"HAND INVADERS" - AN INTERACTIVE SHOOT 'EM UP VIDEO GAME WITH OBJECT DETECTION AND CLASSIFICATION

Github: https://github.com/thesunsavior/Hand-Invader

Team member: Pham Quoc Trung, Lai Dac Tien, Nguyen Tiet Nguyen Khoi

Advisor: Professor Pham Huy Hieu

1. Introduction

In recent years, advancements in computer vision technology have propelled the development of object detection and classification systems. These systems enable machines to perceive and understand the world around them, opening up a myriad of applications across various domains. One such application is hand detection and classification, where the goal is to accurately identify and classify hand gestures or poses in real-time.

Several frameworks and libraries exist to facilitate hand detection and classification tasks. Among them, MediaPipe [1] stands out as a comprehensive toolkit that offers pre-trained models and API pipelines for various computer vision tasks, including hand detection, tracking, and pose estimation. Other frameworks, such as OpenCV and TensorFlow, also provide functionality for building custom hand detection and classification models. Hand detection and gesture classification provide not only an interactive gaming experience but also open up a wide range of applications, including simulating prototypes in fields like civil and mechanical engineering.

In line with the growing interest in interactive gaming experiences, we present our interactive gaming application, Hand Chicken Invader, enhanced with hand detection and hand pose classification capabilities. Chicken Invader is a classic arcade-style game where players control a spaceship to defend Earth against waves of invading chickens. Chicken Invader is a “shoot’ em up” game, where the main mechanism surrounds positioning and shooting. By integrating hand detection and classification, players can now navigate the game and perform actions like shooting and enable special power using hand gestures. With our custom-built system, players can control the spaceship's movements by moving their hands in specific directions, and perform actions such as firing lasers or activating power-ups by making predefined hand poses. Overall, our core targets and achievements for this project are:

- We offer a new level of interactivity for game players by combining computer vision with classic gaming experiences.
- We successfully deploy this interactive game with high detection and classification accuracy, as well as low latency.
- We provide an end-to-end reimplementation without using off-the-shelf APIs and frameworks, and have achieved acceptable results.

2. Problem Overview & Solution

2.1 Context and Objectives

As is well understood, in any real-time gaming scenario, it's crucial for the system to swiftly and accurately react to the user's inputs. In the case of Chicken Invaders, precise positioning of the spaceship is crucial to navigate through flocks of chickens and obstacles. Additionally, monitoring whether the weapon is firing to avoid overheating is also important. Consequently, our project is centered around two primary objectives: achieving high accuracy in hand detection and classification, and optimizing the system's frames per second (FPS) to ensure swift generation of control actions.

2.2 Overall Pipeline

This report presents a real-time on-device hand tracking and gesture classification pipeline for intuitive game control using a single RGB camera. The cornerstone of this pipeline lies in its three distinct yet interconnected modules:

1. **Hand Detector**: This module identifies the presence and location of hands within the camera frame. Its lightweight design ensures minimal processing overhead, crucial for maintaining real-time performance. The model locates the hand within a bounding box region. If a hand is found in the frame, a close-up bounding box region of the image is cropped and sent to the Hand Gesture Classification.

2. **Hand Gestures Classifier**: Once a hand is detected, this module classifies the hand gesture into a predefined set of gestures. This classification relies on applying robust deep CNNs, and achieving a high accuracy rate with low computing cost.

3. **Game Agent**: This component serves as the interpreter and action instigator. It receives the hand position and gesture label data from the previous modules and translates them into meaningful actions within the game using the pyautogui library. An open palm is translated to character movement, while a clenched fist triggers weapon fire.

To maximize performance and frame rate, the pipeline employs multi-threading. Each module operates on a separate thread, enabling parallel processing and minimizing latency. Communication between threads occurs through shared memory, ensuring efficient data exchange without compromising processing speed. This multithreaded architecture is particularly crucial for real-time gaming applications where responsiveness is paramount.

