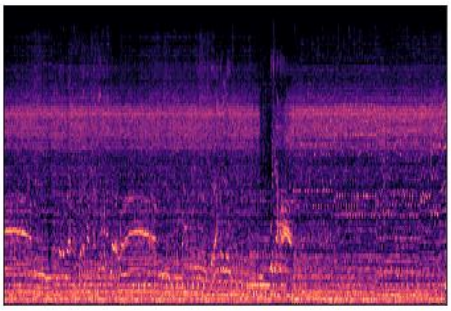
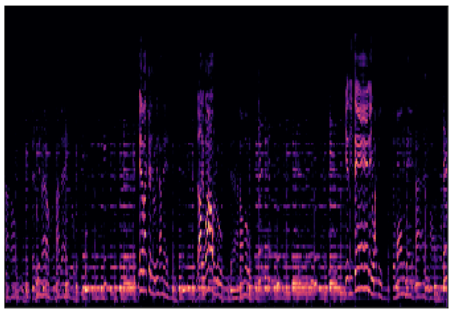
DATASET

The dataset used for this music genre classification task comprises images organized into 10 genre-specific folders: **blues, classical, country, disco, hiphop, jazz, metal, pop, reggae, and rock.** Each folder contains spectrogram or waveform representations of audio clips corresponding to its genre, serving as input for CNN-based models. These images were extracted from the GTZAN dataset, a widely used benchmark in music information retrieval tasks. The data was preprocessed by resizing each image to a uniform size of 128×128 pixels and normalized for training. This structured organization allows for efficient loading and labeling of genre data, making it suitable for supervised classification tasks using deep learning architectures.

https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification/data





EXTRACTED FEATURES

In addition to image-based data, a tabular dataset containing pre-extracted audio features was used for sequence modeling. This CSV file includes 30-second audio segments with 20 Mel-Frequency Cepstral Coefficients (MFCCs), each represented by its mean and variance—resulting in 40 features per sample. These MFCCs capture the timbral texture of audio clips, making them highly relevant for genre classification tasks. The data was reshaped into a (20, 2) structure to simulate temporal sequences, enabling the use of LSTM and CNN-LSTM models. Each sample is labeled with one of the 10 genres, encoded using one-hot encoding for categorical classification.

filename	length	chroma_st	chroma_st	rms_mean	rms_var	spectral_c	spectral_c	spectral_b	spectral_b	rolloff_mea	rolloff_var	zero_cross	zero_cross	harmony_r	harmony_v	perceptr_r	perceptr_v
blues.000(	661794	0.350088	0.088757	0.130228	0.002827	1784.166	129774.1	2002.449	85882.76	3805.84	901505.4	0.083045	0.000767	-4.53E-05	0.008172	7.78E-06	0.005698
blues.000(	661794	0.340914	0.09498	0.095948	0.002373	1530.177	375850.1	2039.037	213843.8	3550.522	2977893	0.05604	0.001448	0.00014	0.005099	-0.00018	0.003063
blues.000(	661794	0.363637	0.085275	0.17557	0.002746	1552.812	156467.6	1747.702	76254.19	3042.26	784034.5	0.076291	0.001007	2.11E-06	0.016342	-1.95E-05	0.007458
blues.000(	661794	0.404785	0.093999	0.141093	0.006346	1070.107	184355.9	1596.413	166441.5	2184.746	1493194	0.033309	0.000423	4.58E-07	0.019054	-1.45E-05	0.002712
blues.000(	661794	0.308526	0.087641	0.091529	0.002303	1835.004	343399.9	1748.172	88445.21	3579.758	1572978	0.101461	0.001954	-1.76E-05	0.004814	-1.01E-05	0.003094
blues.000(	661794	0.302456	0.087532	0.103494	0.003981	1831.994	1030482	1729.653	201910.5	3481.518	3274440	0.094042	0.006233	1.96E-07	0.008083	-2.65E-05	0.003242

