**Gesture Recognition**

**Introduction and background**

Gesture recognition is one of the main part of our projects. Our *armband* collects the MMG and accelerometer signals from the users and uses these inputs to interpret the commands from the user. The user commands include two aspects: directions and gestures. Directional commands are interpreted by the NU interface with the accelerometers, gyroscopes and magnetometers, while the gesture commands are classified with the mechanomyogram signals (MMG, muscle vibration signals). Gesture interpretations are carried out with a pretrained Convolutional Neural Network (CNN) calibrated for each user. Constructed in python, the CNN receives data transmitted from the NU interface, classifies the gestures, and either pass the information back to the NU interface for simulation or uses them directly for quadcopter control.

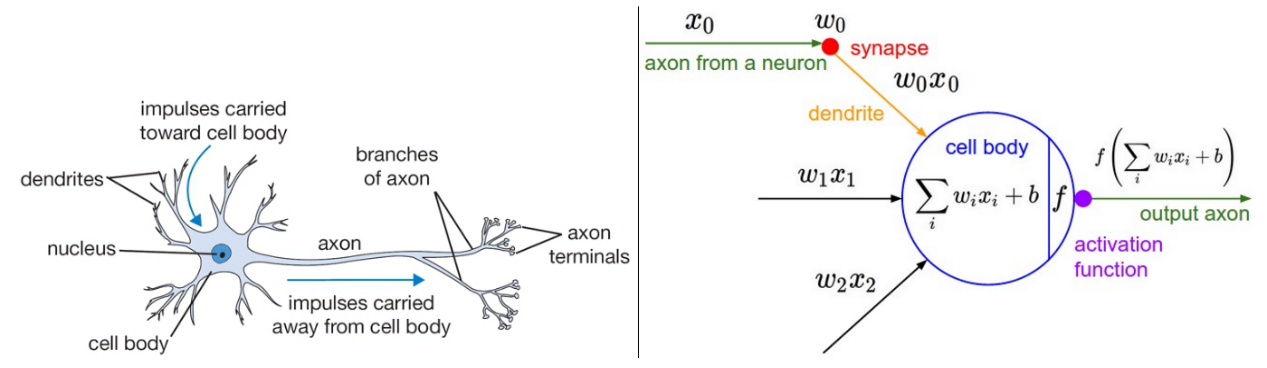
**Backgrounds**

**Neural Networks**

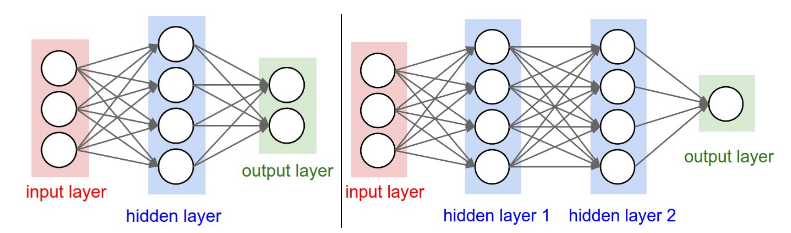
Neural networks are biologically inspired universal approximators for functions that may have large numbers of inputs and outputs. Supervised learning is a branch of machine learning problems, where the desired outputs are fed with the inputs to the machine learning algorithms, so that the underlying function that maps the outputs to the inputs could be approximated. In this project, our neural network learns to map the MMG signals to the gesture/noise classifications.

**Fully connected layers and non-linear activation functions**

The most basic neural network is constructed with the fully connected layers joined by activation functions. Fig \_\_\_ shows one unit of the fully connected layer and its biological analogy. In these layers, each neuron unit in the neural network is non-linearly activated weighted sum of the neuron layer. The weights w are the trainable parameters that approximates the desired patterns. The activation function f induces non-linearity in the neural network, providing it the ability to approximate non-linear functions.



The neurons between adjacent layers are fully connected as shown in examples in Fig \_\_\_. This is a rather simplistic structure in that the network output is independent of the input and output in the previous time steps, and that the adjacency of the input data points is not explicitly considered.



