DEEP LEARNING ASSIGNMENT

22N211 - Catherine Mary J

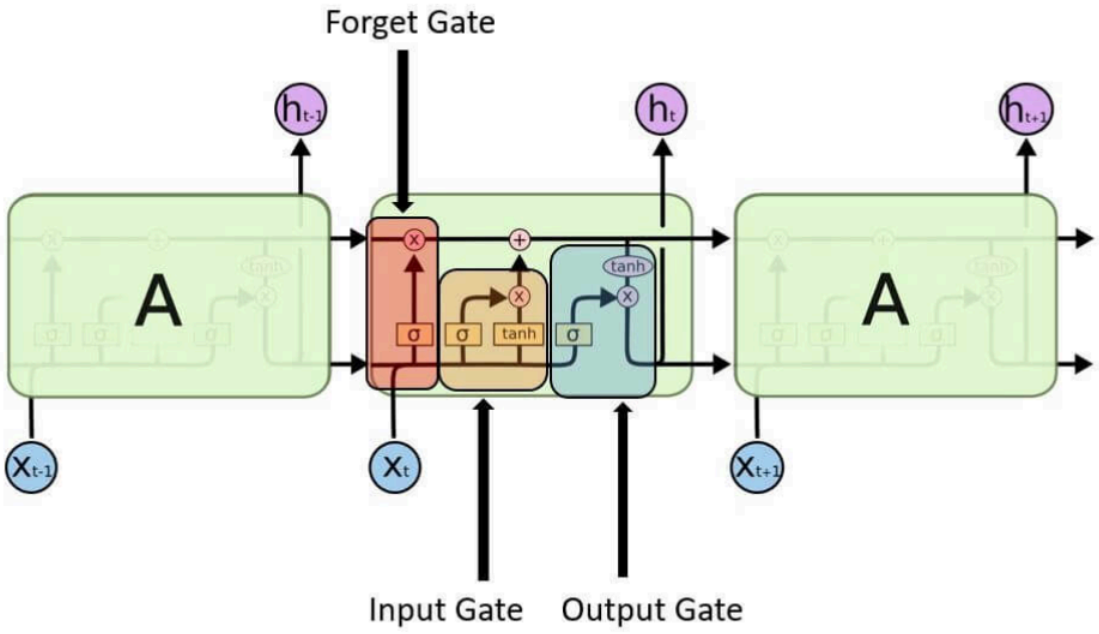
22N215 - Dhanushrii SJ

22N252 - Sneha R

PROBLEM STATEMENT:

To generate music using LSTM with the maestro dataset

LSTM (long short term memory) ARCHITECTURE:



Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning. Unlike standard feed-forward neural

networks, LSTM has feedback connections. It can process not only single data points (such as images) but also entire sequences of data (such as video or music). A general LSTM unit is composed of a cell, an input gate, an output gate, and a forget gate. The cell remembers values over arbitrary time intervals, and three gates regulate the flow of information into and out of the cell. LSTM is well-suited to classify, process, and predict the time series given of unknown duration.
Long Short- Term Memory (LSTM) networks are a modified version of recurrent neural networks, which makes it easier to remember past data in memory.

Input gate- It discovers which value from input should be used to modify the memory. The Sigmoid function decides which values to let through 0 or 1. And tanh function gives weightage to the values which are passed, deciding their level of importance ranging from -1 to 1.

$$i_t = \sigma(W_i.[h_t - 1, x_t] + bi)$$
$$C_t = \tanh(W_C.[h_t - 1, x_t] + b_C)$$

Forget gate- It discovers the details to be discarded from the block. A sigmoid function decides it. It looks at the previous state (ht-1) and the content input (Xt) and outputs a number between 0(omit this) and 1(keep this) for each number in the cell state Ct-1.

$$f_t = \sigma(W_f.[h_t - 1, x_t] + b_f)$$

Output gate- The input and the memory of the block are used to decide the output. The Sigmoid function decides which values to let through 0 or 1. And the tanh function decides which values to let through 0, 1. And tanh function gives weightage to the values which are passed, deciding their level of importance ranging from -1 to 1 and multiplied with an output of sigmoid.

$$O_t = \sigma(W_o[h_t - 1, x_t] + b_o$$
$$h_t = o_t * \tanh(C_t)$$

It represents a full RNN cell that takes the current input of the sequence xi, and outputs the current hidden state, hi, passing this to the next RNN cell for our input sequence. The inside of an LSTM cell is a lot more complicated than a traditional RNN cell, while the conventional RNN cell has a single "internal layer" acting on the current state (ht-1) and input (xt).

HOW IS LSTM USED TO GENERATE MUSIC?

Dataset: Maestro v2 Overview

Content:

1. **Audio Files**: Maestro v2 primarily consists of high-quality audio recordings of piano performances. Each audio file captures a performance of classical piano pieces.
2. **MIDI Files**: Accompanying each audio file is a MIDI file that represents the same performance in terms of note pitch, duration, and dynamics. This allows for analysis and machine learning applications.

Structure:

- The dataset is organized into various folders, often categorized by year (e.g., 2018, 2004, 2006). Each folder contains audio files and their corresponding MIDI files.

Details within Files:

- **Audio Files**: These are high-fidelity recordings of piano performances, typically in WAV format. They capture the nuances of the performance, including dynamics and expression.
- **MIDI Files**: Each MIDI file contains:
    - **Note Pitch**: The specific notes played (e.g., C4, A#5).
    - **Duration**: How long each note is held.
    - **Velocity**: The intensity with which each note is played, affecting its loudness.

