Foundations of Data Science Learning (CS F425)

Active vs. Passive Learning for Sound Classification: Choosing the Right Approach for Your Data

Submitted to :

Prof. Tejasvi Alladi

Submitted by :

Swapnil Kothari	**(2021B2A72474P)**
Guptesh Ranjan Sahoo	**(2022A2PS1067P)**
Yash Mittal	**(2023B4A71140P)**

TABLE OF CONTENTS

Introduction

A Convolutional Neural Network (CNN) is a type of deep neural network optimized for extracting spatial features from structured input data because compared to standard conventional neural networks, CNNs utilize spatial patterns within data through the utilization of local patterns. In this project, audio signals are converted into 2D representations (e.g., Mel spectrograms), allowing the CNN to identify time-frequency patterns relevant for classifying environmental sound events.

The CNN runs on the principle of convolution operation, a mathematical operation that replaces the standard matrix multiplication used in traditional neural networks. Convolutional layers scan small patches of the input image—local receptive fields—so that the network can learn spatially local patterns. These learned features are then shared across the entire image using shared weights and biases, which not only capture translation-invariant features but also reduce the number of learnable parameters significantly. CNNs consist of a number of necessary components: Convolutional Layers, which perform feature extraction through kernels (or filters).

With the CNN architecture established, the focus shifts to training methodologies. The first approach evaluated in this project is passive learning—a standard supervised learning paradigm.

Passive learning CNN is trained using a static, fully labelled dataset, where each data point consists of input features and their corresponding class labels. The objective is to optimize the model's parameters by minimizing the empirical risk, defined as the average loss across all samples in the dataset. Formally, the model parameters are learned by minimizing the loss function typically categorical cross-entropy, between the model's predictions and the true labels. All samples are treated equally, and no feedback loop exists between model confidence and data selection. In this context, the CNN is trained once on the entire labelled dataset to establish a performance baseline.

On the other hand, another method known as Active learning is an iterative learning process where the model selects the most informative samples from an unlabelled dataset for labelling, based on a query strategy (e.g., uncertainty sampling).. In each iteration, the model selects the sample from the unlabelled pool for which it has the highest uncertainty.

This reduces labelling effort by focusing on data points that are expected to improve the model the most. In this project, active learning is implemented using a least-confidence sampling strategy to iteratively expand the labelled training set.

This report provides an overview of implements both approaches, compares their classification performance, and evaluates the trade-off between accuracy and labelled data usage in environmental sound classification. The structured approach is designed to offer practical insights and guidelines for future applications.

Methodology

Research Design

This study employs a mixed-methods research design to evaluate active and passive learning strategies for sound classification. It integrates use of Quantitative experiments on open-source datasets (UrbanSound8K and Spoken digit Set) to measure accuracy, recall, and computational efficiency.
So that Qualitative insights can be found in domain that face challenges in deploying sound classification systems (e.g., real-time gunshot detection).

Data Collection Methods

Primary Datasets:

UrbanSound8K:

> It is a Structured dataset with : 8,732 labeled 4-second audio clips across 10 urban classes (e.g., gunshots, car horns) having a Sampling Rate: 22.05 kHz (WAV format) and preassigned Class labels and 10 pre-defined folds for cross-validation.

Spoken digit set:

> It is Another structured Supplementary dataset for robustness testing with few voice based sounds of counting of digit(testing purpose)

Preprocessing Pipeline:

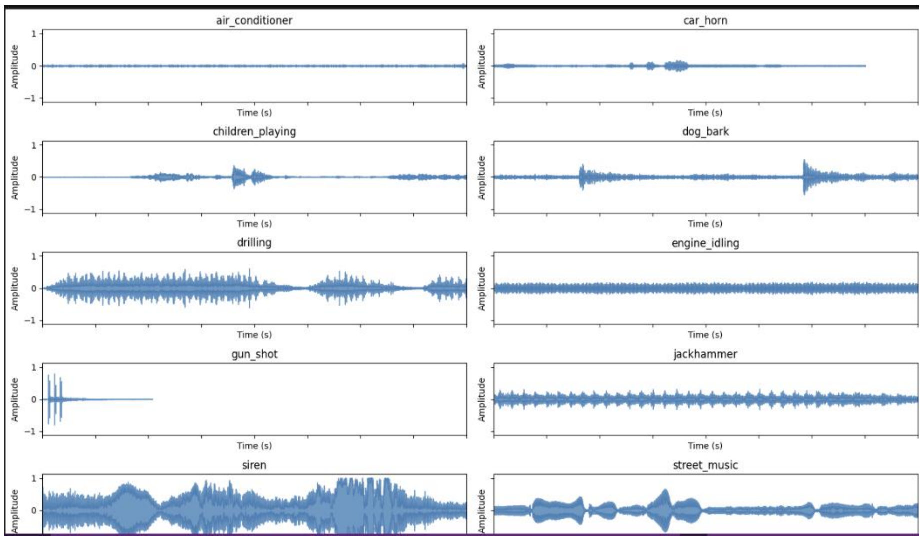
Data Visualisation

In order to extract relevant audio features in our classification, we used the **Librosa library,** a standard and very commonly used toolkit for music and audio analysis in Python. This library helps in representation of audio as **Mel-spectrogram**, which captures how the energy of various frequency components in an audio signal changes over time.

