ILLINOIS INSTITUTE OF TECHNOLOGY

ECASP

Vision-Based Gesture-Controlled Drone: A Comprehensive Python Package to Deploy Embedded Pose Recognition Systems

Summer 2021 Research Project – Final Report

August 11, 2021

Arthur FINDELAIR

Advisor: Dr. Jafar Saniie

Embedded Computing and Signal Processing Research Laboratory

Departement of Electrical and Computer Engineering

Illinois Institute of Technology, Chicago, Illinois, USA

CONTENTS

***Abstract*—This project focuses on the creation and deployment of gesture control systems. A Python package is presented to ease dataset creation, models evaluation, and processing pipeline deployment. The critical element in the proposed architecture is the intermediate representation of human body poses as keypoints to perform efficient classification. Several classification models will be compared to reach an accurate and robust system that performs well even on partial inputs. The resulting processing pipeline will then be optimized and deployed on a drone's companion computer to provide a robust proof-of-concept of gesture control interface for embedded applications.**

I. Introduction

A. Context

1) The growing use of drones:

Autonomous systems are rapidly growing in popularity. Self-driving vehicles, assistant robots, aerial drones, to name just a few, are devices that would significantly shift the way we experience our day-to-day life for the better. The benefits are evident for the two first examples. Autonomous cars could save tremendous time to commuters, and assistant robots already provide great help in the health care and social assistance contexts. However, for most people outside the drone industry, these systems are a simple combination of a camera with a small copter or airplane. Yet drones are a lot more than that. Both related software and hardware are now very advanced. It pushed unmanned aerial vehicles to play critical roles in many industry verticals that indirectly help citizens with everyday services. Besides the massive use of filming/photography drones in the media, a recent survey [1] states that mapping, surveying, or inspection tasks have a rate of use of drone-based methods of 80% in construction, 83% in energy, and 67% in real-estate and industrial plants.

Industry Vertical	Mapping & Surveying	Inspection	Photography & Filming	Localization & Detection	Other¹	Delivery	Spraying & Seeding
Energy (Utilities)	14%	83%	0%	1%	1%	0%	0%
Construction	80%	15%	0%	0%	2%	2%	0%
Transportation and Warehousing	48%	34%	0%	6%	5%	7%	0%
Agriculture	59%	3%	0%	8%	0%	0%	30%
Mining, Quarrying, and Oil & Gas Extraction	55%	38%	0%	0%	6%	0%	2%
Public Administration	28%	5%	15%	32%	14%	0%	6%
Information	3%	6%	74%	0%	16%	0%	0%
Real Estate, Rental & Leasing, and Industrial Plants	16%	67%	17%	0%	0%	0%	0%
Arts, Entertainment, and Recreation	5%	0%	60%	0%	30%	5%	0%
Insurance	46%	10%	22%	14%	8%	0%	0%
Health Care and Social Assistance	35%	0%	2%	17%	8%	38%	0%
Professional, Scientific, and Technical Services	51%	6%	2%	15%	24%	2%	0%
Safety & Security	11%	0%	0%	69%	20%	0%	0%
Educational Services	53%	18%	0%	12%	17%	0%	0%
Waste Management & Remediation Services	34%	14%	5%	17%	30%	0%	0%

Fig. 1: Applied drone-based methods per industry [1]

However, these applications do not exploit drones' full potential as they are currently mainly used as prospection tools. Excluding these sorts of applications, there are only two growing uses of industrial drones: spraying and seeding in agriculture and urgent deliveries in the healthcare industry. The absence of drones in more practical applications is mainly due to the limits in the interaction between drones and their environment, including humans. An operator is currently mandatory in most cases to ensure safe flights. While an operator is flying a machine, he naturally cannot perform other tasks. Thus, the drone is just a tool and not an autonomous agent that can perform tasks independently. Cultivated fields are the perfect experimentation zone for autonomous tasks. Indeed, such terrains are mainly flat, unobstructed, and without humans potentially hurt by a malfunction. This is precisely why seeding and spraying are two of the few first tasks automated by drones.

2) Current limitations:

There are two crucial characteristics to develop to deploy autonomous drones in more applications. First, drones must perfectly perceive their environment to avoid obstacles and adapt trajectories to reach specific positions. Secondly, the human-machine interaction must be flawless. An assistant drone is useless if it cannot efficiently receive orders from users who could simultaneously perform other tasks. Simplified controllers could work when all persons interacting with the drone can have access to a controlling device when needed. This would limit interaction with the drone to users who are known beforehand. Otherwise, drones must perceive orders through embedded sensors. Besides speech, the most natural way for humans to communicate is through visual cues. Audio-based solutions are hardly possible due to the loud nature of drones' propulsion systems and the user's possible far distance from the machine. This leads us to the latter communication medium: visual communication.

With the advancement of Machine Learning (ML) techniques and especially Deep Learning (DL), computer vision systems have become more powerful and reliable. It is possible to capture a ton of data from a single image – take the proliferation of screens as the primary human-machine interface as proof for this claim. So naturally, almost all daily tasks performed by humans revolve around their vision: moving in crowded spaces, driving cars, manufacturing, exploring new environments, communicating. The interpretation of human movement is a critical aspect for autonomous systems to better understand their surroundings and efficiently understand humans. Such technology would open the way to a new kind of human-machine interface. While our project focuses on controlling a drone using body gestures, many other applications exist. Assistant robots, IoT hubs, or autonomous cars are some other applications that can be augmented by gesture recognition to enhance control interfaces and activity monitoring such as workout and habits trackers, non-invasive patient monitoring, or even autonomous rescue systems.

B. Objectives

The main objective of this project is to create a fully autonomous gesture-controlled drone. Robustness and extendability are the core characteristics of the implementation. Indeed, the project is split into two distinct steps to enforce these properties and dissociate the creation of the video processing pipeline from the deployment on the drone:

– The first fold of this project offers a kit of tools to develop, customize, and deploy pose recognition neural networks. The solution we settled on to respond to these

objectives is the development of a Python package called *Pose-Classification-Kit (PCK)*. Indeed, Python is the most used programming language in the ML community. Even though most frameworks (e.g. TensorFlow and PyTorch) are implemented in more efficient languages such as C++, the ease of using Python makes it a perfect choice to abstract as much as possible the creation and training of new models. This abstraction layer is critical to focus on the development of a model instead of the implementation details. Following the same objective, it makes sense to offer tools that fit into the same development environment. In its current state, the PCK package includes two main components: an application to ease dataset creation and model testing and an API to import existing datasets and pre-trained models. This paper will present our Python package, focusing on using the application and creating a dataset and the design of pose classification models.

– The second part focuses on integrating a video processing pipeline onto a drone to allow gesture control. The hardware platform has to be carefully selected to support the heavy computation overhead involved in the deployment of such a program. Most pre-assembled drones are not powerful enough to perform neural network inference. In addition, an API is mandatory to control the drone. These constraints will naturally lead us toward open-source projects, and most notably ArduPilot. Also, the integration of the pipeline of the drone will be developed such as it can be easily adapted to any other autonomous system. Indeed, one of the key aspects of this second fold of the project is to provide a comprehensive deployment process of the PCK processing pipeline.

II. Related works

Other works already presented similar pipelines to achieve gesture recognition. The system can always be split into two parts: the pose estimation and the pose recognition models. However, the degree of complexity varies greatly to match different use cases.

Brandon Yam-Viramontes and Diego Alberto Mercado-Ravell [2] present a gesture-controlled drone. The autonomous platform used in this project is not powerful enough to run a pose estimation model. Thus, the video stream of the embedded camera is sent to a ground station that runs OpenPose and a pose classification system to send orders to the drone. Still, the pose classification system is very efficient and flexible as it is not a trainable model. Four features are drawn from the keypoints: the angles between the torso and both arms and the distances between the hands and the torso. Orders are deducted based on this value using a threshold look-up table. While a new class can be defined without a dataset, the system's scalability is very limited. The authors deployed a system that detects ten poses. It seems to be the limit of the system. In addition, the user necessarily has to be oriented toward the cameras for the system to operate.

Feiyu Chen [3] presents a very similar processing pipeline as ours as it also uses OpenPose and a neural network for classification. The significant difference is the capacity to detect dynamic movements. This is achieved by using a recurrent neural network for the classification. The dataset contains approximately 20 minutes of video of the author performing nine different actions. Unfortunately, the authors state that the model's performance is pretty low as it only works on himself and in specific positions toward the camera.

Ngoc-Hoang Nguyen et al. [4] present one of the most advanced gesture recognition systems. In addition to the temporal dimension, this model also includes the third spatial dimension. This model certainly is one of the most performant gesture recognition today. The third dimension allows a very accurate representation of the pose, which is almost always the bottleneck of gesture recognition systems. However, the model is massive. It combines two 3D_ResNet networks and one LSTM network to benefit from RGB images, 3D skeleton joint information, and color body part segmentation. Such architecture cannot currently be deployed on most embedded applications.

III. Technical description of the project

The two folds of this project – processing pipelines creation and deployment – are presented in figure 2. While the processing pipeline describes in-depth the data flow of the inference process, the deployment pipeline should be interpreted as a general guideline on how to leverage the PCK.

A. Processing pipeline

Gesture control can be thought about in two ways: discrete and continuous. A discrete gesture control draws a direct association between a body pose or gesture to a label. This is a classification problem. On the other hand, a continuous gesture control analyzes each part of the body's exact position to offer refined control over continuous value. This is a regression problem. All user interfaces fall under one or the other category. For example, a keyboard is a discrete control interface where a mouse offers continuous control. Discrete interfaces are generally easier to implement. Besides, virtually all standard continuous control interfaces also include discrete interaction to operate appropriately – imagine a mouse without a button. Thus, the discrete approach is the first natural step toward a complete gesture control system. However, we will see in this section that a partial continuous representation of the body pose allows a more efficient system. In its current state, the order manager only supports discrete control of the drone. However, only minor additions are required to allow continuous control.

1) Object detection method:

First, one could treat this problem as a classical object detection problem. A person with a specific pose can be labeled like any other object in an image. A dataset of images of persons with body/hand poses would be needed, associated with a specific message. For example, an object detection model can be trained to find seated and standing people in an image and dissociate the two kinds of poses. This kind of neural network has been massively studied in recent years. Current

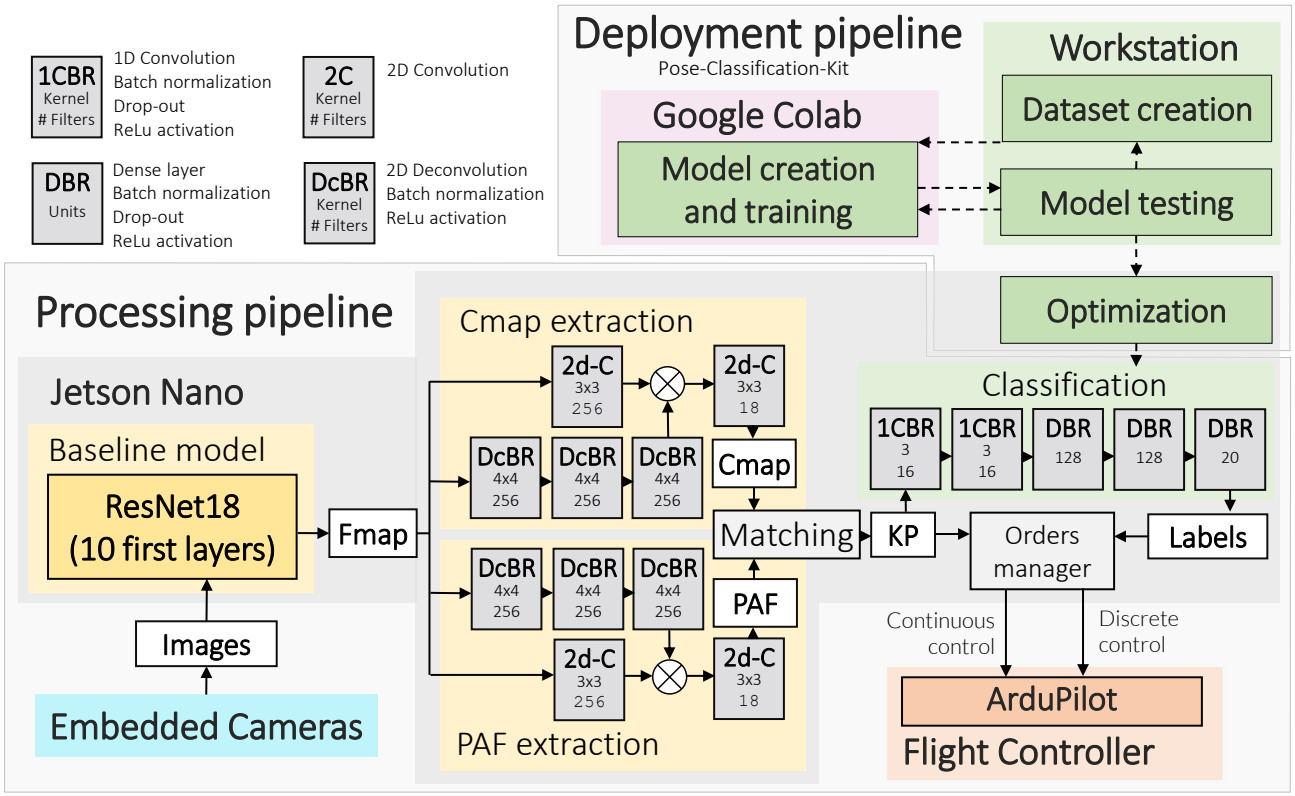


Fig. 2: Deployment and processing pipeline

state-of-the-art architectures are Faster RCNN, R-FCN, and Single Shot Detector (SSD). Each of these architectures can be implemented using a different feature extractor. A feature extractor is a model that only predicts the class of an image, with no consideration of the object's position. Note that such a model would not perform well for this application since the position of people in the images captured by the drone fluctuates heavily. Figure 3 presents the accuracy and inference time per image of some object detection architecture using different backbone CNN feature extractors. Note that the GPU used for these experiments runs at approximately 1 GFLOPS. Most of these models could then run at a high frame rate, even on embedded platforms.

However, the object detection approaches have a significant issue: the creation of an images dataset. Each body pose that the system should detect must be added to the dataset. Image dataset creation is highly complex due to the high risk of bias involved. In addition, the training of an image classifier of flexible objects - such as humans - is more demanding than rigid objects. For example, the corners of bricks are always at the same distance from one another, no matter the orientation of the brick. A neural network quickly learns such a pattern. However, flexible objects should be recognized under all their forms, leading to more complex features to analyze. This higher level of generalization leads to the necessity of more training images. Tens of hours of human labor would be necessary to create a large enough dataset. This problem has motivated the development of architectures more suited to pose classification that fall under the broader pose estimation

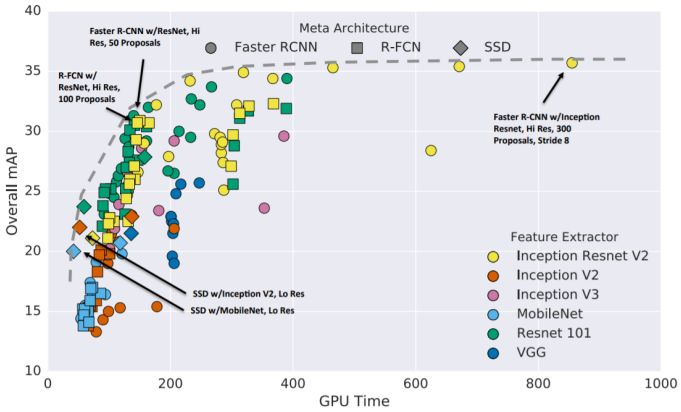


Fig. 3: Accuracy vs time, with marker shapes indicating meta-architecture and colors indicating feature extractor [5]

task.

The need for a large dataset of images to train such a model is a significant issue in our case. It would complexify the implementation of additional gestures in the system. Especially that, the training of an image classifier of flexible objects - such as humans - is more demanding than rigid objects in terms of training dataset size. To avoid this problem, we will develop a processing pipeline that dissociates pose estimation (continuous, regression) and pose classification. It is way more flexible in terms of class additions, and it also opens the way to continuous control.

2) Keypoints-based intermediate representation:

The processing pipeline describes the whole inference process from image acquisition to the transmission of orders based on the body pose of the user. The pipeline presented in this project (see figure 2) contains two main DL models that can be further decomposed: the pose estimation model (yellow) and the pose classification model (green). This project does not cover the training of a pose estimation model. Still, we will cover the architecture to clearly understand the whole processing pipeline and avoid potential pitfalls associated with its use. The intermediate data representation between both DL models is crucial to the understanding of the pipeline and the model creation process. The pose estimation model generated keypoints that represent the main articulations of a person in an image. These coordinates can then be used as input for a classification model to detect the body pose described by the skeletons. The dataset contains samples under this same keypoints format. The body representations vary only slightly from a pose estimation model to the other. It is thus possible to deploy the classification model on top of a pose estimation model different from the one used to create the dataset. We will later present the two body models used in this project.

B. Deployment process

The deployment process is composed of three main interconnected tasks. The origin of any model is the creation of a dataset. The dataset's quality is the most critical factor in the performance of the resulting classification model. In addition, the dataset defines the body poses that the system can detect. The creation of the dataset takes place on the workstation of the user where the PCK application runs. Once the dataset is created, a model can be created and trained. This can be done locally or on a computation server such as Google Colab to speed up the process. This step aims at creating a model as accurate as possible while minimizing its complexity to ensure an efficient inference process. Several data augmentation techniques will also be covered in this step. Still, the accuracy of a model does not reflect the actual performance of a model. The PCK application allows testing the model in real-time to improve its performance in real use-case. Based on these observations, the user can decide to augment the dataset, improve the architecture or the model's training process, or deploy it on the processing pipeline. Indeed, the development of a performant classification model is an iterative process. Nevertheless, we will not dive into the several iterations to reach our current architecture and training process. This paper will mainly cover our best-performing model.

IV. Hardware

A. Development platform

1) Workstation:

In the current state of the PCK application, a reasonably powerful computer is needed. The dataset's quality is largely defined by the accuracy of the pose estimation of its composed keypoints. The pose estimation model used in the PCK application thus has to be as accurate as possible. However, accuracy comes at the cost of efficiency. The current most well-performing pose estimation models are highly complex. Only a powerful computer equipped with a recent GPU (CUDA enabled) can run the inference process of these models in real-time at a decent frame rate. The workstation used in this project is equipped with an NVIDIA GeForce RTX 2060 Max-Q for a theoretical performance of 9.1 TFLOPS (FP16) GPU and an AMD Ryzen 9 4900HS CPU. In addition, a webcam is a crucial element to the use of the PCK application and the creation of new dataset samples. Any webcam can be used at this stage. Still, it is preferable to avoid a large field of view cameras that might introduce distortion into the generated samples. A Canon EOS M50 mirrorless camera with a 15mm focal lens ($77°$ of FOV) is used to capture images in this experiment.

2) Google Colab: After creating the dataset, the user can either carry on model training on its local machine or use a more powerful computation platform such as computation servers. Both solutions have been used during the development of this project. Google Colab is an online computation service running on top of Google Cloud Platform (GCP). It leverages the massive computational power of these servers under a simple Jupyter Notebook. Given the relatively low complexity of the classification model created in this project, the free tier service is plenty enough. The Python 3 kernel runs on a Linux virtual environment which is allocated up to 25 GB of RAM, 100 GB of disk memory, and an NVidia Tesla T4 (16GB RAM, 320 Tensor Cores) to reach a peak performance of 65.1 TFLOPS (FP16); roughly seven times faster than our local workstation. The user can also mount its Google Drive storage space as an additional disk to upload and save permanent files such as the trained model. Most classical Python packages comes pre-installed in the environment. However, it is still possible to install additional packages by passing console commands to the Linux environment through the Jupyter Notebook. We will use this functionality to install the PCK through the package manager Pypi.

B. Deployment platform

The choice of the deployment platform is crucial to the success of this project. A good balance between flexibility and abstraction must be found. On the one hand, a readily available commercial drone such as DJI Mavic or Skydio 2 can be considered the highest level of abstraction. The user has complete control of the drone at its fingertips, and the only pre-flight procedure is to unfold the machine's arms. The whole back-end system is hidden and cannot be adjusted. On the other hand, one can build a drone from spare parts and develop the flight control system on any microcontroller.