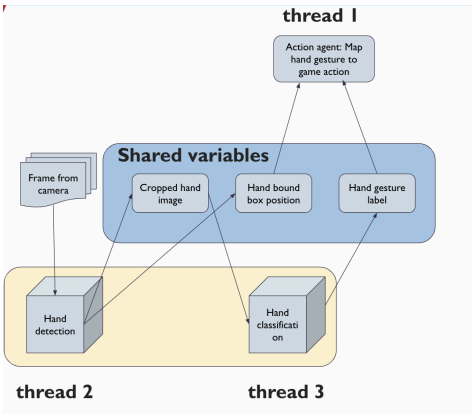

Fig 1. Pipeline architecture

Reasons for the two-staged pipeline: Our experiments revealed that training a single model for both hand detection and gesture classification, similar to classic architecture like R-CNN, on our limited dataset and resources compromised both accuracy and training efficiency. The combined models suffered from lower detection and classification accuracy rates, increased training time, and limitations in parallelism, leading to lower frame rates. To address these issues, we adopted a two-stage pipeline, adopting the similar approach from MediaPipe.

This pipeline separates detection and classification into distinct models. The hand detection model locates hands in the image, providing a tightly cropped region for the gesture classification model. This approach offers several advantages:

- **Improved accuracy**: Each model specializes in its task, potentially leading to higher individual accuracy in both detection and classification.
- **Faster training**: Training two smaller models is quicker and more efficient than training a single, complex model.
- **Parallel processing**: Separate models enable parallel processing on different threads, boosting frame rates for real-time applications.
- **Reduced data augmentation**: By providing a pre-cropped image, the need for complex data augmentation techniques (rotations, translations) is minimized, simplifying preprocessing and potentially improving training efficiency.
- **Scalability**: Adding new gestures becomes easier, as only the classification model needs to be expanded, leaving the detection model unchanged.

While introducing some additional complexity, the two-stage pipeline provides a solid foundation for a scalable and performant system suitable for real-time gaming experience.

2.3 Hand Detection

2.3.1 Introduction

In this part, we present a hand-detection model customized for real-time gameplay of Chicken Invader.

To address the challenge of real-time hand detection and tracking with high frame rates, which

are crucial for smooth gameplay, we introduce a custom-designed hand detection model based on a deep learning architecture. Leveraging a single-shot detection model with a MobileNet backbone ensures both accuracy and efficiency. This combination enables the model to rapidly locate and identify hands within the camera frame, paving the way for real-time game control via hand gestures.

We place particular emphasis on achieving high FPS suitable for gaming environments. This focus ensures minimal latency and responsiveness, guaranteeing a smooth and uninterrupted gaming experience where hand movements translate instantly into in-game actions.

2.3.2 Dataset and Preprocessing

We leverage the Egohands Dataset [2] for training our hand detection model due to its numerous advantages. It boasts over 15,000 high-quality, pixel-level annotations for hand locations across 4,800 images. Notably, these images capture multiple perspectives of hands across diverse environments (48 in total) and activities (playing cards, chess, puzzles, etc.)

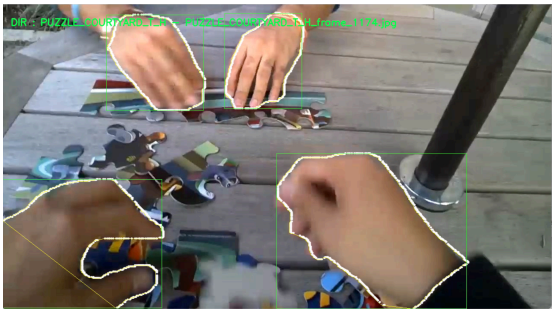

Fig 2. A data example of the Egohands dataset

However, the Egohands Dataset requires preprocessing to align with our training framework. We developed a custom Python script to perform the following crucial steps:

1. **Download and Organization**: The script automatically downloads the dataset and renames all files to incorporate their directory names, guaranteeing unique identification within the training process.
2. **Data Split**: We split the dataset into training (70%), validation(20%), and testing (10%) sets, enabling robust evaluation of the trained model's generalization ability.
3. **Bounding Box Generation**: For each image, the script extracts hand location information from the provided annotation files and generates the corresponding bounding boxes.
4. **Visualization and Verification**: To ensure accuracy, the script visualizes the generated bounding boxes overlaid on the images, allowing for manual inspection and correction if needed.
5. **TFRecord Conversion**: Finally, the script converts the preprocessed dataset (images and generated csv files) into the TFRecord format, which is optimized for efficient training with TensorFlow.

This streamlined preprocessing pipeline prepares the Egohands Dataset for seamless integration into our hand detection model training.

2.3.3 Single Shot Detection (SSD)

Object detection tasks often require real-time performance, especially in applications like autonomous vehicles or, in our case, real-time gaming experience. While models like Faster R-CNN [4] offer high accuracy, their two-head architecture with a region proposal network (RPN) limits their frame rate. This is where the Single-Shot MultiBox Detector (SSD) [5] comes in.