Training the LSTM Model

1. **Data Preparation**:
    - **MIDI Conversion**: The MIDI files are often converted into a format that the LSTM can understand. This involves creating sequences of notes and their corresponding timings.
    - **Encoding**: Notes are typically encoded as numerical values. For example, each pitch might be represented by a unique integer, and timing can be encoded based on a set resolution (like quarter notes or eighth notes).
2. **Sequence Generation**:
    - The dataset is divided into overlapping sequences of a fixed length. For example, if the sequence length is 50 notes, the model will learn from the first 50 notes, then the next 50, and so on.
    - Each sequence is paired with the note that follows it, which acts as the target output for the model to predict.

Training Process:

- The model is trained using backpropagation and optimization techniques (like Adam or RMSprop). During training, the LSTM adjusts its internal weights based on the error between its predictions and the actual following notes.
- The loss function (commonly categorical cross-entropy for classification tasks) measures how well the model's predictions match the actual next notes.

Learning Patterns:

- As training progresses, the LSTM learns various musical patterns, including:
    - **Note Sequences**: Which notes frequently follow others (e.g., C might be followed by E).
    - **Rhythmic Structures**: Patterns in timing, such as common note durations and rests.
    - **Harmonic Relationships**: How chords and harmonies are constructed within the music.

Validation:

- To ensure that the model generalizes well to unseen music, a portion of the dataset is set aside for validation. This helps check that the model isn't just memorizing the training data.

Prediction: The LSTM takes this sequence and predicts the next note based on what it has learned. It considers the previous notes to make its prediction.

Iteration: The predicted note is then added to the sequence, and the process repeats. The LSTM keeps predicting the next note until a desired length of music is created.

Output: The final output is a new piece of music that resembles the style of the training data.

```
class MaestroDataset(Dataset):
def __init__(self, midi_files):
        self.midi_files = midi_files
        self.scaler = MinMaxScaler()
        self.data = self.preprocess_data()

    def preprocess_data(self):
        data = []
        for file in self.midi_files:
            # Load your MIDI file and convert to a suitable format
            midi_data = self.load_midi(file)
            data.append(midi_data)

        # Normalize data
        data = np.array(data)
        data = self.scaler.fit_transform(data.reshape(-1,
               data.shape[-1])).reshape(data.shape)
```

```
        return data
    def load_midi(self, file):
        pass
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        return self.data[idx]

midi_files = glob.glob('/content/drive/My Drive/maestro', recursive=True)
# Load dataset
dataset = MaestroDataset(midi_files)
```

Imports:

- The code starts by importing necessary libraries:
  - **os** and **glob** for file handling.
  - **numpy** for numerical operations.
  - **torch** and **torch.utils.data** for working with PyTorch, a deep learning library.
  - **MinMaxScaler** from **sklearn** for normalizing data.

MaestroDataset Class:

- This class represents our dataset and contains methods to load and preprocess MIDI files.

__init__ Method:

- This method is called when you create an instance of the class. It takes a list of MIDI files as input.
- It initializes the **scaler** to normalize data and calls **preprocess_data** to process the MIDI files.

preprocess_data Method:

- This method is where the actual data processing happens:
    - It creates an empty list called data.
    - For each MIDI file, it loads the data (though the actual loading logic needs to be implemented).
    - After loading, it appends the MIDI data to the data list.
    - Finally, it normalizes the data so that all values are between 0 and 1 (this helps in training models effectively).

load_midi Method:

- This is a placeholder method where you would write the code to read the MIDI files and convert them into a usable format (like a piano roll). It's currently empty and needs to be filled in.

__len__ Method:

- This method returns the number of samples in the dataset. It tells how many MIDI files have been processed.

__getitem__ Method:

- This method allows you to access a specific item in the dataset by its index (like getting the 5th MIDI file).

Loading the Dataset:

- The last part of the code uses **glob** to find all MIDI files in a specified directory.
- It creates an instance of the **MaestroDataset** class using the list of MIDI files.

```
import os
import pandas as pd
import pretty_midi
from sklearn.model_selection import train_test_split
```

```python
def midi_to_dataframe(midi_file):  # to extract features from a MIDI file
    try:
        midi_data = pretty_midi.PrettyMIDI(midi_file)
        notes = []
        for instrument in midi_data.instruments:
            for note in instrument.notes:
                notes.append({
                    'pitch': note.pitch,
                    'start': note.start,
                    'end': note.end,
                    'duration': note.end - note.start
                })
        return pd.DataFrame(notes)
    except Exception as e:
        print(f"Error processing {midi_file}: {e}")
        return pd.DataFrame()  # Return empty DataFrame on error

base_dir = '/content/drive/My Drive/maestro' # Set your base directory

dataframes = [] # Initialize a list to hold DataFrames

# Walk through the directory and read MIDI files
for root, dirs, files in os.walk(base_dir):
    for file in files:
        if file.endswith('.midi') or file.endswith('.mid'):
            midi_path = os.path.join(root, file)
            df = midi_to_dataframe(midi_path)
            if not df.empty:
                df['file_name'] = file  # Add file name for reference
                dataframes.append(df)

# Concatenate all DataFrames into one
dataset = pd.concat(dataframes, ignore_index=True)

# Now you can split the dataset into train, validation, and test sets
```

```
train_data, test_data = train_test_split(dataset, test_size=0.2,
random_state=42)
val_data, test_data = train_test_split(test_data, test_size=0.5,
random_state=42)

# Display the sizes of the splits
print(f"Training data size: {len(train_data)}")
print(f"Validation data size: {len(val_data)}")
print(f"Test data size: {len(test_data)}")
```

Importing Libraries

The libraries that are imported are

- **Os** for providing functions to interact with the operating system, particularly for file and directory handling.
- **pandas (pd)** for A library for data manipulation and analysis. It's used here to store the musical note data in a structured format (DataFrame).
- **Pretty_midi** for A library to process and analyze MIDI files. It can extract musical information such as notes, pitch, timing, and duration from MIDI files.
- **Train_test_split** for A function from sklearn.model_selection used to split the dataset into training, validation, and test sets.