FCNN

PRE-PROCESSING AND TRAINING

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Load the dataset (CSV)
df = pd.read_csv(r"C:\Users\menda\Downloads\DNN-music _dataset\Data\features_30_sec.csv")

# Drop non-feature columns
X = df.drop(['filename', 'length', 'label'], axis=1)

# Encode the labels (genres)
le = LabelEncoder()
y = le.fit_transform(df['label'])

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
```

This code performs the essential preprocessing steps for building a machine learning model to classify music genres. It begins by loading a CSV file containing audio features extracted from 30-second music clips. Non-informative columns like filenames and clip length are dropped, leaving only the numerical features for training. The genre labels are then encoded into numerical values using LabelEncoder, making them suitable for classification. The feature values are standardized using StandardScaler to ensure consistent scaling across all features, which improves model performance and training stability. Finally, the dataset is split into training and testing sets, with 80% used for training and 20% for evaluating the model's performance.

```python
import numpy as np

class FCNN:
    def __init__(self, input_size, hidden_sizes, output_size, learning_rate=0.01):
        self.lr = learning_rate
        self.params = {}
        self.init_weights(input_size, hidden_sizes, output_size)

    def init_weights(self, input_size, hidden_sizes, output_size):
        layers = [input_size] + hidden_sizes + [output_size]
        for i in range(len(layers) - 1):
            limit = np.sqrt(6 / (layers[i] + layers[i+1]))
            self.params[f"W{i+1}"] = np.random.uniform(-limit, limit, (layers[i], layers[i+1]))

            self.params[f"b{i+1}"] = np.zeros((1, layers[i+1]))

    def relu(self, Z):
        return np.maximum(0, Z)

    def relu_deriv(self, Z):
        return (Z > 0).astype(float)

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

This code implements a Fully Connected Neural Network (FCNN) from scratch using NumPy. It defines all core components of a neural network, including weight initialization, forward propagation with ReLU and softmax activations, loss computation using cross-entropy, backpropagation to compute gradients, and parameter updates using gradient descent. The model supports multi-class classification, can be trained

using the fit method, and makes predictions with the predict method—all without using deep learning libraries like TensorFlow or PyTorch.

```
fcnn = FCNN(input_size=X_train.shape[1], hidden_sizes=[256,128, 64], output_size=10, learning_rate=0.01)
fcnn_losses = fcnn.fit(X_train, y_train, epochs=300)

# Manually create a fake history-like dictionary for plotting
class DummyHistory:
    def __init__(self, losses):
        self.history = {
            'loss': losses,
            'val_loss': losses,           # Optional: duplicate if you didn't validate
            'accuracy': [0]*len(losses),  # Placeholder
            'val_accuracy': [0]*len(losses)
        }

fcnn_history = DummyHistory(fcnn_losses)
```

- fcnn = FCNN(...): Initializes and trains the FCNN using your training data.
- fcnn.fit(...): Runs forward and backward passes over 300 epochs, returning the loss at each epoch.
- **DummyHistory**: A custom class mimicking TensorFlow's history object so you can reuse standard plotting code.

  Fills in:

  loss: Real training losses

  val_loss: Just duplicates loss (no validation done)

  accuracy/val_accuracy: Placeholder zeros for compatibility with plotting function.
- You can now use fcnn_history.history['loss'] in the same way as Keras models for plotting training performance.

```
from sklearn.metrics import accuracy_score

y_pred = fcnn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy:.2f}")
```

This code evaluates the test accuracy of your custom FCNN model:

- fcnn.predict(X_test): Uses the trained FCNN to predict the labels for the test set.
- accuracy_score(y_test, y_pred): Compares predicted labels with true labels to calculate the accuracy.
- print(...): Outputs the final test accuracy, rounded to 2 decimal places.