Fortunately, a lot of intermediate alternatives are available. Some drone constructors provide SDKs to access core functionalities of their drone, allowing the user to easily add custom features (e.g. Ryze Tello, Parrot Anafi). However, these solutions are generally heavily limited, both in terms of software and hardware. The system developed in this project will need high computational power. More specifically, neural

network inference processes that hugely benefit from parallel computing are hardly deployable on most microcontrollers used in commercial drones. Admittedly the best solution, in this case, is to use a drone kit flexible enough to accommodate a companion computer and an open-source flight controller, which allows for - and even encourages - the use of external devices. An abundance of kits available on the market suits the needs for this development platform, given the high expansion of drone development in recent years.

1) Holybro X500 quadcopter:

Besides a radio controller, the Holybro X500 includes everything one needs to fly. It ensures that all components work flawlessly with one another. This is especially important for relatively large drones for which the propulsion system can be complex to design. As its name suggests, the X500 is a quad-rotor in X configuration with an amplitude of 500mm. This ample wheelbase offers plenty of room for additional hardware and a powerful propulsion system with a maximum thrust of $\sim$ 1300g per motor. Such propulsion allows for a maximum payload of approximately 2.1kg with a 14.8V battery. Additional hardware that weighs up to 500g can be mounted, given that the weights of the frame, the battery, and the flight controller sum up to 1.6kg.

Fig. 4: Holybro X500 kit

2) Flight controller:

As shown in figure 4, the X500 kit also includes a flight controller, a compatible power distribution board, a telemetry kit, and a GPS module. The proposed flight controller is one of the main reasons behind the selection of the Holybro kit. It contains a Pixhawk 4. This board, made by Holybro, stands on the open-source Pixhawk standards that define hardware specifications and drone systems development guidelines. The Pixhawk platform has proven extremely powerful and reliable over the past decade and has become an industry standard for custom machines. The Pixhawk 4 from Holybro implements the FMUv5 design, released in 2018.

The central FMU processor is a 32bit ARM Cortex-M7 running at 216 MHz with 512 KB of RAM. Another lighter processor, 32bit ARM Cortex-M3 - 24 MHz - handles all interface operations such as GPS data analysis. In addition, the FMU design includes a couple of inertial measurement units for redundancy, a magnetometer, and a barometer. Note that an external GPS module is connected to the board. It contains a U-Blox GPS/GLONASS receiver and an additional magnetometer. This module is mounted on a pole, as far as possible from all magnetic fields generated by the motors and high power cables. This external magnetometer will thus be preferred as the main component for orientation detection for our system.

The Pixhawk 4 handles the flight control firmware, and the power management is completely separated. A Power Management Board (PMB) covers all these operations. It distributes the power coming from the battery to the flight controller under 5 V and power the ESC directly from a 6000 mAh, four cells (14.8 V) LiPo battery. The 5V voltage converter of the board is limited to 15W. While this is enough power for the flight controller, the companion controller might require another power supply. Also, all additional actuators are connected to the PMB to isolate logic and power circuits from each other. The Pixhawk 4 and the PMB are connected through 2 connections to power the flight controller and convey motor control signals.

Finally, two radio modules are connected to the flight controller. First, a 2.4GHz *X8R FrSky* radio receiver is connected through *SBUS* to control the drone with a *Taranis X9D+* controller. This nine-channel interface will be mainly used as a fail-safe to recover manual control if the autonomous system fails to follow the desired course of action. Secondly, a 915MHz, 100mW radio telemetry connection is used between the drone and a ground station. Using the MAVLink protocol, complex orders can be transmitted to the drone.

3) Jetson Nano:

Some of the most exciting applications of gesture control are embedded applications. Any autonomous system, including our autonomous drone, could be controlled through hand and body gestures. Our goal is to develop a flexible and easily deployable human-machine interface. It is thus essential to have a computational platform with low power consumption and a small footprint to fit in the platform. Still, pose estimation systems, and more generally, deep learning techniques, are particularly computationally heavy. The hardware platform must support parallel computing to operate neural network inference. Manufacturers understand this need for powerful embedded computers. One of the most common solutions is the Jetson platform from NVidia, which comprises multiple GPU-based single-board computers. We will use the entry offer of this product range; the Jetson Nano. Even this entry solution will prove powerful enough to handle the gesture-control processing pipeline thanks to its high performance and NVidia's neural network optimization tools.

The Jetson Nano reaches a top performance of 472 GFLOPS (FP16) on a Maxwell GPU running at 10W; roughly 22 times slower than the workstation. It can also operate using only 5W, but its performances are significantly reduced than the 10W configuration. In addition, this power consumption estimation only accounts for the naked Jetson Nano module. A carrier board is needed to extend connectivity and interact with the module. The carrier board included in the Development Kit

provided by NVidia requires an additional 1.25W for the module to operate at maximum power. The overall power consumption of the processing pipeline should thus be restrained under 12W.

The image capture device can vary heavily based on the application. The processing pipeline has been developed with this characteristic in mind, and it has been tested with USB, CSI, and Wi-Fi cameras. Still, a CSI camera is the most reliable and suited option in our case. A compact version of the IMX477 camera module from Arducam is mounted on the drone: the Arducam Mini with a 3.9mm lens adapted to the M12 mount of the module. With a 1/2.3", 12.3MP sensor, the camera has roughly the same horizontal FOV (80°) as the camera used on the workstation for dataset creation.

4) Integration:

figure 5 shows all the components of our systems, including the ground station and their interconnections. Note that all *B+* pads and all *GND* pads on the PMB are respectively connected to the positive and negative terminals of the battery. Thus, ESCs and the additional UBEC can be soldered interchangeably on any of these pads, given that polarity is respected. In addition, be aware of the logical ports used between the Pixhawk 4 and the PMB. ESCs signal ports are usually soldered to *Mi* (for i between 1 and 8). These pads are mapped on the *I/O-PWM-in* connector of the board. However, an alternative connection port is used to avoid ESC signal soldering. These alternative signal connections are mapped on the *FMU-PWM-in* connector of the board and are usually used to connect additional actuators such as servo motors. Still, these connections can easily be swapped, as shown in figure 5. *I/O-PWM-out* of the Pixhawk 4 is connected to *FMU-PWM-in* of the PMB, and ESCs signals are wired to *FMU-PWM-out* pins of the PMB.

The Jetson Nano and the Pixhawk 4 can communicate using two methods. The first option is to connect both devices through a USB connection. However, the USB protocol would lead to an extra encoding layer, given that both devices present serial ports. Additionally, most flight controllers' firmware documentation recommends not to use the USB connection of a board for real-time operation during flight. Thus, the Jetson Nano will be connected to the Pixhawk 4 using the UART ports *D15 (RX) - D14 (TX)* on the *J41* expansion header pins of the carrier board, and *TELEM2* on the Pixhawk 4. Note that both devices already share common electrical ground. Only the Transmit and Receive connections can be wired.

V. Pose Estimation

A. Two approaches

Human pose estimation is the task of inferring the precise pose of a person by identifying and locating keypoints on the body, such as major joints (elbow, knee, shoulders, etc.). There are two approaches to this problem:

– **Bottom-up**: the model localizes every instance of each keypoints in the image and tries to assemble these into skeletons for distinct persons.

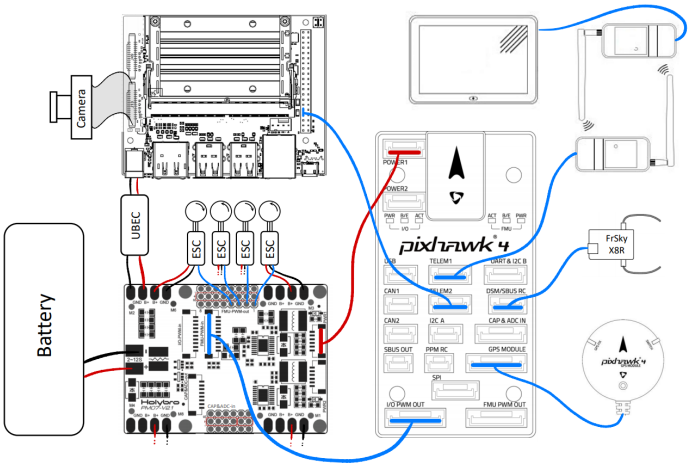


Fig. 5: Wiring diagram of the system

– **Top-down**: the model first identifies all persons in the image and then estimates the keypoints within each cropped region.

The bottom-up approach is more complex than the other to train due to the more generalized approach to keypoints detection and skeleton reconstruction. However, it is way more efficient thanks to the single-shot keypoints extraction, especially for crowd analysis. The present system will potentially be confronted with images of large groups of people. Constant time complexity is crucial to the reliability of gesture recognition in embedded applications with limited computational resources. For this reason, the current state-of-the-art pose estimation system takes on the bottom-up approach.

B. Part Affinity Fields Architecture

The current most popular bottom-up architecture is based on the fusion of Part Affinity Fields (PAFs), and part Confidence Maps (CMaps) [6]. This architecture is behind some of the most used pose estimation model such as **OpenPose** [7] from the same authors and **TensorRT-Pose** from NVidia, an simplified model for embedded applications. Note that both models have slight differences from the original architecture. However, the processing steps shown in figure 6 remain the same.

- First, the image goes through a backbone CNN which generates a feature map from the input image. As we saw earlier, this backbone model heavily influences the performance of a model. There is no strong recommendation about the choice of CNN in the original architecture, but it has been tested using the ten first layers of a pre-trained VGG-19. Note that virtually all models submitted to ILSVRC [8] can be used with different numbers of layers kept for feature extraction. Indeed, these models are all trained to classify images on an enormous dataset of a couple of thousands of labels and more than than 14 million images. Thus, their first layers are extremely efficient at extracting relevant information toward a pseudo-global understanding of images. Still, the backbone feature extraction model is fine-tuned during

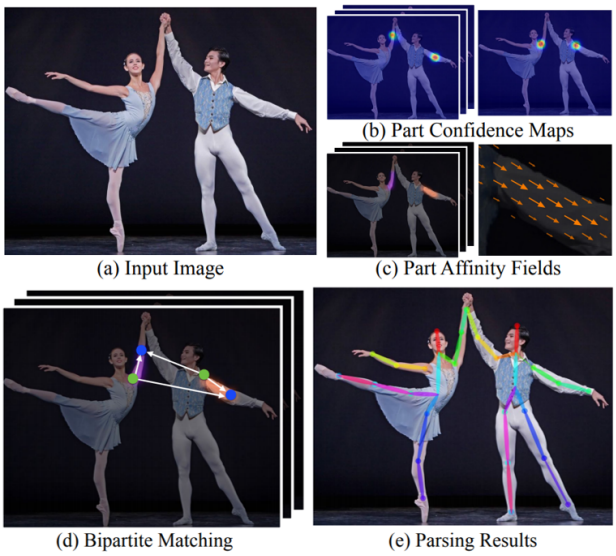


Fig. 6: PAF & CMap based architecture - "(a) Our method takes the entire image as the input for a CNN to jointly predict (b) confidence maps for body part detection and (c) PAFs for part association. (d) The parsing step performs a set of bipartite matchings to associate body part candidates. (e) We finally assemble them into full body poses for all people in the image." [7]

the training process of the pose estimation pipeline to interact well with subsequent networks.

- Then, the feature map is fed to a couple of multi-stage CNN that produces a Part Confidence Maps and a Part Affinity Fields – see figure 7. The CMap represents the probability that a particular human joint can be located in any given pixel. It contains as many channels as the number of types of body joints detected in the image. The PAF is a vector field that encodes the orientation and location of limbs. Again, it contains as many channels as the number of types of limbs (i.e. joints pairs).

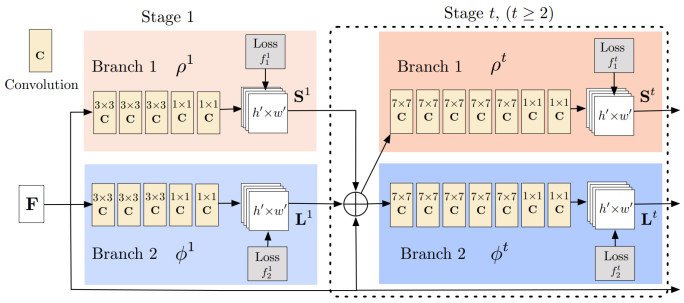


Fig. 7: Architecture of the two-branch multi-stage CNN. Each stage in the first branch predicts confidence maps S^t, and each stage in the second branch predicts PAFs L^t. After each stage, the predictions from the two branches, along with the image features, are concatenated for next stage. [6]

- Finally, the CMap and the PAF are processed by a greedy bipartite matching algorithm to output the skeleton estimation for each person in the image. This operation has two benefits. First, it allows connecting joints from the same person. Secondly, it eliminates the possible erroneous detection of joints in the CMap. Such algorithms have no trainable parameters; they belong to the realm of graph theory.

C. TensorRT-Pose

Even when using a lightweight backbone feature extractor, OpenPose [7], the original model developed by the authors of the PAF architecture, is way too computationally heavy to run on the Jetson Nano. Note that the current version of this tool has slightly changed since the publication of the first paper. The authors have achieved impressive results, but the model is still quite heavy. Fortunately, NVidia has developed an alternative model optimized for the Jetson platform. According to its documentation, **TensorRT-Pose** [9] manage to infer poses at 12 or 22 frame per seconds using respectively Resnet18 or Densenet121 as backbone feature extractors. However, the model had to be greatly simplified in comparison to OpenPose to reach such performance. Thus, the system is slightly less accurate.

1) Single stage CMap–PAF generation:

The critical simplification made to the architecture in TensorRT-Pose is that there is only a single processing stage ($t = 1$ on figures 6) while OpenPose has three. Consequently, there is no need for concatenation at the end of the stage—the deep learning model pipeline output S^1 as CMap and L^1 as PAF. There is no interaction between branches. While this characteristic enormously simplifies the model computationally, it is also a massive drawback in model capacity. Indeed, in multiple-stage analysis, each stage takes as input the combination of the results of both precedent output and the initial features map. Even though branches are trained independently and aim to produce different outcomes, they can benefit from one another by sharing their output. They can access additional information that might have been only extracted from the other branch. Such a technique artificially broadens the model and its learning capability.

Another distinction between the original architecture and TensorRT-Pose is the fundamental layer. Initially, simple convolution layers are used. However, due to the strong downsizing occurring during the feature extraction process of this optimized model, CBR (Convolution, Batch normalization, ReLu activation) up-sampling layers have a decent spatial resolution in the CMap and PAF. Also, as shown in figure 7 both branch architectures are strictly similar beside the final activation function.

2) Attention mechanism:

The model is augmented using an Attention map [10] to compensate for the lack of interaction between branches. An Attention map is a mask that allows focusing on relevant areas of the image. Such mask can be generated in parallel to the central processing pipeline and highlight its relevant output features as shown in figure 8. TensorRT-Pose deploys two attention mechanisms, one for each branch. The attention estimator is a single trainable convolution layer, and the

weighted combination is obtained through a simple element-wise multiplication with the output of the main convolution layers. Note that the attention mask and the weighted feature map must match in size.

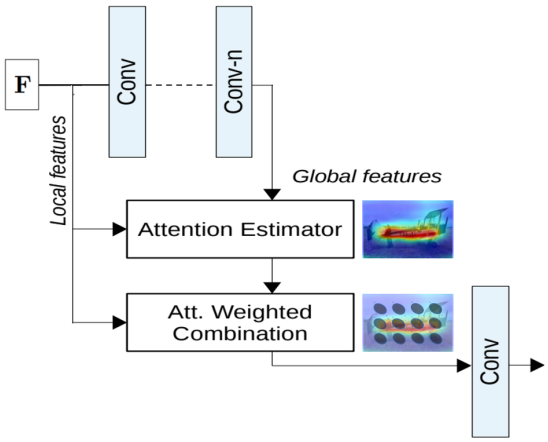


Fig. 8: The attention mechanism

See appendix A for a simplified version of the PyTorch definition of the CMap–PAF generation model with attention mechanism.

3) Backbone model:

Now that the head of the model is defined, the model only needs a feature extraction baseline. The TensorRT-Pose project currently supports almost all variations of ResNet, DenseNet, and Mnasnet. More than 20 models are defined using the pre-trained models provided in the Torch framework [11]. However, only two of these TensorRT-Pose variants have an available set of trained parameters: one using the Resnet18 as a baseline and the other using Densenet121. Again, the choice of the backbone model significantly impacts the performance of the overall model balance between efficiency and accuracy. For better efficiency, we will use the Resnet18 based model in our project. Similar to OpenPose, only the first layers of the CNN are used as the baseline, in this case: CBR, MaxPooling, and four successive convolutions.

VI. Python package – *pose-classification-kit*

Data collection is one of the most time-consuming and critical tasks of a Machine Learning project. A good dataset eases the models' training process, and it is also vital to a model's actual performance. As a rule of thumb, the larger the dataset, the better the resulting model. Indeed, exposing a model to a large number of samples is the best and most straightforward way to increase its generalization capacity on unseen samples. However, ML practitioners generally are limited by the size of already available datasets. This Python package proposes a tool to efficiently add instances to a dataset to train pose classification models.

Such tools are not suited for most computer vision tasks related to human analysis. Indeed, a good dataset must include various parameters that should not interfere with the classification. Suppose we were to analyze poses from images. In that case, the dataset should consist of many people displaying different features – sizes, skin tones, hair cuts, clothes – and in different contexts – various backgrounds, lighting conditions, and camera qualities. The number of variables is almost infinite in images, so it is tough to cover all cases in a dataset. Such limitations introduce bias in the final model that can only be detected during experimentation. Even the most extensive image datasets struggle to eliminate biases [12]. However, our dataset is based on keypoints, not images. This is where the power of this tool lies. A few hundred samples per class are enough to eliminate most biases from the dataset, thanks to the low number of variables in a skeleton. All the variables mentioned above do not apply here. However, the orientation and height of the camera still are some variables to eliminates from the dataset. Note that the size of the skeleton is normalized so that the person's size does not introduce biases in the dataset.

This Python package is fully open-source. The code is available on GitHub [13], and it can be easily imported in any Python environment thanks to the package manager PyPi:

```
$ pip install pose-classification-kit
```

Besides the dataset creation and model testing application, the package contains other interesting features:

- Two datasets for body and hand gesture classification of more than 11 000 samples each.
- Several pre-trained models to support multiple body models and different sets of classes.
- Utility functions for data augmentation and video analysis.
- Jupyter Notebooks examples to ease DL models creation.

A. Implementation

The PCK package is created using the Python packaging and dependencies management tool Poetry. This tool ensures that all dependencies in the project are compatible with one another. It can also build and publish the package on PyPI automatically. All of the package's parameters are defined in a single file. This mainly includes diverse info such as the author, the package's name, a description, the license, and the dependencies, and the callable scripts. We have defined three main categories of dependencies: mandatory, development, and extra-app. The mandatory set includes all the primary processing tools involved in the main package functionalities (e.g. Numpy, Tensorflow, Pandas). Only these are installed on the user system by default. Then, the dev-dependencies define the tools used during the development process. This includes Black for code formatting and pytest for unit testing. Only contributors need this additional set of dependencies. The extra-app dependencies cover the requirements to run the PCK application: OpenCV, PyQt5, and Matplotlib. The installation of these three packages is optional to ease the package deployment on systems that are not used for dataset creation. For example, the Jetson Nano and the Google Colab environment never call functions from the application. The installation of these three heavy packages is thus a waste of precious computational resources in an embedded scenario.

Note that no pose estimation models are deployed as part of the package. The user has to give access to one already installed on the system. The PCK package currently only supports the Python API of OpenPose [7]. Given the massive computation overhead involved in this model's inference process, the GUI application runs on multiple threads. For the graphical interface to be responsive, it should run on a different thread than computationally heavy tasks. Fortunately, the Python package used to develop the application includes multi-threading capabilities. The back-end system continuously runs the pose estimation inference on the video feed of the user's webcam. The output is sent to the front-end thread for further data management. The whole interface is event-driven and leverages several pre-defined and custom Qt widgets. One of the most critical components of the application is the data management widget, as it defines the storage format of samples. Internally, each class of the dataset is saved as an independent JSON file. This JSON file contains all the information related to the set, most notably the body format used for keypoints representation, the accurate detection of each sample, and obviously, the label associated with the file (which is also the name of the file). Such simple flat data representation certainly is the best balance between loading/saving efficiency and ease of use.

The source code of the application is separated from the general API of the PCK package. The functions composing the API are defined in three categories *datasets*, *models*, and *scripts*. The script folder mainly includes a function to export the whole dataset in a single CSV file used to import the dataset at once easily. Still, the end-user would not have to deal with the CSV file as the *datasets* API include a function to import the dataset as Numpy arrays. The package Pandas is used to read and process the dataset.