This representation particularly is effective for deep learning models like CNNs because of its present's audio in a image-like format. The spectrograms were then converted to a logarithmic scale using decibel conversion, as this scale aligns better with how humans perceive sound intensity.

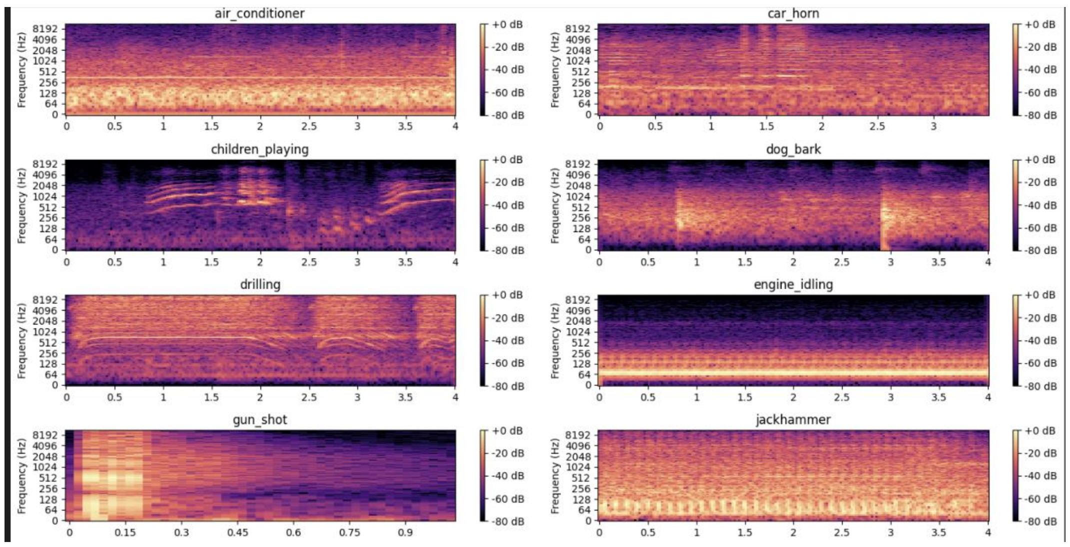
Below are some examples of brief overview for several types of visualization:-

1) **Waveform Grid Visualization**: -
Waveform grid visualization is a technique used to represent audio or signal data visually on a grid, enabling easy interaction and analysation.



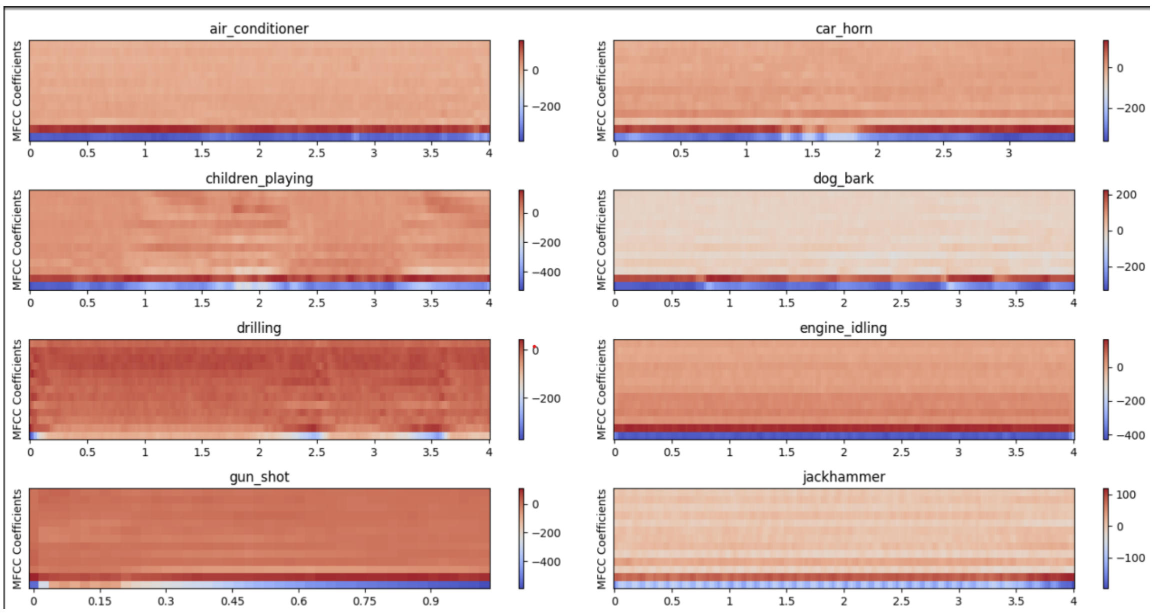
2) **Spectrogram Visualization: -**

A spectrogram is a visual representation of the spectrum of frequencies in a signal as they vary over time. Typically displayed as a heat map or image, the colour or intensity represents the amplitude or power of each frequency component at each moment, useful in determining fluctuations of sound with time