Function to Extract Features from MIDI Files midi_to_dataframe(midi_file):

- This function reads a single MIDI file and extracts note-related features (pitch, start time, end time, duration).
- **pretty_midi.PrettyMIDI(midi_file)**: Loads the MIDI file using the pretty_midi library.
- **Notes extraction**: It loops through all the instruments in the MIDI file and extracts all the notes they play.
    - **note.pitch:** The pitch of the note (e.g., C, D, E).

    - **note.start:** The start time of the note in seconds.
    - **note.end:** The end time of the note.
    - **note.end - note.start:** The duration of the note.
- Returns a pandas DataFrame containing the extracted features for each note.
- Error handling: If an error occurs while processing the file, it prints an error message and returns an empty DataFrame.

Directory Setup and Data Processing

base_dir: Specifies the directory where the MIDI files are stored. In this case, it's mounted to Google Drive.

dataframes: Initializes an empty list to store the DataFrames generated from each MIDI file.

Walking Through Directory and Processing Files

os.walk(base_dir): Walks through the directory tree starting at base_dir, traversing all subdirectories and files.

- **root:** The current directory path.
- **dirs:** List of subdirectories.
- **files:** List of files in the current directory.

Processing MIDI files:

- For each file, the code checks if the file extension is either .midi or .mid (MIDI file formats).
- If it is a MIDI file, the full file path is created using **os.path.join**(root, file).
- The **midi_to_dataframe(midi_path)** function is called to process the MIDI file and extract features.
- If the DataFrame is not empty, the file name is added as a new column **(df['file_name'] = file)** for reference, and the DataFrame is appended to the **dataframes** list.

Concatenating DataFrames

pd.concat(dataframes, ignore_index=True): Concatenates all DataFrames in the **dataframes** list into a single large DataFrame called **dataset**. The **ignore_index=True** ensures that the resulting DataFrame will have a continuous index.

Splitting the Dataset

train_test_split(dataset, test_size=0.2, random_state=42)

- Splits the dataset into training (80%) and test (20%) sets.
- The **test_size=0.2** parameter specifies that 20% of the data should be allocated to the test set.
- **random_state=42** ensures reproducibility by fixing the randomness of the split.

train_test_split(test_data, test_size=0.5, random_state=42)

- Splits the previously created test set into validation (50%) and final test (50%) sets, creating equal-sized validation and test datasets.

Printing Dataset Sizes

Prints the size of the training, validation, and test sets.

```
import torch
import torch.nn as nn

class SimpleModel(nn.Module):# Example model definition
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc1 = nn.Linear(10, 5)
        self.fc2 = nn.Linear(5, 1)
        # Make the second layer non-trainable
        for param in self.fc2.parameters():
            param.requires_grad = False
```

```
      def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        return x

model = SimpleModel() # Create the model

trainable_params = [] # Lists to hold parameters and their counts
non_trainable_params = [] # Iterate through model parameters

for name, param in model.named_parameters():
    if param.requires_grad:
        trainable_params.append(param)

      else:
        non_trainable_params.append(param)

# Calculate total number of trainable and non-trainable parameters
total_trainable = sum(param.numel() for param in trainable_params)
total_non_trainable = sum(param.numel() for param in
non_trainable_params)

print(f"Number of trainable parameters: {total_trainable}")
for param in trainable_params:
    print(f"Trainable: {param.shape}")
print(f"Number of non-trainable parameters:{total_non_trainable}")
for param in non_trainable_params:
    print(f"Non-Trainable: {param.shape}")
```

Importing Necessary Libraries

torch: The main PyTorch library for building neural networks.

torch.nn as nn: A submodule that provides all the building blocks for defining and using neural networks, such as layers and activation functions.

Defining the SimpleModel Class

class SimpleModel(nn.Module): This defines a neural network model class by inheriting from PyTorch's **nn.Module.**

def __init__(self): The constructor method where the layers of the model are initialized.

self.fc1 = nn.Linear(10, 5):

- Defines a fully connected layer (also known as a dense layer).
- It takes an input of size 10 and outputs 5 units (neurons).

self.fc2 = nn.Linear(5, 1)

- Defines the second fully connected layer, which takes an input of size 5 (output from **fc1**) and outputs a single value.

Making the second layer non-trainable:

- The loop for param in self.fc2.parameters(): **param.requires_grad = False** disables gradient computation for the parameters (weights) of the second layer.
- This means that the weights of **fc2** won't be updated during training, making it a non-trainable layer.

Forward Pass Method

forward (self, x): Defines how the input x flows through the network (the forward pass).

x = self.fc1(x): The input is passed through the first layer (fc1).

x = self.fc2(x): The output of fc1 is passed through the second layer (fc2).

return x: The output of the second layer is returned as the final output.

Model Creation

model = SimpleModel(): Instantiates the model, creating an object of the **SimpleModel** class.

Parameter Tracking

Two empty lists are initialized:

- **trainable_params:** To store the parameters (weights) that will be trained (updated during training).
- **non_trainable_params:** To store the parameters that are non-trainable (will not be updated).

Iterating Through Model Parameters

for name, param in model.named_parameters(): Iterates over all the parameters of the model. The **named_parameters()** function returns both the parameter names and their values (weights).

if param.requires_grad: Checks if the parameter has **requires_grad=True**, which indicates whether it will be trained.

- If **requires_grad=True**, it appends the parameter to **trainable_params.**
- Otherwise, it appends the parameter to **non_trainable_params.**

Calculating and Printing Parameter Counts

param.numel(): Returns the total number of elements (weights) in a parameter (e.g., for a layer with a weight matrix of size [10, 5], **param.numel()** would return 50).

sum(param.numel() for param in trainable_params): Calculates the total number of trainable parameters by summing up the number of elements for each trainable parameter.

sum(param.numel() for param in non_trainable_params): Similarly, calculates the total number of non-trainable parameters.