B. Utilization

1) The application:

As explained above, the primary purpose of this application is to create datasets easily. Once the pose-classification-kit is installed in a Python environment in which Python's Script folder is in the system's environment variable, the user can launch the application. Note that the pose estimation model (OpenPose) location must be defined by the user in the app's configuration file.

```
$ python -m pose-classification-app
```

Once everything is set up, the user can improve the dataset by adding samples to an existing class or creating a new one. To do so, one can respectively choose to *Open* (Ctrl+O) or *Create new* (Ctrl+N) in the *Dataset* drop-down of the menu bar – see figure 9. A configuration window will ask for the label and the newly created samples set's accuracy threshold in case of creating a new class. The accuracy threshold defines the minimum accuracy of hand keypoints detection from OpenPose of any sample in the set. This accuracy is displayed on top of the keypoints graph.

In both cases, a class is loaded in the application. The user can now record new samples from the video feed or inspect and delete the already available samples. At the end of a session, samples are saved in the app's internal database as JSON files. The CSV exportation of the dataset is executed with the following command:

```
$ export-datasets
```

2) Python API:

The second fold of the pose-estimation-kit is its Python API. Its most useful component, from a model training point of view, is the importation of the dataset and the data augmentation function:

```
from pose_classification_kit.datasets import
↪  bodyDataset, BODY18, dataAugmentation
dataset = bodyDataset(
    testSplit=0.15, shuffle=True, bodyModel=BODY18
)
x_scale_augm, y_scale_augm = dataAugmentation(
    dataset['x_train'], dataset['y_train_onehot'],
    augmentation_ratio=.15,
    scaling_factor_standard_deviation=.08,
    rotation_angle_standard_deviation=10,
    random_noise_standard_deviation=.03,
    remove_rand_keypoints_nbr=4,
)
```

The *bodyDataset* function has three parameters: *testSplit* defines the percentage of samples in the dataset to reserve for model testing, *shuffle* defines if the dataset should be shuffled, *bodyModel* defines the format of the keypoints. The function returns the training and testing samples – **x_train**, **x_test** – with their respective class – **y_train**, **y_test** –, as well as the label associated to each class – **labels** –. In addition, the class of each entry is also provided as one-hot encoded vectors – **y_train_onehot**, **y_test_onehot** –. With the exception of **labels**, this objects are all Numpy arrays which is compatible with most DL library, including TensorFlow. All the values are extracted from the CSV file dataset mentioned above which is hosted on GitHub[1].

The API also gives access to some pre-trained models, but we will not dive into more details as this part of the package is not yet fully functional.

C. The dataset

1) Structure & body formats:

The PCK package includes two datasets, one for full-body pose recognition and another for hand gesture recognition. The hands' dataset is out of this project's scope, so we will only focus on the full-body dataset. There is a total of 20 body dataset classes which contains between 500 and 600 samples each for a total of 10680 entries. Even if the number of samples from one class to the other varies in the raw dataset, the API yields a balanced dataset of 503 samples per class. Also, by default, 20% of these are reserved for final testing of the model. Each entry in the dataset is an array of 25 2D coordinates. The mapping of these keypoints

[1] raw.githubusercontent.com/ArthurFDLR/pose-classification-kit/master/pose_classification_kit/datasets/BodyPose_Dataset.csv

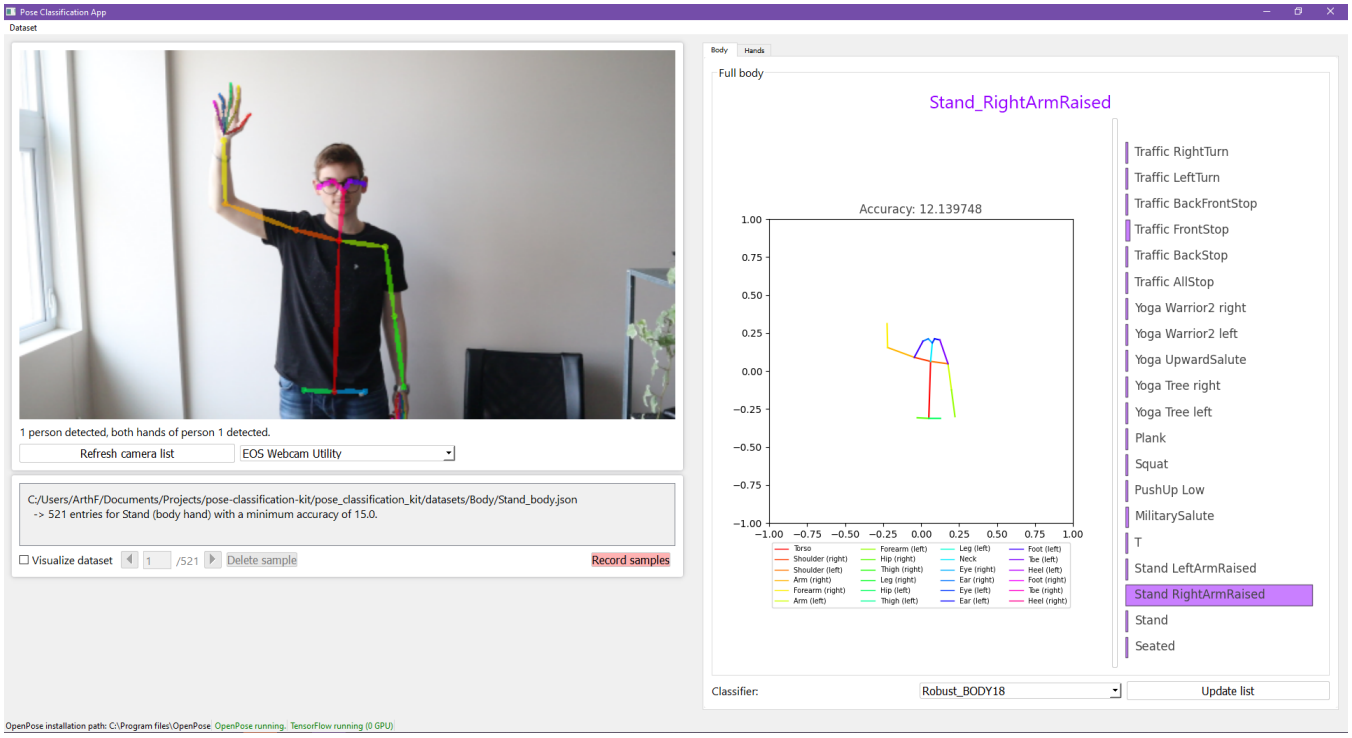


Fig. 9: PCK application in-use example

follows the BODY25 body model. Note that the connections between keypoints presented in figure 11 are only used for visualization. Only the keypoints coordinates are used for classification.

We created the dataset using the BODY25 representation as it is one of the most comprehensive standard body models. However, some pose estimation models, such as the one we will use on the Jetson Nano, use an 18 keypoints representation (BODY18). The seven missing keypoints do not strongly influence classification as 6 of them are used for feet representation, and the last one is a central hip keypoint. Still, the dataset must be converted to the BODY18 representation. This is done by reindexing the samples based on the comparison of the mapping of both body models as shown in figure 10. The user can choose which body model to use when importing the dataset with the API.

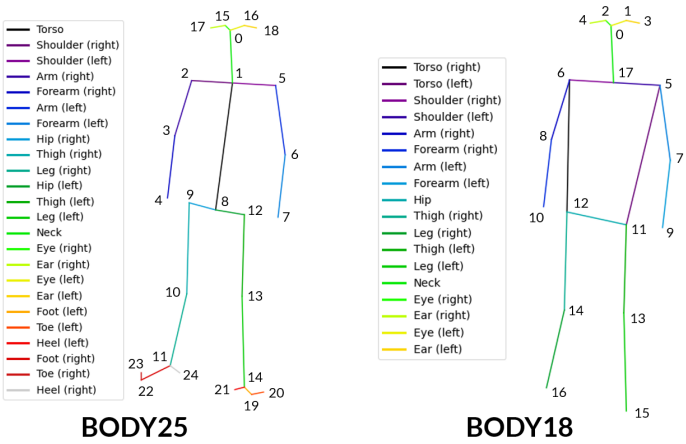


Fig. 10: Body models

2) Categories:

figure 11 presents example samples from all classes currently included in the dataset. The goal of this initial dataset is to allow several simple classification applications. Besides some diverse natural body poses, we can distinguish three main categories: work-out exercise, Yoga pose, and traffic signs. While the first two categories can efficiently train a work-out monitoring application, we will focus here on the traffic hand signals. These are the movement used by police officers to direct traffic. Aircraft marshallers also use such gestures in airports. They are thus perfectly suited to control our drone.

3) Data augmentation:

One of the most challenging parts of this project is the robustness and the real performance of the processing pipeline. Given the high accuracy and quality of the dataset, fairly high classification accuracy is reachable even with elementary models. However, it does not reflect the performance in real use scenarios. The most critical issue is missing keypoints. Some keypoints will not be detected if the image analyzed by the pose estimation does not include a person's whole body. These partial inputs are particularly hard to deal with. The most effective solution used in this project is data augmentation. The data augmentation procedure consists in altering existing samples from the dataset. These new samples can then be used to train the model, in addition to the original dataset. We developed four augmentation techniques:

– Scaling: a random scaling factor drawn from a normal distribution of mean 0 and standard deviation σ_{scale} is

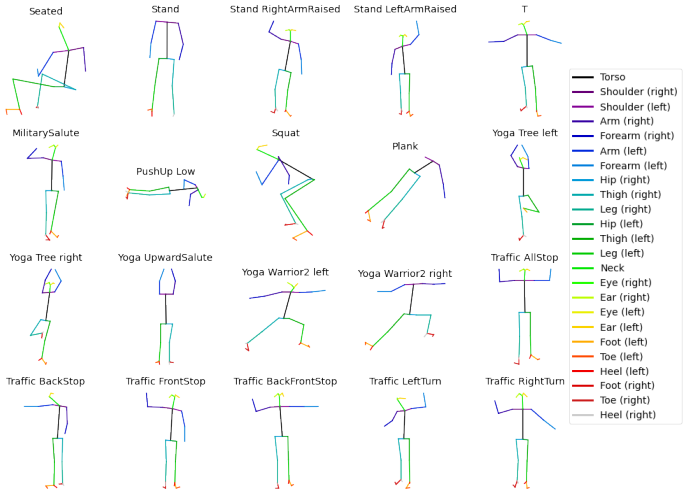


Fig. 11: Random samples from each class of the dataset

Augmentation Ratio	σ_{scale}	$\sigma_{rotation}$	σ_{noise}	Remove keypoints
10%	0.08	0.0	0.0	None
10%	0.0	10.0	0.0	None
15%	0.0	0.0	0.03	Legs
15%	0.0	0.0	0.03	Legs & Hip
20%	0.0	0.0	0.03	2 random

TABLE I: Detailed augmentation technique

applied to all sample coordinates.

- Rotation: a rotation of an angle randomly drawn from a normal distribution of mean 0 and standard deviation $\sigma_{rotation}$ is applied to the sample.
- Noise: Gaussian noise of standard deviation σ_{noise} is added to coordinates of the sample.
- Remove keypoints: a pre-defined or random list of keypoints are removed (coordinates set to 0) from the sample.

The dataset has been augmented by 70% (5628 samples) using a fusion of these techniques on the initial samples. Approximately half of these new samples only include keypoints above the waist. Indeed, most classes in the dataset can be detected without consideration of the position of the legs. The precise augmentation process is shown in Table I.

D. Classification model

As shown in the deployment pipeline presented in figure 2, the creation of a classification model is an iterative process. The actual performance of all trained models is evaluated in the PCK application. It gives insight that the validation accuracy cannot provide. Given these experiments, we can choose to either improve the model architecture or the training process. There have been two main iteration stages involved in the development of our classification model. First, we trained and compared two very efficient architectures on our original dataset. However, the testing experiments showed that these models are not robust enough in the case of partial inputs for real applications. So we improved robustness by developing a more complex model trained with data augmentation.

1) Architectures:

The body representation of the input of the classification model is quite atypical. Thus, no standard architecture is accepted as more performing than others, such as CNN for image analysis. Our first experimentation focuses on the comparison of two possible approaches toward the creation of efficient models. First, the most straightforward technique is to flatten the body representation and use dense layers. The 2D representation of 18 elements $[(x,y)_1,(x,y)_2\dots(x,y)_{18}]$ is fed to the neural network as an array of 36 elements $[x_1,y_1,x_2,y_2\dots x_{18},y_{18}]$. Trainable models would certainly adapt to such representation. Still, an architecture adapted to the natural input format generally performs better. The 1D convolution layer is one of the few supporting multiple channel arrays such as BODY18 (or BODY25).

The 1D convolution layer is very similar to the more common 2D convolution. In both cases, a kernel of trainable parameters is defined. In the 1D scenario, the kernel has two dimensions, one associated with the number of input channels (2 in our case – x and y coordinates), and the other defines the size (or the number of keypoints) computed at each convolution step, generally 3. All our convolution layers will thus use kernels of size 3×2. The intuition behind both convolution layers is the same. They aim at finding patterns in the input. This technique is particularly efficient when analyzing images as most tasks involved finding a specific object in an image. The local computation thus allows focusing on several parts of the image independently. Similarly, the 1D convolution excels at finding patterns in temporal inputs by analyzing several temporal windows independently. The order of the body keypoints is thus critical to allow similar local processing. Indeed, given a kernel length of 3, the convolutional layer will detect templates embedded in the coordinates of 3 consecutive keypoints. The most efficient way to order the keypoints is thus to place next to each other keypoints that form an anatomical part of the body. For example, shoulder–elbow–wrist, and hip–knee–foot are consecutive in BODY18 and BODY25.

Given the added complexity of the new dataset, classification is more challenging. The learning capacity of the initial models used on the original dataset is now too low. We developed a more complex neural network, including convolution layers and dense layers. This second model contains more than ten times the number of trainable parameters than the first models. This increase in complexity also complicates the training process. A couple of regularization techniques are used to tackle this issue:

- Dropout: At each new training batch, a defined rate of random nodes in a layer is set to 0. Such a process forces other neurons to replace the classification behavior of the dropped out. This heavily increases robustness and limits model overfitting. However, the training process is significantly slower.
- Batch normalization: the values flowing data in-between trainable layers are normalized – centered and scaled. The underlying reason for this technique is still unclear, but empirical analysis demonstrates significant performance

improvement.

A dropout rate of 30% is applied to all layers, and all layers are augmented with batch normalization.

2) Training and comparison:

Figure 12 presents the training history – losses and accuracies – over 15 epochs of 3 architectures:

- *Light ANN*: 2 hidden dense layers of 48 neurons each.
- *Light CNN*: 2 hidden 1D convolution layers of kernels of length 3 and 12 filters.
- *Hybrid* 2 1D convolution layers of kernels of length 3 and 16 filters and 2 dense layers of 128 neurons each.

Each model also includes a final layer of 20 neurons to match the number of classes in the dataset. Also, models are trained using the Adam optimizer with a categorical cross-entropy loss function and a validation split of 20%. Only the curves annotated *Augmented* in figure 12 are trained on the augmented dataset.

The first critical observation is that even minimal neural networks can achieve great results on the original dataset. Both light models achieve similar validation accuracies of approximately 98%. However, the CNN model only contains 75% of the ANN model's total number of trainable parameters. This results in a more efficient inference process for similar performance. The accuracy of the *Light CNN* model is 98.25% on the testing dataset. However, as explained earlier, this high accuracy does not fully reflect the performance of the model. Indeed, the testing accuracy of the model is only 50.95% on the same samples without legs keypoints. This model is thus selected as a great classifier for non-critical applications where the camera detects the whole body. Figure 13.a-b presents both confusion matrix on the original and modified testing dataset. As expected body pose that heavily rely on the position of the legs are heavily misclassified (see *Yoga_Tree_Right*, *Yoga_Tree_Left*, and *Yoga_UpwardSalute*).

The validation accuracy of the light models falls to around 80% when trained on the augmented dataset. Regularization methods show virtually no improvements in these scenarios. However, the regulated hybrid models yield great results. Even though the validation accuracy only reached 88%, the testing accuracies are astonishing. While the testing accuracy on the original dataset is roughly the same as on the light models, the accuracy on the modified dataset, which only contains upper-body keypoints, is up to 95%. As shown in the confusion matrices (see figure 13.c-d), even poses that are hardly distinguishable by human (only looking at the upper-body) is almost perfectly classified by the model.

VII. Deployment on the drone

A. Flight controller firmware

First, let us define some of the technologies and terminologies to understand the software stack used in our final system. We already saw earlier that Pixhawk refers to a collection of open-source hardware designs which manufacturers used to create Pixhawk-based flight controllers. These boards generally come without any firmware installed on them. The user has to flash them himself with the firmware of his choice.

There are currently two principal firmware suites: Ardupilot and PX4. Ardupilot certainly is the most advanced, full-featured, and reliable open source autopilot software available. It has been under development since 2010 by a large team of engineers, computer scientists, and pilots. It can be deployed on many platforms such as airplanes, multirotors, helicopters, rovers, boats, balance-bots, and even submarines. ArduPilot is composed of multiple projects, each specialized in the control of a specific platform. ArduCopter is used for multirotor and helicopters. It also is the most supported and advanced project in the ArduPilot family. PX4 is an alternative set of firmware. While this firmware has been specifically developed for the Pixhawk platform, it is very similar to ArduPilot in terms of features and compatibility with Pixhawk boards. The significant difference between the two suites is their licensing. PX4 operates under the BSD License, while the GPL License protects ArduPilot. This means that PX4 does not require contributors to share their source code. While this is a crucial requirement for commercial applications, it restrains innovation and support from the open-source community. Still, PX4 is generally more user-friendly for simple applications. However, our project will require advanced features of the firmware. It is primordial that these are well documented. For this reason, ArduCopter will be used on this project's flight controller.

Once the flight controller suite used on the drone is defined, one needs another software to flash, setup, and tune the firmware - and potentially communicate with the machine during flight. The choice of this ground control station software is mainly dictated by the firmware. Even though they can work interchangeably, Mission Planner is developed in concert with ArduPilot, while QGroundControl is developed for PX4. Thus, the most natural choice is to use Mission Planner in concord with ArduCopter.

B. Video capture

Video encoding is computationally heavy. Given the limited capabilities of the CPU, it is best to leverage dedicated video encoder hardware. The Tegra X1 chip used on Nvidia Jetson devices includes a Nvidia Encoder (NVENC). It can be used with the GStreamer plugin included in the Jetpack software suite. GStreamer allow the user to define pipelines to connect the source to any output interface with a variety of processing blocks in between. The pipeline used in our implementation is as follow:

```
nvarguscamerasrc ! video/x-raw(memory:NVMM),
↪  width=(int)398, height=(int)224,
↪  format=(string)NV12, framerate=(fraction)60/1 !
↪  nvvidconv top=0 bottom=224 left=87 right=311
↪  flip-method=0 ! video/x-raw, width=(int)224,
↪  height=(int)224, format=(string)BGRx !
↪  videoconvert ! video/x-raw, format=(string)BGR !
↪  appsink
```

The embedded camera is connected to the first MIPI-CSI Camera Connector on the carrier board. *nvarguscamerasrc* define this connection as input for our pipeline. Note that

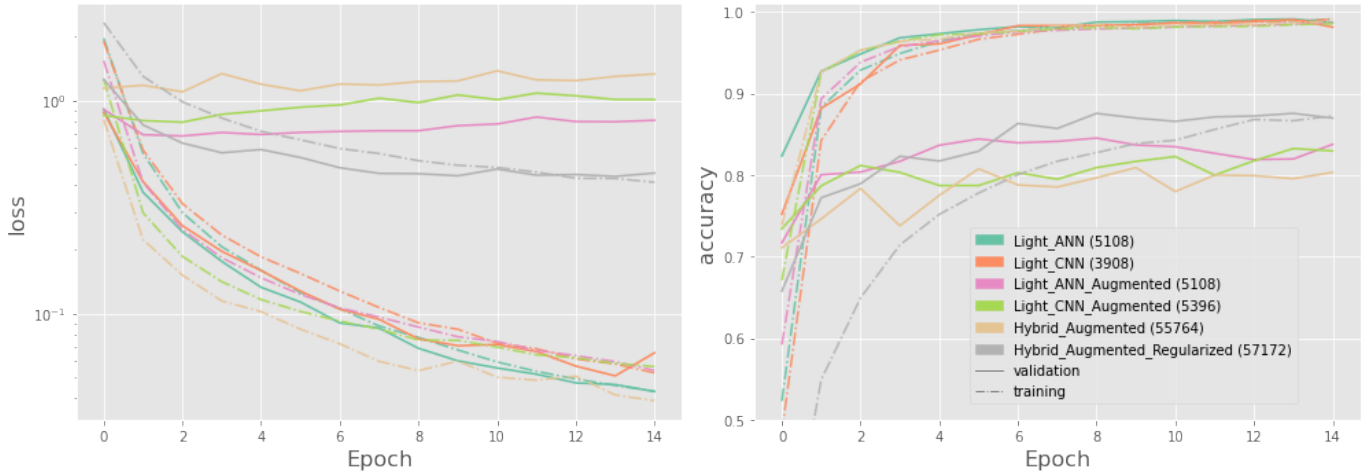


Fig. 12: Training history – losses and accuracies – of the studied models

the Jetson Nano B01 developer kit has two CSI camera slots. The *sensor_mode* attribute can be used with *nvarguscamerasrc* to specify the camera. Its value is 0 by default, which coincides with the CSI slot connected to the camera. The pose estimation model process image of size 224×224. However, the embedded camera streams images with an aspect ratio of $16/9$. The image is thus first scaled using the same aspect ratio and then cropped in the center of the image to avoid warping while maintaining the vertical field of view. Finally, the image is formated using the OpenCV default format: 32-bit per pixel, BGR. Images are received in a *VideoCapture* instance of OpenCV, which is accessed using the Python API. The embedded video-processing Python application is multi-threaded to reduce delay. One of the thread exclusively read images and return the latest image when needed.

