.zeros_like(self.weights[-1]))
            self.velocities_b.append(np.zeros_like(self.biases[-1]))


        self.weights.append(np.random.randn(hidden_sizes[-1], output_size) * 0.01)
        self.biases.append(np.zeros((1, output_size)))
        self.velocities_w.append(np.zeros_like(self.weights[-1]))
        self.velocities_b.append(np.zeros_like(self.biases[-1]))

    def forward(self, X):

        self.activations = [X]
        self.z_values = []

        for i in range(self.layers - 1):
            z = np.dot(self.activations[-1], self.weights[i]) + self.biases[i]
            self.z_values.append(z)
            activation = relu(z)
            self.activations.append(activation)


        z = np.dot(self.activations[-1], self.weights[-1]) + self.biases[-1]
        self.z_values.append(z)
        output = softmax(z)
        self.activations.append(output)

        return output

    def backward(self, X, y, output):

        m = y.shape[0]
        dz = cross_entropy_loss_derivative(output, y)
        dw = np.dot(self.activations[-2].T, dz)
        db = np.sum(dz, axis=0, keepdims=True)

        self.velocities_w[-1] = self.momentum * self.velocities_w[-1] - self.learning_rate * dw
        self.velocities_b[-1] = self.momentum * self.velocities_b[-1] - self.learning_rate * db
        self.weights[-1] += self.velocities_w[-1]
        self.biases[-1] += self.velocities_b[-1]

        for i in range(self.layers - 2, -1, -1):
            dz = np.dot(dz, self.weights[i + 1].T) * relu_derivative(self.z_values[i])
            dw = np.dot(self.activations[i].T, dz)
            db = np.sum(dz, axis=0, keepdims=True)

            self.velocities_w[i] = self.momentum * self.velocities_w[i] - self.learning_rate * dw
            self.velocities_b[i] = self.momentum * self.velocities_b[i] - self.learning_rate * db
            self.weights[i] += self.velocities_w[i]
            self.biases[i] += self.velocities_b[i]


    def train(self, X, y, X_test, y_test, epochs=1000, batch_size=32):

        self.epoch_losses = []
        self.epoch_test_losses = []

        for epoch in range(epochs):

            indices = np.arange(X.shape[0])
            np.random.shuffle(indices)
            X = X[indices]
            y = y[indices]

            for start in range(0, X.shape[0], batch_size):
                end = start + batch_size
                X_batch = X[start:end]
                y_batch = y[start:end]


                output = self.forward(X_batch)
                self.backward(X_batch, y_batch, output)
            train_output = self.forward(X)
            train_loss = cross_entropy_loss(train_output, y)
            test_output = self.forward(X_test)
            test_loss = cross_entropy_loss(test_output, y_test)
            self.epoch_losses.append(train_loss)
            self.epoch_test_losses.append(test_loss)
            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Training Loss: {train_loss}, Test Loss: {test_loss}")
        return self.weights, self.biases
    def predict(self, X):
        output = self.forward(X)
        return np.argmax(output, axis=1)
def k_fold_cross_validation(X, y, X_test3, y_test3, k=5, hidden_layer_options=[[16], [8,
8],[8],[16,16]], learning_rates=[0.01,0.005,0.06], batch_sizes=[32], epochs=300):
    fold_size = len(X) // k
    indices = np.random.permutation(len(X))
    best_accuracy = 0
    best_params = {}
    losses_per_hyperparameter = {}
    for hidden_sizes in hidden_layer_options:
        for learning_rate in learning_rates:
            for batch_size in batch_sizes:
                fold_accuracies = []
                fold_test_losses = []
                fold_losses = []


                for fold in range(k):

                    val_indices = indices[fold * fold_size: (fold + 1) * fold_size]
                          train_indices = np.concatenate([indices[:fold * fold_size], indices[(fold + 1) *
fold_size:]])

                    X_train, y_train = X[train_indices], y[train_indices]
                    X_val, y_val = X[val_indices], y[val_indices]


                    input_size = X_train.shape[1]
                    output_size = 4

                    global model
                                     model = Neural_Network(input_size, hidden_sizes, output_size,
learning_rate=learning_rate, momentum=0.9)
                                  w, b = model.train(X_train, y_train, X_val, y_val, epochs=epochs,
batch_size=batch_size)
                    fold_losses.append(model.epoch_losses)
                    fold_test_losses.append(model.epoch_test_losses)
                    y_pred = model.predict(X_val)
                    accuracy = np.mean(y_pred == y_val)
                    fold_accuracies.append(accuracy)
                    print(f"Fold {fold + 1}, Validation Accuracy: {accuracy}")

                avg_accuracy = np.mean(fold_accuracies)
                avg_loss_test = np.mean(fold_test_losses, axis=0)
                avg_loss = np.mean(fold_losses, axis=0)

                hyperparam_key = f"hidden_{hidden_sizes}_lr_{learning_rate}_batch_{batch_size}"
                losses_per_hyperparameter[hyperparam_key] = {
                    "avg_loss": avg_loss,
                    "avg_loss_test": avg_loss_test,
                    "avg_accuracy": avg_accuracy
                }
                    print(f"\nAverage Validation Accuracy for hidden sizes {hidden_sizes}, learning rate
{learning_rate}, batch size {batch_size}: {avg_accuracy}\n")

                if avg_accuracy > best_accuracy:
                    best_accuracy = avg_accuracy
                    best_params = {
                        'hidden_sizes': hidden_sizes,
                        'learning_rate': learning_rate,
                        'batch_size': batch_size,
                        'avg_accuracy': avg_accuracy
                    }
                    best_weight = {
                        'weights': w,
                        'biases': b,
                    }
    for key, losses in losses_per_hyperparameter.items():
        plt.figure(figsize=(10, 5))
        plt.plot(range(epochs), losses["avg_loss"], label='Training Loss', color='blue')
        plt.plot(range(epochs), losses["avg_loss_test"], label='Test Loss', color='orange')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.title(f'Training and Test Loss for {key}')
        plt.legend()
        plt.grid(True)
        plt.show()

    print(f"Best hyperparameters: {best_params}")
                model_test = Neural_Network(input_size, best_params['hidden_sizes'], output_size,
learning_rate=best_params['learning_rate'], momentum=0.9)
    model_test.weights = best_weight['weights']
    model_test.biases = best_weight['biases']
    y_test_out = model.predict(X_test3)
    test_accuracy = np.mean(y_test_out == y_test3)
    print(f"Test Accuracy with Best Set of Hyperparameters: {test_accuracy}")

    return best_params
k = 5
average_accuracy = k_fold_cross_validation(X_train, y_train,X_val1,y_val1, k=k, epochs = 300)
```