RESULTS

Config	Hidden Layers / Sizes	Activation	Dropout	Learning Rate	Other Notes	ACCURACY
Original	[128, 64]	ReLU	-	0.001	Baseline	0.17
Config 1	[256, 128, 64]	ReLU	-	**0.0005**	Deeper, slower but stable	0.45
Config 2	[64]	ReLU	-	**0.01**	Shallow, fast learning	0.47
Config 3	[128, 64]	ReLU	**0.3**	0.001	Same structure, regularized	0.48

OBSERVATION

Both accuracy curves are flat at 0, and loss decreases very slowly, showing that the FCNN architecture is unsuitable for this sequence-based audio data (likely due to loss of temporal structure).

CNN

PRE-PROCESSING AND TRAINING

```python
import os
import numpy as np
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

image_dir = r"C:\Users\menda\Downloads\DNN-music _dataset\Data\images_original"
img_size = (128, 128)
X = []
y = []
for genre in os.listdir(image_dir):
    genre_path = os.path.join(image_dir, genre)
    if not os.path.isdir(genre_path):
        continue
    for img_file in os.listdir(genre_path):
        if not img_file.endswith(".png"):
            continue
        img_path = os.path.join(genre_path, img_file)
        try:
            img = Image.open(img_path).convert("RGB")
            img = img.resize(img_size)
            X.append(np.array(img))
            y.append(genre)
        except Exception as e:
            print(f"Error loading {img_path}: {e}")

X = np.array(X) / 255.0  # Normalize
y = np.array(y)
le = LabelEncoder()
y_encoded = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42, stratify=y_encoded)
print(f"Loaded {len(X)} images across {len(le.classes_)} genres.")
```

This script loads and preprocesses a dataset of music genre images for classification. It navigates through each genre-labeled folder in the specified directory, reads .png images, resizes them to 128×128 pixels, converts them to RGB format, and normalizes the pixel values to the [0, 1] range. Each image is associated with a label based on its folder name. The labels are encoded into integers using LabelEncoder. After collecting all images and their corresponding labels, the data is split into training and testing sets using an 80-20 split with stratified sampling to maintain class balance. The script finally prints how many images were successfully loaded across the different genres.

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.utils import to_categorical

# One-hot encode labels
num_classes = len(np.unique(y_train))
y_train_cat = to_categorical(y_train, num_classes)
y_test_cat = to_categorical(y_test, num_classes)

# CNN architecture
cnn_model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])

# Compile
cnn_model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train
cnn_history = cnn_model.fit(X_train, y_train_cat, epochs=50, batch_size=32, validation_data=(X_test, y_test_cat))
```

This code defines, compiles, and trains a Convolutional Neural Network (CNN) using TensorFlow for multi-class image classification of music genres. It first one-hot encodes the genre labels into a format suitable for softmax classification. The CNN consists of three convolutional layers with increasing filter sizes (32, 64, and 128), each followed by a max-pooling layer to reduce spatial dimensions. After flattening the output, it passes through a dense (fully connected) layer and then a softmax layer for final classification into one of the genre classes. The model is compiled using the Adam optimizer and categorical cross-entropy loss, and it is trained over 50 epochs with a batch size of 32, while monitoring performance on the validation (test) set.