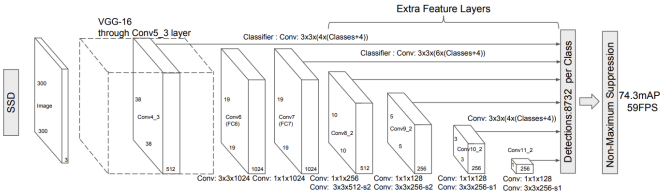

Fig 3. Original Single Shot MultiBox Detector with VGG-16 backbone

Designed for Efficiency: SSD sacrifices some accuracy for significant speed gains by eliminating the RPN. Instead, it predicts both object class and bounding box directly from feature maps extracted by a convolutional neural network (CNN) like VGG16 [3]. This "single-shot" approach significantly reduces computational cost, making SSD ideal for real-time applications.

Multi-Scale Features and Default Boxes: To compensate for the accuracy loss and handle objects of various sizes, SSD employs two key techniques:

- **Multi-scale features**: The CNN extracts features at different scales, allowing SSD to

detect objects regardless of their size in the image.

- **Default boxes**: Instead of predicting bounding boxes from scratch, SSD uses predefined "default boxes" anchored at different scales and locations within the image, based on the dataset ground truth. The ground truth boxes are clustered into groups of similarity, each group has a bounding box representing it. The model then refines these boxes to match the actual objects. This technique reduces complexity and improves convergence during training.

Fig 4. Multiple scales of a class object detection

Matching Predictions to Ground Truth: SSD predicts class probabilities and bounding box offsets relative to the default boxes. These predictions are then matched to ground truth annotations. Matches with high Intersection-over-Union (IoU) are considered positive and contribute to the training process.

SSD provides a compelling balance between accuracy and speed, making it a valuable tool for real-time object detection tasks. This is why we employ SSD as our detection architecture for hand detection.

2.3.4 Single Shot MultiBox Detector with MobileNet Backbone Approach

Our Hand detection architecture employs SSDLite architecture. SSDLite is an object detection model that aims to produce bounding boxes around objects in an image. We use MobileNet V2 [2] with a feature pyramid network for feature extraction to enable real-time object detection on low-end devices and quick inference response time. This is a fast and compact model, with an acceptable accuracy rate that suits our application. The model is pre-trained on the Pascal VOC 2012 [6] dataset. We finetuned the last 2 layers of MobileNet V2 backbone, Single Shot Detection model on hand detection with a single class of whether there is a hand in the image with 225 epochs.

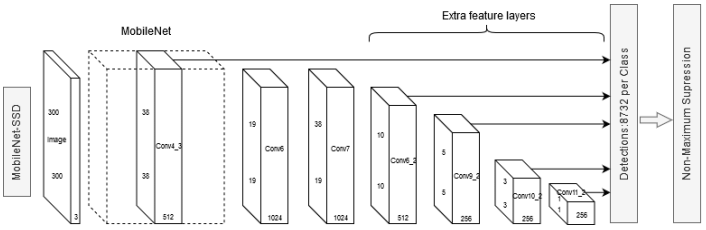

Fig 5. Single shot detection with MobileNet backbone

Training objective: As in the original paper, The model loss is a weighted sum between localization loss and confidence loss. The SSD training objective is derived from the MultiBox objective.

2.4 Hand Classification

2.4.1 Introduction

Following hand detection, the next critical step involves the precise prediction of the hand gestures as gameplay commands. From the output of the detection, the detected hand within each image frame is cropped based on the bounding box parameters, ensuring that only the relevant hand area is extracted for further analysis. Once cropped, this hand image is then fed into our fine-tuned MobileNet classification model. The output label is then used by PyAutoGUI to facilitate the gameplay.

2.4.2 Dataset and Preprocessing

We utilized the Roboflow Hand Gesture Dataset [10] for fine-tuning, which comprises 8,561 images across 20 hand gesture classes. Although the gameplay of Chicken Invader only requires 3 hand gestures for commanding, we still fine-tuned with all 20 classes, as we observed the results of the fine-tuning model was high even for multiple classes. We built a preprocessing pipeline as follows:

1. **Download and Organization**: The script automatically downloads the dataset from Roboflow website. The dataset was already divided into 3 subsets: train (80%), valid (10%), and test (10%), and organized in the COCO format.
2. **Bounding Box Generation**: As this dataset closely mirrors the output of our hand detection model, for each image, the script extracts hand location information from the provided annotation files and generates the corresponding bounding boxes.