The non-linear function used in this project is the Rectified Linear Unit (ReLU). It is popular in recent years as it greatly simplifies the computation process compared to traditional choices like sigmoid and tanh function. []

**Convolutional Neural Networks**

CNN is a type of neural network widely used in modern computer vision tasks that excels at identifying local features. CNN treats the adjacent inputs as local patterns as oppose to unrelated data points. CNNs are characterised by convolutional layers, where neurons are arranged in 3 dimensions: width, height and depth, and each layer of neuron in the depth direction is a learnable filter, or feature extractor. For instance, in image classifications, these neural groups may be excited by local features like edges or brightness gradient in a certain direction, converting the images to a map of neuron excitation. A few fully connected layers are then trained to classify the image contents based on the neuron excitation map.

Figure \_\_ shows an example of the convolutional layer. The input image is of dimension 32x32x3, and the example neurons are arranged in groups of 5x5x3 to form filters, so that each filter examines a 5x5 square in the input image. There are 5 groups, and by sweeping the filters on the input image, a result excitation map of dimension 28x28x5 is produced, assuming stride is 1 and no padding is used.

**Overfitting**

Overfitting is a problem commonly encountered when approximating functions. In gesture recognition, the gesture classification accuracy may be greatly reduced if the neural network is overfitted on the training data, as it may lead to failure to recognise a gesture only because it is slightly different from the training example. Apart from trying to obtain more training data, maxpooling and dropout is applied to ameliorate overfitting.

**Pooling layers**

The pooling layer is another component in convolutional neural networks. It reduces the spatial size of the neuron activation map, hence reducing the number of parameters in the network. This may help reduce the problem of overfitting, and saves some computational time for each neural network prediction. In the neural network we constructed, a maxpooling layer of size 1x2 is applied to shorten the activation map by 1/2, while the depth of the map remains unchanged. It is used instead of average pooling as it has been shown to work better in practice.

**Dropout**

Dropout is another popular method to prevent over-fitting. It intentionally covers random neurons in the neural network during the training process, so that each of the neuron node has a chance of being absent in the network. As such, the remaining of the network would be forced to look for distinct features from the input. An intuitive understanding of this technique is that, for a neural network trained to identify cats in pictures, each of the neuron node may correspond 'is white', 'has ears', or 'furry'. Covering parts of the neuron network forces the network to make predictions only based on the remaining of the information available, hence would approximate more generalised functions. A dropout of 50% dropping chance is applied to the fully connected layers of the neural network.

**Neural network structure**

**rationale**

One challenge in this project is that gestures must be identified among the noise signals of other muscle movements, which could make up over 90 percent of the signals received. Specifically, the users would move their arms to control the drone directions. These muscle movements should be rejected by the gesture recognition algorithm.

Earlier in our project, classic signal processing techniques like filtering and applying thresholds was considered. For instance, the MMG signals caused by moving arms to control the drone direction is usually smaller than those caused by gestures, the signals can be squared and then smoothed. The processed signals below a certain magnitude can be treated as noise, and we only need to investigate in the values above the threshold to classify the gestures. However, this method requires manual adjustments for each user. For instance, some people may require a larger threshold value as they have more vigorous muscle activities. In addition, some gestures may have weaker or more abrupt signals than others, and these gestures may be accidentally rejected. Neural networks, however, is not constrained by such a threshold value, as it examines the shape as well as the magnitude of the input to differentiate between gestures and noise.

Various types of neural network structures including Long-short term memory (LSTM) and multilayer perceptron (MLP) are tested, and the CNN is chosen because it gives the best result, and is able to produce these results in real time.

**drawbacks**

The most significant disadvantage of neural networks are that they work like black boxes. It is hard to locate the exact issues when the networks do not work as desired due to their intrinsic random initialisation process. However, several visualisation techniques have been utilised to interpret the neural network.

Another drawback is that the neural network we used requires more computational power than many other signal processing techniques, and the computation is carried out at every time step. Although we have shown in our project that real time processing is feasible, there is still a large incentive for us to simplify the neural network.

**Structure**

A schematic of the CNN structure is shown in fig\_\_. We are attaching 3 MMG sensors on the armband, producing three streams of data at each time step. 50 time steps of the most recent data streams are kept, corresponding to 1 second, so that effectively, a running window of 1 second of muscle vibration is fed to the neural network at each new time step. The weights in the first convolutional layer are of dimension 1x3x3x32, with each unit covering a local connectivity of height 1, width 3, and depth 3. This transforms the original 1-by-50-by-3 input to an neuron activation map of depth of 32. The weights in the second layer are of dimension 1x5x32x64. The final neuron activation map is reshaped to a vector, which is then processed with two successive fully connected layers, producing the final result. The classification result is expressed as a 9 dimensional one-hot vector, with 8 dimensions corresponding to the gesture and 1 corresponding to other muscle movements. For instance, if the neural network classifies the input as gesture 3, the output would be [0,0,1,0,0,0,0,0,0]’, and if the input is classified as noise, the output should return [0,0,0,0,0,0,0,0,1]’.

