HUMAN ACTION RECOGNITION IN THE DARK: AN EXPLORATION WITH IMAGE ENHANCEMENT AND LATE FUSION

Nurfitri Anbarsanti G2104045K
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore
{nurfitri006}@e.ntu.edu.sg

ABSTRACT

In this work, we explore the effects of image enhancement as a crucial preprocessing step to mitigate the loss of visual fidelity due to low light. We implement some image enhancement configuration on traditional machine learning, particularly Kernel support vector machine (SVM); and compare its results with late fusion techniques of deep learning. We analyze whether the leverage of image enhancement improves the performance of human action recognition systems in dark environments or not. Late fusion techniques are the fusion of obtained prediction from LSTM and RNN using simple averaging in order to enable end-to-end training and classification processes. The experimental results show that the image enhancement configuration does improve the performance of traditional machine learning, but not significantly influence the outcome of deep learning-based classification.

1 INTRODUCTION

Human action recognition has long been researched due to its applications in many different fields including video surveillance, person identification (e.g., gait recognition), smart home automation (e.g., gesture recognition), autonomous systems, and human-computer interaction Yang et al. (2022). However, current research still mainly focuses on videos captured under normal illumination and therefore recognizing human actions in the dark can result in high cost and the difficulties in its large-scale deployments. So, in this study, we will focus on human action recognition (HAR) in dark videos.

The robust detection and classification of human actions under low-light scenarios remains a significant challenge. For example, anomaly actions are more common at night time and in dark environments, yet current HAR models are obscured by darkness are unable to recognize any actions effectively. Darkness has hampered the effectiveness of onboard cameras so severely that most vision-based autonomous driving systems are strictly prohibited at night Brown (2009), while those who do allow night operation could cause severe accidents Boudette (2021).

Traditional machine learning algorithms play a vital role in the early-stage research on device-free HAR. By extracting hand-crafted features, classic classifiers, such as SVM, random forest, and Naive Bayes, have made significant results Yang et al. (2022). Though these methods yield excellent results in controlled environments or small datasets, these models cannot deal with real-world scenarios that are more complex and dynamic such as the heterogeneity of human activity and insufficient illumination of nighttime scenes.

Deep learning extracts more robust features using massive data. Deep learning methods learn the feature space that mostly contributes to the task automatically by novel network architectures and back-propagation, which overcomes the shortcomings of hand-crafted features Yang et al. (2022).

In this work, we present some exploration of HAR with different experimental settings. We explore different methods of samplings. Then, we explore the effects of image enhancement as a critical pre-

processing step to mitigate the loss of visual fidelity due to low light. Other than that, we implement traditional machine learning, particularly Kernel SVM and compare its results with an approach that combines late fusion techniques with advanced image enhancement algorithms to improve the performance of HAR systems in the dark.

2 The DataSet and Experimental Setting

In this study, we are given a set of training data, which includes 25 random videos for each of the six action classes, and a set of testing/validation data. The dataset that are given to us are identical and are small subsets to those in Action Recognition in the Dark-Plus (ARID-plus) (Xu et al. (2023)).

2.1 Action Classes

The dataset used in this study includes a total of six common human action classes: Jump, Run, Sit, Stand, Turn, and Walk. The classes of *Jump, Run, Turn and Walk* can be categorized as Singular Person Actions, while Sit and Stand could be categorized as *Person Actions with Objects*. The dataset utilized in this experiment is dedicated to be used in human action recognition in the dark.

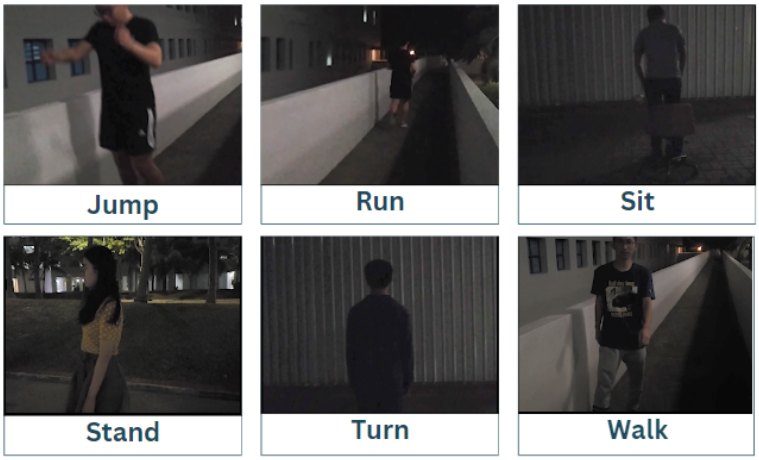

Figure 1: Sample frames for each of the six action classes of the given dataset. All samples are brightened $30\% - 100\%$ for display purposes.

2.2 Basic Statistic

In the training dataset, each classes contain 25 videos so that 150 training videos in total; while the validation dataset (for testing purpose) contains 96 videos. The dataset are divided into 60.97% for the training dataset and the rest of 39.03% for the validation data set. The distribution and proportion of the dataset can be seen in Table 1 and Figure 2.

Table 1: The distribution of clips among all classes in the training and validation dataset

Classes	**Number of Training Clips**	**Number of Validation Clips**
Jump	25	17
Run	25	15
Sit	25	15
Stand	25	16
Turn	25	17
Walk	25	16

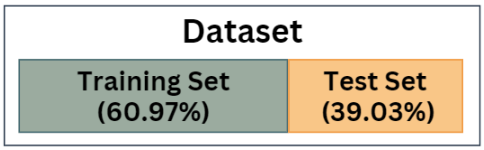

Figure 2: The partition of the dataset

Each size of the video frame is $240x320$ RGB pixels. The video clips are fixed to a frame rate of 30 FPS. The videos are saved in $.mp4$ format. The minimum clip length is 1.2 seconds with 36 frames (Xu et al. (2022)).

2.3 Experimental Setting

To obtain the human action recognition system, we utilize Python programming language along with Numpy, OpenCV, Tensorflow, Scikit-Learn, Keras library.

For our experiments, we sampled the videos into 10 sampled frames, then we normalize the sampled frames using predefined mean and standard deviation. We compare different values of mean and standart deviation for normalization process. The output sampled frames are already normalized.

Then we resized each of the normalized sampled frames into $224x224$ pixels. To accelerate training, we leverage the pre-trained models that has been pre-trained on ImageNet dataset, i.e., ResNet50 for being feature extractor. The output feature type and size from ResNet50 is array of 2048 values from each frame.

Those aforementioned stage of sampling, normalization and feature extraction process are applied both into all training videos and all validation videos. Then, we flatten the frames of each clips using Numpy Array before bringing them into traditional classifier phase.

In the traditional classifier phase, we select Kernel SVM and train the classifier with the obtained feature and labels provided. The experimental setting that leverages Kernel SVM can be seen in Figure 3.

