International Journal of Information System Modeling and Design
Volume 13 • Issue 10

Deep Learning Model for Dynamic Hand Gesture Recognition for Natural Human-Machine Interface on End Devices

Tsui-Ping Chang, National Taichung University of Science and Technology, Taiwan*

Hung-Ming Chen, National Taichung University of Science and Technology, Taiwan

Shih-Ying Chen, National Taichung University of Science and Technology, Taiwan

Wei-Cheng Lin, National Taichung University of Science and Technology, Taiwan

ABSTRACT

As end devices have become ubiquitous in daily life, the use of natural human-machine interfaces has become an important topic. Many researchers have proposed the frameworks to improve the performance of dynamic hand gesture recognition. Some CNN models are widely used to increase the accuracy of dynamic hand gesture recognition. However, most CNN models are not suitable for end devices. This is because image frames are captured continuously and result in lower hand gesture recognition accuracy. In addition, the trained models need to be efficiently deployed on end devices. To solve the problems, the study proposes a dynamic hand gesture recognition framework on end devices. The authors provide a method (i.e., ModelOps) to deploy the trained model on end devices, by building an edge computing architecture using Kubernetes. The research provides developers with a real-time gesture recognition component. The experimental results show that the framework is suitable on end devices.

KEYWORDS

Deep Learning, Hand Gesture Recognition, ModelOps, Natural Human–Machine Interface, Object Detection

1. INTRODUCTION

With the rapid development and popularization of computers and information technology, people can use their end devices (i.e., modern smartphones and Raspberry Pi) nearly anywhere, resulting in considerable research being devoted to the development of new applications for these ubiquitous end devices. Although these new applications provide significant benefits to users, their human–machine interfaces are still keyboards, mouses, or touch screens (Wang et al., 2017). Hand gesture recognition (Kim & Toomajian, 2016) can provide users with a more lively, natural, and convenient human–machine interface to operate and invoke applications on end devices. Also, hand gesture recognition can be used in human-robot interaction to create user interfaces that are natural to use and easy to learn. However, locating the hands and segmenting them from the background in an image sequence is a problem for hand gesture recognition.

In recent years, many studies (Costante et al., 2014; Dhingra & Kunz, 2019; Kim & Toomajian, 2016; Nanni et al., 2017; Shin & Sung, 2016; Žemgulys et al., 2018; ZOU et al., 2018) have used

DOI: 10.4018/IJISMD.306636 *Corresponding Author

hand gesture recognition models for human–machine interface applications. These models are largely based on handcrafted features and feature extraction through deep learning. These models can be divided into static and dynamic gesture recognition. Static gesture recognition methods consider spatial features of hands, whereas dynamic gesture recognition methods extract not only spatial features but also temporal features.

In contrast to models based on handcrafted features, models (Costante et al., 2014; Dhingra & Kunz, 2019; Shin & Sung, 2016) based on deep learning perform well in automatic feature learning from image frames. Feature deep learning provides new insights into gesture recognition, and many researchers have attempted to use deep learning methods to extract gesture features from RGB, depth, and skeleton data. In (Shin & Sung, 2016), a dynamic hand gesture recognition technique was developed using a recurrent neural network (RNN) algorithm. In (Costante et al., 2014), deep CNNs and random forest (RF) algorithms were compared, and the results indicated that CNN slightly outperformed RF with sufficient data and achieved significantly better accuracy than other methods. A deep learning CNN can learn hand gesture features from single-mode data or multimodal fusion data. As the appearance and optical flow sequences are relatively easy to obtain, most deep learning methods adopt these two as their input, with few depth-based techniques.

The existing deep learning models are not suitable for dynamic hand gesture recognition in real-time applications on end devices. For methods based on handcrafted features, spatial and temporal features are acquired by different methods from RGB data in the image frames. However, in real-time applications on end devices, the image frames are captured continuously, and the starting image frame is difficult to identify. In contrast, in the deep learning methods, these image frames are captured continuously, many irrelevant features are also extracted from the image frames and learned by the system. In addition, the trained gesture recognition models need to be deployed automatically in real-time applications on user end devices. However, the existing hand gesture recognition methods update and deploy their trained models manually and tend to be inconvenient for users.

Conversely, 3D-CNNs are indeed suitable solutions for hand gesture recognition in real-time applications on end devices. 3D-CNNs can capture appearance and motion simultaneously from a sequence of image frames, from low-level details to high-level semantics. Moreover, because of their ability to be processed in parallel, 3D-CNNs are faster during training and inference compared with other CNN models and can be executed in real time. However, the traditional 3D-CNN models are mainly based on the C3D (Tran et al., 2015) structure, which has only eight convolutional layers. It is shallower than most of the successful CNN models (Hitawala, 2018; Li et al., 2018) used in the image classification domain, resulting in limited representation capacity. In addition, 3D-CNNs are unable to identify the initial image frame in a gesture recognition image sequence, leading to misclassification of gesture actions in recognition phases.

Three main factors make our research unique. First, a combined CNN (namely **E3D**) framework is proposed for dynamic hand gesture recognition in real-time applications on end devices. As far as our knowledge, E3D is the first combined CNN model with static and dynamic gesture recognition model. Our E3D improves real-time feature extraction ability in comparison to the original 3D-CNN model. Second, an object detection method is combined with a 3D-CNN to identify the starting image frame in real-time applications, as images are captured in a continuous stream. Finally, we provide an update and deployment method (namely ModelOps) based on a cloud edge architecture we designed to automatically deploy and update the trained gesture recognition model to a real-time application on end devices. The remainder of this paper is organized as follows. An overview of related research is presented in Section 2. Our proposed **E3D** method is described in Section 3. An edge computing platform using Kubernetes is presented and illustrated in Section 4, and the experimental results and analysis are explained in Section 5. Our results and conclusions are presented in the last section.

2. LITERATURE REVIEW

This section briefly describes extant hand gesture recognition methods based on CNNs, object detection methods, and the software architecture DevOps for automatically deploying and updating the trained model to the end device application. CNNs have achieved superior performance in visual tracking based on their strong feature learning capabilities. They can directly learn features from raw data without resorting to manual modifications. (Gao et al., 2014) used a fully CNN for human tracking analysis, taking the entire frame as input to predict a foreground heat map by one-pass forward propagation. (Wang et al., 2018) proposed a deep tracking framework using a candidate pool of multiple CNNs. Moreover, a deep auto encoder was first pertained offline and then fine-tuned for binary classification in online tracking in (Wang & Yeung, 2013).

In the training process of a CNN, the feature weights are updated by an optimizer at a learning rate η to obtain a better trained model. Gradient descent (GD) is the most widely used optimizer in CNNs. GD provides two types of η, including stochastic gradient descent (SGD) and learning rate decay (LRD). SGD extracts small batches from datasets to train the CNN model and is an iterative method for minimizing/maximizing an objective function, whereas LRD is a method that decreases the value of η as the iterations are increased in the training process. Furthermore, Adam (Kingma & Ba, 2014) is an efficient GD method that computes individual adaptive learning rates for different parameters, and the values of η are calculated for each parameter. In Eq. (2.1) and Eq. (2.2), we use Adam to compute the decaying averages of past and past squared gradients m_t and v_t, respectively. m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, respectively.

$$m_t = \beta_1 m_{t-1} + \left(1-\beta_1\right) g_t \tag{2.1}$$

$$v_t = \beta_2 v_{t-1} + \left(1-\beta_2\right) g_t^2 \tag{2.2}$$

Adam computes bias-corrected first and second moment estimates, as shown in Eq. (2.3) and Eq. (2.4). Its update rule is shown in Eq. (2.5). We propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε, demonstrating empirically that Adam works well in practice and compares favorably to other adaptive learning algorithms.

$$\hat{m}_t = \frac{m_t}{\left(1-\beta_1^t\right)} \tag{2.3}$$

$$\hat{v}_t = \frac{v_t}{\left(1-\beta_2^t\right)} \tag{2.4}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \varepsilon}} \hat{m}_t \tag{2.5}$$

Two-stream CNNs (Li et al., 2018), ResNets (Shamir, 2018), and ResNeXt (Hitawala, 2018) are extensions of CNNs for image classification. A two-stream CNN is an effective approach that trains two CNNs using static frames and temporal motion separately. The temporal motion stream is converted to successive optical flow images so that the CNN designed for images can be directly deployed. Many studies extend the proposed concept of two-stream CNNs from varying perspectives. (Li et al., 2018) attempted to improve the results by using deeper networks and proposed an action detection method based on a two-stream network. (Shamir, 2018) extended the two-stream network by implementing several different fusion methods in different layers instead of using late fusion in the score layer as in the two-stream networks of (Li et al., 2018). ResNets provide state-of-the-art performance across numerous applications. They identify shortcut connections enabling the flow of information across layers without attenuation. ResNeXt is inspired by ResNet and can improve performance in image classification. ResNeXt involves stacking a series of multi-branch residual blocks. These branches perform sets of convolution networks and then aggregate at the ends of blocks.