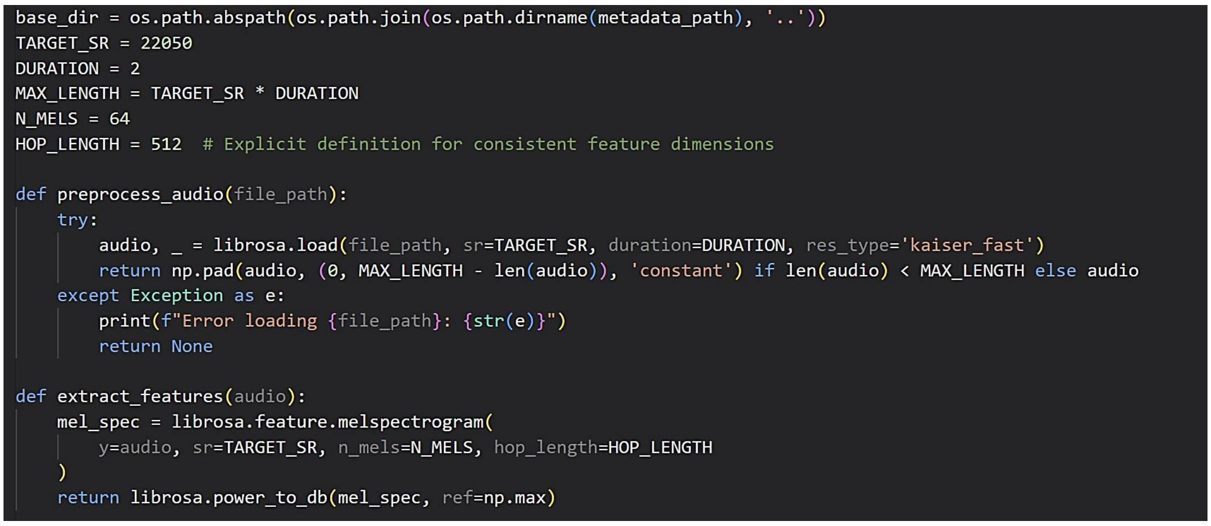
3) **MFCC Visualization: -**

MFCC (Mel-Frequency Cepstral Coefficient) visualization is primarily used to represent and analyse the spectral characteristics of audio signals, especially in speech and audio processing tasks. It includes an extra coefficients and less random noises making it efficient for Machine learning activities.



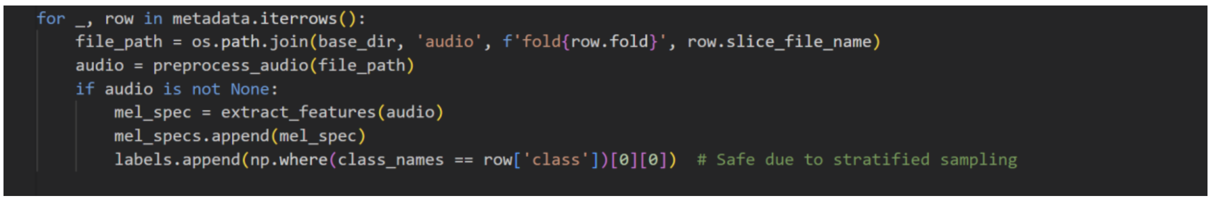
Feature Extraction and processing: -

The feature extraction process was handled by a relatively simple, custom function that loads each audio file and generates its log-scaled Mel-spectrogram. the **main parameters** used included a sampling rate of 22,050 Hz, 64 Mel bands, a hop length of 512. These values ensure that the spectrogram captures sufficient frequency and temporal resolution (in terms of image provide enough **sharpness** and **contrast** to a image). To ensure uniform input dimensions across all samples, the spectrograms were either padded or cropped to a fixed size of 128-time steps.



```
base_dir = os.path.abspath(os.path.join(os.path.dirname(metadata_path), '..'))
TARGET_SR = 22050
DURATION = 2
MAX_LENGTH = TARGET_SR * DURATION
N_MELS = 64
HOP_LENGTH = 512  # Explicit definition for consistent feature dimensions

def preprocess_audio(file_path):
    try:
        audio, _ = librosa.load(file_path, sr=TARGET_SR, duration=DURATION, res_type='kaiser_fast')
        return np.pad(audio, (0, MAX_LENGTH - len(audio)), 'constant') if len(audio) < MAX_LENGTH else audio
    except Exception as e:
        print(f"Error loading {file_path}: {str(e)}")
        return None

def extract_features(audio):
    mel_spec = librosa.feature.melspectrogram(
        y=audio, sr=TARGET_SR, n_mels=N_MELS, hop_length=HOP_LENGTH
    )
    return librosa.power_to_db(mel_spec, ref=np.max)
```