Output Results

Print the total count of trainable and non-trainable parameters

- The total number of trainable parameters is printed, followed by the shape of each parameter.
- The total number of non-trainable parameters is printed, followed by the shape of each non-trainable parameter.

```
import torch.nn as nn

class MusicGeneratorLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(MusicGeneratorLSTM, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
 batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.activation = nn.Softmax(dim=-1)  # Output layer activation

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.fc(out[:, -1, :])  # Take the last time step
        return self.activation(out)

# Initialize model
input_size = 128  # Example size
hidden_size = 256
num_layers = 2
output_size = 128  # Example size
model = MusicGeneratorLSTM(input_size, hidden_size, num_layers,
output_size)
```

Importing Required Libraries

torch.nn as nn: This imports PyTorch's neural network module, which contains layers and functions to define and train neural networks.

Defining the MusicGeneratorLSTM Class

This line defines a new class **MusicGeneratorLSTM,** which inherits from **nn.Module,** PyTorch's base class for all neural network models. The class will represent the architecture of the LSTM-based model for music generation.

Initializing the Model Layers

__init__: This is the constructor where the model's layers are defined.

input_size: The number of input features (e.g., size of the input data). In music generation, this could represent the size of the input features like pitch, duration, or other musical characteristics.

hidden_size: The number of units in the hidden layer of the LSTM. This controls the capacity of the LSTM model.

num_layers: The number of LSTM layers stacked on top of each other. More layers allow the model to capture more complex patterns.

output_size: The size of the output layer, which could correspond to the number of possible musical notes or other output features.

LSTM Layer

nn.LSTM: This defines the Long Short-Term Memory (LSTM) layer, which is useful for processing sequences (such as sequences of musical notes).

input_size: The number of input features at each time step.

hidden_size: The size of the hidden state and the output of the LSTM.

num_layers: The number of LSTM layers stacked.

batch_first=True: This argument indicates that the input data will have the shape **(batch_size, sequence_length, input_size),** meaning that the batch dimension comes first.

Fully Connected Layer

nn.Linear: A fully connected (dense) layer that connects the LSTM output to the final output. It maps the hidden state of the LSTM (of size **hidden_size**) to the desired output size **(output_size)**, which could represent the number of possible musical notes or actions.

Activation Function

nn.Softmax(dim=-1): This activation function is applied to the final output of the model to convert it into a probability distribution across the output classes (e.g., possible musical notes). The **dim=-1** argument ensures that the softmax is applied along the last dimension, which corresponds to the output size.

Defining the Forward Pass

forward(self, x): This method defines how the input data x will flow through the model (the forward pass).

out, _ = self.lstm(x): Passes the input data x through the LSTM layer. The output contains the hidden states for all time steps, and _ represents the final hidden state and cell state of the LSTM (which can be ignored here).

out[:, -1, :]: Takes the hidden state from the last time step of the sequence. This reduces the LSTM's output to just the final time step, which is then passed to the next layer.

self.fc(out): Passes the final hidden state through the fully connected layer to generate the output.

return self.activation(out): The output is passed through the softmax activation function to produce a probability distribution over the output classes.

Data Preparation & Preprocessing

1. Data Creation

```
# Example data preparation
data = {
    'feature1': [0.1 * i for i in range(1, 21)],
    'feature2': [1.0 + 0.2 * i for i in range(20)],
    'feature3': [0.5 + 0.1 * i for i in range(20)],
    'feature4': [1.5 + 0.1 * i for i in range(20)],
    'feature5': [2.0 + 0.1 * i for i in range(20)],
    'feature6': [2.5 + 0.1 * i for i in range(20)],
    'feature7': [3.0 + 0.1 * i for i in range(20)],
    'feature8': [3.5 + 0.1 * i for i in range(20)],
    'feature9': [4.0 + 0.1 * i for i in range(20)],
    'feature10': [4.5 + 0.1 * i for i in range(20)],
    'target': [0.5 + 0.1 * i for i in range(20)]
}
dataset = pd.DataFrame(data)
```

Here, we are creating a **synthetic dataset** with 20 samples and 10 features (**feature1** to **feature10**) along with the **target** value. This represents the dataset that will be used to train and validate the model.

2. Data Splitting

```
# Split the dataset into training and validation sets
train_data = dataset.sample(frac=0.8, random_state=42)
val_data = dataset.drop(train_data.index)
```

Training Set: 80% of the data is used for training (**train_data**).

Validation Set: The remaining 20% is used for validation (**val_data**).

This split allows the model to learn on the training set and be evaluated on unseen data to assess its generalization ability.

3. Normalization

```
def normalize_data(train_data, val_data): # Normalize data
    scaler = StandardScaler()
    train_features = train_data.iloc[:, :-1]
    val_features = val_data.iloc[:, :-1]

    train_features_scaled = scaler.fit_transform(train_features)
    val_features_scaled = scaler.transform(val_features)
    train_data.iloc[:, :-1] = train_features_scaled
    val_data.iloc[:, :-1] = val_features_scaled

    return train_data, val_data

# Normalize data
train_data, val_data = normalize_data(train_data, val_data)
```

The features in the training and validation sets are normalized using **StandardScaler,** which standardizes the data by removing the mean and scaling to unit variance. This step ensures that the model's input features have similar scales, making the training process more stable and faster.
The scaler is fitted only on training features (**scaler.fit_transform(train_features)**) and then applied to validation features (**scaler.transform(val_features)**) to avoid data leakage.