3D-CNNs have been widely used in the field of action recognition. They use 3D convolutional kernels, which can directly extract spatiotemporal features from low to high levels. Because 3D-CNNs have many more parameters than other CNN models, they are more difficult to train, and their performance is also limited. The earliest 3D convolutional network is C3D (Tran et al., 2015), which is designed based on the VGG ConvNet and has eight convolutional layers. Subsequently, many deeper 3D-CNN models have been designed based on 2D-CNNs and successfully used in the image classification field. For example, Res3D is designed based on ResNeXt (Hitawala, 2018). In addition, S3D proposes replacing some of the 3D convolutional layers with 2D convolutional layers to save computational resources while maintaining the same accuracy. In S3D, the $3 \times 3 \times 3$ convolutional filters are replaced with one $1 \times 3 \times 3$ convolutional filter for the spatial domain and one $3 \times 1 \times 1$ convolutional filter for the temporal domain.

These related CNN models (including 3D-CNNs, ResNeXt, and two-stream CNNs) can be used to learn the features of hand gestures. The relevant features (i.e., the swing direction) can be learned by these models through numerous hand gesture images and videos, and thus the accuracy of gesture recognition can be improved. However, in real-time applications on end devices, the initial or start image of a gesture recognition sequence is important. This is because the start image is the time point to begin the process of running related CNN models for gesture recognition.

Object detection methods can be used to determine start images for gesture recognition in real-time applications. The aim of an object detection method is to find the location of all targets and specify each target category on a given image or video. Many successful methods (i.e., Faster R-CNN, Mask R-CNN, and YOLO) have been proposed for object detection in images or videos. Faster R-CNN (Ren et al., 2015) can be regarded as a system consisting of a regional proposal network. In Faster R-CNN, the regional proposal network is used instead of the selective search algorithm of Fast R-CNN. Mask R-CNN (Huang et al., 2019) detects objects in an image while simultaneously generating a high-quality segmentation mask for each instance. It extends Faster R-CNN by adding a branch for predicting an object mask in parallel with the existing branch for bounding box recognition. YOLO (Zhong & Deng, 2019) unifies target classification and localization into a regression problem. It directly performs regression to detect targets in the image and provides much faster detection. YOLOv2 improves on YOLO prediction accuracy by using a new network structure called Darknet-19, which was designed by removing the fully connected layers of the network. However, YOLO and YOLOv2 are not fast enough to run on end devices. Tiny-YOLO is a lightweight version of YOLO and has a real-time deep neural network for object detection. Tiny-YOLO was designed to create a smaller, faster, and more efficient model to increase the accessibility of real-time object detection for end devices (i.e., the Raspberry Pi and NVIDIA Jetson Nano).

DevOps (Jabbari et al., 2016) is a novel software engineering paradigm that can provide a new way to automatically update and deploy trained CNN models to applications. (Jabbari et al., 2016) described the main concepts of DevOps and investigated how DevOps can mitigate various challenges.

(Jabbari et al., 2016) aimed to specify the concept of DevOps and what practitioners perceive as impediments to adopting it. They defined DevOps by proposing three main attributes: capabilities, culture, and technology. Furthermore, Kubernetes (Hightower et al., 2017) is an open-source version of a container that can be used to implement the concepts of DevOps to automatically update and deploy software into applications. For instance, regardless of which programming language is used to write a given app, the app can be directly mapped to a Kubernetes Service, which then communicates with the Internet or other Services via standard TCP-based protocols. Kubernetes adds a novel Pod layer between server nodes and the containers running on these nodes. Multiple containers can run simultaneously in the same Pod, thereby effectively enhancing data communication efficiency between these containers.

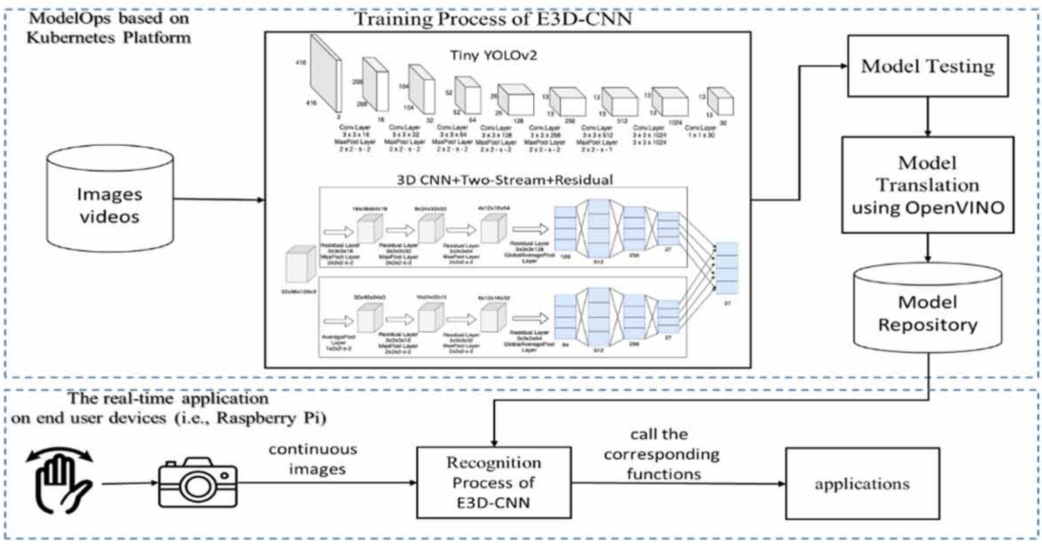
3. ENHANCED 3D-CNN (E3D) FOR DYNAMIC GESTURE RECOGNITION

Based on the related works (i.e., Fast R-CNN and Mask R-CNN) in Section 2, dynamic hand gestures can be identified well through CNN models in deep feature learning. Two issues must be considered in the problem of identifying dynamic hand gestures. Firstly, for real-time applications on end devices, the hand gesture images or videos are input continuously and the starting image must be identified to begin the process of dynamic hand gesture recognition. Secondly, the trained CNN model should be automatically deployed and updated in real-time applications on end devices to provide convenience for users. Therefore, to solve the issues mentioned above, in this study we design a dynamic hand gesture recognition model called Enhanced 3D-CNN (E3D). In our E3D, the lightweight version of the object detection method (i.e., Tiny-YOLO) is first used to identify the start image to initialize the process of recognition. Then, the concepts of two-stream 3D-CNN and the residual method are combined to enhance the accuracy of dynamic hand gesture recognition in real-time applications on end devices. Furthermore, to automatically deploy and update the trained E3D model into an application, we present our design ModelOps based on the concept of DevOps and implemented on the Kubernetes platform. Finally, to reduce the response time required by the CNN processing on the end devices, acceleration devices (i.e., Intel NCS2 and Jetson Nano) are used in our model.

3.1 E3D

Our E3D includes two parts (i.e., a training component and a recognition component), as shown in Figure 1 In the training component, the Kubernetes Pods are implemented to train the E3D model. A Kubernetes Pod is a group of one or more containers with shared resources and a specification for how to run the container. Then, a dataset including static gesture images and dynamic gesture videos is input to train the E3D-CNN model. The static gesture images are input for Tiny YOLOv2 to identify the start image, while the dynamic gesture videos are input for 3D CNN to recognize the hand gesture of users. In the training process, Tiny YOLOv2 and 3D CNN can be executed simultaneously. After the training process, E3D is tested and translated using OpenVINO (Gorbachev et al., 2019). Finally, the translated model is stored in the model repository. In addition, as more images or videos are input for Tiny YOLOv2 or 3D CNN respectively, the new training process is started. Therefore, after the new training process finished, the testing and translating process are automatically started to store the new version model into the model repository.

Figure 1. E3D



In the recognition component, the trained recognition model is deployed on the end device (i.e., a Raspberry Pi) from the model repository through DL Models Cloud Hub. In addition, the real-time application is installed on an end user device to capture dynamic gestures from users through the camera. After the recognition process of the E3D model, the corresponding functions are called. The real-time application and the trained E3D model are set up on the user's end devices. It can continuous capture the images and videos for user gestures and send these images and videos to the trained E3D-CNN model to recognize the user gesture and then call the corresponding applications.