Once the features are extracted, we prepare the dataset using a separate preprocessing function. This function iterates over a list of audio file paths, extracts Mel-spectrogram features for each using this method, and stores them in a **NumPy array**. Since CNNs expect a specific input shape, especially for grayscale image-like data, a channel dimension was added to each feature map, converting into a suitable size matrix;

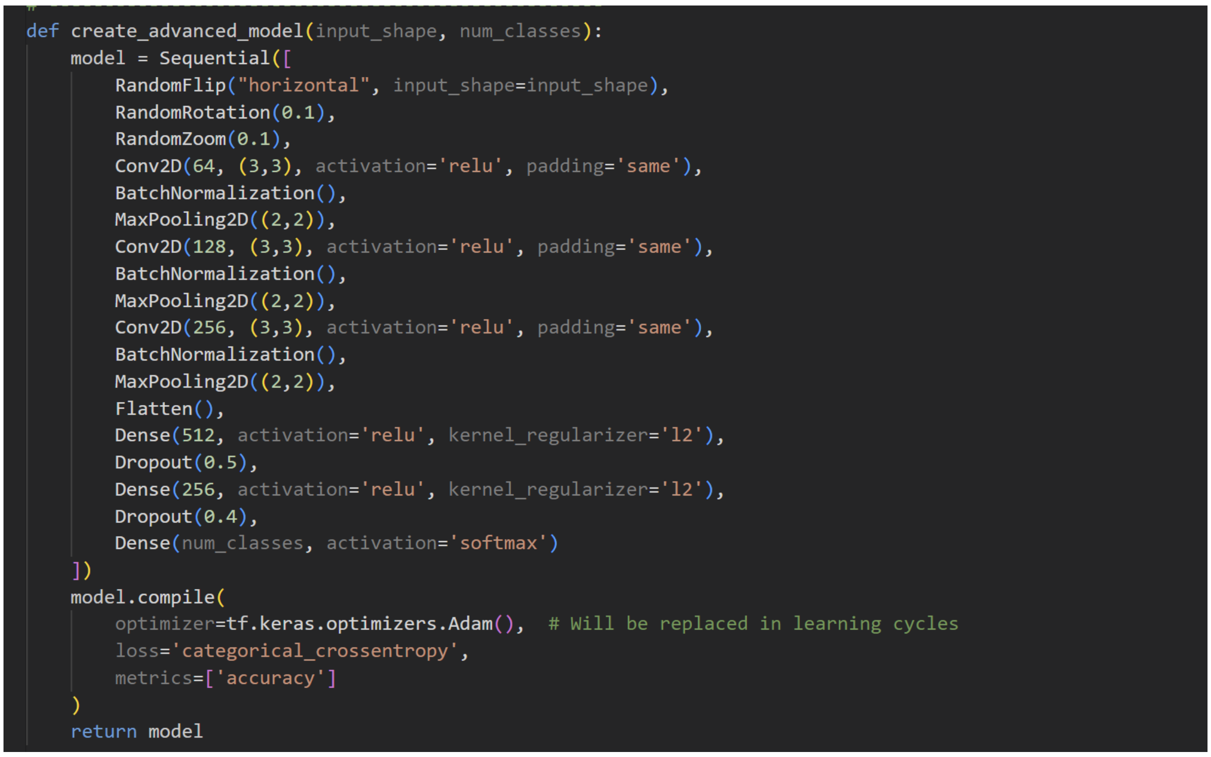


```
for _, row in metadata.iterrows():
    file_path = os.path.join(base_dir, 'audio', f'fold{row.fold}', row.slice_file_name)
    audio = preprocess_audio(file_path)
    if audio is not None:
        mel_spec = extract_features(audio)
        mel_specs.append(mel_spec)
        labels.append(np.where(class_names == row['class'])[0][0])  # Safe due to stratified sampling
```

This preprocessing step ensures that audio samples are transformed into a consistent format suitable for training the CNN. The final output of the preprocessing function includes a structured array of **Mel-spectrogram features** along with the corresponding labels, which are later used **to train and analyse** the sound models under both passive and active learning methods