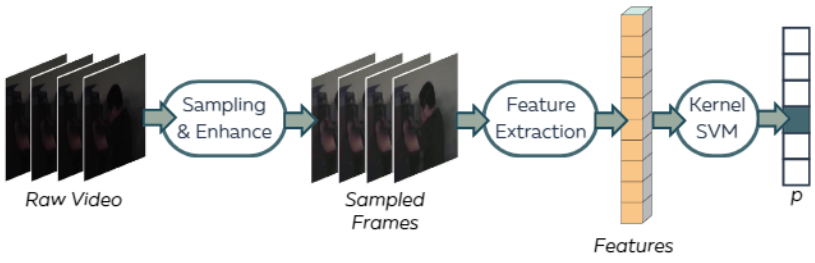

Figure 3: Experimental diagram leveraging Kernel SVM

Besides, we also use late fusion approach that combine LSTM and RNN and fused the obtained prediction using simple averaging in order to enable end-to-end training and classification processes. The experimental setting that utilize late fusion approachs shown in Figure 4.

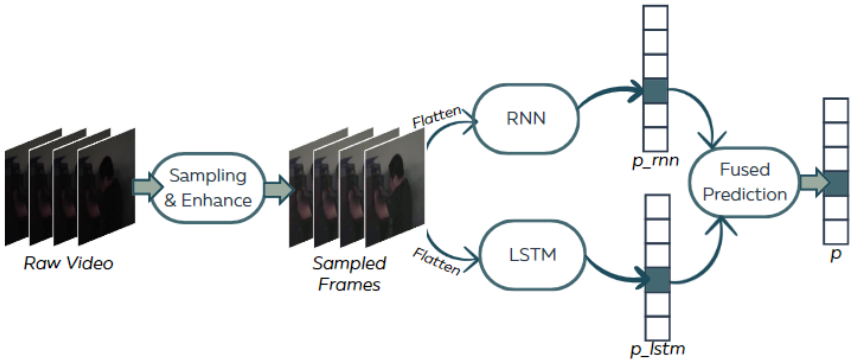

Figure 4: Experimental setting utilizing late fusion technique

3 Frame Sampling

In order to make feature extraction phase more feasible, we start the experiment with sampling video frames. We pick uniform sampling and random sampling.

3.1 Uniform Sampling: Deterministic and Temporal Consistency

Uniform sampling involves selecting frames from a video at regular intervals. Uniform sampling follows a specific pattern. Besides that, uniform sampling maintains the temporal consistency of the video, which can be important for understanding the context of actions in videos (Karpathy et al. (2014)). The uniform sampling results of one video clip can be seen in Figure 5.

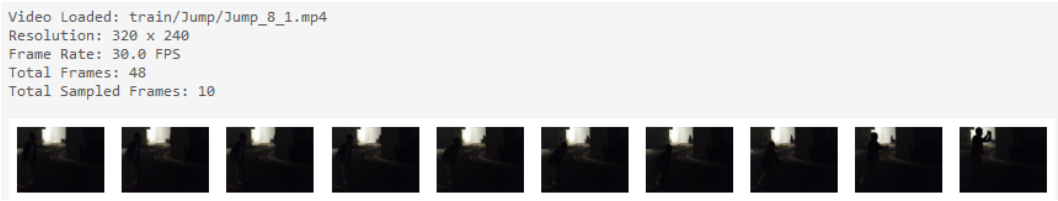

Figure 5: The uniform sampling results of a ”Jump” video clip

3.2 Random Sampling: Non-deterministic and Potential Loss of Temporal Structure

Random sampling involves selecting frames from a video based on a random process. This means that the frames are selected without a fixed pattern, and different runs will yield different results unless the random seed is fixed.

Random sampling is inherently non-deterministic, which means it can give different results each time we sample from the video unless you set a specific seed for the random number generator. It can potentially disrupt the temporal flow of events in the video, which might be crucial for understanding certain actions.

However, random sampling is chosen if the randomness is a desired feature to introduce variability in the training dataset for machine learning models (Misra et al. (2016)).

The random sampling results of same video clip can be seen in Figure 6.

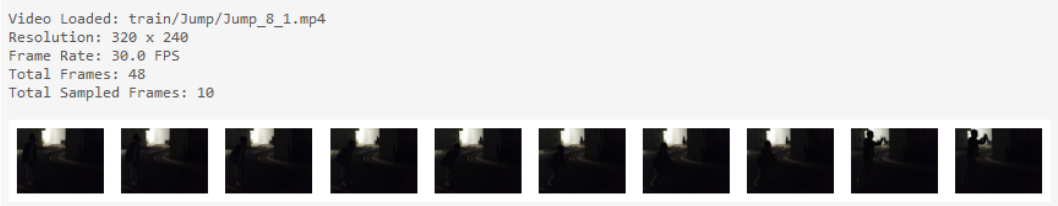

Figure 6: The random sampling results of a ”Jump” video clip

3.3 Similarities between Uniform and Random Sampling

Both methods reduce the number of frames and reduce the size of the dataset that need to be processed, thus saving computational resources. They can both be used to prevent overfitting in machine learning models by providing a manageable and representative subset of the data.

In the context of machine learning and computer vision, since it's crucial to ensure that the sampled frames are representative of the entire video to avoid introducing biases in the analysis or model training, we considered to pick uniform sampling first.

4 Feature Scaling (Normalization)

To mitigate performance degradation of HAR models in dark environments, one intuitive method is to perform pre-processing of dark videos which could improve the visibility of the dark videos. Such a method is indeed effective from the human vision perspective (Xu et al. (2023)).

In this experiment, we use NumPy along with OpenCV to normalize pixel values of a video frame to have some provided mean and standard deviation. In the Python code, this step is actually inside the aforementioned sequential sampling function.

Here's what the code is doing:

- It reads a frame from the video file.
- It converts the pixel values to float32 for proper scaling.
- It uses the provided reference mean and standard deviation for each color channel and scales the pixel values accordingly.
- It clips the values to ensure they are still in the range [0, 255] after normalization.

We try three references values for mean and standard deviation, and also Gamma Intensity Correction Xu et al. (2022) that the uniform sampling results are shown below.

4.1 Zero Mean and Unit Standard Deviation

Five sampling from one of ”Jump” video clip with zero mean and unit standard deviation can be seen in Figure 7.

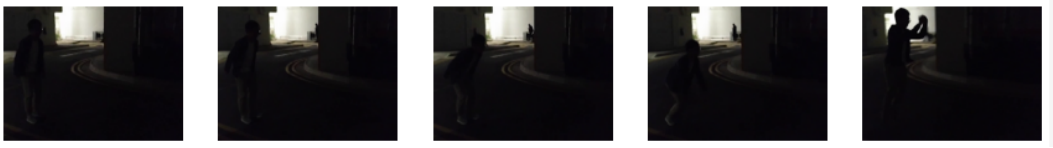

Figure 7: Sampling with zero mean and unit standard deviation

4.2 Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08]

Five sampling from one of "Jump" video clip with Mean [0.07,0.07,0.07] and standard deviation [0.1,0.09,0.08] can be seen in Figure 8.