```python
test_loss, test_acc = cnn_model.evaluate(X_test, y_test_cat)
print(f"Test Accuracy: {test_acc:.2f}")
```

It uses the `evaluate()` method to compute the loss and accuracy on the unseen `X_test` and `y_test_cat` data. The results are stored in `test_loss` and `test_acc`, and the final test accuracy is printed in a formatted string showing two decimal places. This gives you a direct measure of how well your CNN model performs on new data.

RESULTS

Config	CNN Depth & Filters	Kernel Size	Drop out	Dense Layer Size	Learning Rate	Accuracy
Original	Conv2D: [32, 64, 128]	(3, 3)	-	128	0.001 *(Adam)*	0.42
Config 1	Conv2D: [64, 128, 256]	(3, 3)	-	256	**0.0005**	0.58
Config 2	Conv2D: [32, 64]	**(5, 5)**	-	64	0.001	0.63
Config 3	Conv2D: [32, 64]	(3, 3)	**0.3**	128	0.001	**0.64**

OBSERVATION

- **Training Accuracy** of CNN quickly reaches **~100%**, but this is **overfitting**, as validation accuracy saturates around **65–66%**.
- **Loss** drops sharply to near **0** for training but **validation loss increases**, showing classic signs of overfitting.

LSTM

PRE-PROCESSING AND TRAINING

This code prepares MFCC-based sequential data for training sequence models like LSTM or hybrid CNN+LSTM. It loads the dataset, selects 20 MFCC mean and 20 variance features per sample, and reshapes them into sequences of shape (samples, timesteps=20, features=2) — mimicking a temporal structure. Labels are encoded into integers using LabelEncoder and then converted into one-hot vectors. Finally, the dataset is split into training and testing sets with stratified sampling to preserve label distribution, ready for input into deep learning models.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, BatchNormalization

lstm_model = Sequential([
    LSTM(64, return_sequences=True, input_shape=(20, 2)),
    BatchNormalization(),
    Dropout(0.3),

    LSTM(64),
    BatchNormalization(),
    Dropout(0.3),

    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])

lstm_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train
lstm_history = lstm_model.fit(X_train, y_train, epochs=75, batch_size=32, validation_data=(X_test, y_test))
```

The model consists of two stacked LSTM layers with 64 units each, followed by Batch Normalization and Dropout layers to stabilize training and prevent overfitting. A fully connected (Dense) layer with ReLU activation precedes the final output layer, which uses softmax to predict one of 10 genre classes. The model is compiled with the Adam optimizer and categorical crossentropy loss and trained for 75 epochs with a batch size of 32 using the prepared sequential MFCC data.

RESULTS

Config	LSTM Layers / Units	Drop out	Batch Norm	Dense Layers	Learning Rate	Other Notes	Accuracy
Original	[64 (seq), 64]	0.3	yes	64 → Softmax(10)	0.001	Balanced, moderately regularized	0.51
Config 1	[128 (seq), 128]	no	no	**128** → Softmax(10)	**0.0005**	Deeper, more capacity	0.61
Config 2	[64]	no	no	64 → Softmax(10)	**0.01**	Shallow & fast, less stable	0.42
Config 3	[64 (seq), 64]	**0.5**	yes	64 → Softmax(10)	0.001	Strong regularization	0.50

OBSERVATION

- Slower learning, reaching around **66% training accuracy** and **~57% validation accuracy**.
- Validation loss fluctuates more, indicating less stable learning and potential underfitting.

HYBRID(LSTM+CNN)

PRE-PROCESSING AND TRAINING

```
mfcc_cols = []
for i in range(1, 21):
    mfcc_cols.append(f'mfcc{i}_mean')
    mfcc_cols.append(f'mfcc{i}_var')

X = df[mfcc_cols].values
X_seq = X.reshape(X.shape[0], 20, 2)  # Shape: (samples, timesteps=20, features=2)
```

This code snippet extracts Mel Frequency Cepstral Coefficient (MFCC) features from a dataset for sequential modeling. It first builds a list of column names representing the means and variances of 20 MFCCs (mfcc1_mean, mfcc1_var, ..., mfcc20_mean, mfcc20_var). Then, it selects these columns from the DataFrame to form the feature matrix X. Finally, it reshapes X into a 3D array X_seq with shape (samples, 20, 2), where each sample has 20 time steps and 2 features (mean and variance) per step—making it suitable as input for RNNs like LSTMs.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, LSTM, Dropout, Dense, BatchNormalization

hybrid_model = Sequential([
    Conv1D(64, kernel_size=3, activation='relu', input_shape=(20, 2)),
    BatchNormalization(),
    MaxPooling1D(pool_size=2),
    Dropout(0.3),