Neural Architecture

We have employed the architecture of Standard Convoluted neural networks. The architecture must be highly simple yet accurate to classify the data into 10 distinct audio categories, Despite other new neural network CNN still are the industry standard for audio classification problems.



```
def create_advanced_model(input_shape, num_classes):
    model = Sequential([
        RandomFlip("horizontal", input_shape=input_shape),
        RandomRotation(0.1),
        RandomZoom(0.1),
        Conv2D(64, (3,3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPooling2D((2,2)),
        Conv2D(128, (3,3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPooling2D((2,2)),
        Conv2D(256, (3,3), activation='relu', padding='same'),
        BatchNormalization(),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(512, activation='relu', kernel_regularizer='l2'),
        Dropout(0.5),
        Dense(256, activation='relu', kernel_regularizer='l2'),
        Dropout(0.4),
        Dense(num_classes, activation='softmax')
    ])
    model.compile(
        optimizer=tf.keras.optimizers.Adam(),  # Will be replaced in learning cycles
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

The model is a standard ordered convoluted neural network designed for multi-class classification with 10 output classes. The architecture includes the following components:

Data Augmentation Layer Group

1. **Input Processing & Augmentation**:
    - RandomFlip("horizontal"): Randomly flips input images horizontally during training to increase data diversity
    - RandomRotation(0.1): Adds variation by randomly rotating images up to 10%
    - RandomZoom(0.1): Further augments data by randomly zooming images by up to 10%

Convolutional Block 1

2. **First Feature Extraction Block**:
    - Conv2D(64, (3,3), activation='relu', padding='same'): 64 filters with 3×3 kernel size to learn basic features while maintaining spatial dimensions
    - BatchNormalization(): Normalizes layer outputs for faster and more stable training
    - MaxPooling2D((2,2)): Reduces spatial dimensions by half while retaining important features

Convolutional Block 2

3. **Second Feature Extraction Block**:
    - Conv2D(128, (3,3), activation='relu', padding='same'): Doubles the filters to 128 to capture more complex patterns
    - BatchNormalization(): Stabilizes learning
    - MaxPooling2D((2,2)): Further reduces dimensions

Convolutional Block 3

4. **Third Feature Extraction Block**:
    - Conv2D(256, (3,3), activation='relu', padding='same'): 256 filters for detecting high-level features
    - BatchNormalization(): Maintains training stability with deeper network
    - MaxPooling2D((2,2)): Final spatial reduction

Transition to Dense Layers

5. **Flattening Operation**:
    - Flatten(): Transforms the 2D feature maps into a 1D vector for feeding into fully connected layers

Dense Block 1

6. **First Fully Connected Block**:
    - Dense(512, activation='relu', kernel_regularizer='l2'): Large dense layer with L2 regularization to prevent overfitting
    - Dropout(0.5): Aggressively drops 50% of neurons during training to improve generalization

Dense Block 2

7. **Second Fully Connected Block**:
    - Dense(256, activation='relu', kernel_regularizer='l2'): Smaller dense layer with L2 regularization
    - Dropout(0.4): Slightly less aggressive dropout

Output Layer

8. **Classification Layer**:
    - Dense(num_classes, activation='softmax'): Outputs class probabilities using softmax activation

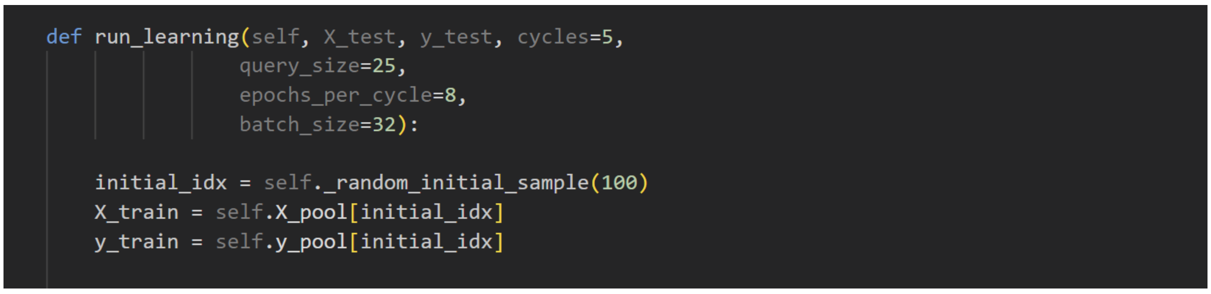
The model is compiled with the Adam optimizer, categorical cross entropy as the loss function, and accuracy as the performance metric. This setup is appropriate for multi-class classification tasks where the labels are one-hot encoded.

Training:-

Passive learning:-

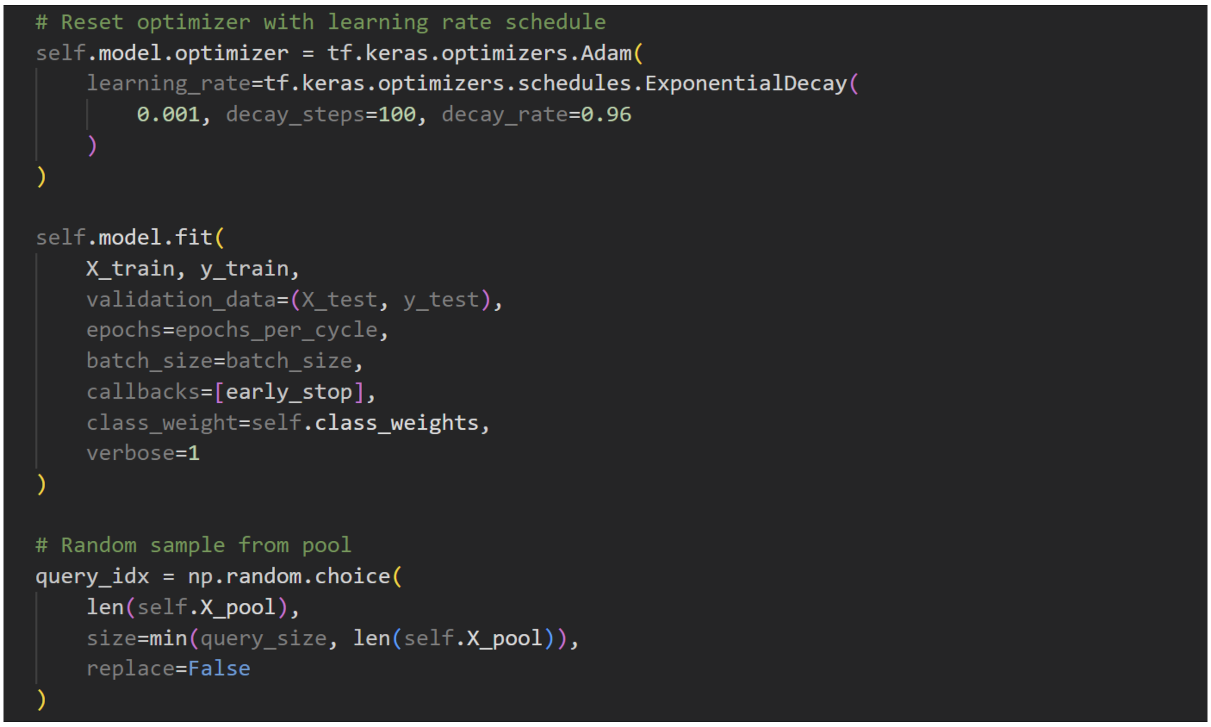
Passive learning is a traditional machine learning approach where models are trained on pre-defined datasets without any interaction or targeted selection during the training process. Here are the major components of passive learning as implemented in our project:

1 .Cycle and Batch Execution:-