Figure 8: Sampling with Mean [0.07,0.07,0.07] and standard deviation [0.1,0.09,0.08]

4.3 Mean [0.485,0.456,0.406] and Standard Deviation [0.229,0.224,0.225]

Five sampling from one of "Jump" video clip with Mean [0.485,0.456,0.406] and standard deviation [0.229,0.224,0.225] can be seen in Figure 9.

Figure 9: Sampling with Mean [0.485,0.456,0.406] and standard deviation [0.229,0.224,0.225]

4.4 Enhancement with Gamma Intensity Correction with Gamma = 0.12

Five sampling from one of "Jump" video clip with Gamma Intensity Correction can be seen in Figure 10.

Figure 10: Sampling with Gamma Intensity Correction

5 Feature Extraction Using Pre-trained ResNet50

To extract the feature from each sampled frame, we use pre-trained models that are trained on large datasets, ResNet50

ResNet50 is one of the most popular architectures for feature extraction in computer vision tasks due to its deep architecture and its use of residual connections. Here are some of the benefits of using ResNet50 for feature extraction:

- Deep Hierarchical Feature Learning: With its 50 layers, ResNet50 can capture high-level features.
- Residual Connections: These skip one or more layers and perform identity mapping. They help to mitigate the vanishing gradient problem, allowing deeper networks to be trained effectively.

- Pre-trained on ImageNet: The ResNet50 model has been pre-trained on the ImageNet dataset, which contains over a million images with 1000 different classes.
- Efficiency: Despite its depth, ResNet50 is relatively efficient because of its use of global average pooling and the efficiency of residual blocks.
- Implementation Availability: Pre-trained ResNet50 models are readily available in TensorFlow and Keras, making it easy to use and integrate into this project.

Generals step of using ResNet50 as a feature extractor for video processing are as follow. After process the video to obtain sequence of normalized frames, we need one more step of preprocess the frames to match the input format expected by ResNet50 ($224x224$ pixels). We pass each frame through ResNet50 to obtain **the output features with the dimension of 2048 for each frames**.

Figure 11: Flattened features from a 10-samples of a video clip, ready to hand over to subsequent training

Next, we flatten the features from one sampled video clip, and append all flattened features to get a video-level representation. We flattened the features of each video clips to make Kernel SVM easier to process.

We extract four set of features and labels from four different normalization settings:

- Set 1: Zero Mean and Unit Standard Deviation
- Set 2: Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08]
- Set 3: Mean [0.485,0.456,0.406] and standard deviation [0.229,0.224,0.225]
- Set 4: Gamma Intensity Correction with Gamma = 0.12

6 KERNEL SVM

A Support Vector Machine (SVM) is a supervised classification method, that after a training phase can identify if a new point belongs to a class or another with the highest mathematical accuracy. In this study, we will use Kernel SVM as a classifier.

Here are the advantages of using Kernel SVM (Abe (2005),Bishop & Nasrabadi (2006),Boser et al. (1992),Cortes & Vapnik (1995)):

- Non-Linear Mapping: Kernel SVM can efficiently handle non-linear data through the use of kernel functions, which map the data into a higher-dimensional space where it becomes linearly separable.
- Flexibility: The choice of kernel functions (linear, polynomial, RBF, sigmoid, etc.) provides the flexibility to model a variety of relationships in the data.
- Sparsity of Solution: Due to the use of support vectors, the solution to the SVM optimization problem is often sparse, which means that only a subset of the training data is used in the decision function.
- Regularization: The regularization parameter in SVM (often denoted as C) allows the user to control the trade-off between achieving a low training error and a low testing error, which helps to avoid overfitting.

While the disadvantages of using Kernel SVM are as follows (Abe (2005),Bishop & Nasrabadi (2006),Boser et al. (1992),Cortes & Vapnik (1995)):

- Choice of Kernel: Selecting the appropriate kernel and tuning its parameters (like degree in polynomial kernel, gamma in RBF, etc.) can be tricky and requires domain knowledge.

- Scalability: Kernel SVMs can be inefficient on very large datasets because the training time complexity can be between quadratic to cubic with respect to the number of samples.
- Interpretability: Kernel SVM models, particularly with non-linear kernels, are less interpretable than simpler models, such as linear models.
- Parameter Sensitivity: The performance of SVMs is highly sensitive to the choice of the regularization parameter C, and the parameters of the kernel function, which need to be carefully chosen using cross-validation or similar techniques.
- Speed and Performance: SVMs, in general, are slower to train compared to some other algorithms like decision trees or linear models, especially as the dataset size grows.
- Binary by Default: SVMs are inherently binary classifiers, and while they can be extended to multiclass problems using strategies like one-vs-rest or one-vs-one, this can complicate the training process.
- Data Preparation: SVMs require preprocessed data (like normalization or scaling) for optimal performance, which adds to the preprocessing workload.
- Not suitable for large datasets with many features: SVMs can be very slow and can consume a lot of memory when the dataset has many features. Not suitable for datasets with missing values: SVMs requires complete datasets, with no missing values, it can not handle missing values.

Support vector machine (SVM) is a powerful technique for data classification. Despite of its good theoretic foundations and high classification accuracy, normal SVM is not suitable for classification of large data sets, because the training complexity of SVM is highly dependent on the size of data set (Abe (2005),Bishop & Nasrabadi (2006),Boser et al. (1992),Cortes & Vapnik (1995)).

6.1 Exploration of Kernel SVM as Classifier

We leverage Kernel SVM to be a classifier with the four set of obtained feature and labels that mentioned in the previous chapter and get the below results.

6.1.1 Zero Mean and Unit Standard Deviation

In first set of normalization setting (zero mean and unit standard deviation), we get the accuracy of Kernel SVM is 14.58%.

We try to utilize PCA into the features and train the Kernel SVM as well. After applying PCA, we get the accuracy of Kernel SVM is 15.62%. The kernel SVM decision boundaries plot in this experiment can be seen in Figure 12.

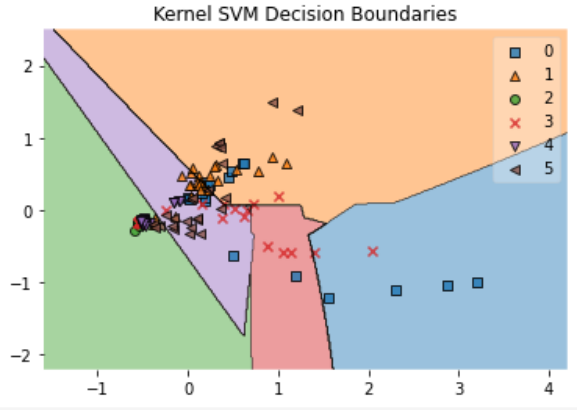

Figure 12: The kernel SVM decision boundaries plot for normalization setting of zero mean and unit standard deviation

6.1.2 MEAN [0.07,0.07,0.07] AND STANDARD DEVIATION [0.1,0.09,0.08]

In the second set of normalization setting (Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08]), we get the accuracy of Kernel SVM is 12.50%.

We try to utilize PCA into the features and train the Kernel SVM as well. After applying PCA, we get the accuracy of Kernel SVM is 20.83%. The kernel SVM decision boundaries plot in this experiment can be seen in Figure 13.