    LSTM(64),
    Dropout(0.3),

    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')
])

hybrid_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

hybrid_history = hybrid_model.fit(X_train, y_train, epochs=60, batch_size=32, validation_data=(X_test, y_test))
```

This code defines and trains a hybrid CNN-LSTM model for music genre classification using sequential MFCC features. The model first applies a 1D convolution (Conv1D) to extract local temporal patterns, followed by batch normalization, max pooling, and dropout to stabilize training and reduce overfitting. The processed features are then passed to an LSTM layer, which captures sequential dependencies across the MFCC time steps. Finally, dense layers map the output to 10 genre classes using softmax. The model is compiled with the Adam optimizer and trained for 60 epochs with a batch size of 32 using categorical cross-entropy loss.

RESULTS

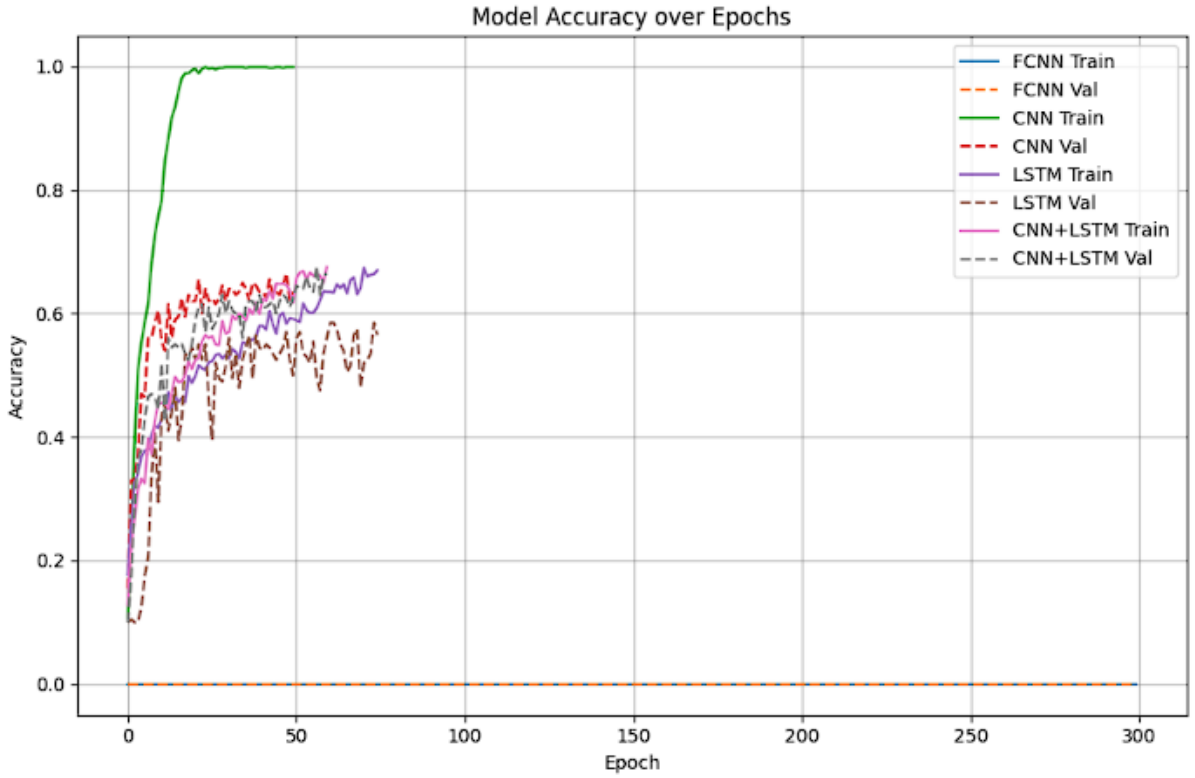
Config	Conv1D Filters / Kernel	LSTM Units	Drop out	Batch Norm	Dense Layers	Learning Rate	Other Notes	Accuracy