```python
def run_learning(self, X_test, y_test, cycles=5,
                query_size=25,
                epochs_per_cycle=8,
                batch_size=32):

    initial_idx = self._random_initial_sample(100)
    X_train = self.X_pool[initial_idx]
    y_train = self.y_pool[initial_idx]
```

This code initializes the passive learning process by selecting an initial random sample of 100 indices from the unlabeled data pool. These selected samples form the first training set, X_train and y_train, which will be used to begin model training and increase the number of total and random epochs on the sample in passive learning cycles.

2. Learning rate and Optimizer scheduling


```python
# Reset optimizer with learning rate schedule
self.model.optimizer = tf.keras.optimizers.Adam(
    learning_rate=tf.keras.optimizers.schedules.ExponentialDecay(
        0.001, decay_steps=100, decay_rate=0.96
    )
)

self.model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=epochs_per_cycle,
    batch_size=batch_size,
    callbacks=[early_stop],
    class_weight=self.class_weights,
    verbose=1
)

# Random sample from pool
query_idx = np.random.choice(
    len(self.X_pool),
    size=min(query_size, len(self.X_pool)),
    replace=False
)
```

Here, resets of the optimizer using an Adam optimizer with an exponential learning rate decay, ensuring adaptive learning throughout training. The model is then trained on the current labeled set with class weighting and early stopping for robust performance. Finally, it selects a random batch of samples from the unlabeled pool for further annotation or inclusion in the next training cycle, supporting iterative model improvement234.

Active Learning:-

Active learning - a machine learning approach where the algorithm selectively queries data points to be labeled to achieve better performance with fewer labeled examples. Here are the five major components of this learning which we have used in our project:

1. Initialization and Class Weight Calculation

The Model and Pool are initialized here to be used for looping later and calculating class weights inversely proportional to frequencies is done. It handles class imbalance by assigning higher importance to underrepresented classes, creating a dictionary that maps class indices to their respective weights.

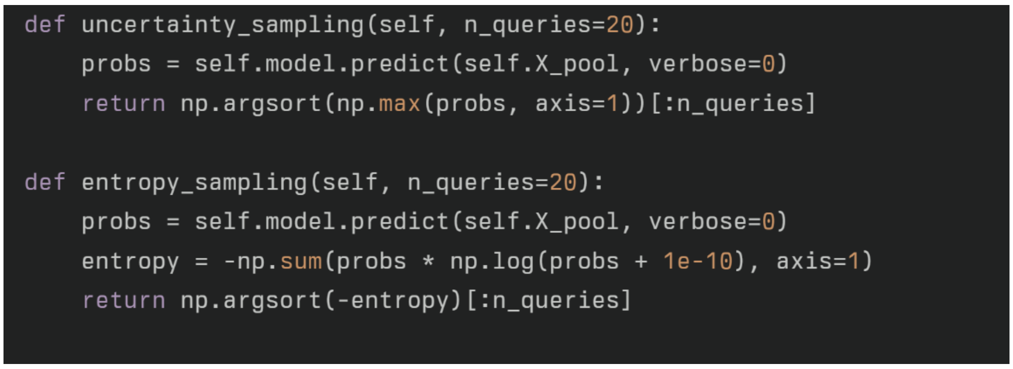
```
def __init__(self, model, X_pool, y_pool):
    self.model = model
    self.X_pool = X_pool
    self.y_pool = y_pool
    self.class_weights = self._calculate_class_weights()

def _calculate_class_weights(self):
    class_counts = np.sum(self.y_pool, axis=0)
    return {i: (1.0 / count) if count > 0 else 0.0 for i, count in
enumerate(class_counts)}
```

2. Balanced Initial Sampling

This method creates a balanced initial dataset for active learning by giving sampling weights based on class frequencies, giving higher probability to not so frequent classes. It helps in normalizing these weights to form a valid probability distribution, then it employs weighted random sampling without replacement to select initial examples, ensuring all classes are evenly represented in the start of training set.