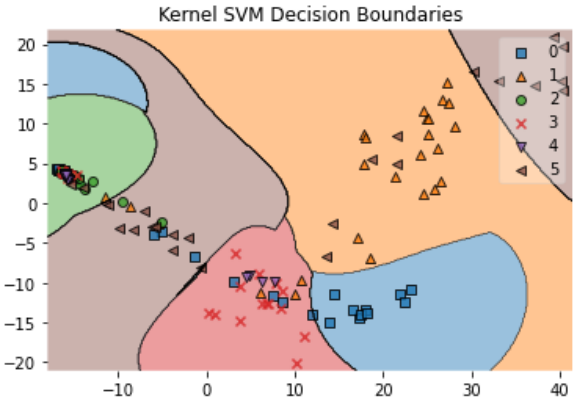

Figure 13: The kernel SVM decision boundaries plot for the second normalization setting (Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08])

6.1.3 MEAN [0.485,0.456,0.406] AND STANDARD DEVIATION [0.229,0.224,0.225]

In the third set of normalization setting (Mean [0.485,0.456,0.406] and Standard Deviation [0.229,0.224,0.225]), we get the accuracy of Kernel SVM is 21.88%. We try to utilize PCA into the features and train the Kernel SVM as well. After applying PCA, we get the accuracy of Kernel SVM is 23.96%. The kernel SVM decision boundaries plot in this experiment can be seen in Figure 14.

This normalization setting does enhance the performance of Kernel SVM and give the best result among other image enhancement configurations.

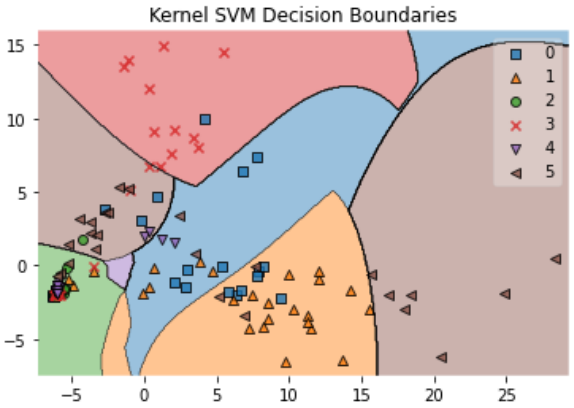

Figure 14: The kernel SVM decision boundaries plot for the third normalization setting (Mean [0.485,0.456,0.406] and Standard Deviation [0.229,0.224,0.225])

6.1.4 Enhancement with Gamma Intensity Correction with Gamma = 0.12

In the fourth set of normalization setting (Gamma Intensity Correction with Gamma = 0.12), we get the accuracy of Kernel SVM is 17.71%.

We try to utilize PCA into the features and train the Kernel SVM as well. After applying PCA, we get the accuracy of Kernel SVM is 20.83%. The kernel SVM decision boundaries plot in this experiment can be seen in Figure 15.

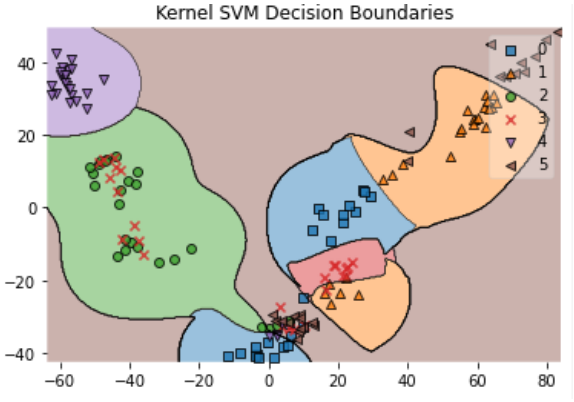

Figure 15: The kernel SVM decision boundaries plot for the fourth normalization setting (Gamma Intensity Correction with Gamma = 0.12)

6.2 Discussion on the Effects of Image Enhancements on Kernel SVM HAR Model

Image enhancement is a crucial preprocessing step in the workflow of human action recognition systems. It aims to improve the quality of images or video frames, making the underlying patterns more discernible for analysis by both human observers and computer algorithms.

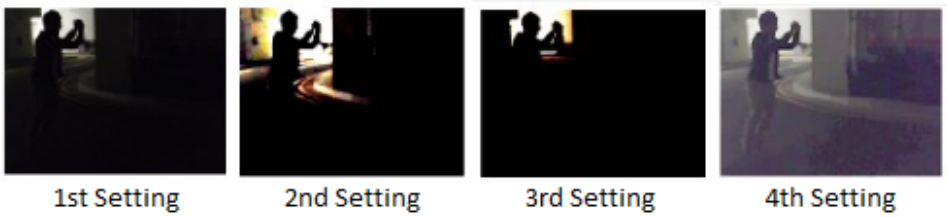

Figure 16: Sampled output frames resulting from different setting of image enhancement

We recapitulate the performance of our Kernel SVM through four different normalization setting in Table 2.

Table 2: The recapitulation results of Kernel SVM through different Normalization Setting

Normalization Setting	**Kernel SVM**	**Kernel SVM with PCA**
Zero Mean and Unit Std (Standard Deviation)	14.58%	15.62%
Mean [0.07,0.07,0.07] and Std [0.1,0.09,0.08]	12.50%	20.83%
Mean [0.485,0.456,0.406] and Std [0.229,0.224,0.225]	21.88%	23.96%
Gamma Intensity Correction with Gamma = 0.12	17.71%	20.83%

In human perception, applying image enhancements could improve our capability in classifying human actions. We can see from our experimental results that our HAR model does follow such intuition as well. By utilizing different normalization setting to enhance each frames of each video clips, the classification results increased by $7 - 9\%$ without PCA, and it improves by $5 - 8\%$ with using PCA.

Enhanced images could significantly improve the performance of human action recognition (traditional machine learning-based) in the following ways:

- Improved Visibility and Contrast: Image enhancement techniques can improve the visibility of the subjects and the contrast between them and the background, making it easier for feature extraction algorithms to identify relevant features.

- Illumination Correction: Uneven or poor lighting can obscure important features. Different normalization setting can standardize lighting across the dataset, helping algorithms to focus on structural and motion information rather than lighting variations.

- Color Enhancement: Sometimes, certain actions are closely related to color information (e.g., a uniform color in sports). Enhancing the color can help in distinguishing between different actors or actions.

- Normalization: Image normalization (standardizing the scale and range of pixel values) ensures that the input data fed into the recognition system is consistent, which is crucial for many machine learning algorithms.

Overall, the application of appropriate image enhancement techniques is proved to lead to more accurate and reliable traditional machine learning-based HAR.

7 Late Fusion Techniques of Deep Learning

This project aims explore HAR in videos shot in the dark leveraging the late fusion technique of deep learning as well. Late fusion, a method of combining the decisions from multiple models at a later stage, offers the advantage of capturing a more holistic view of the action from different representations. Furthermore, late fusion techniques in this study is designed to enable end-to-end training. The structure along training and testing procedures can be seen in Figure 17.