C. Deep Learning models compilation

The efficiency of the processing pipeline is critical. The higher the frame rate, the better the responsiveness and robustness. The main challenge of the deployment of the processing pipeline is thus to leverage the computational power offered by the embedded platform fully. While the Jetson Nano is a competent platform thanks to its GPU, the CPU is minimal. Our goal is thus to rely as much as possible on the GPU. We will load both neural networks (estimation and classification models) on the GPU during the initialization of the processing pipeline. This can be done thanks to TensorRT, the neural network optimizer of the Jetson platform. However, the model first has to be optimized and compiled. As shown in figure 2, the optimization step must be executed on the target platform, the Jetson Nano in our case. There are two main parameters associated with this process: the data length and the maximum memory size allocated to the model on the GPU at runtime.

Weights, Biases, and layer inputs are all numbers. Their digital representation can thus influence the performance of a neural network. Most computational tasks in computer science require precise numbers representation. 32 (or sometimes 64) bits floating-point representation is thus the go-to on most systems. However, neural networks are robust in data representation. 16, 12, or 8-bit length representation is viable for neural network computation. The model network memory size and the bandwidth of data flow can thus easily be divided by a factor of 2 without significant loss in model accuracy. In addition, the values of trainable parameters and intermediate features do not matter per se – proportional values would perform similarly. Thus, even if a model is developed using parameters between 0 and 1, values can be scaled up to be represented as fixed-point numbers. Arithmetic operations are thus heavily simplified. However, except for the Xavier module with Tensor cores hardware, Jetson devices do not support the INT8 representation. We will thus only consider 32 and 16-bit quantization.

Original: 453		Data length FP32	Data length FP16
Size	1GB	89	68
	33MB	86	67

Pose estimation model

Original: 184		Data length FP32	Data length FP16
Size	1GB	6	6
	33MB	5	5

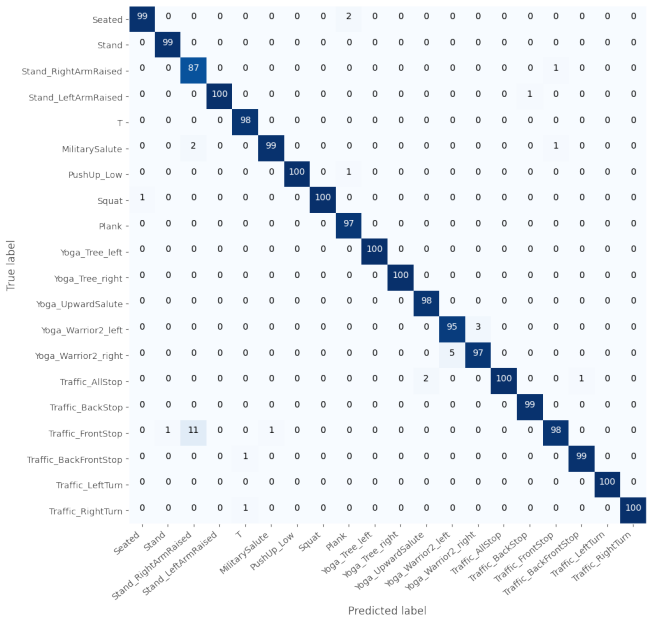
Pose classification model

TABLE II: Comparison of the inference times with different optimization parameters $[ms]$

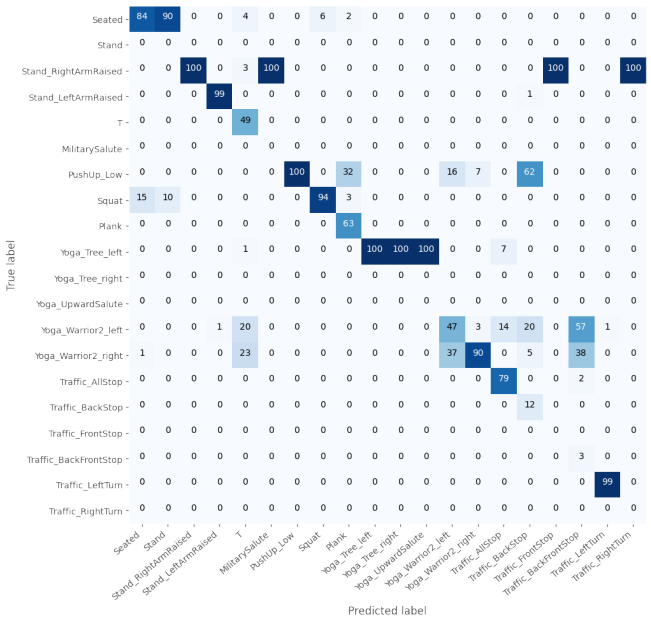
Given the optimization results presented in Table II, 33MB is enough for both models. In addition, while the pose estimation model reaches far better inference times using FP16 data representation, this is not the case for the pose classification model. This is undoubtedly due to its very low complexity. Indeed, narrow neural networks generally do not gain from quantization. Still, the FP16–33MB compiled models are selected for deployment. The model compilation process is highly effective as it improved the cumulative inference time of both models from 637ms to 73ms.

D. Communication protocol

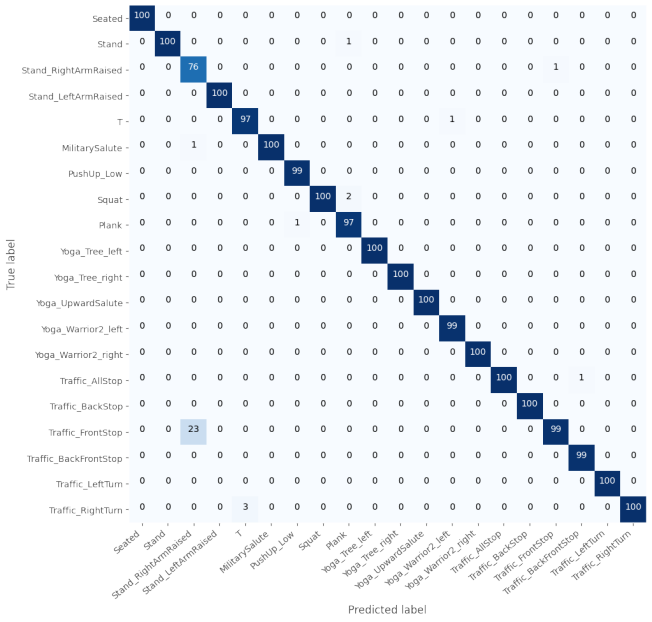
The compatibility between all software and firmware is ensured by the communication protocol MAVLink. Micro Air Vehicle Link is a protocol for communicating with small unmanned vehicles such as the quadrotor drone developed



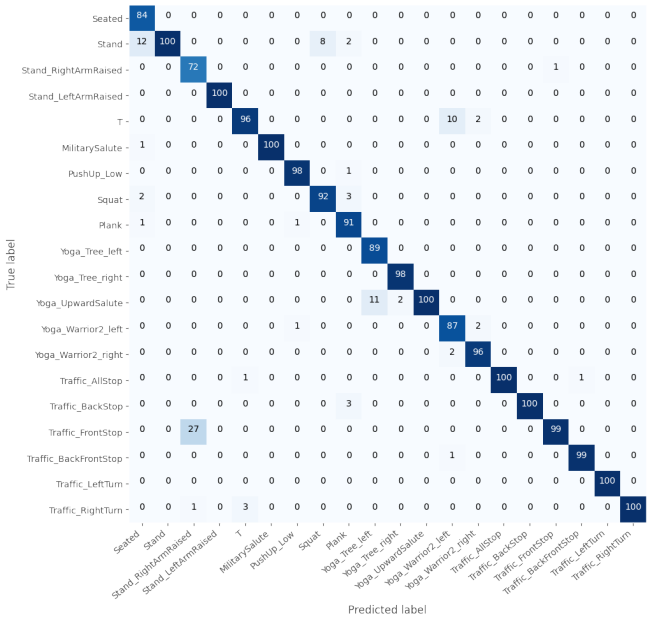
(a) *light weight* model on the testing dataset



(b) *light weight* model on the truncated testing dataset



(c) *robust* model on the testing dataset



(d) *robust* model on the truncated testing dataset

Fig. 13: Confusion matrices

for this project. Not only is it very efficient and reliable, but MAVLink is also flexible and configurable. Messages are defined within XML files. Each XML file defines the message set supported by a particular MAVLink system, also referred to as a *dialect*. New messages can easily be added by modifying the *common.xml* file available on all communicating systems: the flight controller, the companion computer, and the ground control station. The most recent firmware and software versions of MAVLink compatible devices use Version 2, whose packet structure is defined in Table III.

Multiple tools exist to process MAVLink communications. However, most of these applications are pretty complex programs and offer many features. The current companion computer should use the most efficient tools to concentrate its computational resources toward neural network inference. Most MAVLink tools use the Pymavlink python library. This is a low-level and general-purpose MAVLink message processing library developed and supported by the MAVLink corporation and ArduPilot. While this tool is highly efficient, it does not include any pre-defined message to ease control. Thus, we will use the lightweight Python package DroneKit, which presents a slightly higher level of abstraction with pre-defined functions to operate the drone efficiently. This python package will

Field name	Index (Bytes)	Purpose
Start-of-frame	0	Denotes the start of frame transmission (v1.0: 0xFE)
Payload-length	1	Length of payload (n)
Packet sequence	2	Each component counts up their send sequence. Allows for detection of packet loss.
System ID	3	Identification of the SENDING system. Allows to differentiate systems on the same network.
Component ID	4	Identification of the SENDING component. Allows to differentiate components of the same system.
Message ID	5	Identification of the message - it defines what the payload "means" and how it should be decoded.
Payload	6 to (n+6)	The data into the message.
Cyclic redundancy check	(n+7) to (n+8)	Check-sum of the entire packet, excluding the packet start sign (LSB to MSB)

TABLE III: MAVLink 2 Packet Format [14].

allow for easy communications with the flight controller whose firmware already uses MAVLink as a standard communication protocol, as seen earlier.

E. Orders management

The order manager is the most critical part of the embedded processing pipeline as it directly controls the drone's actions. Every movement must be carefully controlled to avoid crashes. Still, the flight controller includes some protections over received orders over MAVLink connections. For example, motor disarming signals are ignored when the drone is in flight. Such critical application requires careful analysis of the DL pipeline output. Misclassification is relatively common and can lead to unwanted actions from the drone. Still, it is improbable that several consecutive frames are similarly misclassified. Thus, orders are sent to the flight controller only when the processing pipeline detects the same pose several times over a short time. A buffer of 7 elements keeps track of the latest poses detected. An order is selected when more than half the buffer is composed of the associated pose. The order is then transmitted to the flight controller only if it is different from the last selected order to avoid repetition.

In addition, the order manager continuously checks the current state of the drone. It allows to create conditional actions and add another layer of security. For example, the take-off order is only sent if the drone is armed and on the ground. Also, orders are only transmitted if the flight controller is in *guided* mode.

The source code of the order manager running on the main thread of the embedded application is available in appendix B.

VIII. Results

A switch on the radio controller allow the user to set the drone's state to *guided* and thus activate gesture controls. It currently supports basics – yet essential – commands:

- *T*: Arm the drone if it is disarmed and landed; Disarm the drone if it is armed and landed.
- *Traffic_AllStop*: Take-off at an altitude of 1.8m if the drone is armed and landed; Land if the drone is in flight.
- *Traffic_RightTurn*: Move 4m to the right if the drone is armed.
- *Traffic_LeftTurn*: Move 4m to the left if the drone is armed.
- *Traffic_BackFrontStop*: Move 2m backward if the drone is armed.
- *Traffic_FrontStop*: Move 2m forward if the drone is armed.
- *Yoga_UpwardSalute*: Return to Launch (RTL).

No delay is perceivable thanks to the processing pipeline's relatively high frame rate, which varies from 9.5 to 12.0 frames per second. The embedded camera's large vertical field of view allows a functional system both in flight and on the ground. The user can thus fully control the drone from take-off to landing using gesture control only. However, the system does not include user tracking. The user necessarily has to stay in front of the drone to perform gesture control. This is particularly restrictive when the drone operates at a high altitude. The maximum altitude at which the drone can operate naturally varies, given the camera's orientation and field of view. The limit is approximately 4 meters in our configuration – camera leveled – for the system to also detect gestures while landed.

IX. Future Works

A. Technical improvements

While the system is fully functional, it is nowhere near being suited for consumer use. First, the initialization procedure is extremely slow. The inference process is highly efficient thanks to the neural networks being loaded on the GPU. However, this procedure is computationally heavy regarding the low-performance CPU. In addition, two different DL frameworks are used in the pipeline (TensorFlow and PyTorch), which leads to a complex dependencies importation procedure. The initialization would greatly benefit from the concatenation of both neural networks under the same architecture to streamline the process.

Fig. 14: Processing pipeline output frame

While the current performance of the pipeline is enough for a single camera, it might not be suited for multiple video feeds. Additional cameras on the drone would extend the gesture detection system coverage around the drone. There are two ways to increase the performances.

- The first and most apparent is improving the computation capability of the hardware platform. We can opt for a more powerful Jetson solution, such as a Jetson Xavier NX. No software modifications would be needed. The whole processing pipeline is indeed deployed using Jetson's SDK JetPack [15] which is similar on all Jetson boards. Also, Machine Learning ASICs accelerators are another great solution toward powerful devices for low-power consumption. For example, Google has developed and optimized a pose estimation model for the Google Coral, a TPU-based ML accelerator. This device can be used in unison or independently from the Jetson Nano to accelerate the processing pipeline.
- Secondly, instead of upgrading the hardware, the processing pipeline could be further optimized for multi-camera systems. We can leverage the constant time complexity of the model regarding the number of persons in the image. The model would undoubtedly support the analysis of multiple images in a single shot. The 224×224 input matrix can be split into four quadrants containing a different image from 4 different cameras. While this solution would accelerate the processing speed by four times, it would also significantly decrease the system's precision as each input image would be reduced to a resolution of 112×112. Fortunately, the bottom-up architecture is quite flexible in terms of input image resolution as it is defined by the size of pre-trained backbone models to generate the feature map as shown in figure 2. The ResNet 18 model can be switched for a larger model supporting larger input images with a resolution of 448×448.

B. Additional features

User-facing features are still minimal. The main objective of this project was to offer a proof-of-concept of an open-source gesture control system – which has been a success. However, the in-flight control is still limited to only two operations – go left or right. Little work is required to improve the interface significantly. The addition of continuous control and dynamic gestures would also be a massive improvement for minimal effort. This would, for example, allow the user to precisely defines the altitude of the drone or the distance to go in a given direction. The PCK package currently only includes static pose recognition models. However, the dataset creation tool and the overall pipeline only need few adjustments to support recurrent neural networks training. Still, we would have to create a new dataset to detect dynamic gestures.

Another fundamental improvement is to add user tracking to allow gesture control without going in front of the camera. The groundwork for such a feature is already implemented, given that people are already detected in the frame. Still, one of the challenges is to distinguish the main user from other people.

X. Conclusion

The significant contribution of this project is the creation of a Python package streamlining the training and deployment of gesture recognition systems. It contains two major components, a desktop application to create a dataset and evaluate the performance of classification models, and an API to access publicly available datasets and pre-trained models.

This paper also presents the creation and training of two neural networks. The first model is highly efficient and yet reaches a testing accuracy of 98.25%. However, it performs poorly on partial inputs, which are pretty common in practice. Thus, a second model has been created leveraging heavy data augmentation and regularization techniques. While the testing accuracy remains similar to the first model, it performs exceptionally well on samples that only display a person's upper body. The accuracy reaches 95% in this scenario.

Finally, we covered the deployment process of the video processing pipeline on resource-limited hardware. An open-source drone platform has been augmented with an embedded Jetson Nano companion computer to allow gesture control. The optimization of the pipeline allows a processing frame rate exceeding 9.5 FPS. Such performance results in a very responsive proof-of-concept.

XI. Acknowledgments

I would like to thank my supervisor, Professor Jafar Saniie, for his guidance throughout this project. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. I am also most thankful for the ECASP Laboratory of the Illinois Institute of Technology that has provided all the necessary hardware to develop this project. Finally, I want to thank Mohan Sridharan from the University of Birmingham who introduced me to vision-based human-machine interface and guided me into developing my first hand sign recognition system.

REFERENCES

[1] Ed Alvarado. 237 ways drone applications revolutionize business. https://droneii.com/237-ways-drone-applications-revolutionize-business. Accessed August 6, 2021.

[2] Brandon Yam-Viramontes and Diego Alberto Mercado-Ravell. Implementation of a natural user interface to command a drone. *CoRR*, abs/2003.02662, 2020.

[3] Feiyu Chen. Multi-person real-time action recognition based-on human skeleton. https://github.com/felixchenfy/Realtime-Action-Recognition, 2019. Accessed July 28, 2021.

[4] Ngoc-Hoang Nguyen, Tran-Dac-Thinh Phan, Guee-Sang Lee, Soo-Hyung Kim, and Hyung-Jeong Yang. Gesture recognition based on 3d human pose estimation and body part segmentation for rgb data input. *Applied Sciences*, 10(18), 2020.

[5] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors, 2017.

[6] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields, 2017.

[7] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *CoRR*, abs/1812.08008, 2018.

[8] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[9] NVidia-AI-IOT. trt_pose repository. https://github.com/NVIDIA-AI-IOT/trt_pose, 2021. Accessed May 13, 2021.

[10] Saumya Jetley, Nicholas A. Lord, Namhoon Lee, and Philip H. S. Torr. Learn to pay attention. *CoRR*, abs/1804.02391, 2018.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[12] Kaiyu Yang, Klint Qinami, Li Fei-Fei, Jia Deng, and Olga Russakovsky. Towards fairer datasets: Filtering and balancing the distribution of the people subtree in the imagenet hierarchy. In *Conference on Fairness, Accountability, and Transparency*, 2020.

[13] Arthur Findelair. pose-classification-kit repository. https://github.com/ArthurFDLR/pose-classification-kit, 2021. Accessed August 6, 2021.

[14] MAVLink. Packet serialization. https://mavlink.io/en/guide/serialization.html. Accessed July 13, 2021.

[15] NVidia. Advanced embedded systems for edge computing. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. Accessed July 13, 2021.