3. **Hand Cropping Image**: The script then proceeds to crop the detected hands based on the bounding box coordinates. This step is crucial as it ensures that the images fed into our classification model are focused solely on the hand gestures, thereby improving the accuracy of the classification. The cropped data images are used for fine-tuning.
4. **Data Augmentation and Normalization**: To improve the robustness of our classification model against variations, the script applies random rotation and horizontal flipping to the training images, varying between 10 to 20 degrees. The image data are then normalized before fine-tuning for faster convergence.

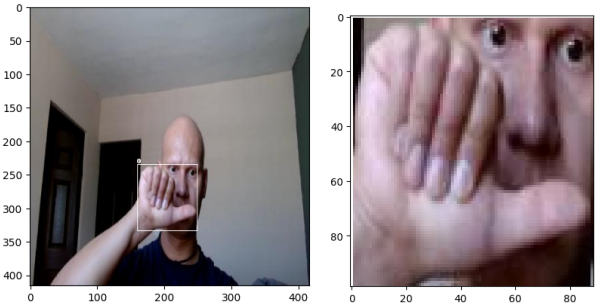

Fig 6. Raw sample from Roboflow Dataset (left) and cropped sample based on provided bounding box (right)

2.4.3. MobileNet for Classification

The MobileNet and its MobileNetV2 models are lightweight, efficient deep neural networks designed for mobile and embedded vision applications. Instead of using only traditional convolution, they utilize depthwise separable convolutions to significantly reduce the computing expense without compromising accuracy. This makes MobileNet an excellent choice for tasks that require high computational efficiency, such as classification with real-time response inference.

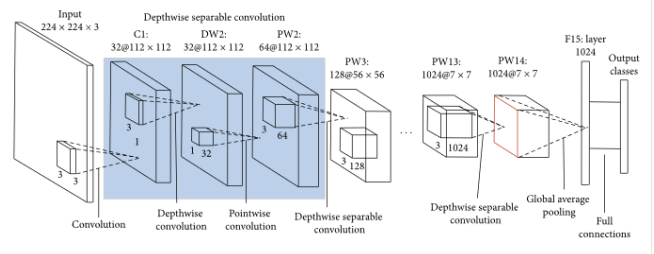

Fig 7. MobileNet architecture with depthwise separable convolution

For fine-tuning, we used a pre-trained version of MobileNetV2 on the ImageNet dataset, which includes over a million images across 1000 categories, providing a robust foundation for a wide range of vision-based tasks. In summary, we chose MobileNet as our hand classifier for several significant advantages:

- **Efficiency and adaptability to resource -constrained environments**: MobileNet's design focuses on minimizing computational resources and parameters, making it highly efficient for training and inference for real-time processing.
- **High-quality feature extraction**: Despite its compact size, MobileNet effectively extracts rich features due to its pre-training on the ImageNet dataset. This makes it excellent for transfer learning, achieving high accuracy with less data.
- **Versatility in application**: MobileNet's adaptable design makes it suitable for a range of computer vision tasks, beyond just classification, without significant changes to its architecture.

Model	Params (million)	Inference Speed	Top-1 Accuracy on ImageNet
MobileNetV2	3.4	Very fast	~71.8%
EfficientNet-B0	5.3	Fast	~77.1%
GoogLeNet	6.8	Fast	~69.8%
ResNet-50	25.6	Fast	~76.2%
VGG16	138	Slow	~71.5%

Table I. A brief comparison of MobileNet to some other vision architectures including EfficientNet [7], GoogLeNet [8], ResNet [9], and VGG16 [3].

2.5 Pyautogui as an Action Agent

After using hand prediction and classification, our job is to control the device to play Chicken Invaders based on the predicted landmarks and classes. We map 3 basic actions of playing Chicken Invaders, which are “no firing”, “firing”, and “firing both guns”, onto 3 shapes of the hand. The mapping is shown in Table II.

In terms of landmarks, we utilized a mapping technique to align camera frames with the screen size. This ensured that the hand's position could be accurately represented across every pixel of the screen.