Custom Dataset and DataLoader Setup

Custom Dataset:

```
class CustomDataset(Dataset): # Custom dataset class
    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)
```

```
    def __getitem__(self, idx):
        features = torch.tensor(self.data.iloc[idx, :-1].values,
dtype=torch.float32)
        target = torch.tensor(self.data.iloc[idx, -1],
dtype=torch.float32)
        return features, target
```

This class wraps around the pandas DataFrame and allows access to features and target values in a PyTorch-friendly format.

DataLoader:

```
# Create dataset objects
train_dataset = CustomDataset(train_data)
val_dataset = CustomDataset(val_data)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=4, shuffle=False)
```

train_loader: Loads the training data in batches (**batch_size=4**) with shuffle=True to ensure randomness during training.
val_loader: Loads the validation data with **shuffle=False** (since validation doesn't need shuffling).

3. **Model Definition**

```
# Improved model definition with customizable activation functions
class ImprovedModel(nn.Module):
    def __init__(self, hidden_layers, dropout_rate, activation_fn):
        super(ImprovedModel, self).__init__()
        self.layers = nn.ModuleList()
        self.layers.append(nn.Linear(10, hidden_layers[0]))
        self.layers.append(nn.BatchNorm1d(hidden_layers[0]))
        # Batch Normalization after the first layer
        for i in range(len(hidden_layers) - 1):
            if activation_fn == 'relu':
                self.layers.append(nn.ReLU())
            elif activation_fn == 'leaky_relu':
                self.layers.append(nn.LeakyReLU())
```

```
            elif activation_fn == 'elu':
                self.layers.append(nn.ELU())
            elif activation_fn == 'swish':
                self.layers.append(nn.SiLU()# PyTorch's implementation of Swish
            else:
                raise ValueError("Unknown activation function.")
            self.layers.append(nn.Linear(hidden_layers[i],hidden_layers[i+1]))
            self.layers.append(nn.BatchNorm1d(hidden_layers[i + 1]))
            # Batch Normalization after each layer
            self.layers.append(nn.Dropout(dropout_rate))
        self.layers.append(nn.Linear(hidden_layers[-1], 1))

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
```

Input Layer: Takes 10 features and maps them to the first hidden layer.
Hidden Layers:

- Customizable number of hidden layers **(hidden_layers)**, each followed by a batch normalization, activation, and dropout layer.
- Activation function (activation_fn) can be customized, such as 'relu', **'leaky_relu'**, **'elu'**, or **'swish'.**
- Batch Normalization: Helps stabilize the network by normalizing inputs to each layer.
- Dropout: Helps prevent overfitting by dropping a fraction of neurons during training.

Output Layer: Maps the output of the last hidden layer to a single value, suitable for regression tasks.

4. Training Process

```
# Improved training function with early stopping
def train_model(model, train_loader, val_loader, epochs=200,
weight_decay=1e-5, lr=0.001, patience=10):
    optimizer = optim.Adam(model.parameters(), lr=lr,
```

```
weight_decay=weight_decay)
    criterion = nn.MSELoss()
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min',
 patience=5, factor=0.5)
    train_losses = []
    val_losses = []
    best_val_loss = float('inf')
    epochs_no_improve = 0
    for epoch in range(epochs):
        model.train()
        epoch_train_loss = 0.0
        for features, targets in train_loader:
            optimizer.zero_grad()
            outputs = model(features)
            loss = criterion(outputs, targets.view(-1, 1))
            loss.backward()
            optimizer.step()
            epoch_train_loss += loss.item()
        avg_train_loss = epoch_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)
        model.eval()
        epoch_val_loss = 0.0
        with torch.no_grad():
            for val_features, val_targets in val_loader:
                val_outputs = model(val_features)
                val_loss = criterion(val_outputs, val_targets.view(-1, 1))
                epoch_val_loss += val_loss.item()
        avg_val_loss = epoch_val_loss / len(val_loader)

        val_losses.append(avg_val_loss)
        scheduler.step(avg_val_loss) # Step the scheduler
        print(f'Epoch [{epoch + 1}/{epochs}], Train Loss:
{avg_train_loss:.4f}, Validation Loss: {avg_val_loss:.4f}')
        if avg_val_loss < best_val_loss: # Early stopping
            best_val_loss = avg_val_loss
            epochs_no_improve = 0
```

```
        else:
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print("Early stopping!")
                break
    return train_losses, val_losses

train_losses, val_losses = train_model(model, train_loader, val_loader,
 epochs=200, weight_decay=weight_decay, lr=learning_rate)
```

Optimizer: **Adam** is used to update model weights. It also includes weight decay (**weight_decay**) to regularize the model.

Loss Function: **nn.MSELoss()** is used for regression tasks to minimize the mean squared difference between actual and predicted values.

Learning Rate Scheduler: Reduces the learning rate when validation loss plateaus.

Training Loop:

- Performs forward propagation, computes loss, and updates weights using backpropagation.
- **optimizer.step():** Updates model parameters.
- **optimizer.zero_grad():** Clears previous gradients to prevent accumulation.