3.2 Datasets

We tested our proposed model using three publicly available datasets: COCO (Patterson & Hays, 2016), 20BN-JESTER Dataset V1 (Materzynska et al., 2019), and the Cambridge Hand Gesture Dataset (Kim et al., 2007). The dataset COCO is used to train Tiny-YOLOv2 to identify the start image frame, while the 20BN-JESTER Dataset V1 and Cambridge Hand Gesture Dataset are used to train the two-stream 3D-CNNs for recognizing dynamic user gestures. The COCO dataset is a large-scale object detection, segmentation, and captioning dataset. COCO contains images of 91 object types, and objects in COCO are labeled using per-instance segmentations to aid precise object localization. The 20BN-JESTER Dataset V1 (as shown in Figure 2) is a recent video dataset for hand gesture recognition, which contains 27 types of predefined hand gestures performed in front of a camera. It has a total of 148,092 gesture samples extracted from the original videos at 12 frames per second. The samples are divided into three sets: 118,562 samples for training, 14,787 samples for validation, and 14,743 samples for testing without providing labels. The average video length is 35 frames. On the other hand, the Cambridge Hand Gesture Dataset consists of 900 image sequences of nine gesture classes, which are defined by three primitive hand shapes and three primitive motions (see Figure 3.3). Therefore, the target task for this dataset is to classify different shapes as well as different motions simultaneously.

Figure 2. 20BN-JESTER (Materzynska et al., 2019)

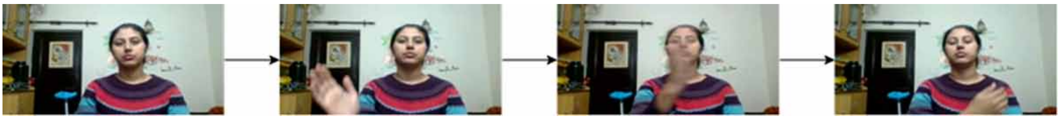


Figure 3. Cambridge Hand Gesture (Kim et al., 2007)

In this study, to improve the recognition performance of E3D, enough images in training datasets are necessary. In dynamic gesture recognition, the most discriminative part in a gesture image is the hand. The area of the region it occupies is relatively small compared to the entire image. As a result, the classifier is easily misguided by environmental variations and complex backgrounds in real-world scenes. Because E3D detects the starting image, it requires not only an image with a category label, but also the location of a gesture. However, in the public datasets, most of the gesture images have category labels but do not have the location of gestures. Therefore, a dataset with sufficient training gesture images was designed in this study. Algorithm 1 shows the pseudocode for generating the gesture images in the dataset. In Algorithm 1, the variables *backgrounds*, *gestures*, and *locations* represent an array of background images, an array of gesture images, and a category file including the widths, heights, and locations of the gesture object frames in the images, respectively. Line 1 initializes a loop for an array of background images. Line 2 sets the number of gesture images needed in the dataset. Lines 3–4 calculate the width and height of the gesture object frame according to the maximum and minimum values in the gesture image category file. Lines 5–7 place the gesture image on the background image through the values of the width and height of the object frame. Lines 8–12 use the function *iou* to determine whether the gesture images overlap or not. Lines 14–16 record the coordinates of the gesture object frame to produce many gesture images in the dataset.

Algorithm 1 Create Images

```
backgrounds ¬ the array of the background images
gestures ¬ the array of the gesture images
locations ¬ the category file including the locations of the gesture in the image

for i=0, 1, …., backgrounds do
for j=0,1,…….10 do
width ¬ (location_xmax[j] – location_xmin[j]) * scale
height ¬ (location_ymax[j] – location_ymin[j]) * scale
x ¬ random(0, backgrounds_width[i] – width)
y ¬ random(0, backgrounds_height[i] – height)
draw ¬ true
for b=0, 1,…., boxes do
u ¬ iou(x, y, width, height, boxes[b])
if iou >0 then draw ¬ false
end if
end for
if draw == true then
boxes ¬ (x, y, width, height)
gesture_img ¬ gestures[j]_ymin:ymax,xmin:xmax
gesture_img ¬ resize(gesture_width * scale, gesture_height * scale)
if gesture_RGB < 200 then backgrounds[i] ¬ gesture_img
end if
end if
end for
export(backgrounds[i], boxes)
shuffle(gestures)
end for
```

For example, in Figure 3.4, a gesture image is captured in front of a pure white wall. Then, the location of the gesture in the image is recorded, and the gesture object frame is produced. Furthermore, to obtain a better gesture dataset, the image is rotated and zoomed to produce more gesture images. In Figure 3.5, the gesture image is rotated, and the size of its object frame is corrected. Then, a new gesture image is produced and the value (i.e., x_{new}) of its x coordinate is set according to Eq (3.1). The value (i.e., y_{new}) of its y coordinate is set according to Eq (3.2). In Eq. (3.1) and Eq. (3.2), the values x_{old} and y_{old} are the original values of the x and y coordinates of the gesture in the original image. On the other hand, in Figure 3.5, when the image is zoomed, the gesture object frame is zoomed based on the original ratio of the image. By rotating and zooming the original image, a number of new gesture images are produced for our custom dataset. In addition, the complex backgrounds are appended to the images to generate additional new gesture images. The number of 1,200 pictures with 230 backgrounds are used to generate the static gesture images. Therefore, 11,353 images that contain two to four gestures are produced in the training dataset.

$$x_{new} = x_{old} * cos\theta - y_{old} * sin\theta \tag{3.1}$$

$$y_{new} = x_{old} * sin\theta + y_{old} * cos\theta \tag{3.2}$$

Figure 4. Process of producing the new gesture images

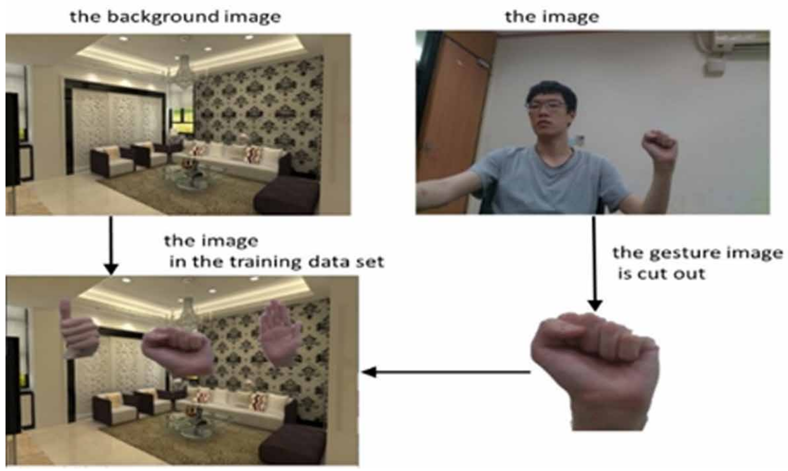
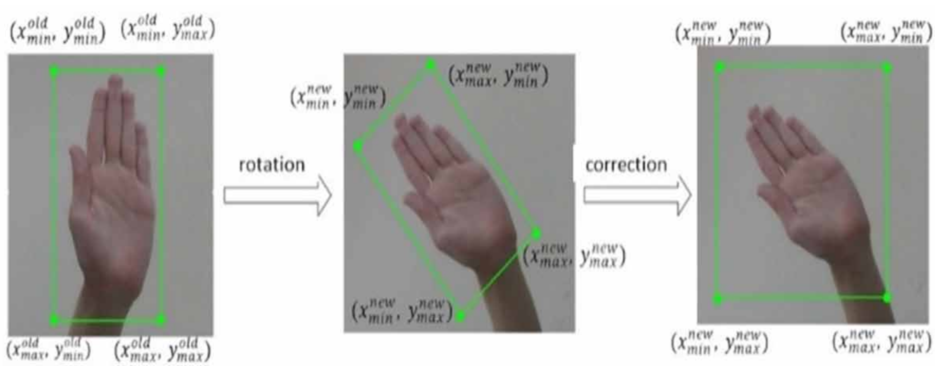


Figure 5. Rotation and zoom for the original gesture image



3.3 E3D Training Process

In E3D, to identify the start image frame in the real-time application on the end device, the object detection method (i.e., Tiny-YOLO) is trained by inputting static gesture images from the COCO dataset (Patterson & Hays, 2016) and our designed dataset including 11,353 images containing two to four gestures. In contrast, the two-stream 3D-CNN is trained by inputting dynamic gesture images from the 20BN-JESTER dataset (Materzynska et al., 2019) and the Cambridge Hand Gesture Dataset (Kim et al., 2007). Prior to the training process of Tiny-YOLO in E3D, the k-means clustering method is used to compute the sizes of the anchor box in the training images in the COCO dataset. This is because in COCO, the anchor boxes are not labeled on the images and these anchor boxes are need to be used in Tiny-YOLO. Also, in Tiny-YOLO, the values of RGB in an image must be normalized between 0 to 1. Therefore, according to Eq. (3.3), the static gesture images are normalized to train Tiny-YOLO to identify the start image in the real-time application on the end device.

$$image_{RGB} = \frac{image_{RGB}}{255} \tag{3.3}$$

Our E3D is based on a two-stream 3D-CNN, which is divided into high-resolution and low-resolution neural networks, as shown in Figure 3.1. In E3D, multi-layer 3×3 convolutional layers are used instead of the single-layer in 3D-CNN models, and the residual method is used for the convolution operation. In the training process of E3D, the sizes of images are adjusted to be input into the neural networks, and the image frames are set according to Eq. (3.4). In the high-resolution neural network, the original size of the image is input, while in the low-resolution neural network, the size of the image is reduced and then input. After the training process, the values from the high-resolution and low-resolution neural networks are obtained. Finally, these values are multiplied to obtain the results.