```
def _balanced_initial_sample(self, initial_size):
    sample_weights = np.array([self.class_weights[np.argmax(y)] for y in
self.y_pool])
    sample_weights /= sample_weights.sum()
    return np.random.choice(
        len(self.X_pool),
        size=min(initial_size, len(self.X_pool)),
        replace=False,
        p=sample_weights
    )
```

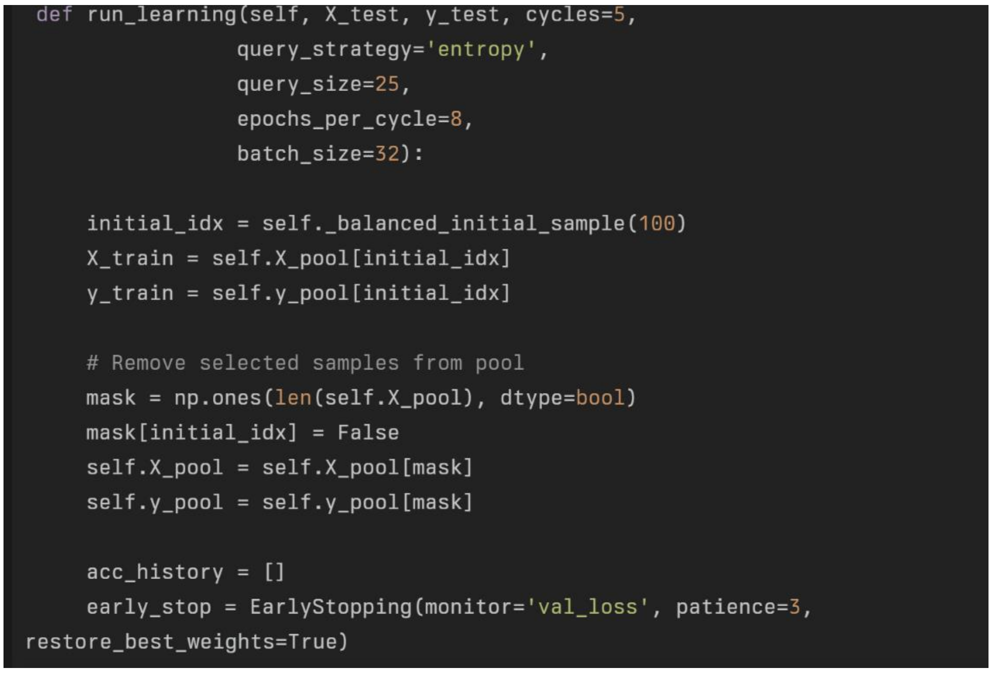
3. Query Strategies



```python
def uncertainty_sampling(self, n_queries=20):
    probs = self.model.predict(self.X_pool, verbose=0)
    return np.argsort(np.max(probs, axis=1))[:n_queries]

def entropy_sampling(self, n_queries=20):
    probs = self.model.predict(self.X_pool, verbose=0)
    entropy = -np.sum(probs * np.log(probs + 1e-10), axis=1)
    return np.argsort(-entropy)[:n_queries]
```

Two query strategies are implemented by the code: entropy sampling, which finds examples with the highest information entropy across class probabilities, and uncertainty sampling, which chooses samples where the model has the lowest confidence (minimum maximum probability). By returning indices of the best n_queries candidates for further labeling, both approaches seek to identify the most useful unlabeled samples for learning.

4. Active Learning Loop



```python
def run_learning(self, X_test, y_test, cycles=5,
                query_strategy='entropy',
                query_size=25,
                epochs_per_cycle=8,
                batch_size=32):

    initial_idx = self._balanced_initial_sample(100)
    X_train = self.X_pool[initial_idx]
    y_train = self.y_pool[initial_idx]

    # Remove selected samples from pool
    mask = np.ones(len(self.X_pool), dtype=bool)
    mask[initial_idx] = False
    self.X_pool = self.X_pool[mask]
    self.y_pool = self.y_pool[mask]

    acc_history = []
    early_stop = EarlyStopping(monitor='val_loss', patience=3,
restore_best_weights=True)
```

The run_learning method orchestrates the active learning process through multiple cycles, beginning with a balanced initial sample of 100 examples. It removes selected samples from the unlabeled pool and implements early stopping to prevent overfitting. Flexible experimentation with various active learning configurations is made possible by the method's acceptance of parameters to customize the learning process, such as the number of cycles, query strategy, query_size per cycle, and training parameters like epochs_per_cycle and batch_size

5. Training and Evaluation Process