**Pretraining and calibration**

Early experiments have shown that different users may have highly distinct muscle activities, and hence mechanomyogram (MMG) signals for the same gesture **\_** insert picture**\_\_**. Thus, a desirable gesture recognition algorithm must be able to adapt to each user.

As mentioned in the background section, the two convolutional layers acts as the filters or feature extractors, and the fully connected layers converts the activation map of these filters to the result classification. Pretraining of the neural network is based on the assumption that although the users may have different muscle movements for the same gesture in the time scale of second, the local MMG input features should be similar in the time scale of millisecond. Hence the trained convolutional layers can be directly applied to the new users, and calibration is only required to be performed on the two fully connected layers. This technique reduces both computational time and gesture examples required for calibration from the new user.

second\_optimizer = tf.train.GradientDescentOptimizer(learning\_rate=learning\_rate).minimize(

  cost, var\_list=[weights['wd2'], weights['out'], biases['bd2'], biases['out']])

...

sess.run(second\_optimizer, feed\_dict={

        x: batch\_x, y: batch\_y, keep\_prob: 0.5, learning\_rate: 0.0001})

sess.run(second\_optimizer, feed\_dict={

        x: noise\_x, y: noise\_y, keep\_prob: 0.5, learning\_rate: 0.0005})

sess.run(second\_optimizer, feed\_dict={

        x: new\_noise, y: new\_noised, keep\_prob: 0.5, learning\_rate: 0.00005})

The calibration is carried out as shown with gradient descent. It only optimises the weights and biases in the fully connected layers and freezes the weights in the convolutional layers.

**Design iterations**

**Length of the running window**

Originally, we our running window of MMG input has the length of 300, corresponding to 0.6s of MMG signals. We have found that the classification accuracy is unaffected if the MMG sampling rate is reduced to 50 times per second. Hence the neural network only takes the input of shape 50x3, saving the computational power significantly. On the other hand, keeping a longer window of data (increased to 1s from 0.6s) has slightly improved our classification accuracy.

**Usage of the accelerometer data**

Intuitively, the accelerometer data should help differentiate between gestures like thumb-up and thumb-down. However, it turned out that the result from 10 trials shows no significant difference between using the accelerometer readings and not using them. We hence decided to discard this branch of the network.

**Data Collection**

The pretraining of the neural network is carried out with data collected from 27 participants. Each participant was requested to preform 8 gestures for 15 times each, followed by 30 seconds of arm movements without gestures. These MMG signals are saved as .csv files and processed in Matlab. Once the raw data are processed, we identified where the gestures took place, and labelled the period from 50 time steps before that point as the gesture input. As for noise (non-gesture muscle movements), the entire duration is labelled as noise and fed to the neural network. When users control quadcopters with the armbands, we predict that gesture should take up only about 5% percent of the total usage time. Hence, for isometric training, we used this percentage to distribute the training data (5% gesture vs 95% noise).

For new users, a calibration procedure is carried out before using the armband. Figure \_\_ shows the calibration form. Once the button 'calibration' is clicked, the user would be prompted to perform the gestures for five times each when the indication light is on. The light is turned on for one second for each gesture trial, and there is a one-second interval between the gesture trials. The data recorded when the light is on (a running window ending at the instance when the light is just switched off) is labelled as input corresponding to the gesture, and the duration of the light being switched off (0.2s after light is switched off until the light is turned on) is fed to the neural network labelled as noise. This is further illustrated in Figure \_\_\_.

**Data Transmission**

Data transmission between the C# form application and the CNN constructed with TensorFlow on python in this project is carried out with named pipes. A pipe server is created in the C# form and is fed with a byte array of the MMG signals as well as the roll, pitch and yaw angle of the IMU. The python program reads the byte array information posted in the pipe by the C# form, passes it through the neural network and obtains the gesture classification. Depending on the task, the gesture results are further used as shown in Figure \_\_\_.

**Test and visualization**

confusion matrix, label significant values.