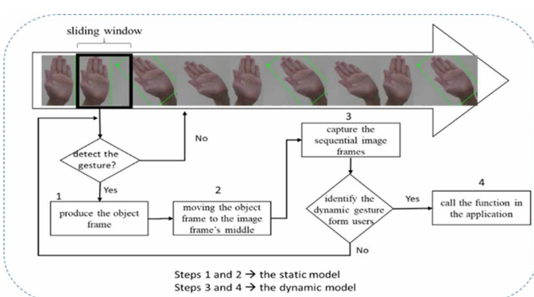
$$image_{frame} = \frac{frame * data_{frame\_sum}}{network_{depth}} \tag{3.4}$$

3.4 E3D Recognition Process

From the model repository shown in Figure 3.1, the trained recognition model can be deployed into the real-time application on an end device (i.e., a Raspberry Pi). The trained model includes the static model to identify the start image and the dynamic model to recognize dynamic gestures from users. In the recognition process of E3D, the image frames are captured by a camera, in which an event is spontaneously detected. This detection stimulates the frame capture module, which is executed for a specific short duration. For example, in Figure 3.4, the image frames are captured by the camera and input into the static model using a sliding window. In the static model, there are two steps (Steps 1 and 2). Step 1 detects the starting image by using Tiny-YOLO to produce the gesture object frame, while Step 2 moves the gesture object frame to the center position of the image and then stores the image in the classification queue for the dynamic model. In the dynamic model, a two-stream 3D-CNN with a residual method is applied to all frames containing hand objects. Algorithm 2 classifies the dynamic gestures from these frames to call the related function in the real-time application on the end devices.

In Algorithm 2, the variable *frame* represents the image obtained from the camera, and the variable *state* represents the three states (i.e., **passive**, **detect**, and **active**). In the **passive** state, there is no gesture image frame detected in the system. In the **detect** state, a gesture image frame is detected by the system, and in the **active** state, the image frames are stacked to input into the neural network. Line 1 defines each image frame obtained from the camera. Lines 2–3 detect the gesture image frame and change the **passive** state into the **detect** state. Lines 4–13 move the gesture object frame to the center position of the image. Lines 14–17 detect that the gesture object frame disappears in the image and place the image obtained from the camera into the image queue (namely *frame_window*). Lines 18–20 input the images of *frame_window* to the 3D-CNN model, while Lines 22–24 output the prediction results and then restart to recognize a new gesture image.

Figure 6. E3D recognition process



Algorithm 2 real-time gesture recognition

input: gesture image frames from camera
output: call the corresponding function

for each $frame_i$ do
if a gesture frame is detected and *state* == “*passive*” then
state ¬ “*detect*”
if $frame_{center\_x} > boxes_{center\_x}$ then
x ¬ $frame_{center\_x}$ - $boxes_{xmax}$;
else
x ¬ $frame_{center\_x}$ - $boxes_{xmin}$
end if
if $frame_{center\_y} > boxes_{center\_y}$ then
y ¬ $frame_{center\_y}$ - $boxes_{ymax}$
else
y ¬ $frame_{center\_y}$ - $boxes_{ymin}$
end if
else if a gesture frame is not detected and *state* == “*detect*” then
image ¬ wrapAffine(*frame,* (*x, y*))
frame_window ¬ *image*
end if
if *frame_window* == 3DCNN_input.shape then
state ¬ “*active*”
end if
end for
if *state* == “*active*” then
output = 3DCNN.predict(*frame_window*)
state ¬ “*passive*”
end if

For example, in Figure 3.5(a), because there is no gesture detected, the system captures the next image frame using the sliding window. Then, in Figure 3.5(b), a gesture is detected, and the object frame is produced. In addition, the object frame is moved to the center of the image frame, as shown in Figure 3.5(c). Consequently, in Figure 3.5(d), the sequence image frames are captured until the gesture is found in the position opposite to the start image frame. Finally, because there is no gesture detected in the image frame, as shown in Figure 3.5(e), the start image and the sequence image frames are stored in the classification queue.

3.5 ModelOps of E3D

In this study, ModelOps was designed and derived from DevOps (Jabbari et al., 2016). Although DevOps focuses on software integration testing and deployment, it does not consider automatic deployment of model training and verification. ModelOps can automatically deploy the trained and tested E3D into applications by extending the continuous integration (CI) and continuous delivery (CD) processes developed in DevOps. To implement the process of ModelOps, the Kubernetes platform was built using the Pod and Deployment components.

Figure 7. Example of detection and recognition process of E3D

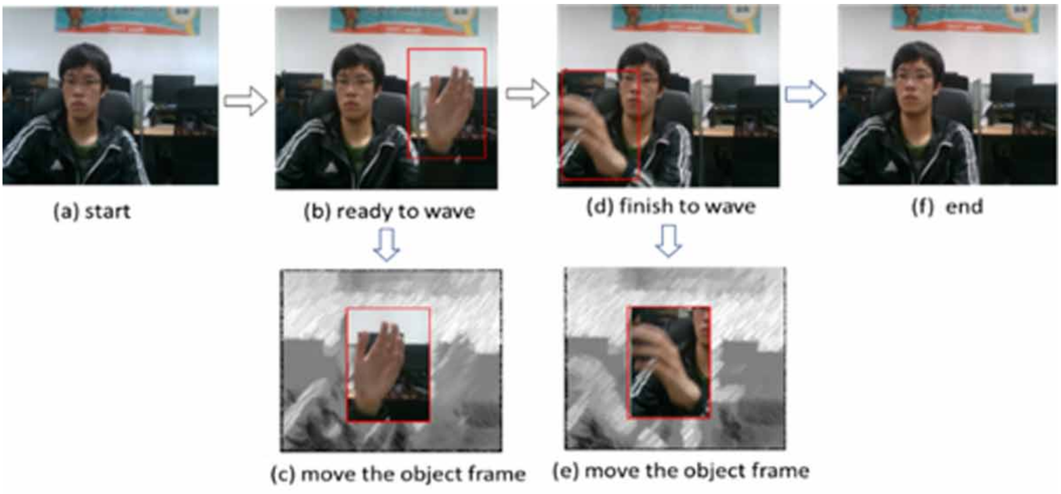
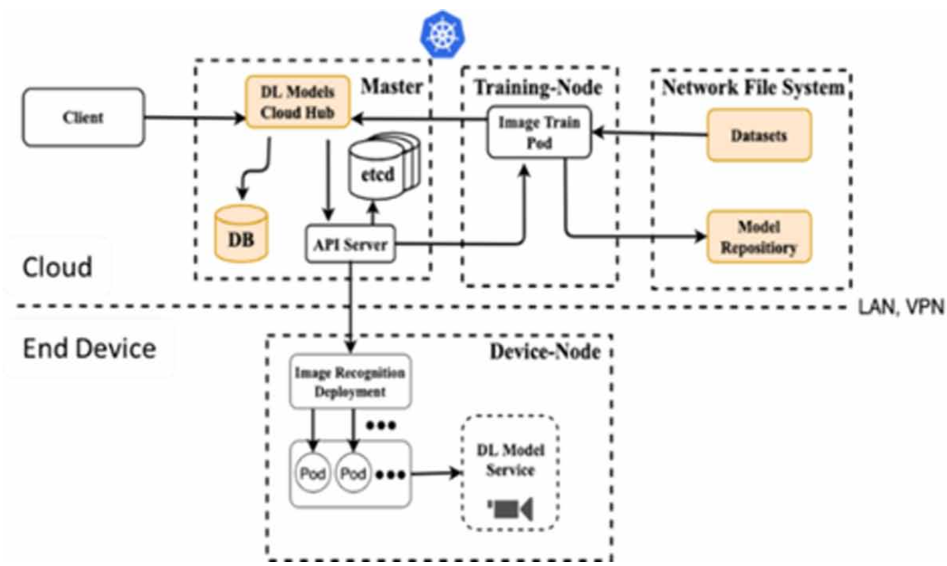
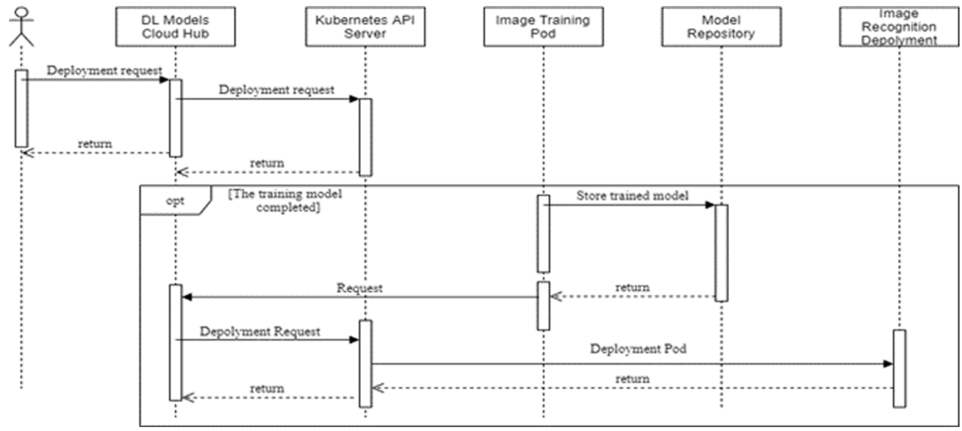