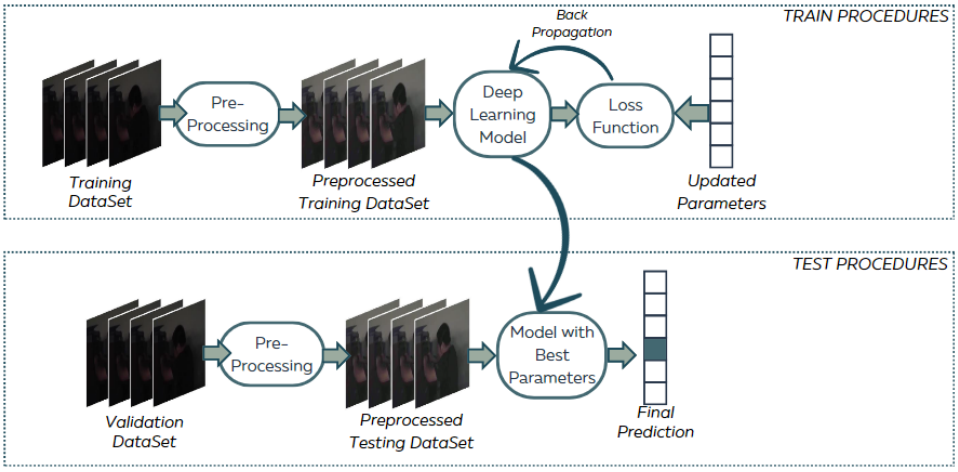

Figure 17: Training and Testing Procedures

To capture both long-term and short-term patterns, Recurrent Neural Network (RNN) was proposed. Long Short-Term Memory (LSTM) is a specialized designs of RNN, which uses multiple gate units to forget or memorize specific signals Hochreiter & Schmidhuber (1997). LSTM is suitable for representation learning of sequence data. However, LSTM requires sufficient training data and the computational complexity of its training is high when compared to CNN. This fusion is shown in Figure 18.

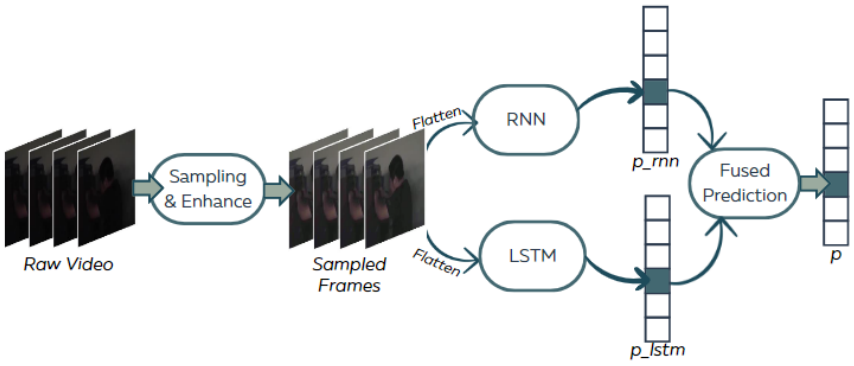

Figure 18: Experimental setting utilizing late fusion technique

7.1 Exploration of Late Fusion Techniques of Deep Learning

We leverage Late Fusion of LSTM and RNN to be a classifier through different enhancement setting that are similar with chapter 6.

7.2 Zero Mean and Unit Standard Deviation

In this normalization setting (zero mean and unit standard deviation), we get that the accuracy of late fusion techniques of deep learning is 23.96%.

7.3 Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08]

In this normalization setting (Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08]), we get that the accuracy of late fusion techniques of deep learning is 22.92%.

7.4 Mean [0.485,0.456,0.406] and Standard Deviation [0.229,0.224,0.225]

In this normalization setting (Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08]), we get that the accuracy of late fusion techniques of deep learning is 16.67%.

7.5 Enhancement with Gamma Intensity Correction with Gamma = 0.12

In this normalization setting (Gamma Intensity Correction with Gamma = 0.12), we get that the accuracy of late fusion techniques of deep learning is 18.75%.

In our experiment, gamma intensity correction improves the performance of deep learning.

7.6 Discussion on the Comparison between Traditional Machine Learning and Deep Learning

We recapitulate the performance of our Kernel SVM and late fusion techniques of deep learning through four different normalization setting in Table 3.

Table 3: The recapitulation results of Kernel SVM through different Normalization Setting

Normalization Setting	**Kernel SVM**	**Kernel SVM with PCA**	**Late Fusion (LSTM-RNN)**
Zero Mean and Unit Std (Standard Deviation)	14.58%	15.62%	23.96%
Mean [0.07,0.07,0.07] and Std [0.1,0.09,0.08]	12.50%	20.83%	22.92%
Mean [0.485,0.456,0.406] and Std [0.229,0.224,0.225]	21.88%	23.96%	16.67%
Gamma Intensity Correction with Gamma = 0.12	17.71%	20.83%	18.75%

The utilization of late fusion of deep learning without any enhancements gives surprising results, with the increase of $8 - 9\%$ compared to the utilization of Kernel SVM. Without any image enhancement techniques applied on the video clips' frames, late fusion techniques of deep learning gives better results than traditional machine learning. Traditional algorithms, which often rely on manual feature extraction and selection, can benefit from pre-processing steps that refine the input data and mitigate issues such as noise, contrast variability, and distortions.

In contrast, deep learning architectures, particularly convolutional neural networks, have inherent capabilities for feature extraction and can implicitly learn to manage raw and unprocessed data. Consequently, the utility of image enhancement as a pre-processing step may not significantly influence the outcome of deep learning-based image analysis, as these advanced models are designed to autonomously learn optimal representations for the task at hand from the data itself.

8 Conclusion

From the exploration in this experiment, we can conclude several points as follows:

- Both uniform sampling and random sampling reduce the number of frames and help reduce the size of dataset. They can both prevent overfitting in machine learning. If we favor temporal consistency, it is better to choose uniform sampling. If the randomness is a desired feature, then the choice of random sampling is in favor.
- The feature scaling or normalization is a method for enhance the video clips' frames. It does improve the visibility of dark videos (from human's perception) and traditional machine learning performance as well, but not increase the performance of deep learning-based classifier.

- Gamma intensity correction that applied on video clips' frames does not improve the performance of both traditional machine learning and deep learning.
- Pre-trained ResNet50 is an effective techniques to do feature extraction. It reduces the size of $240x320x3$ pixels data into 2048 float number. However, some dimension manipulation should be done before the features are being handed over to the classifier.
- Kernel SVM gives classification accuracy that can be significantly improved by image enhancement and PCA application. However, Kernel SVM is not suitable for classification of large datasets.
- Late fusion techniques of LSTM and RNN give better accuracy for raw dark images and videos, but gives lower accuracy compared to traditional machine learning that integrated with enhancement.
- Deep learning has capabilites to manage raw data, so that hand-crafted feature engineering step could be skipped. Furthermore, deep learning provides end-to-end training and evaluation processes.

REFERENCES

Shigeo Abe. *Support vector machines for pattern classification*, volume 2. Springer, 2005.

Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pp. 144–152, 1992.

N.E. Boudette. 'it happened so fast': Inside a fatal tesla autopilot accident. `https://www.nytimes.com/2021/08/17/business/tesla-autopilotaccident.html`, 2021.

P Brown. Autonomous vehicle at night. `https://www.autonomousvehicleinternational.com/opinion/autonomousvehicles-at-night.html/`, 2009.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20:273–297, 1995.

S Hochreiter and J Schmidhuber. Long short-term memory. *Neural Computation*, 8:1735–1780, 1997.

Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1725–1732, 2014. doi: 10.1109/CVPR.2014.223.

Ishan Misra, C. Lawrence Zitnick, and Martial Hebert. Shuffle and learn: Unsupervised learning using temporal order verification, 2016.

Yuecong Xu, Jianfei Yang, Haozhi Cao, Kezhi Mao, Jianxiong Yin, and Simon See. Arid: A new dataset for recognizing action in the dark, 2022.

Yuecong Xu, Jianfei Yang, Haozhi Cao, Jianxiong Yin, Zhenghua Chen, Xiaoli Li, Zhengguo Li, and Qianwen Xu. Going deeper into recognizing actions in dark environments: A comprehensive benchmark study, 2023.

Jianfei Yang, Yuecong Xu, Haozhi Cao, Han Zou, and Lihua Xie. Deep learning and transfer learning for device-free human activity recognition: A survey. *Journal of Automation and Intelligence*, 1, 2022.

CONTENTS

1 Introduction **1**

2 The DataSet and Experimental Setting **2**

2.1 Action Classes 2

2.2 Basic Statistic 2

2.3 Experimental Setting 3

3 Frame Sampling **4**

3.1 Uniform Sampling: Deterministic and Temporal Consistency 4

3.2 Random Sampling: Non-deterministic and Potential Loss of Temporal Structure . . 4

3.3 Similarities between Uniform and Random Sampling 5

4 Feature Scaling (Normalization) **5**

4.1 Zero Mean and Unit Standard Deviation 5

4.2 Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08] 6

4.3 Mean [0.485,0.456,0.406] and Standard Deviation [0.229,0.224,0.225] 6

4.4 Enhancement with Gamma Intensity Correction with Gamma = 0.12 6

5 Feature Extraction Using Pre-trained ResNet50 **6**

6 Kernel SVM **7**

6.1 Exploration of Kernel SVM as Classifier 8

6.1.1 Zero Mean and Unit Standard Deviation 8

6.1.2 Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08] 9

6.1.3 Mean [0.485,0.456,0.406] and Standard Deviation [0.229,0.224,0.225] . . 9

6.1.4 Enhancement with Gamma Intensity Correction with Gamma = 0.12 10

6.2 Discussion on the Effects of Image Enhancements on Kernel SVM HAR Model . . 10

7 Late Fusion Techniques of Deep Learning **11**

7.1 Exploration of Late Fusion Techniques of Deep Learning 12

7.2 Zero Mean and Unit Standard Deviation 12

7.3 Mean [0.07,0.07,0.07] and Standard Deviation [0.1,0.09,0.08] 12

7.4 Mean [0.485,0.456,0.406] and Standard Deviation [0.229,0.224,0.225] 13

7.5 Enhancement with Gamma Intensity Correction with Gamma = 0.12 13

7.6 Discussion on the Comparison between Traditional Machine Learning and Deep Learning 13

8 Conclusion **13**

9 APPENDIX A: The Code **16**

9 APPENDIX A: THE CODE

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Nov  5 19:07:40 2023

@author: Nurfitri Anbarsanti G2104045K
"""
# Bismillahirrahmanirrahim, EE6222 Assignment

import cv2
import numpy as np
import glob
import os
import tensorflow as tf
import matplotlib.pyplot as plt
import random
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import Model
from sklearn.metrics import classification_report, accuracy_score
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.utils.multiclass import unique_labels
from sklearn import metrics
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from array import array
import plotly.graph_objs as go
import plotly.offline as py
py.init_notebook_mode(connected=True)
from mlxtend.plotting import plot_decision_regions
from mpl_toolkits.mplot3d import Axes3D
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, SimpleRNN
from keras.layers import Conv3D, MaxPooling3D, Flatten, Dense
from keras.utils import to_categorical

### ------------------------------------------------------------------------
### --------------- FUNCTION DEFINITION -----------------------------------


# Function to Check if the video was opened successfully
def check_video_opened(cap, video_path):
    if not cap.isOpened():
        print("Error: Could not open Video")
    else:
        # Get video properties
        width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
        height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        fps = cap.get(cv2.CAP_PROP_FPS)
        total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

        print(f"Video Loaded: {video_path}")
        print(f"Resolution: {width} x {height}")
        print(f"Frame Rate: {fps} FPS")
        print(f"Total Frames: {total_frames}")
        print(f"Total Sampled Frames: 10")
    cap.release()

# Function to get frame count
def get_frame_count(video_path):
    cap = cv2.VideoCapture(video_path)
    frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    cap.release()
    return frame_count

# Function to sample frames uniformly
def sample_frames_uniformly(video_path, num_samples, ref_mean, ref_std, GC, gamma):
    frame_count = get_frame_count(video_path)
```

```
    interval = frame_count // num_samples
    frames = []

    cap = cv2.VideoCapture(video_path)
    for i in range(0, frame_count, interval):
        cap.set(cv2.CAP_PROP_POS_FRAMES, i)
        ret, frame = cap.read()
        frame = frame.astype('float32')
        if GC==True:
            # Apply gamma correction
            standardized_frame = cv2.pow(frame, gamma)
            # Scale back to range 0-255 and convert to uint8
            # standardized_frame = np.uint8(standardized_frame*255)
            # Getting mean and standard deviation value
            # ref_mean, ref_std = cv2.meanStdDev(standardized_frame)
            # print("ref_mean", ref_mean)
            # print("ref_std", ref_std)
        else:
            # Standardize the frame using provided mean and standard deviation
            standardized_frame = (frame - ref_mean*255) / (ref_std*255)
        if ret:
            frames.append(standardized_frame)
        else:
            break
    cap.release()
    return frames[0:num_samples]

# Function to sample frames randomly
def sample_frames_randomly(video_path, num_samples, ref_mean, ref_std, GC, gamma):
    frame_count = get_frame_count(video_path)
    frames = []

    # Generate num_samples random unique frame indices
    random_indices = random.sample(range(frame_count), num_samples)

    cap = cv2.VideoCapture(video_path)
    for idx in sorted(random_indices):  # Sort the indices to maintain temporal order
        cap.set(cv2.CAP_PROP_POS_FRAMES, idx)
        ret, frame = cap.read()
        frame = frame.astype('float32')
        if GC==True:
            # Apply gamma correction
            standardized_frame = cv2.pow(frame, gamma)
            # Scale back to range 0-255 and convert to uint8
            # standardized_frame = np.uint8(standardized_frame*255)
            # Getting mean and standard deviation value
            # ref_mean, ref_std = cv2.meanStdDev(standardized_frame)
            # print("ref_mean", ref_mean)
            # print("ref_std", ref_std)
        else:
            # Standardize the frame using provided mean and standard deviation
            standardized_frame = (frame - ref_mean*255) / (ref_std*255)
        if ret:
            frames.append(standardized_frame)
        else:
            print(f"Frame at index {idx} could not be read")
            continue  # Continue to try to read the next frame if this one fails
    cap.release()
    return frames[0:num_samples]