Parts of the input can be systematically occluded to determine the part of the input that is most crucial in classification. Concretely, the input with a small patch set to 0 can be fed to the neural network, and the result is compared between the original input and the patched input. If the classification result has changed, it is implied that the covered region has been responsible for the gesture classification. This patch is iterated across all time steps in the input signal to compare the importance of the input local regions. The results shown in Fig\_\_ has clearly shown that the regions critical for gesture classification corresponds fairly well to our intuitions.

Further tests of the gesture classification and directional control can be tested in the simulation form in section \_\_\_.

**Drone Control**

**Introduction**

A swarm of quadcopters (referred to as 'drones' below) is controlled to demonstrate the human-machine interface. To enable the precise control of the drone swarm, a drone locating system is built with webcams. A PID control loop is then used to guide the drone to the target location as specified by the user with the armband.

**Drone locating system**

**Setup**

Three webcams are arranged in an orthogonal manner as shown in Fig \_\_\_ to form a drone arena of 1.63x1.63x1.90 m^3. The direction that the webcams point to are covered with grey backgrounds, but as will be explained blow, these backgrounds are not necessary.

**Algorithm**

The flowchart of the drone locating system is shown in Fig \_\_\_. The function frame\_loc() identifies the drones on each frame captured by the webcams. get\_angle() calculates the tangent of angle ​ and ​ from the location of the drones on each frame. These angles is fed to get\_coordinate() to calculate the drone coordinate in the three-dimensional space. The functions in the dotted box are grouped and named colored\_cam(), and run in a separate thread from the drone control function, which itself runs from a separate thread from the gesture recognition function, to increase the frequency of commands sent to the drone, hence improving performance.

**frame\_loc()**

**Alternatives**

Two methods have been tested and compared to locate the drone on the camera frames.

The first algorithm compares the first frame that the webcam captures and the current frame. Assuming there are no other moving objects, the only difference between the frames would be the drones, and the centre of the moving object approximately corresponds to the centre of the drone.

The advantage of this method is that no additional parts needs to be attached to the drone. However, there are two main disadvantages with this method. Firstly, this method is sensitive to ambient light changes. Since it only compares the absolute difference between the frames, there is no direct way to tell whether the moving object is a shadow or a drone. This is especially undesirable for demonstrations where such light changes are expected. Secondly, it is hard to differentiate between two quadcopters with the subsequent functions if the function only returns the two locations of the two drones on the frame. For instance, with a pair of cameras, there are ​ possible drone locations for the same frame\_loc() output pairs, and we have 3 pairs of webcam combinations. Sophisticated algorithms may be implemented to reduce the number of possibilities, but such algorithms may not be robust enough.

The second algorithm is to attach a uniquely coloured ball on each drone, and let the webcams track the balls instead of the drone. This method easily rejects the other objects in the frame by applying HSV filters. The main disadvantage with this method is that the attached ball somewhat affects the aerodynamics of the drones, but as we are only implementing non-aggressive control of the drone, such effects are not significant. This method is hence chosen as our preferred alternative.

**Implementation**

A 640x480 frame is read from the webcam and applied a Gaussian blur (11x11 pixel). The frame is initially encoded in RGB, but is converted to HSV scale to improve performance. For each colour we are interested in, a mask is created by defining the upper and lower bound of the colour in the HSV scale. The masks are applied to the frame, followed by a morphological opening and closing to produce a smoothed result as shown in fig\_\_\_. The largest circular contour is found on the smoothed result image, and the centre corresponds to the centre of the ball, and hence approximately a fixed length above the centre of the drone under small angle approximation. The locations of the drones on the frame is returned as four variables: orange\_x, orange\_y, blue\_x, blue\_y. x coordinates range from 0 to 640 and y coordinates range from 0 to 480, the top left-hand corner being the origin.

**get\_angle()**

The coordinates of the drones in the 3-dimensional space are calculated from the tangent of angle ​ and ​ as shown in fig \_\_\_. The field of view of the webcam has a horizontal span of 60 degrees and vertical spans of 50 degrees. From this,get\_angle() calculates the tangent of the angles from a pair of x,y coordinates obtained from frame\_loc().

**get\_coordinates()**

Coordinate (x, y) would result in tangents of angle ​ and ​ as shown in equation \_\_, where a, b, c describes the shape of the drone arena as shown in fig~~~~. Equation ~~ can be converted to forms in equation ~~, which can be converted to simultaneous equations ~~. This system of linear equation returns the location ~ from the angle tangents, and is re-written in matrix form to be programmed in python.