Figure 8. System components in the Kubernetes platform



As shown in Figure 3.6, there are three types of nodes (i.e., Master, Training-Node, and Device-Node) in our Kubernetes platform. Master and Training-Node are set as a cloud system, Device-Node is set on end devices, and these nodes are connected and communicate through a VPN. In our Kubernetes platform, a user can send a request to the API Server through DL Model Cloud Hub. A Pod with the training model is deployed to the Training-Node with access to a GPU and trained on the datasets in the network file system. Then, the trained model is stored in the Model Repository and deployed on the Device-Node.

Figure 9. Development process of ModelOps



The model deployment process of ModelOps is shown in Figure 3.7. The user sends a deployment model training request to DL Models Cloud Hub, and this request is deployed to the Image Training Pod via the Kubernetes API Server. After training, the trained model is verified and stored in the Model Repository. Finally, the Kubernetes API Server deploys the model into Image Recognition Deployment. In addition, ModelOps provides users the ability to query the deployment process of E3D.

Algorithm 3 shows the pseudocode for model verification in ModelOps. In Algorithm 3, the variable $train_{image}$ represents the image in the training dataset. Variable $test_{image}$ represents the image in the testing dataset. Variable *label* represents the label of the image, variable *threshold* represents the definition standard during the testing process, and variable *send* represents whether the verification is successful. Line 1 acquires the corresponding gesture model, Line 2 preprocesses the image in the training dataset, Line 3 defines the loss function used in the study, and Line 4 chooses the gradient descent method to reduce the values of the loss function. Lines 5–7 train the model by setting the number of iterations. Lines 8–10 predict the results of the testing images. Lines 11–15 use the optimizer in the OpenVINO tool to transform and save the verified model into an intermediate representation (IR) file.

Algorithm 3 model evaluation($train_{image}$, $test_{image}$, *label*, *threshold*, *send*)

input: the trained model
output: the verified model

1: *model* ¬ gestureModel()
2: *img* ¬ preprocess($train_{image}$)
3: *loss* ¬ $model_{loss}$($model_{output}$, *label*)
4: *optimizer* ¬ ADAM(*minimize*($model_{loss}$))
5: for *i*=0, 1, ……, *iterations* do
6: *model.run*(*optimizer*, *image*)
7: end for
8: *output* ¬ *model.predict*($test_{image}$)
9: if *output* > *threshold* then *send* ¬ *True*
10: end if
11: if *send*==*True* then
12: *IR* ¬ $OpenVINO_{\text{optimizer}}$(*model*)
13: *save*(*IR*)
14: *sendRestFulAPI*()
15: end if

4. IMPLEMENTATION AND EXPERIMENTS

This section describes the implementation of the components of the Kubernetes platform to automatically deploy the trained E3D model to end devices. In addition, the experiments are designed to analyze the gesture recognition performance of E3D. Section 4.1 presents the implementation details of E3D, Section 4.2 evaluates the performance of E3D, and Section 4.3 discusses and analyzes the experimental results.

4.1 Implementation

There are three units (i.e., the cloud platform, end devices, and external acceleration device) in our implementation environment. The cloud platform includes master and training nodes. The end devices tested included a Raspberry Pi 4 and a Jetson Nano, and the external acceleration device was an Intel NCS2. The specifications of each unit are listed in Table 4.1.

Table 1. Specifications of experiment unit devices

devices	CPU	RAM	OS	GPU
Master node	8 Core Intel® Xeon® CPU E3-1230 v3 @ 3.30GHz	8GB	Ubuntu 16.04	none
Training node	32 Core Intel® Xeon® Silver 4110 CPU @ 2.10GHz	16GB	Ubuntu 16.04	NVIDIA RTX 2080-ti
Raspberry pi 4	4 Core quad-core ARM Cortex-A53 @1.5GHz	4GB	Raspbian 10	none
Jetson Nano	4 Core quad-core ARM A57@1.43GHz	4GB	Ubuntu 18.04	128-core NVIDIA Maxwell GPU
Intel NCS2	16 Core	4 Gbit LPDDR4	VPU: Movidius Myriad X 4GB	

4.2 Experimental Results and Analyses

In the training process, the learning rate is an important parameter for ensuring the higher accuracy of E3D. If the initial learning rate is too big, the diverge will be happen in the training process of E3D; otherwise, the time cost will increase. Through the method of learning rate decay, the model can quickly reach the local or global maximum in the early stage of training. Using a smaller learning rate may achieve convergence in the later stage of training

In this section, we evaluate the performance of our proposed E3D model. Six experiments were conducted to evaluate the performance of E3D. The first experiment compared the performance of dynamic hand recognition using different datasets (i.e., 20BN-JESTER Dataset V1 and Cambridge Hand Gesture Dataset), optimizers (i.e., SGD and Adam), and learning rates (i.e., **fixed**, **decay**, and **cycle**). Here, **fixed** means that the value of the learning rate is fixed in each iteration. **decay** implies that the value of the learning rate is reduced after several iterations. With smaller value of learning rate, the model can reach the local or global maximum in the early stage of training. And **cycle** sets the value of learning rate between the maximum and minimum values. In our experiments, the value for **fixed** was set at 0.0001, the value for **decay** was set from 0.001 to 0.0001, and the value for **cycle** was set between 0.001 and 0.0001. Tables 4.2 and 4.3 present the results of the first experiment for the datasets **20BN-JESTER Dataset V1** and **Cambridge Hand Gesture Dataset**, respectively. In Table 4.2, **cycle** exhibits higher accuracy with the SGD and Adam optimizers. In addition, the accuracy of Adam exceeds that of SGD. However, Adam cannot converge under the learning rate **decay**.

Table 2. Accuracy of E3D with different optimizers and learning rates

Learning rates	SGD	Adam
fixed	39.30%	91.59%
decay	73.17%	9.93%
cycle	75.59%	92.08%

Figures 4.1 and 4.2 depict the values of the loss function (y-axis) for different numbers of iterations (x-axis) in SGD and Adam, respectively. In Figure 4.1, the convergence speeds of **cycle** and **decay** are similar. Moreover, the accuracy of **cycle** is higher than that of **decay**. In Figure 4.2, the convergence speeds of **cycle** and **fixed** are similar. In addition, the accuracy of **cycle** is higher than those of **fixed** and **decay**.