APPENDIX

A. *Source code – CMap–PAF generation model with attention mechanism*

Simplified version of TensorRT-Pose open-source code [9].

```
class UpsampleCBR(torch.nn.Sequential):
    def __init__(self, input_channels, output_channels):
        layers = []
        for i in range(3):
            if i == 0:
                inch = input_channels
            else:
                inch = output_channels
            layers += [
                torch.nn.ConvTranspose2d(inch, output_channels,
                                         kernel_size=4, stride=2, padding=1),
                torch.nn.BatchNorm2d(output_channels),
                torch.nn.ReLU()
            ]
        super(UpsampleCBR, self).__init__(*layers)

class CmapPafHeadAttention(torch.nn.Module):
    def __init__(self, input_channels, cmap_channels, paf_channels, upsample_channels=256):
        super(CmapPafHeadAttention, self).__init__()
        self.cmap_up = UpsampleCBR(input_channels, upsample_channels)
        self.paf_up = UpsampleCBR(input_channels, upsample_channels)
        self.cmap_att = torch.nn.Conv2d(upsample_channels, upsample_channels,
                                        kernel_size=3, stride=1, padding=1)
        self.paf_att = torch.nn.Conv2d(upsample_channels, upsample_channels,
                                       kernel_size=3, stride=1, padding=1)
        self.cmap_conv = torch.nn.Conv2d(upsample_channels, cmap_channels,
                                         kernel_size=1, stride=1, padding=0)
        self.paf_conv = torch.nn.Conv2d(upsample_channels, paf_channels,
                                        kernel_size=1, stride=1, padding=0)
    def forward(self, x):
        xc = self.cmap_up(x)
        ac = torch.sigmoid(self.cmap_att(xc))
        xp = self.paf_up(x)
        ap = torch.tanh(self.paf_att(xp))
        return self.cmap_conv(xc * ac), self.paf_conv(xp * ap)

class ResNetBackbone(torch.nn.Module):
    def __init__(self, resnet):
        super(ResNetBackbone, self).__init__()
        self.resnet = resnet
    def forward(self, x):
        x = self.resnet.conv1(x)
        x = self.resnet.bn1(x)
        x = self.resnet.relu(x)
        x = self.resnet.maxpool(x)
        x = self.resnet.layer1(x) # /4
        x = self.resnet.layer2(x) # /8
        x = self.resnet.layer3(x) # /16
        x = self.resnet.layer4(x) # /32
        return x

def resnet18_baseline_att(cmap_channels, paf_channels, upsample_channels=256, pretrained=True):
    resnet = torchvision.models.resnet18(pretrained=pretrained)
    return _resnet_pose_att(cmap_channels, paf_channels, upsample_channels, resnet, 512)
```

B. Source code – Orders manager

```
while True:
    latest_label = video_processing.get_pose()
    if latest_label == last_label:
        label = None
    else:
        label = latest_label
        last_label = latest_label

    if label and (drone.vehicle.mode.name=="GUIDED"):

        if label == "T":
            if not drone.vehicle.armed: # Arm
                drone.vehicle.armed = True
            else:
                if drone.vehicle.location.global_relative_frame.alt < THRESHOLD_ALT: # Disarm if landed
                    drone.vehicle.armed = False

        elif (label == "Traffic_AllStop") and drone.vehicle.armed:
            if drone.vehicle.location.global_relative_frame.alt < THRESHOLD_ALT:  # Take off if landed
                takeoff_alt = 2.5
                drone.vehicle.simple_takeoff(takeoff_alt) # Take off at two meters
                while drone.vehicle.location.global_relative_frame.alt<(takeoff_alt - THRESHOLD_ALT): #
                ↪  Wait to reach altitude
                    pass
                    #print("Altitude: ", vehicle.location.global_relative_frame.alt)
                time.sleep(.5)
            else:
                if drone.vehicle.location.global_relative_frame.alt > THRESHOLD_ALT:
                    drone.vehicle.mode = VehicleMode("LAND")

        elif (label == "Traffic_RightTurn") and drone.vehicle.armed:
            x, y = 0., 1.  #meters
            yaw = drone.vehicle.attitude.yaw
            drone.send_global_velocity(
                x*np.cos(yaw) - y*np.sin(yaw),
                x*np.sin(yaw) + y*np.cos(yaw),
                0,
                2
            )
            drone.send_global_velocity(0,0,0,1)

        elif (label == "Traffic_LeftTurn") and drone.vehicle.armed:
            x, y = 0., -1.  #meters
            yaw = drone.vehicle.attitude.yaw
            drone.send_global_velocity(
                x*np.cos(yaw) - y*np.sin(yaw),
                x*np.sin(yaw) + y*np.cos(yaw),
                0,
                2
            )
            drone.send_global_velocity(0,0,0,1)

        elif (label == "Yoga_UpwardSalute") and drone.vehicle.armed:
            drone.vehicle.mode = VehicleMode("RTL")

    time.sleep(.25)
    print('FPS: {fps:.2f}\tCurrent label: {lab1}\tSend order:
    ↪  {lab2}'.format(fps=video_processing.get_fps(), lab1=latest_label, lab2=label).ljust(80)[:80],
    ↪  end='\r')
```

C. Source code – PCK datasets exploration

Open in Colab

Pose Classification Kit: datasets exploration

This Notebook can be used to explore in depth the dataset of Pose Classification Kit. The default body model for the representation of the keypoints is BODY25.

Note that datasets can also be imported using the package API:

```
from pose_classification_kit.datasets import bodyDataset, handDataset
```

In [1]:

```
from IPython.display import display, HTML, Markdown

import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

import numpy as np
import pandas as pd
pd.set_option('display.max_columns', None)
pd.set_option('display.min_rows', 5)
```

Body dataset

In [2]:

```
dataset_df_body = pd.read_csv("https://raw.githubusercontent.com/ArthurFDLR/pose-classification
labels_body = dataset_df_body.label.unique()

display(Markdown("### Complete dataset view"))
display(dataset_df_body)

df_body_labels = dataset_df_body.groupby('label')

display(Markdown("### Number of samples per label"))
display(
    pd.DataFrame(
        [df_body_labels.size()],
        columns=labels_body,
        index=["Nbr of entries"],
        )
)
```

Complete dataset view

	label	accuracy	x0	y0	x1	y1	x2	y2	x3	
0	Seated	19.508690	-0.060776	0.497554	-0.085367	0.360305	-0.206404	0.359822	-0.254659	0.157
1	Seated	19.548573	-0.064262	0.501585	-0.088978	0.364251	-0.210033	0.363897	-0.250737	0.161
...	...	...	...	...	...	...	...	...	...	...
10679	Traffic_RightTurn	21.680399	-0.021659	0.501895	-0.015683	0.355273	-0.119264	0.361134	-0.266577	0.355
10680	Traffic_RightTurn	21.511433	-0.024451	0.499075	-0.023881	0.353047	-0.121352	0.358971	-0.273786	0.352

10681 rows × 52 columns

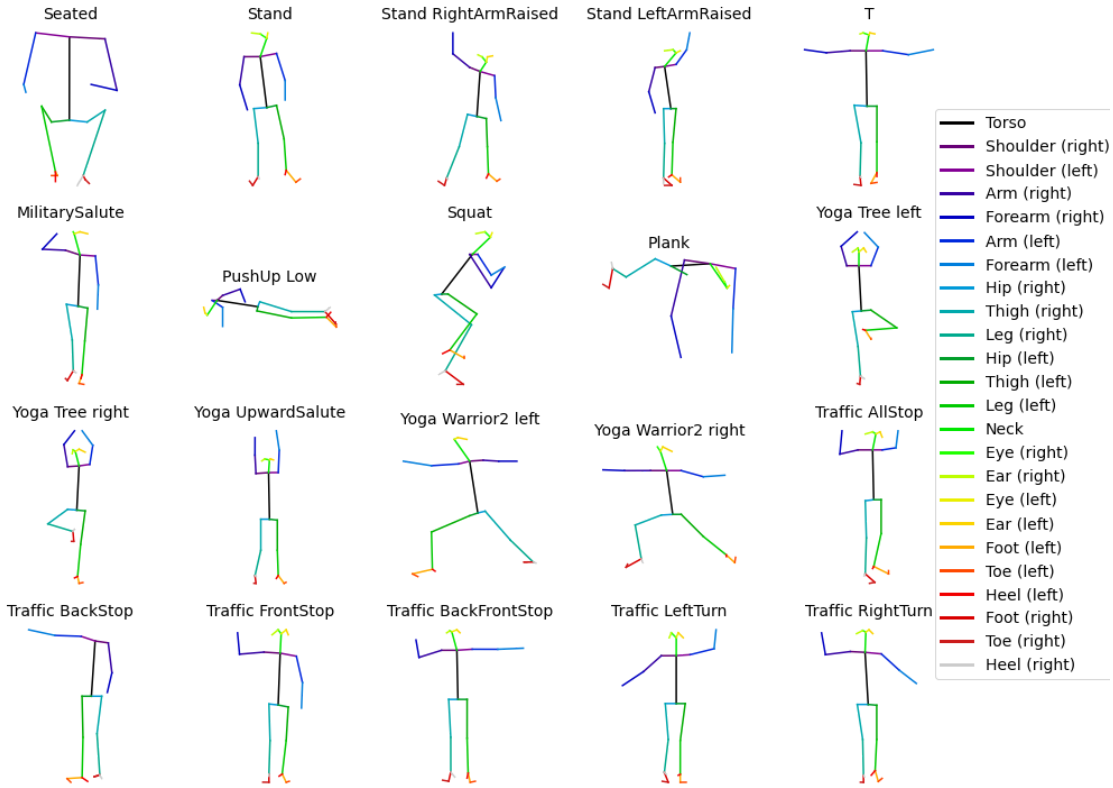
Number of samples per label

	Seated	Stand	Stand_RightArmRaised	Stand_LeftArmRaised	T	MilitarySalute	PushUp_Low	Squat	Pl
Nbr of entries	606	521	503	534	503	524	508	552	

In [3]:

```
posePartPairs={
    "Torso":[1, 8],
    "Shoulder (right)":[1, 2],
    "Shoulder (left)":[1, 5],
    "Arm (right)":[2, 3],
    "Forearm (right)":[3, 4],
    "Arm (left)":[5, 6],
    "Forearm (left)":[6, 7],
    "Hip (right)":[8, 9],
    "Thigh (right)":[9, 10],
    "Leg (right)":[10, 11],
    "Hip (left)":[8, 12],
    "Thigh (left)":[12, 13],
    "Leg (left)":[13, 14],
    "Neck":[1, 0],
    "Eye (right)":[0, 15],
    "Ear (right)":[15, 17],
    "Eye (left)":[0, 16],
    "Ear (left)":[16, 18],
    "Foot (left)":[14, 19],
    "Toe (left)":[19, 20],
    "Heel (left)":[14, 21],
    "Foot (right)":[11, 22],
    "Toe (right)":[22, 23],
    "Heel (right)":[11, 24],
}

color_map = plt.cm.get_cmap("nipy_spectral", len(posePartPairs))

fig, axs = plt.subplots(4, len(labels_body)//4, figsize=(12,10))
for ax, label in zip([item for sublist in axs for item in sublist], labels_body):
    samples = dataset_df_body[(dataset_df_body.label==label)].drop(['label', 'accuracy'], axis=
    sample_mean = samples.mean(axis=0)
    sample_rand = samples[np.random.randint(0,samples.shape[0])]
    sample_data_2D = np.stack([sample_rand[::2], sample_rand[1::2]]).T
    for i,p in enumerate(posePartPairs.values()):
        if np.all(sample_data_2D[p]):
            ax.plot(*sample_data_2D[p].T, c=color_map(i))
    ax.set_aspect("equal")
    ax.axis('off')
    ax.set_title(label.replace('_', ' '), fontsize=14)

handles = [Line2D([0], [0], color=color_map(i), lw=3, ls='-', label=l) for i, l in enumerate(po
fig.legend(handles=handles,
           loc='center left', bbox_to_anchor=(0.95, 0.5), prop={'size': 14},
           borderaxespad=1)
fig.tight_layout()

display(Markdown("### Class visualization"))
```

Class visualization



👍 Hand dataset

In [4]:

```
dataset_df_hand = pd.read_csv("https://raw.githubusercontent.com/ArthurFDLR/pose-classification
labels_hand = dataset_df_hand.label.unique()

display(Markdown("### Complete dataset view"))
display(dataset_df_hand)

display(Markdown("### Number of samples per label"))
df_hand_labels = {hand_i : dataset_df_hand.loc[dataset_df_hand['hand'] == hand_i].groupby('labe
display(
    pd.DataFrame(
        [df.size() for df in df_hand_labels.values()],
        columns=labels_hand,
        index=df_hand_labels.keys(),
        )
)
```

Complete dataset view

	label	hand	accuracy	x0	y0	x1	y1	x2	y2	x3	
0	0	left	15.139381	0.403383	-0.366229	0.194999	-0.361383	-0.013385	-0.264461	-0.149077	-0.162
1	0	left	14.065027	0.333266	-0.430984	0.169325	-0.368800	0.016691	-0.255737	-0.152903	-0.142
...	...	...	...	...	...	...	...	...	...	...	...
11202	Ok	right	16.430834	-0.262744	-0.477646	-0.070791	-0.408216	0.100741	-0.355123	0.215096	-0.314
11203	Ok	right	15.640315	-0.258065	-0.478518	-0.070381	-0.404227	0.105572	-0.353396	0.211144	-0.318

11204 rows × 45 columns

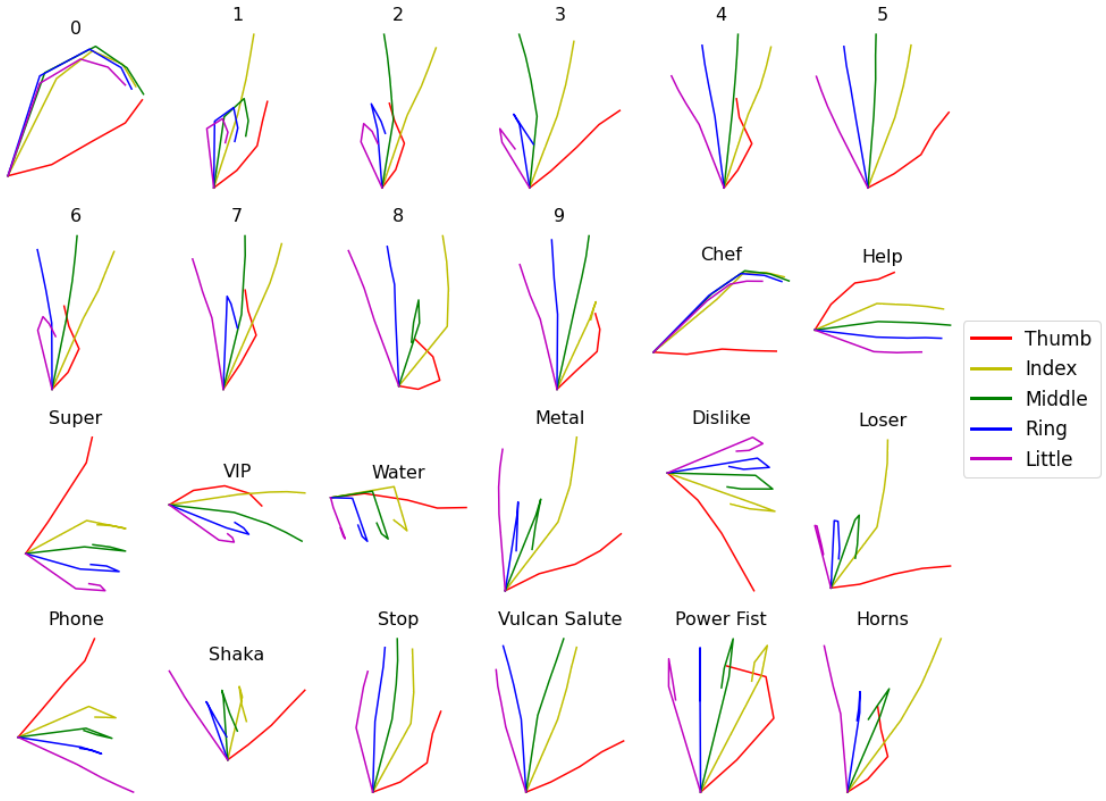
Number of samples per label

	0	1	2	3	4	5	6	7	8	9	Chef	Help	Super	VIP	Water	Metal	Dislike	Loser
left	207	201	213	203	204	206	208	202	241	221	205	235	203	202	207	208	205	206
right	208	207	203	205	213	206	202	202	205	209	206	209	208	221	204	214	206	205

In [5]:

```
colors = ["r", "y", "g", "b", "m"]
hand_visualization = 'right'

fig, axs = plt.subplots(4, len(labels_hand)//4, figsize=(12,10))
for ax, label_viz in zip([item for sublist in axs for item in sublist],labels_hand):
    sample_data = dataset_df_hand[(dataset_df_hand.label==label_viz) & (dataset_df_hand.hand==h
    sample_data_2D = np.stack([sample_data[::2], sample_data[1::2]])

    sample_fingers = {
        "Thumb": sample_data_2D[:, 0:5],
        "Index": np.insert(sample_data_2D[:, 5:9].T, 0, sample_data_2D[:, 0], axis=0).T,
        "Middle": np.insert(sample_data_2D[:, 9:13].T, 0, sample_data_2D[:, 0], axis=0).T,
        "Ring": np.insert(sample_data_2D[:, 13:17].T, 0, sample_data_2D[:, 0], axis=0).T,
        "Little": np.insert(sample_data_2D[:, 17:21].T, 0, sample_data_2D[:, 0], axis=0).T,
    }
    for (name, data), c in zip(sample_fingers.items(), colors):
        ax.plot(data[0], data[1], color=c)
    ax.set_aspect("equal")
    ax.set_title(label_viz.replace('_', ' '), fontsize=16)
    ax.axis('off')

handles = [Line2D([0], [0], color=colors[i], lw=3, ls='-', label=l) for i, l in enumerate(sampl
fig.legend(handles=handles,
           loc='center left', bbox_to_anchor=(0.97, 0.5), prop={'size': 17},
           borderaxespad=1)
fig.tight_layout()

display(Markdown("### Class visualization"))
```

Class visualization



D. Source code – PCK models creation

Open in Colab

Pose Classification Kit: Body pose classification model creation

This Notebook can be used to create Neural Network classifiers running in the Pose Classification Kit.