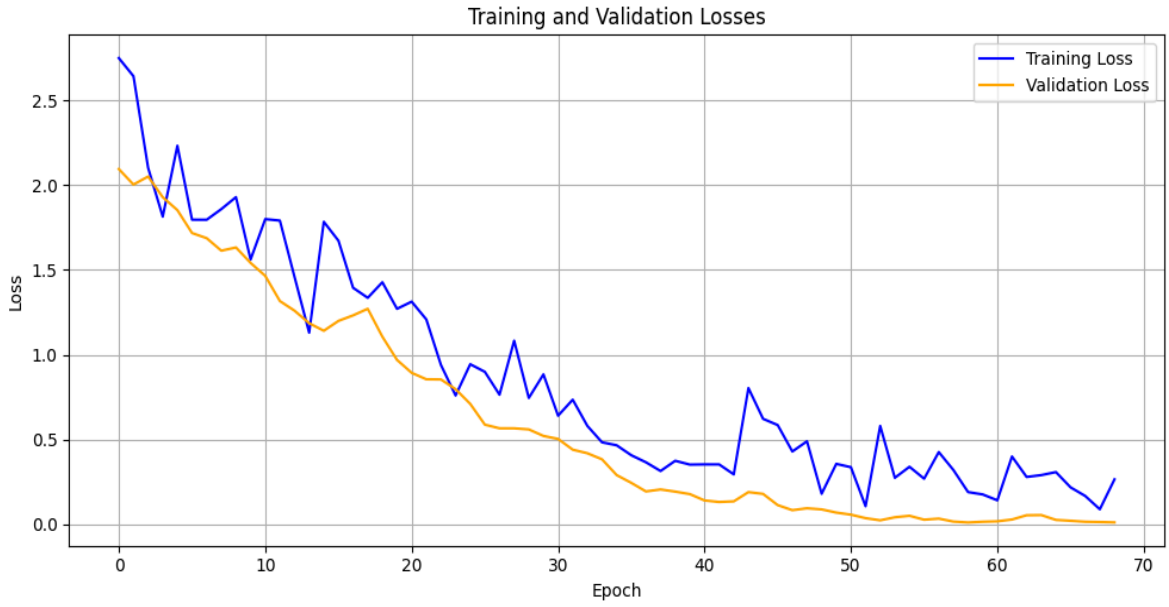
Validation Loop: Evaluates the model on the validation dataset without updating the model weights.

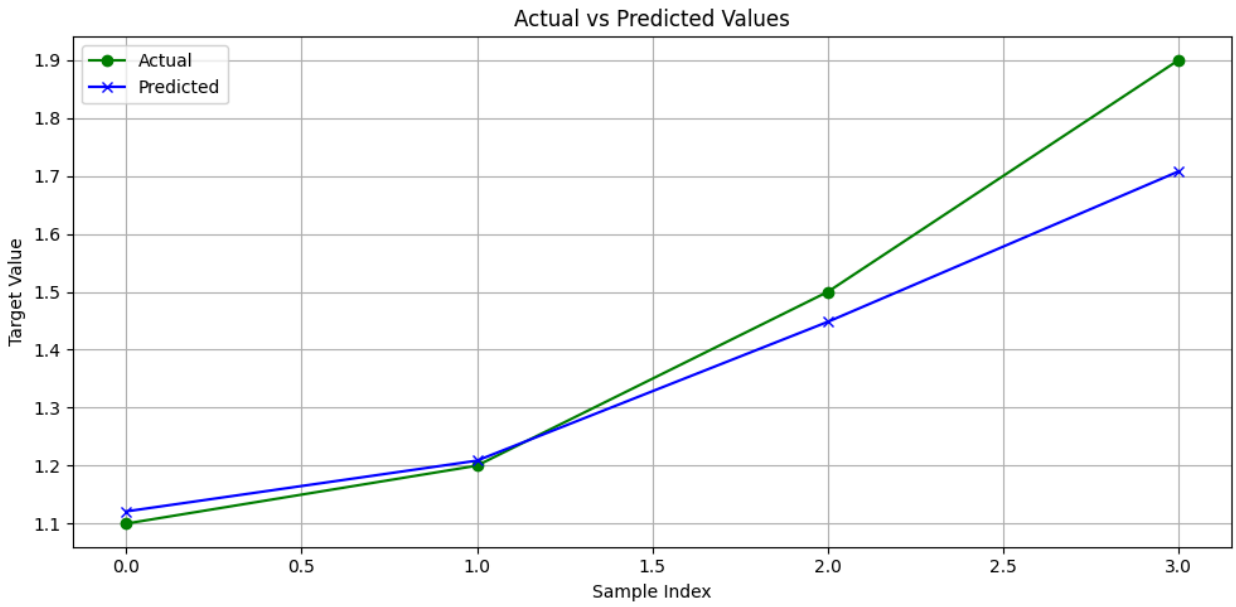
Early Stopping: Monitors validation loss and stops training if it doesn't improve for a given number of epochs (patience).

5. **Visualization and Evaluation**

Loss Plot: Plots training and validation losses across epochs, helping identify overfitting (where validation loss is high compared to training loss).

```python
# Plot the losses

def plot_loss(train_losses, val_losses):

    plt.figure(figsize=(10, 5))

    plt.plot(train_losses, label='Training Loss', color='blue')

    plt.plot(val_losses, label='Validation Loss', color='orange')

    plt.title('Training and Validation Losses')

    plt.xlabel('Epoch')

    plt.ylabel('Loss')

    plt.legend()

    plt.grid()

    plt.tight_layout()

    plt.show()

plot_loss(train_losses, val_losses)
```





Predictions Visualization:

```python
# Visualize actual vs. predicted values
def visualize_predictions(model, val_loader):
    model.eval()
    all_targets = []
    all_predictions = []

    with torch.no_grad():
        for val_features, val_targets in val_loader:
            val_outputs = model(val_features).view(-1)
            all_targets.extend(val_targets.numpy())
            all_predictions.extend(val_outputs.numpy())

    all_targets = np.array(all_targets)
    all_predictions = np.array(all_predictions)

    plt.figure(figsize=(10, 5))
    plt.plot(all_targets, marker='o', label='Actual', color='green')
    plt.plot(all_predictions, marker='x', label='Predicted', color='blue')
    plt.title('Actual vs Predicted Values')
    plt.xlabel('Sample Index')
    plt.ylabel('Target Value')
    plt.legend()
    plt.grid()
```

```
    plt.tight_layout()
    plt.show()
    return all_targets, all_predictions

# Calculate metrics after training
all_targets, all_predictions = visualize_predictions(model, val_loader)
```

Plots actual vs. predicted values for the validation set, helping visualize the model's performance and any major deviations.