It can be observed that each pair of the webcams (0-1, 0-2 or 1-2) is able to determine the location in two dimensions, so with 3 pairs of webcam combinations, each of the x, y, z coordinates is found with two sources. Average is taken between the results from the two sources to improve the robustness of the locating system.

**Data types and safety**

The coordinates and read\_failed variables are declared in python as lists. A list is a mutable object in python, so that when it is updated in the camera thread, it is also updated globally. However, a list object does not support matrix manipulations, so the locations of the orange and blue balls are first declared as numpy arrays, and then assigned element-wise to the coordinate list.

If a coloured ball is not found by the one of the webcams, frame\_loc() would return 0, 0 as coordinates for that ball. Hence if ox0\*ox1\*ox2 != 0 or oy0\*oy1\*oy2 != 0 is True only when all three webcams captures the orange ball. The read\_failed variable is assigned 0 or 1 accordingly, and if the ball is not found for 10 consecutive frames, the corresponding drone would be switched off.

**Drone Control**

**setup**

Two crazyflie 2.0 quadcopter were used to demonstrate the controlling of the drone swarm with the armband. The crazyflie drone is controlled with the crazyflie-python-library cflib, which handles the wireless communication with the drones through radio. In this project, we are interested in the non-aggressive, linear control and small roll, pitch angles are assumed. cflib further simplifies the control in that it allows us to directly send the commanded roll, pitch angle and the drone throttle, effectively turning the drone into an approximately second order system. A PID controller is applied on each drone, guiding it to its target location. The program flowchart and the control loop are shown in figure \_\_.

**Drone class**

A drone is declared as an instance of a Drone() class object (a sample instance will be referred to as cf below). The Drone() class is an extension of the cflib.crazyflie.Crazyflie() class, with the additional inclusion of the location and velocity information (cf.loc and cf.vel respectively, recorded as numpy arrays). The Drone() class also keeps track of the drone's location and velocity in the previous time step, and the number of frames that the drone is not captured on the webcams.

**Initialise()**

A Drone() class object is initialised with its assigned link\_uri. The link\_uri is the radio ID of the crazyflie quadcopter. A cflib.crazyflie.Commander() class object is one that handles the commands sent to the drone through the real-time radio protocol. After communication is established with the drone, a Commander() class object is declared under cf as cf.cmd. This enables sending of roll, pitch and thrust commands to the crazyflie.

**get\_loc()**

The get\_loc() function returns a weighted sum of the drone coordinate from the camera and the inferred drone coordinate from its previous coordinate and velocity. From this weighted sum, the new drone velocity is calculated. This function smoothes the raw coordinate from the webcams and updates these new values to cf.

If the camera fails to find the drone, however, the function would increment cf.not\_found\_counter which records the number of frames that the webcams do not find the drone. The new coordinate and velocity is then purely inferred from the previous time step.

**Go\_to()**

Go\_to() function is the main component of the PID control. Firstly, it reads cf.not\_found\_counter. If the counter is greater than 10, corresponding to 0.1 seconds, the drone will be powered off. This is to prevent the scenario that the drone flies out of the arena.

If the drone is found by all three cameras, line 57 calculates the drone commands. These commands are then capped between sensible values to ensure that the small angle approximation is enforced. The pitch and roll control uses the same set of Kp and Kd value, while throttle is controlled on a different scale, scaling up the command value by 3000 times. Due to a small delay of about 0.15s from the webcams, the drone can hardly reach a steady state, and increasing Kp may lead to instability above certain values.

**Start\_up()**

This function sends an initial command of high thrust to raise the drone to the height that can be captured by all three cameras. The command lasts for 0.5s, and is cut when the cameras find the drone.

**Gesture control**

The target coordinates of the drones are set by the armband interface. Three gestures are used to switch between states that the IMU affect the target location.

The first gesture, snapping, locks or unlocks the target coordinate. Initially, the drone target coordinates are locked, and tilting the IMUs does not have any effect to the drone coordinates. Snapping once unlocks the directional control, and snapping again would lock the coordinates again. The second gesture is tensing muscle. This gesture enables the user to switch between single drone control and controlling the drones as a group. The third gesture, double tensing (tensing muscle twice, like double click), changes the height of the drones.

The target coordinates are constrained to a cube of 40x40x40 cm due to the limited field of view of the webcams. However, other drone locating methods like Kinect can be developed to relax or remove this constraint in the future.