First, we have to import several libraries to create and train a new model.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.lines import Line2D
plt.style.use('ggplot')
import os

try:
    import google.colab
    IN_COLAB = True
except:
    IN_COLAB = False

if IN_COLAB:
    %tensorflow_version 2.x
    !pip install pose-classification-kit

import tensorflow
from tensorflow import keras
from pose_classification_kit.datasets import BODY18, bodyDataset, dataAugmentation
```

```
Requirement already satisfied: pose-classification-kit in /usr/local/lib/python3.7/dist-package
s (1.1.5)
Requirement already satisfied: numpy<1.20.0,>=1.19.2 in /usr/local/lib/python3.7/dist-packages
(from pose-classification-kit) (1.19.5)
Requirement already satisfied: pandas<2.0.0,>=1.1.5 in /usr/local/lib/python3.7/dist-packages
(from pose-classification-kit) (1.1.5)
Requirement already satisfied: tensorflow in /usr/local/lib/python3.7/dist-packages (from pose-
classification-kit) (2.5.0)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-packages (from pan
das<2.0.0,>=1.1.5->pose-classification-kit) (2018.9)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist-packages
(from pandas<2.0.0,>=1.1.5->pose-classification-kit) (2.8.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-
dateutil>=2.7.3->pandas<2.0.0,>=1.1.5->pose-classification-kit) (1.15.0)
Requirement already satisfied: wrapt~=1.12.1 in /usr/local/lib/python3.7/dist-packages (from te
nsorflow->pose-classification-kit) (1.12.1)
Requirement already satisfied: keras-nightly~=2.5.0.dev in /usr/local/lib/python3.7/dist-packag
es (from tensorflow->pose-classification-kit) (2.5.0.dev2021032900)
Requirement already satisfied: typing-extensions~=3.7.4 in /usr/local/lib/python3.7/dist-packag
es (from tensorflow->pose-classification-kit) (3.7.4.3)
Requirement already satisfied: tensorboard~=2.5 in /usr/local/lib/python3.7/dist-packages (from
tensorflow->pose-classification-kit) (2.5.0)
Requirement already satisfied: h5py~=3.1.0 in /usr/local/lib/python3.7/dist-packages (from tens
orflow->pose-classification-kit) (3.1.0)
Requirement already satisfied: flatbuffers~=1.12.0 in /usr/local/lib/python3.7/dist-packages (f
rom tensorflow->pose-classification-kit) (1.12)
Requirement already satisfied: keras-preprocessing~=1.1.2 in /usr/local/lib/python3.7/dist-pack
ages (from tensorflow->pose-classification-kit) (1.1.2)
Requirement already satisfied: tensorflow-estimator<2.6.0,>=2.5.0rc0 in /usr/local/lib/python3.
7/dist-packages (from tensorflow->pose-classification-kit) (2.5.0)
Requirement already satisfied: opt-einsum~=3.3.0 in /usr/local/lib/python3.7/dist-packages (fro
m tensorflow->pose-classification-kit) (3.3.0)
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/dist-packages (from
tensorflow->pose-classification-kit) (3.17.3)
Requirement already satisfied: google-pasta~=0.2 in /usr/local/lib/python3.7/dist-packages (fro
m tensorflow->pose-classification-kit) (0.2.0)
Requirement already satisfied: absl-py~=0.10 in /usr/local/lib/python3.7/dist-packages (from te
```

```
nsorflow->pose-classification-kit) (0.12.0)
Requirement already satisfied: wheel~=0.35 in /usr/local/lib/python3.7/dist-packages (from tens
orflow->pose-classification-kit) (0.36.2)
Requirement already satisfied: grpcio~=1.34.0 in /usr/local/lib/python3.7/dist-packages (from t
ensorflow->pose-classification-kit) (1.34.1)
Requirement already satisfied: astunparse~=1.6.3 in /usr/local/lib/python3.7/dist-packages (fro
m tensorflow->pose-classification-kit) (1.6.3)
Requirement already satisfied: termcolor~=1.1.0 in /usr/local/lib/python3.7/dist-packages (from
tensorflow->pose-classification-kit) (1.1.0)
Requirement already satisfied: gast==0.4.0 in /usr/local/lib/python3.7/dist-packages (from tens
orflow->pose-classification-kit) (0.4.0)
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-packages (from
h5py~=3.1.0->tensorflow->pose-classification-kit) (1.5.2)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/python3.7/dis
t-packages (from tensorboard~=2.5->tensorflow->pose-classification-kit) (0.4.4)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/python3.7/dist-p
ackages (from tensorboard~=2.5->tensorflow->pose-classification-kit) (1.8.0)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-packages (from
tensorboard~=2.5->tensorflow->pose-classification-kit) (3.3.4)
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.7/dist-packages (f
rom tensorboard~=2.5->tensorflow->pose-classification-kit) (2.23.0)
Requirement already satisfied: google-auth<2,>=1.6.3 in /usr/local/lib/python3.7/dist-packages
(from tensorboard~=2.5->tensorflow->pose-classification-kit) (1.32.1)
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/dist-packages (fro
m tensorboard~=2.5->tensorflow->pose-classification-kit) (1.0.1)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/local/lib/python3.
7/dist-packages (from tensorboard~=2.5->tensorflow->pose-classification-kit) (0.6.1)
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.7/dist-packages (fr
om tensorboard~=2.5->tensorflow->pose-classification-kit) (57.2.0)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.7/dist-packages
(from google-auth<2,>=1.6.3->tensorboard~=2.5->tensorflow->pose-classification-kit) (4.2.2)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-packages (from go
ogle-auth<2,>=1.6.3->tensorboard~=2.5->tensorflow->pose-classification-kit) (4.7.2)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7/dist-packages
(from google-auth<2,>=1.6.3->tensorboard~=2.5->tensorflow->pose-classification-kit) (0.2.8)
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.7/dist-packag
es (from google-auth-oauthlib<0.5,>=0.4.1->tensorboard~=2.5->tensorflow->pose-classification-ki
t) (1.3.0)
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/dist-packages (fr
om markdown>=2.6.8->tensorboard~=2.5->tensorflow->pose-classification-kit) (4.6.1)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist-packages
(from pyasn1-modules>=0.2.1->google-auth<2,>=1.6.3->tensorboard~=2.5->tensorflow->pose-classifi
cation-kit) (0.4.8)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (fr
om requests<3,>=2.21.0->tensorboard~=2.5->tensorflow->pose-classification-kit) (2021.5.30)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python
3.7/dist-packages (from requests<3,>=2.21.0->tensorboard~=2.5->tensorflow->pose-classification-
kit) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (fro
m requests<3,>=2.21.0->tensorboard~=2.5->tensorflow->pose-classification-kit) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from req
uests<3,>=2.21.0->tensorboard~=2.5->tensorflow->pose-classification-kit) (2.10)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from
requests-oauthlib>=0.7.0->google-auth-oauthlib<0.5,>=0.4.1->tensorboard~=2.5->tensorflow->pose-
classification-kit) (3.1.1)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from import
lib-metadata->markdown>=2.6.8->tensorboard~=2.5->tensorflow->pose-classification-kit) (3.5.0)
```

Import dataset

In [2]:

```
dataset = bodyDataset(testSplit=.2, shuffle=True, bodyModel=BODY18)
x_train = dataset['x_train']
y_train = dataset['y_train_onehot']

x_train.shape, y_train.shape
```

Out[2]:

```
((8040, 18, 2), (8040, 20))
```

Data augmentation

In [3]:

```
x, y = [x_train], [y_train]

# Scaling augmentation
x[len(x):],y[len(y):] = tuple(zip(dataAugmentation(
    x_train, y_train,
    augmentation_ratio=.1,
    scaling_factor_standard_deviation=.08,
    #random_noise_standard_deviation=.03,
)))

# Rotation augmentation
x[len(x):],y[len(y):] = tuple(zip(dataAugmentation(
    x_train, y_train,
    augmentation_ratio=.1,
    rotation_angle_standard_deviation=10,
    #random_noise_standard_deviation=.03
)))

# Upper-body augmentation
lowerBody_keypoints = np.where(np.isin(BODY18.mapping,[
    "left_knee", "right_knee", "left_ankle", "right_ankle" #,"left_hip", "right_hip",
]))[0]
x[len(x):],y[len(y):] = tuple(zip(dataAugmentation(
    x_train, y_train,
    augmentation_ratio=.15,
    remove_specific_keypoints=lowerBody_keypoints,
    random_noise_standard_deviation=.03
)))
lowerBody_keypoints = np.where(np.isin(BODY18.mapping,[
    "left_knee", "right_knee", "left_ankle", "right_ankle", "left_hip", "right_hip",
]))[0]
x[len(x):],y[len(y):] = tuple(zip(dataAugmentation(
    x_train, y_train,
    augmentation_ratio=.15,
    remove_specific_keypoints=lowerBody_keypoints,
    random_noise_standard_deviation=.03
)))

# Random partial input augmentation
x[len(x):],y[len(y):] = tuple(zip(dataAugmentation(
    x_train, y_train,
    augmentation_ratio=.2,
    remove_rand_keypoints_nbr=2,
    random_noise_standard_deviation=.03
)))

x_train_augmented = np.concatenate(x, axis=0)
y_train_augmented = np.concatenate(y, axis=0)

x_train_augmented.shape, y_train_augmented.shape
```

Out[3]:

```
((13668, 18, 2), (13668, 20))
```

Models exploration

This section is optional. The following blocks can be used to compare different architecture and training processes.

In [4]:

```
model_train_history = {}
input_dim = x_train.shape[1:]
output_dim = len(dataset['labels'])
validation_split = 0.20
epochs = 15
```

Architectures comparison

In [5]:
```
model = keras.models.Sequential(
    name = 'Light_ANN',
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Flatten(),
        keras.layers.Dense(48, activation=keras.activations.relu),
        keras.layers.Dense(48, activation=keras.activations.relu),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
)

model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

model_train_history[model] = model.fit(
    x=x_train,
    y=y_train,
    epochs=epochs,
    batch_size=16,
    validation_split=validation_split,
    shuffle=True,
    verbose=1,
)
```

```
Model: "Light_ANN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 36)                0
_________________________________________________________________
dense (Dense)                (None, 48)                1776
_________________________________________________________________
dense_1 (Dense)              (None, 48)                2352
_________________________________________________________________
dense_2 (Dense)              (None, 20)                980
=================================================================
Total params: 5,108
Trainable params: 5,108
Non-trainable params: 0
_________________________________________________________________
Epoch 1/15
402/402 [==============================] - 1s 2ms/step - loss: 1.9572 - accuracy: 0.5227 - val_
loss: 0.9086 - val_accuracy: 0.8234
Epoch 2/15
402/402 [==============================] - 0s 1ms/step - loss: 0.5643 - accuracy: 0.8831 - val_
loss: 0.3706 - val_accuracy: 0.9272
Epoch 3/15
402/402 [==============================] - 1s 1ms/step - loss: 0.2972 - accuracy: 0.9286 - val_
loss: 0.2417 - val_accuracy: 0.9484
Epoch 4/15
402/402 [==============================] - 0s 1ms/step - loss: 0.2056 - accuracy: 0.9490 - val_
loss: 0.1754 - val_accuracy: 0.9683
Epoch 5/15
402/402 [==============================] - 0s 1ms/step - loss: 0.1593 - accuracy: 0.9639 - val_
loss: 0.1326 - val_accuracy: 0.9733
Epoch 6/15
402/402 [==============================] - 0s 1ms/step - loss: 0.1252 - accuracy: 0.9739 - val_
loss: 0.1131 - val_accuracy: 0.9782
Epoch 7/15
402/402 [==============================] - 1s 1ms/step - loss: 0.1052 - accuracy: 0.9768 - val_
loss: 0.0902 - val_accuracy: 0.9820
Epoch 8/15
402/402 [==============================] - 1s 1ms/step - loss: 0.0873 - accuracy: 0.9821 - val_
loss: 0.0854 - val_accuracy: 0.9807
Epoch 9/15
402/402 [==============================] - 0s 1ms/step - loss: 0.0771 - accuracy: 0.9831 - val_
loss: 0.0686 - val_accuracy: 0.9876
```

```
Epoch 10/15
402/402 [==============================] - 0s 1ms/step - loss: 0.0673 - accuracy: 0.9851 - val_
loss: 0.0599 - val_accuracy: 0.9882
Epoch 11/15
402/402 [==============================] - 0s 1ms/step - loss: 0.0592 - accuracy: 0.9866 - val_
loss: 0.0554 - val_accuracy: 0.9894
Epoch 12/15
402/402 [==============================] - 0s 1ms/step - loss: 0.0534 - accuracy: 0.9882 - val_
loss: 0.0518 - val_accuracy: 0.9882
Epoch 13/15
402/402 [==============================] - 0s 1ms/step - loss: 0.0492 - accuracy: 0.9894 - val_
loss: 0.0472 - val_accuracy: 0.9907
Epoch 14/15
402/402 [==============================] - 0s 1ms/step - loss: 0.0459 - accuracy: 0.9882 - val_
loss: 0.0463 - val_accuracy: 0.9913
Epoch 15/15
402/402 [==============================] - 0s 1ms/step - loss: 0.0431 - accuracy: 0.9914 - val_
loss: 0.0430 - val_accuracy: 0.9869
```

In [6]:

```
model = keras.models.Sequential(
    name = 'Light_CNN',
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Conv1D(12, 3, activation='relu'),
        keras.layers.Conv1D(12, 3, activation='relu'),
        keras.layers.Flatten(),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
)

model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

model_train_history[model] = model.fit(
    x=x_train,
    y=y_train,
    epochs=epochs,
    batch_size=16,
    validation_split=validation_split,
    shuffle=True,
    verbose=1,
)
```

```
Model: "Light_CNN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 16, 12)            84
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 14, 12)            444
_________________________________________________________________
flatten_1 (Flatten)          (None, 168)               0
_________________________________________________________________
dense_3 (Dense)              (None, 20)                3380
=================================================================
Total params: 3,908
Trainable params: 3,908
Non-trainable params: 0
_________________________________________________________________
Epoch 1/15
402/402 [==============================] - 1s 2ms/step - loss: 1.8955 - accuracy: 0.4845 - val_
loss: 0.8772 - val_accuracy: 0.7519
Epoch 2/15
402/402 [==============================] - 1s 2ms/step - loss: 0.5850 - accuracy: 0.8424 - val_
loss: 0.4157 - val_accuracy: 0.8818
Epoch 3/15
```

```
402/402 [==============================] - 1s 2ms/step - loss: 0.3278 - accuracy: 0.9145 - val_
loss: 0.2582 - val_accuracy: 0.9117
Epoch 4/15
402/402 [==============================] - 1s 2ms/step - loss: 0.2333 - accuracy: 0.9411 - val_
loss: 0.1957 - val_accuracy: 0.9590
Epoch 5/15
402/402 [==============================] - 1s 2ms/step - loss: 0.1842 - accuracy: 0.9532 - val_
loss: 0.1588 - val_accuracy: 0.9608
Epoch 6/15
402/402 [==============================] - 1s 2ms/step - loss: 0.1540 - accuracy: 0.9664 - val_
loss: 0.1273 - val_accuracy: 0.9720
Epoch 7/15
402/402 [==============================] - 1s 2ms/step - loss: 0.1272 - accuracy: 0.9726 - val_
loss: 0.1046 - val_accuracy: 0.9832
Epoch 8/15
402/402 [==============================] - 1s 2ms/step - loss: 0.1065 - accuracy: 0.9792 - val_
loss: 0.0938 - val_accuracy: 0.9832
Epoch 9/15
402/402 [==============================] - 1s 2ms/step - loss: 0.0903 - accuracy: 0.9846 - val_
loss: 0.0759 - val_accuracy: 0.9832
Epoch 10/15
402/402 [==============================] - 1s 1ms/step - loss: 0.0843 - accuracy: 0.9835 - val_
loss: 0.0705 - val_accuracy: 0.9845
Epoch 11/15
402/402 [==============================] - 1s 1ms/step - loss: 0.0723 - accuracy: 0.9873 - val_
loss: 0.0714 - val_accuracy: 0.9863
Epoch 12/15
402/402 [==============================] - 1s 2ms/step - loss: 0.0683 - accuracy: 0.9874 - val_
loss: 0.0661 - val_accuracy: 0.9857
Epoch 13/15
402/402 [==============================] - 1s 2ms/step - loss: 0.0612 - accuracy: 0.9896 - val_
loss: 0.0566 - val_accuracy: 0.9882
Epoch 14/15
402/402 [==============================] - 1s 2ms/step - loss: 0.0579 - accuracy: 0.9887 - val_
loss: 0.0509 - val_accuracy: 0.9907
Epoch 15/15
402/402 [==============================] - 1s 1ms/step - loss: 0.0526 - accuracy: 0.9910 - val_
loss: 0.0653 - val_accuracy: 0.9813
```

Data augmented training

In [7]:

```
model = keras.models.Sequential(
    name = 'Light_ANN_Augmented',
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Flatten(),
        keras.layers.Dense(48, activation=keras.activations.relu),
        keras.layers.Dense(48, activation=keras.activations.relu),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
)

model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

model_train_history[model] = model.fit(
    x=x_train_augmented,
    y=y_train_augmented,
    epochs=epochs,
    batch_size=16,
    validation_split=validation_split,
    shuffle=True,
    verbose=1,
)
```

```
Model: "Light_ANN_Augmented"
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 36)                0
_________________________________________________________________
dense_4 (Dense)              (None, 48)                1776
_________________________________________________________________
dense_5 (Dense)              (None, 48)                2352
_________________________________________________________________
dense_6 (Dense)              (None, 20)                980
=================================================================
Total params: 5,108
Trainable params: 5,108
Non-trainable params: 0
_________________________________________________________________
Epoch 1/15
684/684 [==============================] - 1s 1ms/step - loss: 1.5349 - accuracy: 0.5920 - val_
loss: 0.9071 - val_accuracy: 0.7169
Epoch 2/15
684/684 [==============================] - 1s 1ms/step - loss: 0.4119 - accuracy: 0.8935 - val_
loss: 0.6899 - val_accuracy: 0.8003
Epoch 3/15
684/684 [==============================] - 1s 1ms/step - loss: 0.2479 - accuracy: 0.9383 - val_
loss: 0.6801 - val_accuracy: 0.8036
Epoch 4/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1827 - accuracy: 0.9576 - val_
loss: 0.7066 - val_accuracy: 0.8168
Epoch 5/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1470 - accuracy: 0.9645 - val_
loss: 0.6911 - val_accuracy: 0.8365
Epoch 6/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1227 - accuracy: 0.9706 - val_
loss: 0.7074 - val_accuracy: 0.8442
Epoch 7/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1061 - accuracy: 0.9758 - val_
loss: 0.7147 - val_accuracy: 0.8394
Epoch 8/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0961 - accuracy: 0.9775 - val_
loss: 0.7199 - val_accuracy: 0.8413
Epoch 9/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0863 - accuracy: 0.9791 - val_
loss: 0.7200 - val_accuracy: 0.8453
Epoch 10/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0777 - accuracy: 0.9802 - val_
loss: 0.7607 - val_accuracy: 0.8369
Epoch 11/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0738 - accuracy: 0.9816 - val_
loss: 0.7743 - val_accuracy: 0.8347
Epoch 12/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0670 - accuracy: 0.9828 - val_
loss: 0.8349 - val_accuracy: 0.8266
Epoch 13/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0634 - accuracy: 0.9834 - val_
loss: 0.7949 - val_accuracy: 0.8189
Epoch 14/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0593 - accuracy: 0.9854 - val_
loss: 0.7936 - val_accuracy: 0.8197
Epoch 15/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0540 - accuracy: 0.9864 - val_
loss: 0.8059 - val_accuracy: 0.8376
```

In [8]:

```
model = keras.models.Sequential(
    name = 'Light_CNN_Augmented',
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Flatten(),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
)
```

```
model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

model_train_history[model] = model.fit(
    x=x_train_augmented,
    y=y_train_augmented,
    epochs=epochs,
    batch_size=16,
    validation_split=validation_split,
    shuffle=True,
    verbose=1,
)
```

```
Model: "Light_CNN_Augmented"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_2 (Conv1D)            (None, 16, 16)            112
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 14, 16)            784
_________________________________________________________________
flatten_3 (Flatten)          (None, 224)               0
_________________________________________________________________
dense_7 (Dense)              (None, 20)                4500
=================================================================
Total params: 5,396
Trainable params: 5,396
Non-trainable params: 0
_________________________________________________________________
Epoch 1/15
684/684 [==============================] - 1s 2ms/step - loss: 1.2583 - accuracy: 0.6709 - val_
loss: 0.8504 - val_accuracy: 0.7341
Epoch 2/15
684/684 [==============================] - 1s 1ms/step - loss: 0.2967 - accuracy: 0.9260 - val_
loss: 0.8033 - val_accuracy: 0.7868
Epoch 3/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1851 - accuracy: 0.9526 - val_
loss: 0.7896 - val_accuracy: 0.8116
Epoch 4/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1409 - accuracy: 0.9641 - val_
loss: 0.8599 - val_accuracy: 0.8032
Epoch 5/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1160 - accuracy: 0.9720 - val_
loss: 0.8927 - val_accuracy: 0.7871
Epoch 6/15
684/684 [==============================] - 1s 1ms/step - loss: 0.1022 - accuracy: 0.9732 - val_
loss: 0.9288 - val_accuracy: 0.7871
Epoch 7/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0920 - accuracy: 0.9763 - val_
loss: 0.9514 - val_accuracy: 0.8029
Epoch 8/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0844 - accuracy: 0.9790 - val_
loss: 1.0236 - val_accuracy: 0.7948
Epoch 9/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0750 - accuracy: 0.9808 - val_
loss: 0.9760 - val_accuracy: 0.8091
Epoch 10/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0747 - accuracy: 0.9795 - val_
loss: 1.0604 - val_accuracy: 0.8168
Epoch 11/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0695 - accuracy: 0.9815 - val_
loss: 1.0075 - val_accuracy: 0.8226
Epoch 12/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0639 - accuracy: 0.9823 - val_
loss: 1.0814 - val_accuracy: 0.7999
Epoch 13/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0620 - accuracy: 0.9823 - val_
```

```
loss: 1.0502 - val_accuracy: 0.8175
Epoch 14/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0582 - accuracy: 0.9844 - val_
loss: 1.0093 - val_accuracy: 0.8325
Epoch 15/15
684/684 [==============================] - 1s 1ms/step - loss: 0.0563 - accuracy: 0.9855 - val_
loss: 1.0089 - val_accuracy: 0.8296
```

In [9]:

```
model = keras.models.Sequential(
    name = 'Hybrid_Augmented',
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Flatten(),
        keras.layers.Dense(128, activation=keras.activations.relu),
        keras.layers.Dense(128, activation=keras.activations.relu),
        keras.layers.Dense(64, activation=keras.activations.relu),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
)

model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

model_train_history[model] = model.fit(
    x=x_train_augmented,
    y=y_train_augmented,
    epochs=epochs,
    batch_size=16,
    validation_split=validation_split,
    shuffle=True,
    verbose=1,
)
```