Metrics Calculation:

```
def calculate_metrics(targets, predictions):
    targets = np.array(targets)
    predictions = np.array(predictions)

    if len(targets) == 0 or len(predictions) == 0:
        print("Error: Targets or predictions are empty.")
        return np.nan, np.nan, np.nan

    mae = np.mean(np.abs(targets - predictions))
    rmse = np.sqrt(np.mean((targets - predictions) ** 2))
    accuracy = np.mean(np.round(targets) == np.round(predictions))
    # May not be suitable for continuous values
    return mae, rmse, accuracy

# Calculate metrics
mae, rmse, accuracy = calculate_metrics(all_targets, all_predictions)
print(f'Mean Absolute Error: {mae:.4f}')
print(f'Root Mean Squared Error: {rmse:.4f}')
print(f'Accuracy: {accuracy:.4f}')
```

MAE (Mean Absolute Error): Measures the average magnitude of the errors in the predictions without considering direction.

RMSE (Root Mean Squared Error): Measures the square root of the average squared differences between predicted and actual values. It penalizes larger errors more.

Accuracy: Measures how often rounded predictions match the rounded target values (not the best metric for regression).

1. Model Definition

```
model = Sequential()
model.add(LSTM(256, input_shape=(x.shape[1], x.shape[2]),
return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(256, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(128))
model.add(Dense(64))
model.add(Dropout(0.3))
model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer='adam')
```

- **Input Layer**: Accepts a sequence of time steps with multiple features (based on **input_shape**).
- **3 LSTM Layers**:
    - The first and second LSTM layers return sequences to allow stacking of LSTMs.
    - The third LSTM layer returns a single output.
- **Dense Layers**:
    - A Dense layer with 64 neurons further processes the output of the last LSTM layer.
    - The final Dense layer with 1 output neuron is used for prediction.
- **Dropout Layers**:
    - Each Dropout layer helps prevent overfitting by dropping units randomly.

This model is particularly designed to capture sequential dependencies in time-series data using LSTMs, while the dense layers and dropout layers help in combining features and preventing overfitting, respectively.

Model Compilation:

Loss Function:

- **loss='mean_squared_error'**: The model is being trained to minimize the Mean Squared Error (MSE), which is common for regression tasks. It calculates the average squared difference between the predicted values and the actual target values.

Optimizer:

- **Optimizer='adam':** The Adam optimizer is being used to adjust the model weights during training. It is an adaptive optimizer that generally provides good results and is computationally efficient.

Increased Model Complexity:

```
# Adjust the model definition for more complexity if needed
hidden_layers = [1024, 512, 256]  # More neurons
dropout_rate = 0.3  # Increased dropout rate to reduce overfitting
# Create the model
model = ImprovedModel(hidden_layers, dropout_rate, activation_function)
# Train the model with adjusted parameters
train_losses, val_losses = train_model(
    model,
    train_loader,
    val_loader,
    epochs=300,  # More epochs for training
    weight_decay=1e-4,  # Adjusted weight decay
```

```
    lr=0.0005,  # A smaller learning rate
    patience=15  # Longer patience for early stopping
)
# Calculate metrics after training
all_targets, all_predictions = visualize_predictions(model, val_loader,
note_names)
# Calculate and print metrics
mae, rmse, accuracy = calculate_metrics(all_targets, all_predictions)
print(f'Mean Absolute Error: {mae:.4f}')
print(f'Root Mean Squared Error: {rmse:.4f}')
print(f'Accuracy: {accuracy:.4f}')
```

- **Hidden Layers**: [1024, 512, 256]—The hidden layers have more neurons compared to the previous configuration. This increased complexity allows the model to learn more nuanced relationships in the data.
- **Dropout Rate**: 0.3—Increased dropout rate compared to 0.2 in the previous version. This helps in regularizing the model and reducing overfitting.
- **Epochs**: Increased from 200 to 300 to allow for longer training and potentially better convergence.
- **Weight Decay**: Adjusted to 1e-4 for stronger L2 regularization, which can help prevent overfitting, especially with the larger model.
- **Learning Rate**: Reduced to 0.0005 from 0.001 to allow for a more careful optimization process, especially given the increased complexity of the model.
- **Patience**: Increased to 15 from 10 to prevent premature stopping and allow the model more chances to improve during training.

```
Sample Name | Actual Value | Predicted Value
-----------------------------------------------
Note 1          | 1.1000       | 1.1557
Note 2          | 1.2000       | 1.2222
Note 3          | 1.5000       | 1.4235
Note 4          | 1.9000       | 1.6754
Mean Absolute Error: 0.0948
Root Mean Squared Error: 0.1224
Accuracy: 0.7500
```

1. Predicted Values Generation

```
# Sample predicted values (for demonstration, generating 30 random values)
np.random.seed(0)  # For reproducibility
predicted_values = np.random.uniform(1.0, 2.0, 30).tolist()  # Random
values between 1.0 and 2.0
```

np.random.seed(0): Sets the seed for the random number generator to ensure that the results are reproducible.

np.random.uniform(1.0, 2.0, 30): Generates 30 random values between 1.0 and 2.0, representing a sample of predicted values. These values are converted to a list with .tolist().

2. Normalize Predictions

```
def normalize_predictions(predictions, min_midi=60, max_midi=71):
    # Normalize predictions to the range of MIDI notes
    min_pred = min(predictions)
    max_pred = max(predictions)
    scaled_predictions = [
        min_midi + (max_midi - min_midi) * (pred - min_pred) / (max_pred -
min_pred)
        for pred in predictions
    ]
    return scaled_predictions
```

This function normalizes the predicted values to fit within the range of MIDI note numbers.