Figure 10. Loss function for SGD using 20BN-JESTER Dataset V1

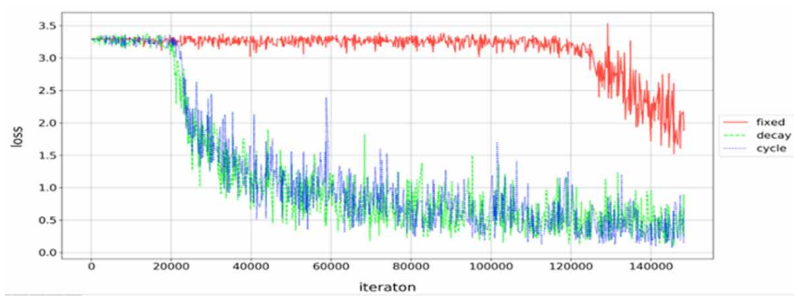
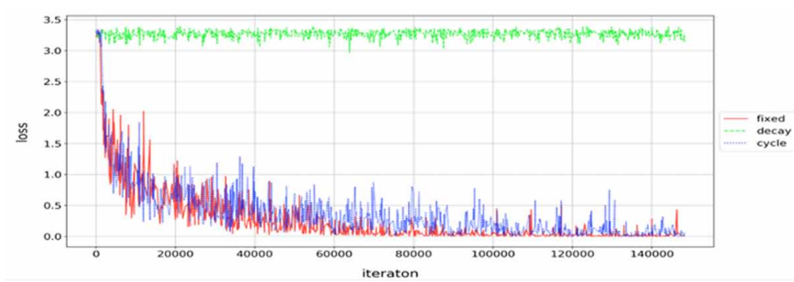


Figure 11. Loss function of Adam using 20BN-JESTER Dataset V1



In Table 4.3, **cycle** has higher accuracy in SGD, whereas **decay** has higher accuracy in Adam. In Figure 4.4, the convergence speed of **cycle** is better than those of **decay** and **fixed**. Moreover, the accuracy of **decay** is higher than that of **fixed**. In Figure 4.3, the convergence speeds of **cycle**, **fixed**, and **decay** are similar. Thus, with the learning rate **cycle**, the results of the first experiment indicate that a higher accuracy is obtained from Adam. Also, the loss function of Adam declines faster than that of SGD.

From the results of the first experiment, in dataset **20BN-jester Dataset V1**, the combination of Adam and cycle has the higher accuracy and that of Adam and decay cannot be converge. However, in dataset **Cambridge Hand Gesture**, the combination of Adam and decay has the higher accuracy. Therefore, the combination of Adam and cycle is chosen for the second experiment.

Table 3. Accuracy of E3D with different optimizers and learning rates

Learning rates	SGD	Adam
fixed	39.30%	96.67%
decay	73.17%	98.33%
cycle	75.59%	97.22%

Figure 12. Loss function for SGD using Cambridge Hand Gesture Dataset

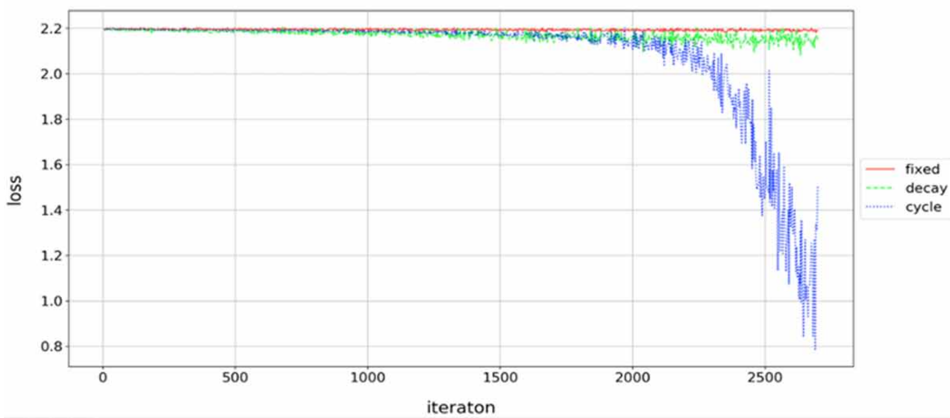
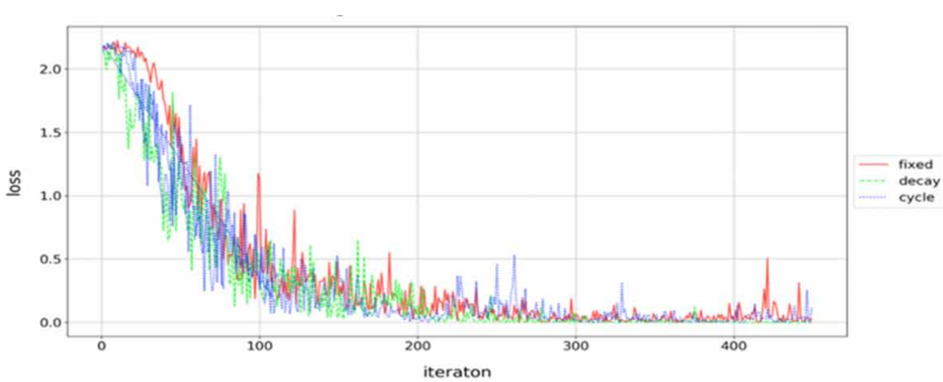


Figure 13. Loss function of Adam using Cambridge Hand Gesture Dataset



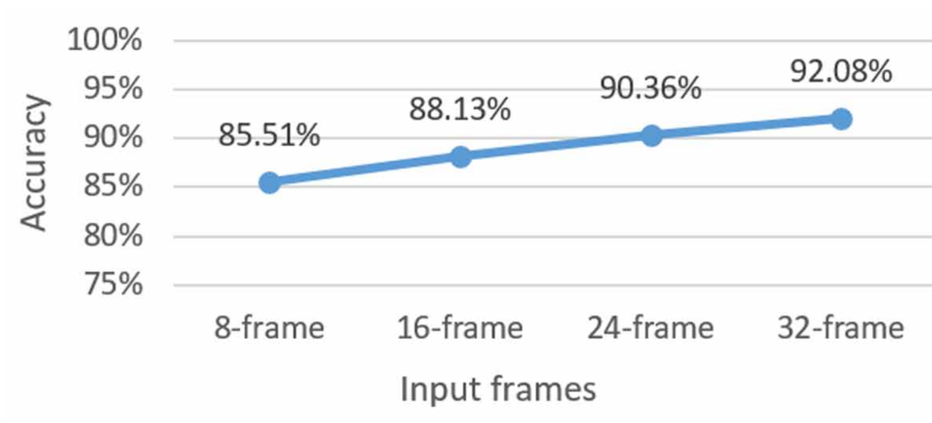
In the second experiment, the neural networks (i.e., ResNeXt and two-stream CNNs) used in E3D were compared in terms of their accuracy of dynamic hand recognition in the dataset **20BN-JESTER Dataset V1**. From Table 4.4, compared with the accuracy of two-stream CNNs, the accuracy of E3D is increased by 8.42% (8.42% = 92.08% - 83.66%). In addition, compared with ResNeXt including a high-resolution neural network, the accuracy of E3D is increased by 1.62% (1.62% = 92.08% - 90.46%), and compared with ResNeXt including a low-resolution neural network, the accuracy of E3D is increased by 2.11% (2.11% = 92.08% - 89.97). According to the results of second experiment, the two-stream CNNs and ResNeXt can be used in E3D to improve the accuracy.

Table 4. Accuracy of various models

Neural networks	Accuracy
Two-stream CNNs	83.66%
ResNeXt within high-resolution neural network	90.46%
ResNeXt within low-resolution neural network	89.97%
E3D (Two-stream CNN + ResNeXt)	92.08%

In the third experiment, the accuracy was compared between different numbers of input image frames in E3D for the dataset **20BN-JESTER Dataset V1** and the optimizer Adam with **cycle.** Figure 4.5 shows the accuracy of E3D by inputting 8, 16, 24, and 32 frames. As the number of image frames increases, the accuracy of E3D increases. The accuracy of E3D obtained by inputting 16 frames is better than that of inputting eight frames. The accuracy of E3D inputting 24 frames is better than that of inputting 16 frames. Similarly, the accuracy of E3D inputting 32 frames is better than that of inputting 24 frames. Therefore, the number of 32 frames has the better accuracy of E3D. On the other hand, Table 4.5 indicates that the training time increases as the number of frames increases. The training time for 16 frames is 143% longer than that of 8 frames, whereas that of 24 frames is 31% longer than that of 16 frames. In addition, the training time of 32 frames is 46% longer than that of 24 frames. According to the results of the third experiment, the number of 32 frames is chosen as input in the fourth experiment.

Figure 14. Accuracy of E3D by inputting different numbers of image frames



In the fourth experiment, the recognition times of E3D with Tiny-YOLOv2 and Tiny-YOLOv3 for different end devices (i.e., Raspberry Pi 4 and Jetson Nano) were compared.