```
Model: "Hybrid_Augmented"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_4 (Conv1D)            (None, 16, 16)            112
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 14, 16)            784
_________________________________________________________________
flatten_4 (Flatten)          (None, 224)               0
_________________________________________________________________
dense_8 (Dense)              (None, 128)               28800
_________________________________________________________________
dense_9 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_10 (Dense)             (None, 64)                8256
_________________________________________________________________
dense_11 (Dense)             (None, 20)                1300
=================================================================
Total params: 55,764
Trainable params: 55,764
Non-trainable params: 0
_________________________________________________________________
Epoch 1/15
684/684 [==============================] - 2s 2ms/step - loss: 0.8132 - accuracy: 0.7384 - val_
loss: 1.1376 - val_accuracy: 0.7107
Epoch 2/15
684/684 [==============================] - 1s 2ms/step - loss: 0.2218 - accuracy: 0.9267 - val_
loss: 1.1774 - val_accuracy: 0.7462
Epoch 3/15
684/684 [==============================] - 1s 2ms/step - loss: 0.1512 - accuracy: 0.9532 - val_
loss: 1.0959 - val_accuracy: 0.7835
```

```
Epoch 4/15
684/684 [==============================] - 1s 2ms/step - loss: 0.1145 - accuracy: 0.9639 - val_
loss: 1.3354 - val_accuracy: 0.7377
Epoch 5/15
684/684 [==============================] - 1s 2ms/step - loss: 0.1019 - accuracy: 0.9667 - val_
loss: 1.1903 - val_accuracy: 0.7751
Epoch 6/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0843 - accuracy: 0.9744 - val_
loss: 1.1088 - val_accuracy: 0.8076
Epoch 7/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0720 - accuracy: 0.9766 - val_
loss: 1.1928 - val_accuracy: 0.7879
Epoch 8/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0595 - accuracy: 0.9833 - val_
loss: 1.1801 - val_accuracy: 0.7853
Epoch 9/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0539 - accuracy: 0.9842 - val_
loss: 1.2253 - val_accuracy: 0.7966
Epoch 10/15
684/684 [==============================] - 2s 2ms/step - loss: 0.0597 - accuracy: 0.9817 - val_
loss: 1.2329 - val_accuracy: 0.8091
Epoch 11/15
684/684 [==============================] - 2s 2ms/step - loss: 0.0502 - accuracy: 0.9846 - val_
loss: 1.3763 - val_accuracy: 0.7798
Epoch 12/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0486 - accuracy: 0.9847 - val_
loss: 1.2473 - val_accuracy: 0.7999
Epoch 13/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0507 - accuracy: 0.9848 - val_
loss: 1.2375 - val_accuracy: 0.7992
Epoch 14/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0414 - accuracy: 0.9871 - val_
loss: 1.2952 - val_accuracy: 0.7955
Epoch 15/15
684/684 [==============================] - 1s 2ms/step - loss: 0.0389 - accuracy: 0.9882 - val_
loss: 1.3309 - val_accuracy: 0.8032
```

Regularization

In [10]:

```
model = keras.models.Sequential(
    name = 'Hybrid_Augmented_Regularized',
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Dropout(.2),
        keras.layers.BatchNormalization(),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Dropout(.2),
        keras.layers.BatchNormalization(),
        keras.layers.Flatten(),
        keras.layers.Dense(128, activation=keras.activations.relu),
        keras.layers.Dropout(.3),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(128, activation=keras.activations.relu),
        keras.layers.Dropout(.3),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(64, activation=keras.activations.relu),
        keras.layers.Dropout(.3),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
)

model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)
```

```
model_train_history[model] = model.fit(
    x=x_train_augmented,
    y=y_train_augmented,
    epochs=epochs,
    batch_size=16,
    validation_split=validation_split,
    shuffle=True,
    verbose=1,
)
```

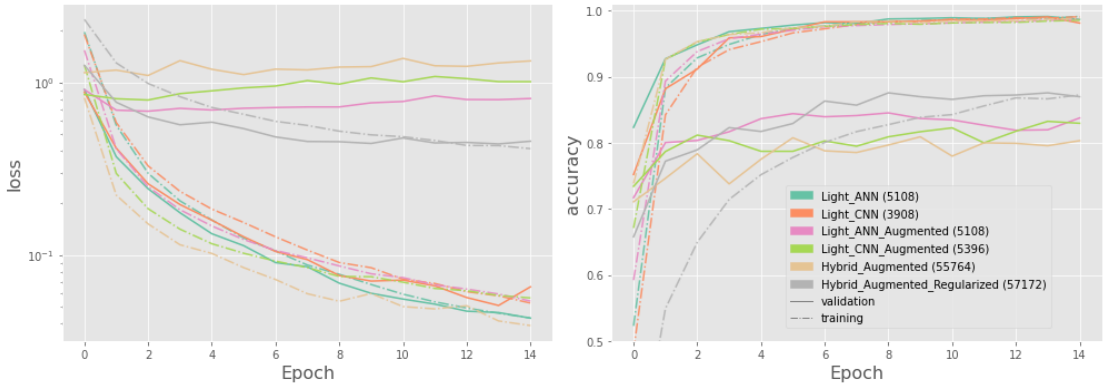
```
Model: "Hybrid_Augmented_Regularized"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_6 (Conv1D)            (None, 16, 16)            112
_________________________________________________________________
dropout (Dropout)            (None, 16, 16)            0
_________________________________________________________________
batch_normalization (BatchNo (None, 16, 16)            64
_________________________________________________________________
conv1d_7 (Conv1D)            (None, 14, 16)            784
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 16)            0
_________________________________________________________________
batch_normalization_1 (Batch (None, 14, 16)            64
_________________________________________________________________
flatten_5 (Flatten)          (None, 224)               0
_________________________________________________________________
dense_12 (Dense)             (None, 128)               28800
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dense_13 (Dense)             (None, 128)               16512
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
batch_normalization_3 (Batch (None, 128)               512
_________________________________________________________________
dense_14 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_4 (Dropout)          (None, 64)                0
_________________________________________________________________
batch_normalization_4 (Batch (None, 64)                256
_________________________________________________________________
dense_15 (Dense)             (None, 20)                1300
=================================================================
Total params: 57,172
Trainable params: 56,468
Non-trainable params: 704
_________________________________________________________________
Epoch 1/15
684/684 [==============================] - 4s 4ms/step - loss: 2.3149 - accuracy: 0.2778 - val_
loss: 1.2543 - val_accuracy: 0.6576
Epoch 2/15
684/684 [==============================] - 2s 3ms/step - loss: 1.2960 - accuracy: 0.5488 - val_
loss: 0.7668 - val_accuracy: 0.7721
Epoch 3/15
684/684 [==============================] - 2s 3ms/step - loss: 0.9847 - accuracy: 0.6494 - val_
loss: 0.6301 - val_accuracy: 0.7893
Epoch 4/15
684/684 [==============================] - 2s 4ms/step - loss: 0.8262 - accuracy: 0.7142 - val_
loss: 0.5676 - val_accuracy: 0.8230
Epoch 5/15
684/684 [==============================] - 2s 3ms/step - loss: 0.7150 - accuracy: 0.7517 - val_
loss: 0.5876 - val_accuracy: 0.8171
Epoch 6/15
684/684 [==============================] - 2s 3ms/step - loss: 0.6527 - accuracy: 0.7778 - val_
loss: 0.5391 - val_accuracy: 0.8292
Epoch 7/15
684/684 [==============================] - 2s 3ms/step - loss: 0.5940 - accuracy: 0.8005 - val_
```

```
loss: 0.4826 - val_accuracy: 0.8632
Epoch 8/15
684/684 [==============================] - 2s 3ms/step - loss: 0.5628 - accuracy: 0.8169 - val_
loss: 0.4542 - val_accuracy: 0.8570
Epoch 9/15
684/684 [==============================] - 2s 3ms/step - loss: 0.5210 - accuracy: 0.8275 - val_
loss: 0.4527 - val_accuracy: 0.8756
Epoch 10/15
684/684 [==============================] - 2s 4ms/step - loss: 0.4961 - accuracy: 0.8386 - val_
loss: 0.4424 - val_accuracy: 0.8698
Epoch 11/15
684/684 [==============================] - 2s 3ms/step - loss: 0.4838 - accuracy: 0.8427 - val_
loss: 0.4758 - val_accuracy: 0.8658
Epoch 12/15
684/684 [==============================] - 2s 3ms/step - loss: 0.4603 - accuracy: 0.8559 - val_
loss: 0.4454 - val_accuracy: 0.8713
Epoch 13/15
684/684 [==============================] - 2s 3ms/step - loss: 0.4307 - accuracy: 0.8680 - val_
loss: 0.4485 - val_accuracy: 0.8723
Epoch 14/15
684/684 [==============================] - 2s 3ms/step - loss: 0.4298 - accuracy: 0.8666 - val_
loss: 0.4395 - val_accuracy: 0.8756
Epoch 15/15
684/684 [==============================] - 2s 3ms/step - loss: 0.4127 - accuracy: 0.8725 - val_
loss: 0.4558 - val_accuracy: 0.8698
```

Comparison

In [11]:

```
fig, axs = plt.subplots(1, 2, figsize=(14,5))
colors_graph = plt.cm.get_cmap("Set2", len(model_train_history))
handles = []

for i, (model, history) in enumerate(model_train_history.items()):
    color = colors_graph(i)
    label = '{} ({})'.format(model.name, model.count_params())
    axs[0].plot(history.history['loss'], c=color, ls='-.', alpha=.9)
    axs[1].plot(history.history['accuracy'], c=color, ls='-.', alpha=.9)
    axs[0].plot(history.history['val_loss'], c=color)
    axs[1].plot(history.history['val_accuracy'], c=color)
    handles.append(mpatches.Patch(color=color, label=label))

for ax in axs:
    ax.set_xlabel('Epoch', fontsize=16)
axs[0].set_ylabel('loss', fontsize=16)
axs[0].set_yscale('log')
axs[1].set_ylabel('accuracy', fontsize=16)
axs[1].set_ylim(0.5,1.01)

handles.append(Line2D([0], [0], color='grey', lw=1, ls='-', label='validation'))
handles.append(Line2D([0], [0], color='grey', lw=1, ls='-.', label='training'))

fig.subplots_adjust(right=0.85)
fig.legend(handles=handles,
           loc="center right",
           borderaxespad=1,
           bbox_to_anchor=(.95,0.33))
fig.tight_layout()
```



Model export

Once you have a good model, you can save it on your Google Drive or local files. A model information JSON file is also added to store labels.

In [12]:

```
from pathlib import Path
import json

if IN_COLAB:
    content_path = Path('/').absolute() / 'content'
    drive_path = content_path / 'drive'
    google.colab.drive.mount(str(drive_path))
    save_path = drive_path / 'My Drive'

    for subfolder in ['Pose Classification Kit', 'Models']:
        save_path /= subfolder
        if not (save_path).is_dir():
            %mkdir "{save_path}"
else:
    save_path = Path('.').absolute()
    %mkdir "{save_path}"
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/con
tent/drive", force_remount=True).
```

In [13]:

```
import itertools

x_test_upper, y_test_upper = dataAugmentation(
    dataset['x_test'], dataset['y_test_onehot'],
    augmentation_ratio = 1.,
    remove_specific_keypoints = np.where(np.isin(BODY18.mapping,[
        "left_knee", "right_knee", "left_ankle", "right_ankle", "left_hip", "right_hip"
        ]))[0],
)

def plot_cm(labels, confusion_matrix):
    fig, ax = plt.subplots(figsize=(10,10), dpi=100)

    ax.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.Blues)
    tick_marks = np.arange(len(labels))
    ax.grid(False)
    ax.set_xticks(tick_marks)
    ax.set_xticklabels(labels, rotation=40, ha='right')
    ax.set_yticks(tick_marks)
    ax.set_yticklabels(labels)

    thresh = np.max(confusion_matrix) / 2.
    for i, j in itertools.product(range(confusion_matrix.shape[0]), range(confusion_matrix.shap
        plt.text(j, i, confusion_matrix[i, j],
            horizontalalignment="center",
            color="white" if confusion_matrix[i, j] > thresh else "black")

    fig.tight_layout()
```

```
    ax.set_ylabel('True label', fontsize=12)
    ax.set_xlabel('Predicted label', fontsize=12)
```

Light weight model

In [14]:

```
model_name = 'LightWeight_CNN_BODY18'
model_path = save_path / '{name}.h5'.format(name = model_name)

model = keras.models.Sequential(
    name = model_name,
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Conv1D(12, 3, activation='relu'),
        keras.layers.Conv1D(12, 3, activation='relu'),
        keras.layers.Flatten(),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
)

model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

model.fit(
    x=x_train,
    y=y_train,
    epochs=15,
    batch_size=16,
    validation_split=0.15,
    shuffle=True,
    callbacks=[keras.callbacks.ModelCheckpoint(filepath=model_path, verbose=2, save_best_only=T
    verbose = 2,
)

with open(save_path / (model_name+'_info.json'), 'w') as f:
    json.dump({'labels':dataset['labels']}, f)
```

```
Model: "LightWeight_CNN_BODY18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_8 (Conv1D)            (None, 16, 12)            84
_________________________________________________________________
conv1d_9 (Conv1D)            (None, 14, 12)            444
_________________________________________________________________
flatten_6 (Flatten)          (None, 168)               0
_________________________________________________________________
dense_16 (Dense)             (None, 20)                3380
=================================================================
Total params: 3,908
Trainable params: 3,908
Non-trainable params: 0
_________________________________________________________________
Epoch 1/15
428/428 - 1s - loss: 1.8760 - accuracy: 0.5003 - val_loss: 0.8079 - val_accuracy: 0.7720

Epoch 00001: val_loss improved from inf to 0.80791, saving model to /content/drive/My Drive/Pos
e Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 2/15
428/428 - 1s - loss: 0.5568 - accuracy: 0.8515 - val_loss: 0.4053 - val_accuracy: 0.8922

Epoch 00002: val_loss improved from 0.80791 to 0.40527, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 3/15
428/428 - 0s - loss: 0.3462 - accuracy: 0.9061 - val_loss: 0.2890 - val_accuracy: 0.9080
```

```
Epoch 00003: val_loss improved from 0.40527 to 0.28896, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 4/15
428/428 - 0s - loss: 0.2631 - accuracy: 0.9318 - val_loss: 0.2234 - val_accuracy: 0.9494

Epoch 00004: val_loss improved from 0.28896 to 0.22336, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 5/15
428/428 - 0s - loss: 0.2135 - accuracy: 0.9453 - val_loss: 0.1846 - val_accuracy: 0.9627

Epoch 00005: val_loss improved from 0.22336 to 0.18459, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 6/15
428/428 - 0s - loss: 0.1820 - accuracy: 0.9574 - val_loss: 0.1715 - val_accuracy: 0.9585

Epoch 00006: val_loss improved from 0.18459 to 0.17152, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 7/15
428/428 - 1s - loss: 0.1578 - accuracy: 0.9637 - val_loss: 0.1419 - val_accuracy: 0.9701

Epoch 00007: val_loss improved from 0.17152 to 0.14186, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 8/15
428/428 - 0s - loss: 0.1406 - accuracy: 0.9674 - val_loss: 0.1241 - val_accuracy: 0.9693

Epoch 00008: val_loss improved from 0.14186 to 0.12411, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 9/15
428/428 - 0s - loss: 0.1205 - accuracy: 0.9737 - val_loss: 0.1179 - val_accuracy: 0.9726

Epoch 00009: val_loss improved from 0.12411 to 0.11793, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 10/15
428/428 - 0s - loss: 0.1087 - accuracy: 0.9773 - val_loss: 0.1101 - val_accuracy: 0.9776

Epoch 00010: val_loss improved from 0.11793 to 0.11008, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 11/15
428/428 - 0s - loss: 0.1056 - accuracy: 0.9775 - val_loss: 0.1034 - val_accuracy: 0.9768

Epoch 00011: val_loss improved from 0.11008 to 0.10335, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 12/15
428/428 - 0s - loss: 0.0886 - accuracy: 0.9835 - val_loss: 0.1110 - val_accuracy: 0.9743

Epoch 00012: val_loss did not improve from 0.10335
Epoch 13/15
428/428 - 1s - loss: 0.0842 - accuracy: 0.9846 - val_loss: 0.0875 - val_accuracy: 0.9867

Epoch 00013: val_loss improved from 0.10335 to 0.08746, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
Epoch 14/15
428/428 - 1s - loss: 0.0762 - accuracy: 0.9854 - val_loss: 0.1111 - val_accuracy: 0.9668

Epoch 00014: val_loss did not improve from 0.08746
Epoch 15/15
428/428 - 0s - loss: 0.0716 - accuracy: 0.9862 - val_loss: 0.0724 - val_accuracy: 0.9818

Epoch 00015: val_loss improved from 0.08746 to 0.07245, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/LightWeight_CNN_BODY18.h5
```

In [15]:

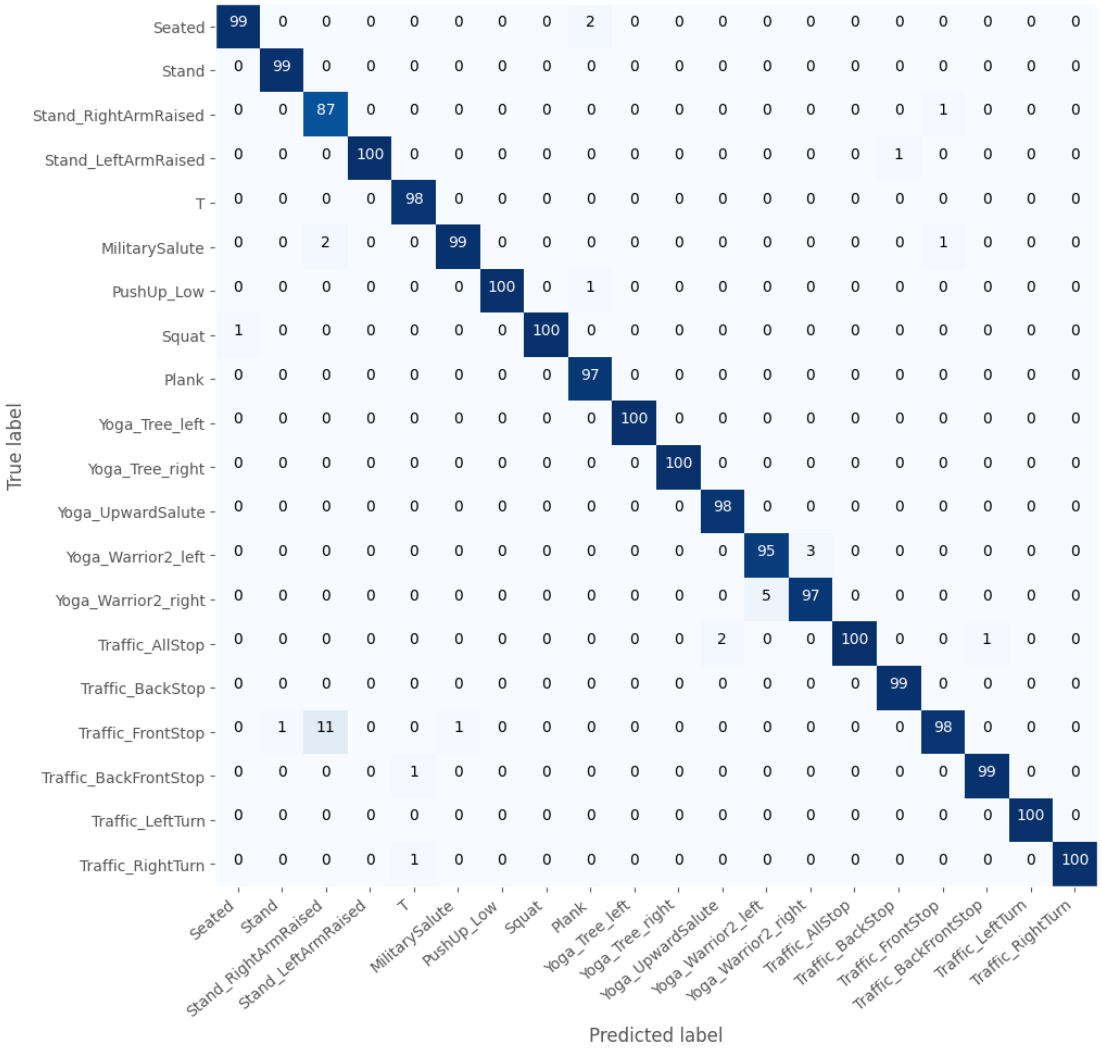
```
model = keras.models.load_model(model_path)

print(model.name + " accuracy on full samples:")
model.evaluate(x=dataset['x_test'], y=dataset['y_test_onehot'])
print(model.name + " accuracy on partial samples:")
model.evaluate(x=x_test_upper, y=y_test_upper)
```