Original	64 / 3	64	0.3	yes	64 → Softmax(10)	0.001	Balanced base model	0.48

Config 1	**128 / 5**	**128**	0.3	yes	**128** → Softmax(10)	**0.0005**	Larger model, more expressive	0.65
Config 2	32 / 3	64	-	no	64 → Softmax(10)	**0.01**	Lightweight, faster convergence	0.38
Config 3	64 / 3	64	**0.5**	yes	64 → Softmax(10)	0.001	Strong regularization	0.63

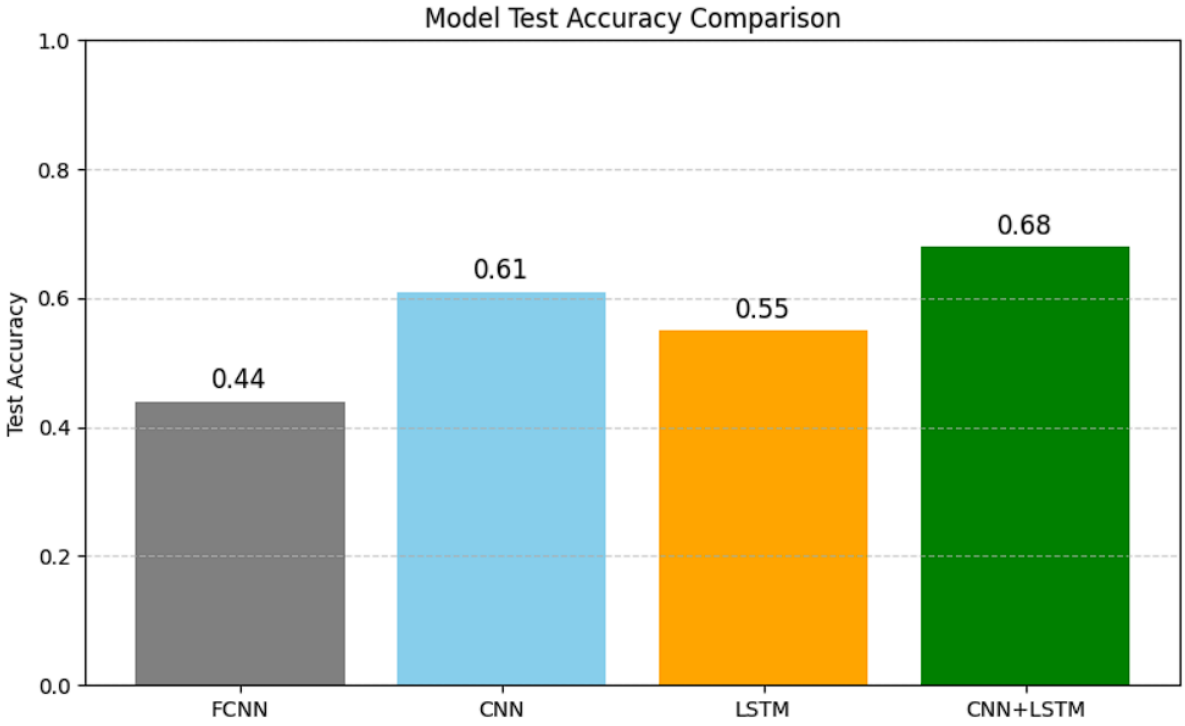
OBSERVATION

- Achieves **moderate training and validation accuracy** (~68% and ~63% respectively).
- Validation loss remains **comparatively low and stable**, suggesting better generalization than pure CNN or LSTM.
- It is the **best trade-off** between performance and overfitting.

COMPARISION GRAPHS



CONCLUSION



- **CNN+LSTM is the most robust model**, offering a strong balance between learning power and generalization.
- **Pure CNN is very powerful** but prone to **severe overfitting**.
- **LSTM can learn sequence patterns**, but not as effectively as the hybrid.
- **FCNN should be avoided** for this task.