No	Actions	Mapped hand shape	Device action	Hand image
1	No firing	Closed	No click	
2	Firing	Opened	Click left mouse button	
3	Firing both guns	“Rock”	Click both mouse buttons	

Table II. Hand gestures mapping for gameplay

We created an agent to perform these actions using the library pyautogui. This library allows the program to control the mouse and the keyboard of the device. The sample coding of the agent is shown below:

```
import pyautogui
class Agent:
  def shoot_first_gun(self, shoot):
    try:
      if shoot:
        pyautogui.mouseDown()
      else:
        pyautogui.mouseUp()
    except KeyboardInterrupt:
      print('\n')

  def shoot_second_gun(self, shoot):
    try:
      if shoot:
        pyautogui.mouseDown(button="right")
      else:
        pyautogui.mouseUp(button="right")
    except KeyboardInterrupt:
      print('\n')

  def move_mouse(self, x_pos, y_pos):
    try:
      pyautogui.moveTo(x_pos, y_pos)
    except KeyboardInterrupt:
      print('\n')
```

We employ the Action Agent on a separate thread, this enables the agent to constantly polling for hand position and gestures output from the two other modules via shared memory. This ensures the system achieves a higher overall frame per second.

3. Evaluation

3.1 Hand Detection

The detection module performed to our expectations by demonstrating both remarkable accuracy and consistent, rapid response times. These qualities are critical for successful integration into our application. During training, the models efficiently converged within 225 epochs with decreasing losses on both train and valid sets.

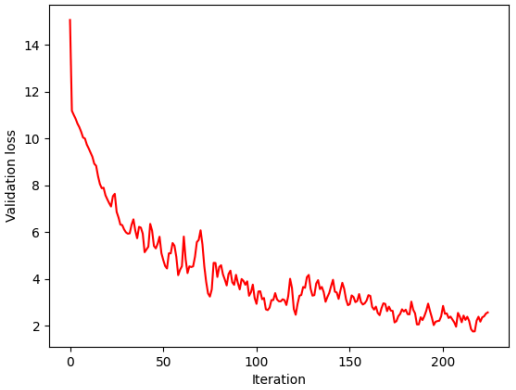

Fig 8. Validation losses throughout iterations

The model's performance was evaluated on a test set of 1,550 images from the Egohands dataset, supplemented by a smaller set of challenging cases we collected. We measured Mean Average Precision (mAP) on a single class (if the hand is available in the picture). Additionally, we compared various backbone architectures to identify the most suitable option, provided in table III.

Backbone	Num. Train Epochs	mAP@0.5	Average FPS
MobileNet V2	225	0.8958	~23 FPS
VGG16	400	0.9342	~11 FPS

Table III. Evaluation of Detection module on EgoHand test set

As shown in the table, MobileNetV2 exhibited a trade-off between accuracy and real-time performance. While achieving a commendable average frame rate of 23 FPS, its accuracy was slightly lower than VGG-16. However, VGG-16's superior accuracy came at the cost of significantly slower inference, hindering real-time requirements and longer training time.

3.1 Hand Classification Evaluation

Similar to the Hand Detection, the fine-tuning for the Classification module met our expectations. During training, the models quickly converged within 10 epochs with decreasing losses on both train and valid sets.

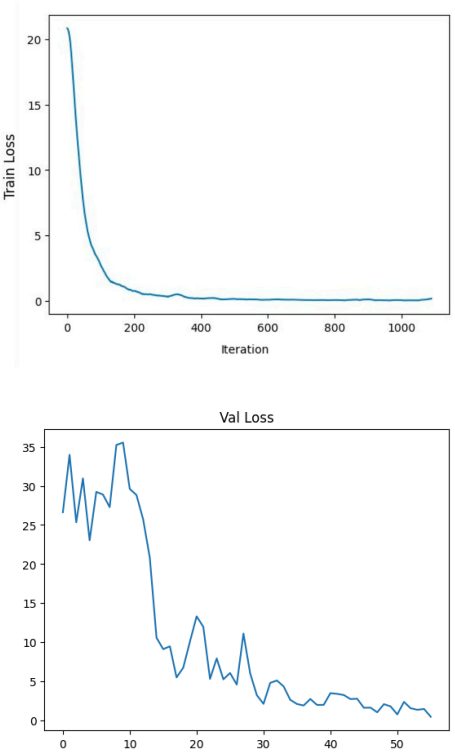

Fig 9. Train and Validation losses throughout iterations

The model's performance was evaluated on a test set of 950 images from the Roboflow Hand Gesture dataset. We measured the average accuracy and F1 score across 20 classes, which details are provided in table IV. We also provided comparisons to other classification architectures. Due to the computation requirements and our limited resources, we only evaluated 2 other fine-tuned models, including EfficientNet-B0 and GooLeNet. The reason we chose these 2 model architectures for comparison was because of their efficiency orientation, as they were designed to be lightweight and computing efficient.