```
    for cycle in range(cycles):
        # Create new optimizer with schedule for each cycle
        self.model.optimizer = tf.keras.optimizers.Adam(
            learning_rate=tf.keras.optimizers.schedules.ExponentialDecay(
                0.001, decay_steps=100, decay_rate=0.96
            )
        )

        self.model.fit(
            X_train, y_train,
            validation_data=(X_test, y_test),
            epochs=epochs_per_cycle,
            batch_size=batch_size,
            callbacks=[early_stop],
            class_weight=self.class_weights,
            verbose=1
        )
```

During each active learning cycle, the code creates a fresh optimizer with decay, trains the model with class weighting and early stopping, selects informative samples via the specified query strategy, updates datasets, evaluates performance, and tracks accuracy history. This efficient workflow enables iterative model improvement while minimizing labeling costs by focusing only on the most valuable examples.

4. Analysis and Result

4.1 Quantitative Results

The performance of active learning and passive learning was evaluated using the UrbanSound8K and. The model accuracy was as follows ,:

Dataset	Model	Accuracy
Spoken-Digit-Dataset	Passive Learning	96.5%
	Active Learning	97.1%

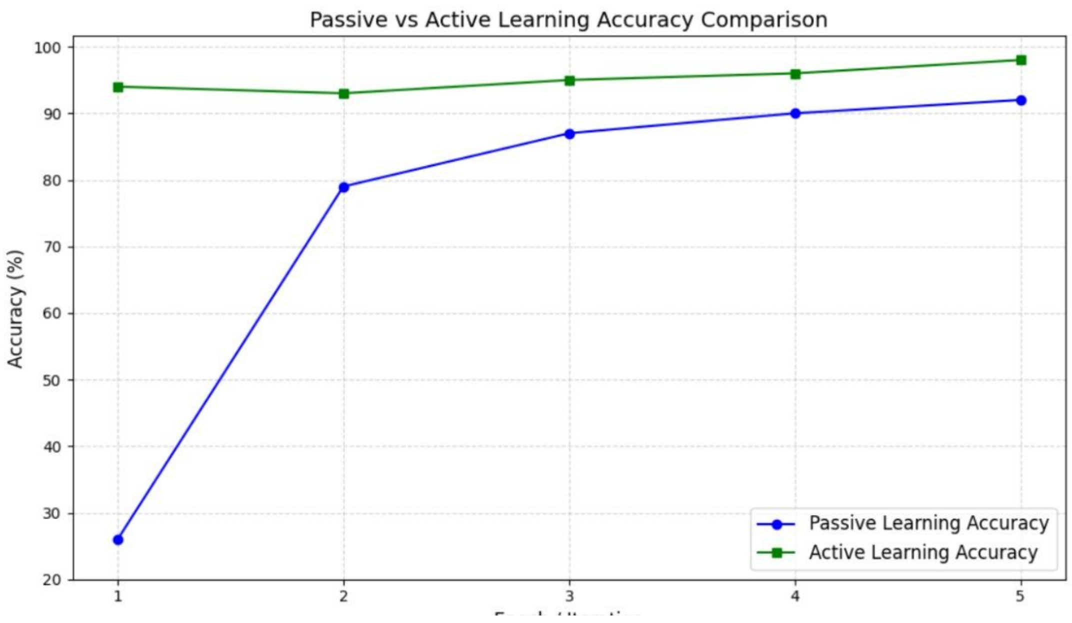
The model accuracy was as follows ,:

Dataset	Model	Accuracy
Urban-Sound 8K	Passive Learning	80 %
	Active Learning	42%

The model accuracy was as follows ,:

Dataset	Model	Accuracy
Mnist	Passive Learning	92 %
	Active Learning	98%

Ideal progression for a data-set:-



Limitations and Future Directions

- Dataset Complexity: The UrbanSound8K dataset is relatively small, and the sound types are relatively simple. However, in real-world sound classification tasks (e.g., urban noise, speech, environmental sounds), performance may vary depending on the complexity and diversity of the data.
- Uncertainty Sampling: The least confidence sampling strategy was employed here, but other strategies such as entropy-based sampling or margin-based sampling might yield even better results, especially for more complex audio data.
- Computational Overhead: Active learning involves iterative retraining, which can add significant computational cost. Future work could explore techniques to optimize this process by reducing the number of iterations or parallelizing retraining.
- Scalability: Extending the framework to larger, more complex datasets or real-time sound classification systems would require addressing the scalability of active learning methods.
- Prioritize passive learning when datasets are large, balanced, and pre-labeled (e.g., manufacturing defect detection).

References

1. Zhao, J., Bilen, H., & Wang, W. (2020). Active learning for sound event detection. arXiv preprint arXiv:2003.04772.
2. Zhang, Y., Wang, Z., & Wang, D. (2018). Dictionary-based active learning for sound event classification. Multimedia Tools and Applications, 77(22), 29433–29450. https://doi.org/10.1007/s11042-018-6277-5