- **Input Arguments**:
  - predictions: A list of predicted values.
  - min_midi=60 and max_midi=71: Define the range of MIDI notes to map the predictions to, where 60 corresponds to C4 and 71 to B4.
- **Normalization Logic**:
  - **min_pred** and **max_pred**: Find the minimum and maximum values from the **predictions** list.
  - **scaled_predictions:** For each pred in **predictions**, normalize it to the range defined by **min_midi** and **max_midi** using the formula:

$$\text{scaled value} = \text{min\_midi} + (\text{max\_midi} - \text{min\_midi}) \cdot \frac{\text{pred} - \text{min\_pred}}{\text{max\_pred} - \text{min\_pred}}$$

- This formula scales the values from their original range to the desired MIDI range.

3. Map Normalized Output to Musical Notes

```
def map_output_to_notes(predictions):
    # Define a mapping from MIDI note numbers to note names
    midi_to_note = {
        60: 'C4', 61: 'C#4', 62: 'D4', 63: 'D#4', 64: 'E4', 65: 'F4',
        66: 'F#4', 67: 'G4', 68: 'G#4', 69: 'A4', 70: 'A#4', 71: 'B4',
        72: 'C5', 73: 'C#5', 74: 'D5', 75: 'D#5', 76: 'E5', 77: 'F5',
        78: 'F#5', 79: 'G5', 80: 'G#5', 81: 'A5', 82: 'A#5', 83: 'B5'
    }
    note_names = [] # Map predictions to notes
    for pred in predictions:
        midi_note = round(pred)
        note_name = midi_to_note.get(midi_note, "Unknown")
        note_names.append(note_name)
    return note_names
```

This function maps the normalized predictions to musical note names.

- **MIDI to Note Mapping**:

    - midi_to_note: A dictionary that maps MIDI note numbers (e.g., 60 for C4) to their corresponding note names.
- **Mapping Logic**:
    - For each normalized value in predictions, round it to the nearest integer to find the corresponding MIDI note.
    - Use midi_to_note.get(midi_note, "Unknown") to get the note name. If the MIDI value isn't in the dictionary, return "Unknown".

4. Normalization and Mapping

```
# Normalize the predicted values
normalized_predictions = normalize_predictions(predicted_values)
# Get the predicted notes from normalized values
predicted_notes = map_output_to_notes(normalized_predictions)

print("Normalized Predictions and Corresponding Notes:") # Print results

for i, (norm_pred, note) in enumerate(zip(normalized_predictions,
predicted_notes), start=1):

    print(f"Note {i}: Normalized Value: {norm_pred:.4f}, Musical Note:
{note}")
```

normalize_predictions(predicted_values): Normalizes the predicted values to the range of MIDI notes.

map_output_to_notes(normalized_predictions): Maps the normalized predictions to the corresponding musical notes.

Audio Synthesis

```
# Function to generate a sine wave for a given note

def generate_sine_wave(note, duration=0.5, sample_rate=44100):

    frequencies = {

        'C4': 261.63, 'C#4': 277.18, 'D4': 293.66, 'D#4': 311.13,

        'E4': 329.63, 'F4': 349.23, 'F#4': 369.99, 'G4': 392.00,
```

```
        'G#4': 415.30, 'A4': 440.00, 'A#4': 466.16, 'B4': 493.88,
        'C5': 523.25, 'C#5': 554.37, 'D5': 587.33, 'D#5': 622.25,
        'E5': 659.25, 'F5': 698.46, 'F#5': 739.99, 'G5': 783.99,
        'G#5': 830.61, 'A5': 880.00, 'A#5': 932.33, 'B5': 987.77
    }
    frequency = frequencies.get(note, 440)#Default to A4 if note not found
    t = np.linspace(0, duration, int(sample_rate * duration), False)
    wave = 0.5 * np.sin(2 * np.pi * frequency * t)
    return wave
# Function to create audio file from predicted notes
def create_audio_from_notes(predicted_notes, filename='output.wav'):
    audio = np.array([])
    for note in predicted_notes:
        wave = generate_sine_wave(note)
        audio = np.concatenate((audio, wave))
    # Normalize audio to the range of int16
    audio = np.int16(audio / np.max(np.abs(audio)) * 32767)
    write(filename, 44100, audio) # Save as a .wav file
    sound = AudioSegment.from_wav(filename) # convert to MP3 using pydub
    sound.export(filename.replace('.wav', '.mp3'), format='mp3')
# List of predicted notes
predicted_notes = [
    'F#4', 'G#4', 'G4', 'F#4', 'F4', 'G4', 'F4', 'A#4',
    'B4', 'E4', 'A4', 'F#4', 'F#4', 'A#4', 'C#4', 'C#4',
    'C4', 'A4', 'A4', 'A#4', 'B4', 'A4', 'F4', 'A4',
    'C#4', 'G4', 'C#4', 'B4', 'F#4', 'F4'
]
```

```
create_audio_from_notes(predicted_notes) # Create audio file

# Play the generated audio file

Audio('output.mp3')  # can also use 'output.wav'
```

Generate Sine Waves: The **generate_sine_wave()** function creates sine waves for each note using a predefined frequency dictionary and **numpy**. A 0.5 amplitude factor prevents clipping.

Combine Waves: The **create_audio_from_notes()** function concatenates sine waves for each note, normalizing the result to fit in the int16 range for saving as a .wav file.

Save Audio: The **.wav** file is saved with a sample rate of 44,100 Hz, and optionally converted to **.mp3** using **pydub** for easier playback.

Playback: The generated audio file can be played directly using the Audio() command, suitable for interactive environments like Jupyter.

Predicted Notes: The code converts a list of predicted notes into an audible format, useful for applications like music generation, transcription, or auditory data analysis.