Additionally, we tracked the accuracy per class to ensure that the model maintains good performance across all classes. Despite its small size, MobileNet performed exceptionally well on all subsets and real trials by us.

Clf model	Num. Train Epochs	F1	Average accuracy
MobileNet V2	10	0.9841	0.9841
EfficientNet-B0	10	0.9901	0.9921
GoogLeNet	10	0.9622	0.9704

Table IV. Evaluation of Classification module on Roboflow test set

Due to their complex architectures for vision tasks and being pretrained on large datasets, their scores are very high for this hand classification task. Thus, we agreed with the decision of choosing MobileNetV2 as our main classifier for its smallest size and efficient inference.

3.3 Pipeline FPS Evaluation

The complete pipeline achieved consistent CPU-based real-time performance, averaging around 18-19 FPS. For comparison, MediaPipe API claims to achieve 30 FPS performance, with lots of complex optimization techniques, including C/C++ based pipeline, quantization, GPU-based, and threadings. Considering the scope of our project of end-to-end reimplementation, we believe that this frame rate falls within an acceptable range for our gameplay, Chicken Invaders.

To experience the pipeline in action, feel free to check out our demo recording available here.

4. Discussion and Future Improvements

Current Bottleneck and Performance Gains: While our hand detection module performs well, it remains the current bottleneck of the pipeline, limiting the overall FPS. This presents an exciting opportunity for significant performance improvement by utilizing state-of-the-art techniques. Exploring lighter-weight models like

EfficientDet or newer SSD variants like SSDLite could potentially boost FPS without drastically sacrificing accuracy. Additionally, investigating hardware acceleration through GPUs or specialized inference chips could further alleviate computational constraints.

Scalability and Future Additions: The current system's modular design facilitates seamless scaling to incorporate additional keys and actions with corresponding hand gestures. This paves the way for richer and more nuanced control schemes within our application.

Beyond simple key mappings, we envision integrating interactive mechanisms that leverage hand movements for other intuitive controls. For example, users could control a virtual car by rotating their hands as if steering a wheel, offering an immersive and engaging experience. Implementing such features would require exploring hand pose estimation techniques alongside hand detection, opening up a wider range of interaction possibilities.

5. Conclusion

Our hand detection pipeline forms a robust foundation for gesture-based interaction. By addressing the identified bottlenecks and exploring innovative hand-based control mechanisms, we aim to further elevate the user experience and push the boundaries of real-time hand gesture recognition in interactive applications.

6. Teamwork

Member	Tasks	Contribution
Pham Quoc Trung	- Detection model fine-tune - Code base set up	40%
Nguyen Tiet Nguyen Khoi	- Classification model fine-tune - Poster preparation	30%
Lai Dac Tien	- Pyautogui for gameplay - Cost efficient model research	30%

7. References

[1] Vivek, P., Prasad, M., & Ye, W. (2020). MediaPipe BlazePose: On-device Real-time Body Pose tracking. Google AI Blog. https://ai.googleblog.com/2020/08/mediapipe-blazepose-on-device-real-time.html

[2] Bambach, S., Lee, S., Crandall, D. J., & Yu, C. (2015). Lending a hand: Detecting hands and recognizing activities in complex egocentric interactions. In The IEEE International Conference on Computer Vision (ICCV) (pp. 1502-1510). Institute of Electrical and Electronics Engineers.

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L. -C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 4510-4520, doi: 10.1109/CVPR.2018.00474.

[3] S. Liu and W. Deng, "Very deep convolutional neural network based image classification using small training sample size," 2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR), Kuala Lumpur, Malaysia, 2015, pp. 730-734, doi: 10.1109/ACPR.2015.7486599.

[4] R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, USA, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.

[5] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single shot multibox detector. In Computer Vision – ECCV 2016 (pp. 21–37).

[6] Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., & Zisserman, A. (2012). The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results [Dataset]. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html

[7] Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. In International Conference on Machine Learning (pp. 6105-6114)

[8] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going Deeper with Convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

[9] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)

[10] Hand Gestures Recognition. (2022, July). Hand gestures dataset dataset. Roboflow Universe. Roboflow. Retrieved February 9, 2024, from https://universe.roboflow.com/hand-gestures-recognition/hand-gestures-dataset