# Function to display frames using matplotlib
def display_frames(frames):
    fig, axes = plt.subplots(1, len(frames), figsize=(15, 5))
    for ax, frame in zip(axes, frames):
        ax.imshow(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
        ax.axis('off')
    plt.show()
```

```
135
136 # Function to extract feature from frames using ResNet50
137 def extract_features(frames, model):
138     features = np.empty((1,0))
139     idx = 0;
140     while idx < len(frames):
141         # Resize the frame to 224 x 224 pixels, assumed that it is already in float32
142         frame = cv2.resize(frames[idx], (224, 224))
143         # Add an extra dimension (for batch size)
144         frame = np.expand_dims(frame, axis=0)
145         # Preprocess the frame for ResNet50
146         frame = preprocess_input(frame)
147         # Get the features for the frame
148         feature = model.predict(frame)
149         # flatten the features\
150         feature = feature.flatten()
151         # Append array of features
152         features = np.append(features, feature)
153         idx += 1
154     return features
155
156 # Function to get sampled, normalized frames from train folder
157 def get_train_frames(train_path, num_samples, ref_mean, ref_std, GC, gamma):
158     frames = []
159     labels = []
160     act = ['Jump', 'Run', 'Sit', 'Stand', 'Turn', 'Walk']
161     idx = 0
162     while idx < len(act):
163         videos_path = train_path + "/" + act[idx]
164         # print(videos_path)
165         for path in glob.glob(os.path.join(videos_path, "*.mp4")):
166             print(path)
167             cap = cv2.VideoCapture(path)
168             # check_video_opened(cap, path)
169             frame = sample_frames_uniformly(path, num_samples, ref_mean, ref_std, GC,
                    gamma)
170             frames.append(frame)
171             labels.append(idx)
172         idx += 1
173     return frames, labels
174
175 # Function to get sampled, normalized frames from validate folder
176 def get_validate_frames(validate_path, num_samples, ref_mean, ref_std, GC, gamma):
177     frames = []
178     labels = [0]*17 + [1]*15 + [2]*15 + [3]*16 + [4]*17 + [5]*16
179     for path in glob.glob(os.path.join(validate_path, "*.mp4")):
180         # print(path)
181         cap = cv2.VideoCapture(path)
182         # check_video_opened(cap, path)
183         frame = sample_frames_uniformly(path, num_samples, ref_mean, ref_std, GC, gamma)
184         frames.append(frame)
185     return frames, labels
186
187 # Function to get features from train folder
188 def get_train_features(train_path, num_samples, model, ref_mean, ref_std, GC, gamma):
189     features = np.empty((0, 2048*num_samples))
190     labels = np.empty((1,0), dtype=np.int32)
191     act = ['Jump', 'Run', 'Sit', 'Stand', 'Turn', 'Walk']
192     idx = 0
193     while idx < len(act):
194         videos_path = train_path + "/" + act[idx]
195         # print(videos_path)
196         for path in glob.glob(os.path.join(videos_path, "*.mp4")):
197             # print(path)
198             cap = cv2.VideoCapture(path)
199             # check_video_opened(cap, path)
200             frame = sample_frames_uniformly(path, num_samples, ref_mean, ref_std, GC,
```

```
                gamma)
                feature = extract_features(frame, model)
                features = np.vstack((features, feature))
                labels = np.append(labels, idx)
            idx += 1
        return features, labels

    # Function to get sampled, normalized frames from validate folder
    def get_validate_features(validate_path, num_samples, model, ref_mean, ref_std, GC, gamma
    ):
        features = np.empty((0, 2048*num_samples))
        labels = np.array([0]*17 + [1]*15 + [2]*15 + [3]*16 + [4]*17 + [5]*16)
        for path in glob.glob(os.path.join(validate_path, "*.mp4")):
            # print(path)
            cap = cv2.VideoCapture(path)
            # check_video_opened(cap, path)
            frame = sample_frames_uniformly(path, num_samples, ref_mean, ref_std, GC, gamma)
            feature = extract_features(frame, model)
            print(feature)
            features = np.vstack((features, feature))
        return features, labels

    # Function to sample frames for CNN
    def sample_frames_CNN(video_path, num_steps, ref_mean, ref_std, GC, gamma):
        frame_count = get_frame_count(video_path)
        interval = frame_count // num_steps
        samples = []

        cap = cv2.VideoCapture(video_path)
        for i in range(0, frame_count, interval):
            cap.set(cv2.CAP_PROP_POS_FRAMES, i)
            ret, frame = cap.read()
            frame = frame.astype('float32')
            if GC==True:
                # Apply gamma correction
                standardized_frame = cv2.pow(frame, gamma)
                # Scale back to range 0-255 and convert to uint8
                standardized_frame = np.uint8(standardized_frame*255)
                # Getting mean and standard deviation value
                # ref_mean, ref_std = cv2.meanStdDev(standardized_frame)
                # print("ref_mean", ref_mean)
                # print("ref_std", ref_std)
            else:
                # Standardize the frame using provided mean and standard deviation
                standardized_frame = (frame - ref_mean*255) / (ref_std*255)
            standardized_frame = standardized_frame.flatten()
            if ret:
                samples.append(standardized_frame)
            else:
                break
        cap.release()
        samples = np.array(samples[0:num_steps])
        return samples

    # Function to get sampled, normalized frames from train folder
    def get_train_samples(train_path, num_steps, ref_mean, ref_std, GC, gamma):
        samples = []
        labels = np.empty((1,0), dtype=np.int32)
        act = ['Jump', 'Run', 'Sit', 'Stand', 'Turn', 'Walk']
        idx = 0
        while idx < len(act):
            videos_path = train_path + "/" + act[idx]
            # print(videos_path)
            for path in glob.glob(os.path.join(videos_path, "*.mp4")):
                # print(path)
                cap = cv2.VideoCapture(path)
                # check_video_opened(cap, path)
```

```
                sample = sample_frames_CNN(path, num_steps, ref_mean, ref_std, GC, gamma)
                samples.append(sample)
                labels = np.append(labels, idx)
            idx += 1
        samples = np.array(samples)
        return samples, labels

# Function to get sampled, normalized frames from validate folder
def get_validate_samples(validate_path, num_steps, ref_mean, ref_std, GC, gamma):
    samples = []
    labels = np.array([0]*17 + [1]*15 + [2]*15 + [3]*16 + [4]*17 + [5]*16)
    for path in glob.glob(os.path.join(validate_path, "*.mp4")):
        # print(path)
        cap = cv2.VideoCapture(path)
        # check_video_opened(cap, path)
        sample = sample_frames_CNN(path, num_steps, ref_mean, ref_std, GC, gamma)
        samples.append(sample)
    samples = np.array(samples)
    return samples, labels