```
LightWeight_CNN_BODY18 accuracy on full samples:
63/63 [==============================] - 0s 929us/step - loss: 0.0583 - accuracy: 0.9825
```

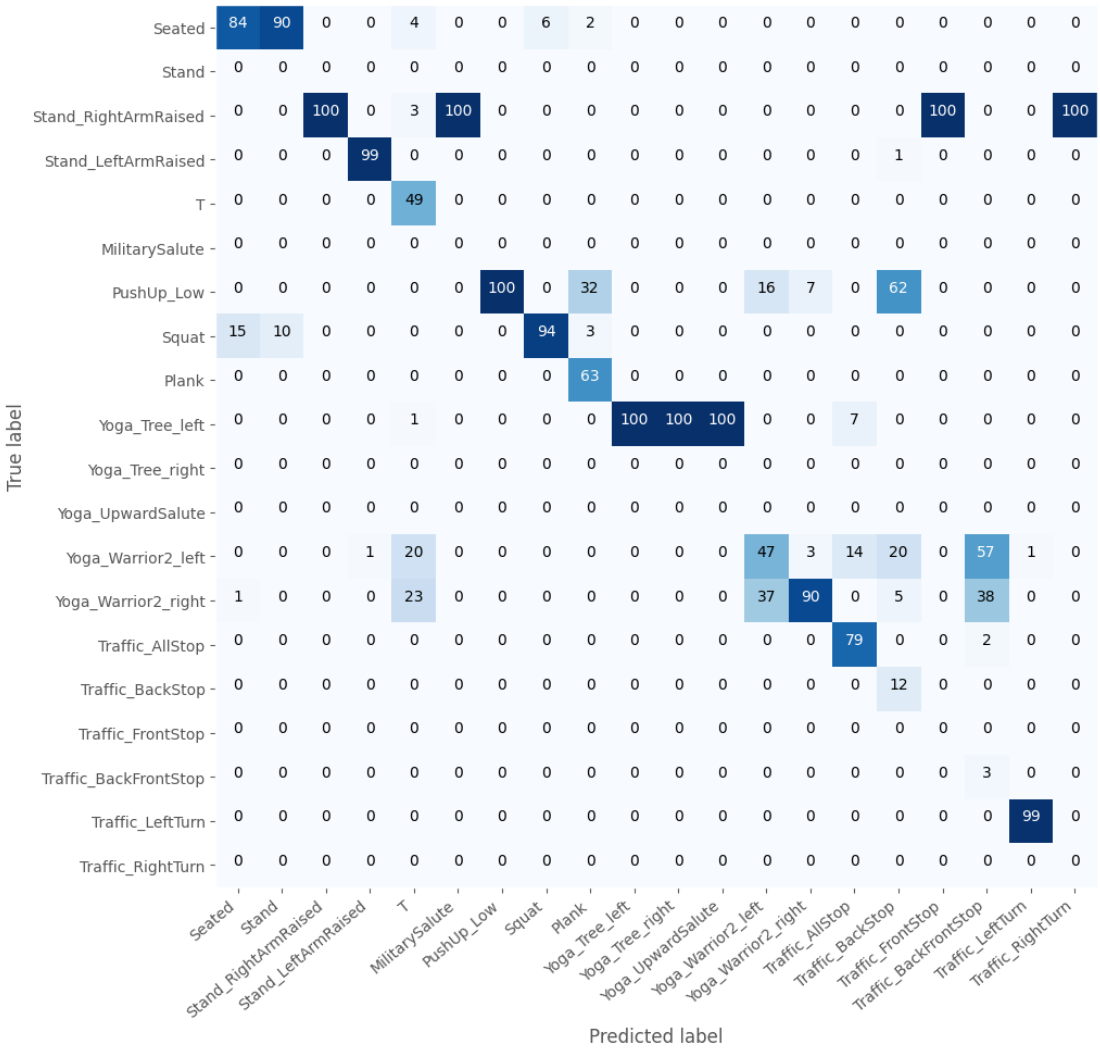
```
LightWeight_CNN_BODY18 accuracy on partial samples:
63/63 [==============================] - 0s 903us/step - loss: 3.1793 - accuracy: 0.5095
```

Out[15]:
```
[3.179300546646118, 0.5095000267028809]
```

In [16]:
```
labels_predict = model.predict(dataset['x_test'])
cm = np.array(
    tensorflow.math.confusion_matrix(
        np.argmax(labels_predict,axis=1),
        np.argmax(dataset['y_test_onehot'],axis=1)
))

plot_cm(labels = dataset['labels'], confusion_matrix = cm)
```



In [17]:
```
labels_predict_upper = model.predict(x_test_upper)
cm_upper = np.array(
    tensorflow.math.confusion_matrix(
        np.argmax(labels_predict_upper,axis=1),
        np.argmax(y_test_upper,axis=1)
))

plot_cm(labels = dataset['labels'], confusion_matrix = cm_upper)
```



Most performant

In [18]:

```
model_name = 'Robust_BODY18'
model_path = save_path / '{name}.h5'.format(name = model_name)

model = keras.models.Sequential(
    name = model_name,
    layers =
    [
        keras.layers.InputLayer(input_shape=input_dim),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Dropout(.2),
        keras.layers.BatchNormalization(),
        keras.layers.Conv1D(16, 3, activation='relu'),
        keras.layers.Dropout(.2),
        keras.layers.BatchNormalization(),
        keras.layers.Flatten(),
        keras.layers.Dense(128, activation=keras.activations.relu),
        keras.layers.Dropout(.3),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(128, activation=keras.activations.relu),
        keras.layers.Dropout(.3),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(64, activation=keras.activations.relu),
        keras.layers.Dropout(.3),
        keras.layers.BatchNormalization(),
        keras.layers.Dense(output_dim, activation=keras.activations.softmax),
    ]
```

```
)

model.summary()
model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)

model.fit(
    x=x_train_augmented,
    y=y_train_augmented,
    epochs=15,
    batch_size=16,
    validation_split=0.15,
    shuffle=True,
    callbacks=[keras.callbacks.ModelCheckpoint(filepath=model_path, verbose=2, save_best_only=T
    verbose = 2,
)

with open(save_path / (model_name+'_info.json'), 'w') as f:
    json.dump({'labels':dataset['labels']}, f)
```

```
Model: "Robust_BODY18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_10 (Conv1D)           (None, 16, 16)            112
_________________________________________________________________
dropout_5 (Dropout)          (None, 16, 16)            0
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16)            64
_________________________________________________________________
conv1d_11 (Conv1D)           (None, 14, 16)            784
_________________________________________________________________
dropout_6 (Dropout)          (None, 14, 16)            0
_________________________________________________________________
batch_normalization_6 (Batch (None, 14, 16)            64
_________________________________________________________________
flatten_7 (Flatten)          (None, 224)               0
_________________________________________________________________
dense_17 (Dense)             (None, 128)               28800
_________________________________________________________________
dropout_7 (Dropout)          (None, 128)               0
_________________________________________________________________
batch_normalization_7 (Batch (None, 128)               512
_________________________________________________________________
dense_18 (Dense)             (None, 128)               16512
_________________________________________________________________
dropout_8 (Dropout)          (None, 128)               0
_________________________________________________________________
batch_normalization_8 (Batch (None, 128)               512
_________________________________________________________________
dense_19 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_9 (Dropout)          (None, 64)                0
_________________________________________________________________
batch_normalization_9 (Batch (None, 64)                256
_________________________________________________________________
dense_20 (Dense)             (None, 20)                1300
=================================================================
Total params: 57,172
Trainable params: 56,468
Non-trainable params: 704
_________________________________________________________________
Epoch 1/15
727/727 - 3s - loss: 2.2377 - accuracy: 0.2892 - val_loss: 1.2828 - val_accuracy: 0.6251

Epoch 00001: val_loss improved from inf to 1.28285, saving model to /content/drive/My Drive/Pos
e Classification Kit/Models/Robust_BODY18.h5
Epoch 2/15
727/727 - 2s - loss: 1.2801 - accuracy: 0.5342 - val_loss: 0.9059 - val_accuracy: 0.7094
```

```
Epoch 00002: val_loss improved from 1.28285 to 0.90593, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 3/15
727/727 - 2s - loss: 0.9708 - accuracy: 0.6528 - val_loss: 0.7362 - val_accuracy: 0.7796

Epoch 00003: val_loss improved from 0.90593 to 0.73621, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 4/15
727/727 - 2s - loss: 0.8059 - accuracy: 0.7222 - val_loss: 0.6922 - val_accuracy: 0.7967

Epoch 00004: val_loss improved from 0.73621 to 0.69222, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 5/15
727/727 - 2s - loss: 0.6782 - accuracy: 0.7751 - val_loss: 0.6168 - val_accuracy: 0.8323

Epoch 00005: val_loss improved from 0.69222 to 0.61683, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 6/15
727/727 - 2s - loss: 0.6203 - accuracy: 0.7964 - val_loss: 0.5901 - val_accuracy: 0.8391

Epoch 00006: val_loss improved from 0.61683 to 0.59006, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 7/15
727/727 - 2s - loss: 0.5891 - accuracy: 0.8074 - val_loss: 0.5427 - val_accuracy: 0.8313

Epoch 00007: val_loss improved from 0.59006 to 0.54271, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 8/15
727/727 - 2s - loss: 0.5468 - accuracy: 0.8244 - val_loss: 0.5381 - val_accuracy: 0.8430

Epoch 00008: val_loss improved from 0.54271 to 0.53814, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 9/15
727/727 - 2s - loss: 0.5050 - accuracy: 0.8358 - val_loss: 0.5649 - val_accuracy: 0.8347

Epoch 00009: val_loss did not improve from 0.53814
Epoch 10/15
727/727 - 2s - loss: 0.4833 - accuracy: 0.8463 - val_loss: 0.4957 - val_accuracy: 0.8610

Epoch 00010: val_loss improved from 0.53814 to 0.49566, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 11/15
727/727 - 2s - loss: 0.4574 - accuracy: 0.8531 - val_loss: 0.4984 - val_accuracy: 0.8591

Epoch 00011: val_loss did not improve from 0.49566
Epoch 12/15
727/727 - 2s - loss: 0.4333 - accuracy: 0.8635 - val_loss: 0.5184 - val_accuracy: 0.8576

Epoch 00012: val_loss did not improve from 0.49566
Epoch 13/15
727/727 - 2s - loss: 0.4384 - accuracy: 0.8673 - val_loss: 0.4993 - val_accuracy: 0.8552

Epoch 00013: val_loss did not improve from 0.49566
Epoch 14/15
727/727 - 2s - loss: 0.3960 - accuracy: 0.8752 - val_loss: 0.4922 - val_accuracy: 0.8654

Epoch 00014: val_loss improved from 0.49566 to 0.49224, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
Epoch 15/15
727/727 - 2s - loss: 0.4260 - accuracy: 0.8661 - val_loss: 0.4847 - val_accuracy: 0.8635

Epoch 00015: val_loss improved from 0.49224 to 0.48472, saving model to /content/drive/My Driv
e/Pose Classification Kit/Models/Robust_BODY18.h5
```

In [19]:

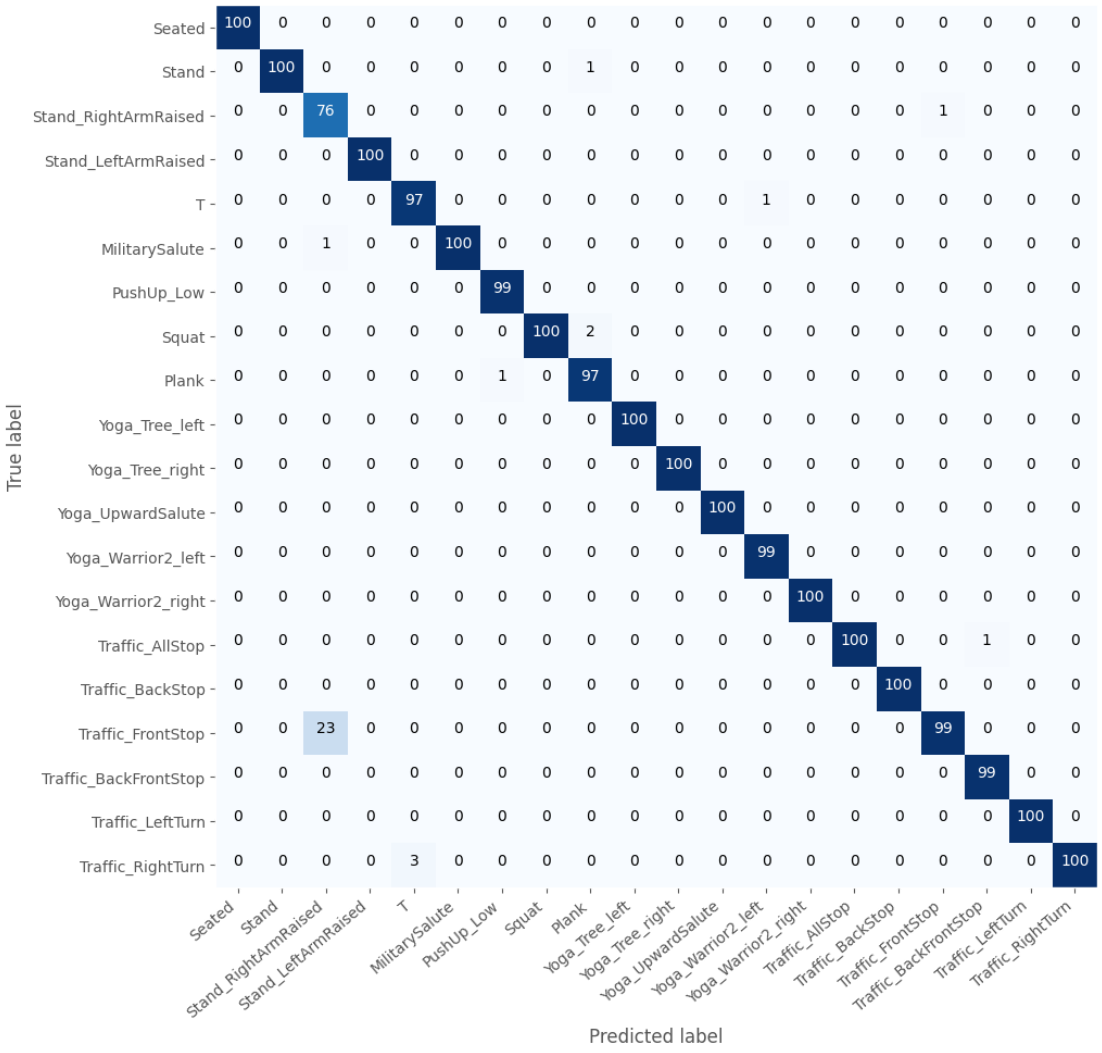
```
print(model.name + " accuracy on full samples:")
model.evaluate(x=dataset['x_test'], y=dataset['y_test_onehot'])
print(model.name + " accuracy on partial samples:")
model.evaluate(x=x_test_upper, y=y_test_upper)
```

```
Robust_BODY18 accuracy on full samples:
```

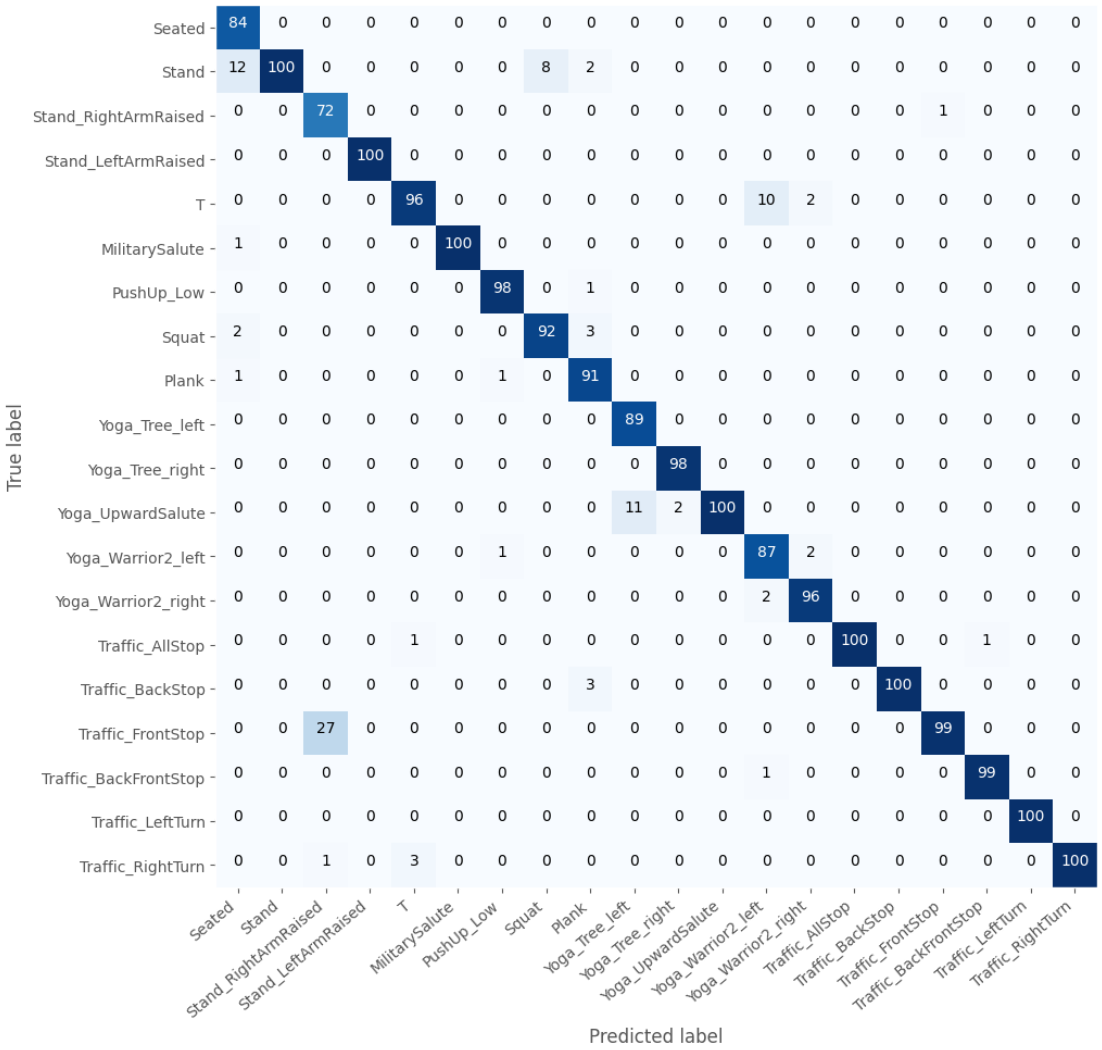
```
63/63 [==============================] - 0s 2ms/step - loss: 0.0567 - accuracy: 0.9830
Robust_BODY18 accuracy on partial samples:
63/63 [==============================] - 0s 1ms/step - loss: 0.1521 - accuracy: 0.9505
```

Out[19]:
```
[0.1520567387342453, 0.9505000114440918]
```

In [20]:
```
labels_predict = model.predict(dataset['x_test'])
cm = np.array(
    tensorflow.math.confusion_matrix(
        np.argmax(labels_predict,axis=1),
        np.argmax(dataset['y_test_onehot'],axis=1)
))

plot_cm(labels = dataset['labels'], confusion_matrix = cm)
```



In [21]:
```
labels_predict_upper = model.predict(x_test_upper)
cm_upper = np.array(
    tensorflow.math.confusion_matrix(
        np.argmax(labels_predict_upper,axis=1),
        np.argmax(y_test_upper,axis=1)
))

plot_cm(labels = dataset['labels'], confusion_matrix = cm_upper)
```


True label
Predicted label
Seated
Stand
Stand_RightArmRaised
Stand_LeftArmRaised
T
MilitarySalute
PushUp_Low
Squat
Plank
Yoga_Tree_left
Yoga_Tree_right
Yoga_UpwardSalute
Yoga_Warrior2_left
Yoga_Warrior2_right
Traffic_AllStop
Traffic_BackStop
Traffic_FrontStop
Traffic_BackFrontStop
Traffic_LeftTurn
Traffic_RightTurn