Table 5. Accuracy and training time of E3D by inputting different numbers of frames

input frames	accuracy	training time
8-Frame	85.51%	22 hours 53 minutes
16-Frame	88.13%	2 days 7 hours 45 minutes
24-Frame	90.36%	3 days 1hours 24 minutes
32-Frame	92.08%	4 days 11hours 35 minutes

Figures 4.6 and 4.7 show the inference time and the frames per second for different end devices. The inference time indicates the time used for hand gesture recognition in E3D, and the frames per second is the number of image frames that are processed in E3D per second. Using an Intel NCS2 in a Raspberry Pi 4 in E3D can greatly reduce the inference time and increase the number of frames per second. By using Tiny-YOLOv2 in E3D, the inference time is reduced by 87.22%, while using Tiny-YOLOv3 in E3D, it is reduced by 84.14%. Consequently, the inference time using the Jetson Nano with an Intel NCS2 for Tiny-YOLOv2 and Tiny-YOLOv3 in E3D can be reduced by 73.57% and 63.33%, respectively. In general, the inference time using Tiny-YOLOv2 is shorter than that using Tiny-YOLOv3. Furthermore, the inference time using the GPU in E3D is shorter than that using Intel NCS2 in E3D. Detailed information is presented in Table 4.6. According to the experimental results, Tiny-YOLOv3 is more effective for detecting small objects than Tiny-YOLOv2. However, the inference time of Tiny-YOLOv3 was greater than that of Tiny-YOLOv2. Therefore, according to the results in the experiment, Tiny-YOLOv2 with Intel NCS2 was used to detect the static gestures and Jetson Nano was used in end devices to recognize the dynamic gestures.

Figure 15. Frames per second on different end devices

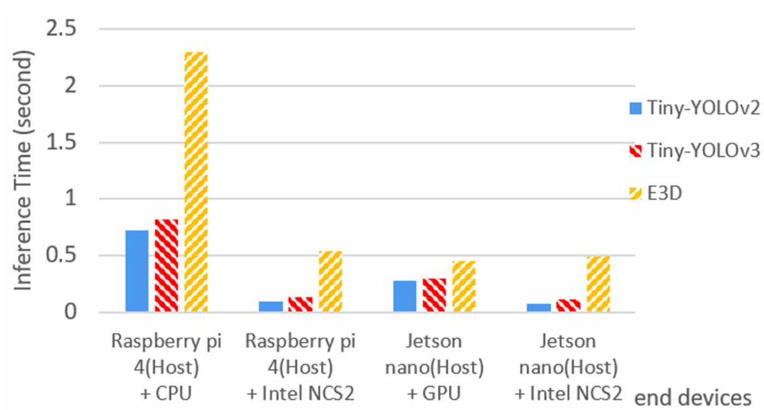
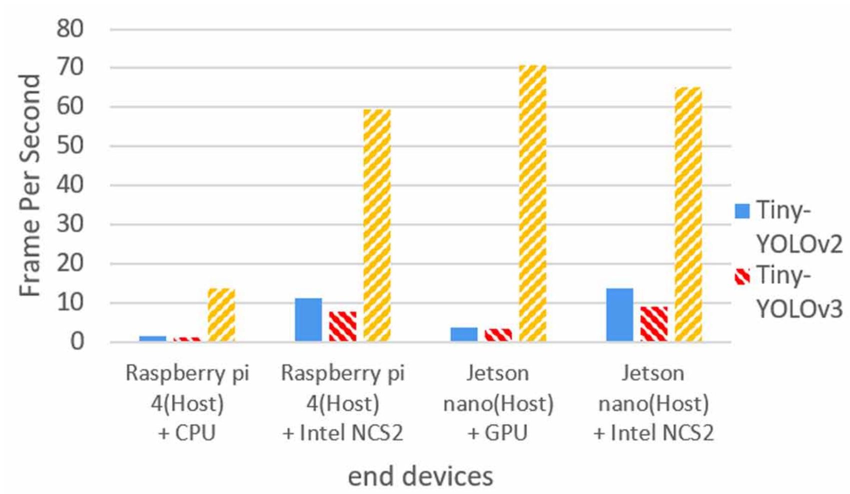


Table 6. Recognition time of E3D for different end devices

End devices	Tiny-YOLOv2	Tiny-YOLOv3	E3D
Raspberry pi 4(Host) + CPU	1.37fps, 0.72s	1.21fps, 0.82s	13.76fps, 2.3s
Raspberry pi 4(Host) + Intel NCS2	11.24fps, 0.09s	7.73fps, 0.13s	59.52fps, 0.54s
Jetson nano(Host) + GPU	3.67fps, 0.28s	3.33fps, 0.3s	70.72fps, 0.45s
Jetson nano(Host) + Intel NCS2	13.51fps, 0.07s	9.09fps, 0.11s	64.96fps, 0.49s

Figure 16. Frames per second on different end devices



In the fifth experiment, the inference time, accuracy, loading time, and number of inference frames of LRCN, C3D, 3D-ResNeXt, and E3D were compared for the dataset **20BN-JESTER Dataset V1** on Raspberry Pi 4 and Jetson Nano with the GPU and NCS. As shown in Table 4.7, the accuracy of E3D is higher than that of LRCN and lower than those of C3D and ResNeXt 101. In addition, the loading time and the number of inference frames of E3D with the GPU are less than those of ResNeXt 101 and C3D. Furthermore, C3D cannot be loaded on the end device with Intel NCS2 because C3D requires too many parameters and thus results in insufficient memory. According to the results of the fifth experiment, although the accuracy of some models is higher than that in E3D, these models take a lot of time in their inference processes.

Table 7. Accuracy and inference time of different models on different end devices

Model	accuracy	On-board Process Delay Time (GPU)	Inference time with GPU	On-board Process Delay Time (NCS)	Inference time with NCS2
LRCN	76.66%	365.62s	54.4fps,0.59s	19.06s	6.4fps, 4.8s
C3D	92.78%	385.44s	21.44fps,1.51s	N/A	N/A
3D-ResNeXt 101	94.89%	227.78s	26.24fps,1.22s	2028.67s	17.28fps,1.84s
E3D	92.08%	24.15s	70.72fps,0.45s	97.57s	64.96fps,0.49s

In the sixth experiment, five types of gestures (i.e., swiping left, swiping right, swiping down, swiping up, and other gestures) from users (as shown in Figures 4.8 and 4.9) were used to test the accuracy and inference time of E3D.

Figure 17. Swiping left gesture including swiping down and up

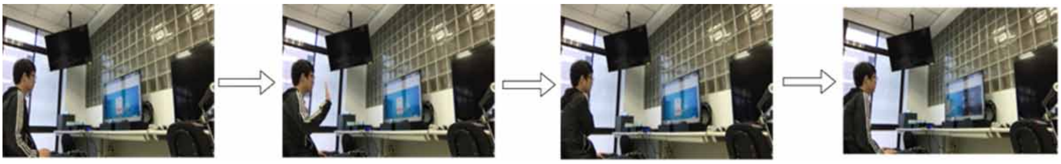
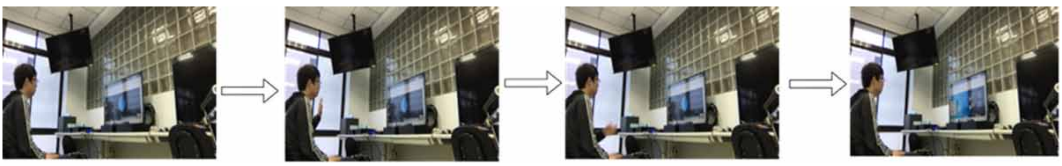


Figure 18. Swiping right gesture including swiping down and up



In this experiment, each user performed a gesture and repeated it five times to generate testing data to compare the recognition accuracy of E3D, E3D with Tiny-YOLOv2, and E3D with Tiny-YOLOv2 and the shift technique (i.e., the gesture object frame in an image can be shifted to the center position of the image). As shown in Figure 4.10, the recognition accuracy of E3D is 65.6%, that of E3D with Tiny-YOLOv2 is 82%, and that of E3D with Tiny-YOLOv2 and the shift technique is 88%. Without Tiny-YOLOv2, the gesture image frames are continuously recognized by E3D, and the complete gesture action cannot be detected. Therefore, the recognition accuracy of E3D cannot be accepted by users. In contrast, using Tiny-YOLOv2 to detect the start image frame can enhance the recognition accuracy of E3D because E3D can focus on identifying a complete gesture action produced by a series of continuous gesture images. In addition, in the object detection process of Tiny-YOLOv2, the shift technology can be used to improve the recognition accuracy of dynamic gestures. This is because most users do not perform their gestures in the center of the images (as shown in Figure 4.11). According to the results in this experiment, E3D with Tiny-YOLOv2 is suitable for user to identify the gestures of users.