### ------------------------------------------------------------------------
### ------------------------- Main Program  ---------------------------------

### ----------------------- Check Samplings of one video clips
video_path = 'train/Jump/Jump_8_1.mp4'

## Create a VideoCapture object
cap = cv2.VideoCapture(video_path)

## Define some variables for normalization
ref0_mean = np.array([0, 0, 0], dtype='float32')
ref0_std = np.array([1, 1, 1], dtype='float32')
ref1_mean = np.array([0.07, 0.07, 0.07], dtype='float32')
ref1_std = np.array([0.1, 0.09, 0.08], dtype='float32')
enh_mean = np.array([0.485, 0.456, 0.406], dtype='float32')
enh_std = np.array([0.229, 0.224, 0.225], dtype='float32')

## Define the feature extraction
model = ResNet50(include_top=False, weights='imagenet', pooling='avg')

## Check if the video was opened successfully
check_video_opened(cap, video_path)

## Getting sampled frames (with normalization)
frames_0 = sample_frames_uniformly(video_path, 10, ref0_mean, ref0_std, GC=False, gamma=0
)
frames_1 = sample_frames_uniformly(video_path, 10, ref1_mean, ref1_std, GC=False, gamma=0
)
frames_enh = sample_frames_uniformly(video_path, 10, enh_mean, enh_std, GC=False, gamma=0
)
frames_gc = sample_frames_uniformly(video_path, 10, ref0_mean, ref0_std, GC=True, gamma =
 0.12)

# # Display sampled frames
display_frames(frames_0)
display_frames(frames_1)
display_frames(frames_enh)
display_frames(frames_gc)

### Extract one Feature from one Frame using defined model
features = extract_features(frames_gc, model)

### ------------------------------------------------------------------------
### ------------------------ SVM for HAR -----------------------------------

###  Get features from All Videos (for Kernel SVM)
```

```
(first Normalization Setting)
features_tr0, labels_tr0 = get_train_features('train', 10, model, ref0_mean, ref0_std, GC=False, gamma=0)
features_val0, labels_val0 = get_validate_features('validate', 10, model, ref0_mean, ref0_std, GC=False, gamma=0)

## (second Normalization Setting)
features_tr1, labels_tr1 = get_train_features('train', 10, model, ref1_mean, ref1_std, GC=False, gamma=0)
features_val1, labels_val1 = get_validate_features('validate', 10, model, ref1_mean, ref1_std, GC=False, gamma=0)

## (third Normalization Setting)
features_tr_en, labels_tr_en = get_train_features('train', 10, model, enh_mean, enh_std, GC=False, gamma=0)
features_val_en, labels_val_en = get_validate_features('validate', 10, model, enh_mean, enh_std, GC=False, gamma=0)

### (fourth Normalization Setting)
features_tr_gc, labels_tr_gc = get_train_features('train', 10, model, ref0_mean, ref0_std, GC=True, gamma=0.12)
features_val_gc, labels_val_gc = get_validate_features('validate', 10, model, ref0_mean, ref0_std, GC=True, gamma=0.12)

# Apply Kernel SVM Classifier into the obtained feature
svm_with_kernel = SVC(gamma=0.01, kernel='rbf', probability=True)
svm_with_kernel.fit(features_tr_gc, labels_tr_gc)
y_pred = svm_with_kernel.predict(features_val_gc)
precision = metrics.accuracy_score(y_pred, labels_val_gc) * 100
print("Accuracy of Kernel SVM: {0:.2f}%".format(precision))

# Using Kernel SVM Classifier into the obtained feature with PCA
pca = PCA(n_components = 2)
features_tr_gc = pca.fit_transform(features_tr_gc)
features_val_gc = pca.fit_transform(features_val_gc)
svm_with_kernel.fit(features_tr_gc, labels_tr_gc)
y_pred = svm_with_kernel.predict(features_val_gc)
precision = metrics.accuracy_score(y_pred, labels_val_gc) * 100
print("Accuracy of Kernel SVM with PCA: {0:.2f}%".format(precision))

# Plotting decision boundaries
plot_decision_regions(features_tr_gc, labels_tr_gc, clf=svm_with_kernel, legend=1)
plt.title('Kernel SVM Decision Boundaries')
plt.show()


### ------------------------------------------------------------------------
### ----------------------- Late Fusion of RNN and LSTM --------------------

# Load the training and validate frames
frames_train, labels_train = get_train_frames('train', 10)
frames_validate, labels_validate = get_validate_frames('validate', 10)

### Load the training and validate sampled frames (first Normalization Setting)
samples_tr0, labels_tr0 = get_train_samples('train', 10, ref0_mean, ref0_std, GC=False, gamma=0)
samples_val0, labels_val0 = get_validate_samples('validate', 10, ref0_mean, ref0_std, GC=False, gamma=0)

### Load the training and validate sampled frames (second Normalization Setting)
samples_tr1, labels_tr1 = get_train_samples('train', 10, ref1_mean, ref1_std, GC=False, gamma=0)
samples_val1, labels_val1 = get_validate_samples('validate', 10, ref1_mean, ref1_std, GC=False, gamma=0)

### Load the training and validate sampled frames (third Normalization Setting)
samples_tr_en, labels_tr_en = get_train_samples('train', 10, enh_mean, enh_std, GC=False,
```

```
 gamma=0)
samples_val_en, labels_val_en = get_validate_samples('validate', 10, enh_mean, enh_std, GC=False, gamma=0)

### Load the training and validate sampled frames (fourth Normalization Setting)
samples_tr_gc, labels_tr_gc = get_train_samples('train', 10, ref0_mean, ref0_std, GC=True, gamma=0.12)
samples_val_gc, labels_val_gc = get_validate_samples('validate', 10, ref0_mean, ref0_std, GC=True, gamma=0.12)

# One-hot encode the labels
y_one_hot = to_categorical(labels_tr_gc, 6)

# Define the RNN model
rnn_model = Sequential()
rnn_model.add(SimpleRNN(50, input_shape=(samples_tr_gc.shape[1], samples_tr_gc.shape[2]), return_sequences=False))
rnn_model.add(Dense(y_one_hot.shape[1], activation='softmax'))

# Define the LSTM model
lstm_model = Sequential()
lstm_model.add(LSTM(50, input_shape=(samples_tr_gc.shape[1], samples_tr_gc.shape[2]), return_sequences=False))
lstm_model.add(Dense(y_one_hot.shape[1], activation='softmax'))

# Compile the models
rnn_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
lstm_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Get predictions from both models
rnn_predictions = rnn_model.predict(samples_val_gc)
lstm_predictions = lstm_model.predict(samples_val_gc)

# Late fusion: here we simply average the predictions
fused_predictions = (rnn_predictions + lstm_predictions) / 2.0

# Convert predictions to actual labels
final_predictions = np.argmax(fused_predictions, axis=1)

# Evaluate the late fusion model
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(labels_val_gc, final_predictions)
print(f'Late Fusion Accuracy: {accuracy * 100:.2f}%')
```