Figure 19. Recognition accuracy between different types of gestures

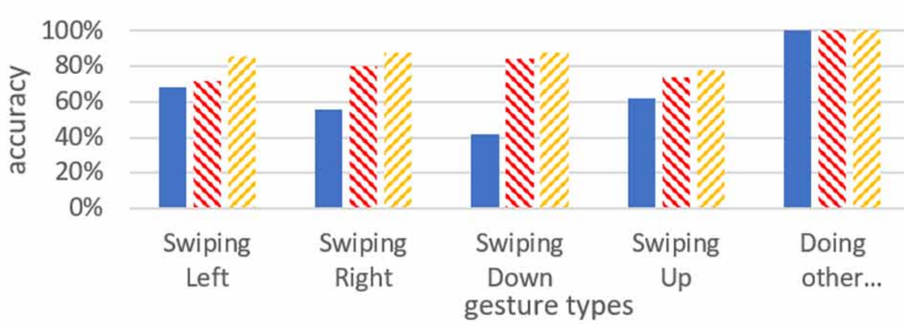


Figure 20. Steps of swiping left gesture

5. CONCLUSION

In this work, a deep learning model, E3D, is proposed for dynamic gesture recognition in real-time applications on end devices. In E3D, an object detection method (i.e., Tiny-YOLOv3) was used in the static model to focus on the parts of the image frame sequence relevant to gesture discrimination in both spatial and temporal dimensions. Then, a 3D-CNN with a two-stream CNN and a residual method were designed in the dynamic model for recognizing dynamic gestures from users. Finally, based on a Kubernetes container, the model of E3D was implemented, and the ModelOps model was designed for automatically deploying and updating the trained E3D model on end devices. We also conducted six experiments to confirm the effectiveness of E3D. From these experiments, Adam with the learning rate **cyclic** is suitable used in E3D. Also, the residual and dual-stream methods can improve the accuracy of dynamic gesture recognition in real-time applications on end devices. Furthermore, in the static gesture recognition, the speed of Tiny-Yolov2 is faster than that of Tiny-Yolov3. Our E3D provides a solution for recognizing the user gestures in the real-time systems. In E3D, the training process should be completed in a cloud server and the inference process can be automatic installation and updated for end devices. However, E3D still has the problems of insufficient training datasets and need to have the end devices with high computing capabilities. Future research can focus on modeling the temporal relations between frames more effectively and reducing the parameters in the 3D-CNN without a decline in performance. Adding version control to ModelOps is also worth investigating.

REFERENCES

Costante, G., Porzi, L., Lanz, O., Valigi, P., & Ricci, E. (2014). Personalizing a smartwatch-based gesture interface with transfer learning. *2014 22nd European Signal Processing Conference (EUSIPCO).*

Dhingra, N., & Kunz, A. (2019). Res3atn-deep 3D residual attention network for hand gesture recognition in videos. *2019 International Conference on 3D Vision (3DV).*

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. IEEE Software, 33(3), 94–100.

Gao, J., Ling, H., Hu, W., & Xing, J. (2014). Transfer learning based visual tracking with gaussian processes regression. *European Conference on Computer Vision*.

Gorbachev, Y., Fedorov, M., Slavutin, I., Tugarev, A., Fatekhov, M., & Tarkan, Y. (2019). OpenVINO deep learning workbench: Comprehensive analysis and tuning of neural networks inference. *Proceedings of the IEEE International Conference on Computer Vision Workshops*.

Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes: Up and running: Dive into the future of infrastructure*. O'Reilly Media, Inc. doi:10.1007/978-3-319-10578-9_13

Hitawala, S. (2018). *Evaluating ResNeXt Model Architecture for Image Classification.* arXiv preprint arXiv:1805.08700.

Huang, Z., Zhong, Z., Sun, L., & Huo, Q. (2019). Mask R-CNN with pyramid attention network for scene text detection. *2019 IEEE Winter Conference on Applications of Computer Vision (WACV).*

Jabbari, R., bin Ali, N., Petersen, K., & Tanveer, B. (2016). What is DevOps? A systematic mapping study on definitions and practices. *Proceedings of the Scientific Workshop Proceedings of XP2016.*

Kim, T.-K., Wong, S.-F., & Cipolla, R. (2007). Tensor canonical correlation analysis for action classification. *2007 IEEE Conference on Computer Vision and Pattern Recognition*.

Kim, Y., & Toomajian, B. (2016). Hand gesture recognition using micro-Doppler signatures with convolutional neural network. IEEE Access: Practical Innovations, Open Solutions, 4, 7125–7130.

Kingma, D. P., & Ba, J. (2014). *Adam: A method for stochastic optimization.* arXiv preprint arXiv:1412.6980.

Li, C., Wu, X., Zhao, N., Cao, X., & Tang, J. (2018). Fusing two-stream convolutional neural networks for RGB-T object tracking. *Neurocomputing*, *281*, 78–85. doi:10.1016/j.neucom.2017.11.068

Martin, S., Yuen, K., & Trivedi, M. M. (2016). Vision for intelligent vehicles & applications (viva): Face detection and head pose challenge. *2016 IEEE Intelligent Vehicles Symposium (IV).*

Materzynska, J., Berger, G., Bax, I., & Memisevic, R. (2019). The Jester Dataset: A Large-Scale Video Dataset of Human Gestures. *Proceedings of the IEEE International Conference on Computer Vision Workshops.*

Nanni, L., Ghidoni, S., & Brahnam, S. (2017). Handcrafted vs. non-handcrafted features for computer vision classification. Pattern Recognition, 71, 158–172.

Patterson, G., & Hays, J. (2016). Coco attributes: Attributes for people, animals, and objects. *European Conference on Computer Vision*.

Pemgulys, J., Raudonis, V., Maskeliûnas, R., & Damaðevièius, R. (2018). Recognition of basketball referee signals from videos using Histogram of Oriented Gradients (HOG) and Support Vector Machine (SVM). Procedia Computer Science, 130, 953–960.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in Neural Information Processing Systems*.

Shamir, O. (2018). Are ResNets provably better than linear predictors? *Advances in Neural Information Processing Systems.*

Shin, S., & Sung, W. (2016). Dynamic hand gesture recognition for wearable devices with low complexity recurrent neural networks. *2016 IEEE International Symposium on Circuits and Systems (ISCAS).*

Tran, D., Bourdev, L., Fergus, R., Torresani, L., & Paluri, M. (2015). Learning Spatiotemporal Features with 3D Convolutional Networks. *2015 IEEE International Conference on Computer Vision (ICCV).*

Wang, G., Yuan, Y., Chen, X., Li, J., & Zhou, X. (2018). Learning discriminative features with multiple granularities for person re-identification. *Proceedings of the 26th ACM international conference on Multimedia.*

Wang, H., Ma, X., & Hao, Y. (2017). Electronic devices for human-machine interfaces. *Advanced Materials Interfaces, 4*(4), 1600709.

Wang, N., & Yeung, D.-Y. (2013). Learning a deep compact image representation for visual tracking. *Advances in Neural Information Processing Systems*.

Zhong, Z., & Deng, J. (2019). Real-Time Detection based on Modified YOLO for Herniated Intervertebral Discs. *Proceedings of the 2019 4th International Conference on Intelligent Information Processing*.

Zou, X., Zheng, Q.-Q., & Li, B. (2018). Static Hand Gesture Recognition Based on Kinect Sensor and HOG Features. *Software Guide,* (2), 9.

Tsui-Ping Chang received her Ph.D. degree in Computer Science and Engineering from National Chung-Hsing University, Taiwan, in 2009. She was the faculty member of the Department of Information Technology, Ling Tung University, Taiwan, since 2009. She has joined the Computer Science and Information Engineering, National Taichung University of Science and Technology, Taiwan, since 2019. Her research interests include database systems and deep learning.

Hung-Ming Chen is a Professor in the Department of Computer Science and Information Engineering of National Taichung University of Science and Technology. He received his PhD in the Department of Electrical Engineering from the National Taiwan University. His research interests focus on deep learning and cloud computing.

Shih-Ying Chen received his Ph.D. degree in Computer Science from National Chung Hsing University in 2006. Currently, he is the faculty of the Computer Science and Information Engineering of National Taichung Institute of Technology. His major research interests include XML databases and cloud computing technologies.