Driving with Gestures

Alex Brown, Devin Thomas, Kyle Brewster, Vinayak Chaturvedi

Introduction

Recent advances in image classification [1] have opened a wide range of possibilities for applications of machine perception. We investigate the use of convolutional neural networks in gesture classification to control a simulated car. Our work is motivated by the real-world scenario in which a large vehicle must be carefully maneuvered into a tight space. This is often achieved by cooperation between a driver and a guide, who gestures to the driver to indicate how to complete the maneuver, but a second person is not always available to provide the second perspective. The driver could fill the role of guide if their vehicle were equipped to respond to gestures as the driver would have. Further applications include docking autonomous boats and positioning aerial drones for photography.

Prior Work

[2] use a convolutional neural network to capture long-range spatial dependencies in human pose. [3] apply a similar structure in their network to predict 3D hand pose, but they add another network which does viewpoint estimation and 3D pose estimation from 2D hand pose. More specifically, they develop a hand segmentation network, HandSegNet, the 2D hand pose prediction network, PoseNet, represented as a score-map of 21 key points, and a 3D pose prediction network called PosePrior.

Gesture Command Net

We base our work closely on that of [3]. Specifically, we use the same first two pieces of their architecture: HandSegNet and PoseNet. We replace their PosePrior network with a network of our own to classify gestures (see below).

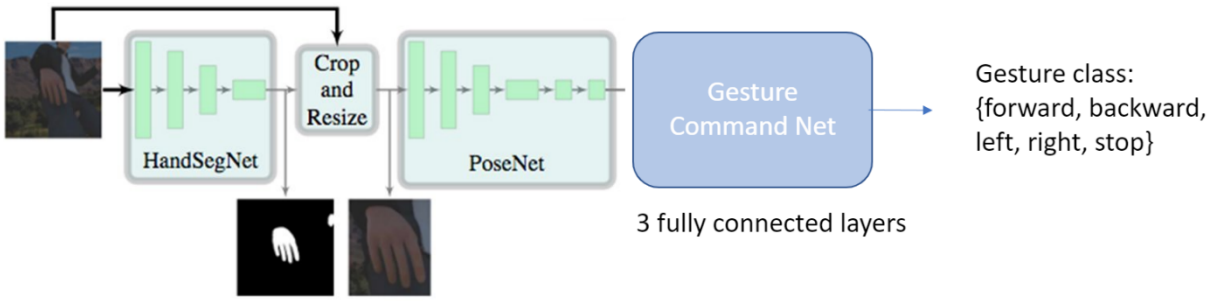
id	Name	Kernel	Dimensionality
	Input image	-	$256 \times 256 \times 3$
1	Conv. + ReLU	3×3	$256 \times 256 \times 64$
2	Conv. + ReLU	3×3	$256 \times 256 \times 64$
3	Maxpool	4×4	$128 \times 128 \times 64$
4	Conv. + ReLU	3×3	$128 \times 128 \times 128$
5	Conv. + ReLU	3×3	$128 \times 128 \times 128$
6	Maxpool	4×4	$64 \times 64 \times 128$
7	Conv. + ReLU	3×3	$64 \times 64 \times 256$
8	Conv. + ReLU	3×3	$64 \times 64 \times 256$
9	Conv. + ReLU	3×3	$64 \times 64 \times 256$
10	Conv. + ReLU	3×3	$64 \times 64 \times 256$
11	Maxpool	4×4	$32 \times 32 \times 256$
12	Conv. + ReLU	3×3	$32 \times 32 \times 512$
13	Conv. + ReLU	3×3	$32 \times 32 \times 512$
14	Conv. + ReLU	3×3	$32 \times 32 \times 512$
15	Conv. + ReLU	3×3	$32 \times 32 \times 512$
16	Conv. + ReLU	3×3	$32 \times 32 \times 512$
17	Conv.	1×1	$32 \times 32 \times 2$
18	Bilinear Upsampling	-	$256 \times 256 \times 2$
19	Argmax	-	$256 \times 256 \times 1$
	Hand mask	-	$256 \times 256 \times 1$

Table 4: Network architecture of the proposed *HandSegNet* network. Except for input and hand mask output every row of the table gives a data tensor of the network and the operations that produced it.

id	Name	Kernel	Dimensionality
	Input image	-	$256 \times 256 \times 3$
1	Conv. + ReLU	3×3	$256 \times 256 \times 64$
2	Conv. + ReLU	3×3	$256 \times 256 \times 64$
3	Maxpool	4×4	$128 \times 128 \times 64$
4	Conv. + ReLU	3×3	$128 \times 128 \times 128$
5	Conv. + ReLU	3×3	$128 \times 128 \times 128$
6	Maxpool	4×4	$64 \times 64 \times 128$
7	Conv. + ReLU	3×3	$64 \times 64 \times 256$
8	Conv. + ReLU	3×3	$64 \times 64 \times 256$
9	Conv. + ReLU	3×3	$64 \times 64 \times 256$
10	Conv. + ReLU	3×3	$64 \times 64 \times 256$
11	Maxpool	4×4	$32 \times 32 \times 256$
12	Conv. + ReLU	3×3	$32 \times 32 \times 512$
13	Conv. + ReLU	3×3	$32 \times 32 \times 512$
14	Conv. + ReLU	3×3	$32 \times 32 \times 512$
15	Conv. + ReLU	3×3	$32 \times 32 \times 512$
16	Conv. + ReLU	3×3	$32 \times 32 \times 512$
17	Conv.	1×1	$32 \times 32 \times 21$
18	Concat(16, 17)	-	$32 \times 32 \times 533$
19	Conv. + ReLU	7×7	$32 \times 32 \times 128$
20	Conv. + ReLU	7×7	$32 \times 32 \times 128$
21	Conv. + ReLU	7×7	$32 \times 32 \times 128$
22	Conv. + ReLU	7×7	$32 \times 32 \times 128$
23	Conv. + ReLU	7×7	$32 \times 32 \times 128$
24	Conv.	1×1	$32 \times 32 \times 21$
25	Concat(16, 17, 24)	-	$32 \times 32 \times 554$
26	Conv. + ReLU	7×7	$32 \times 32 \times 128$
27	Conv. + ReLU	7×7	$32 \times 32 \times 128$
28	Conv. + ReLU	7×7	$32 \times 32 \times 128$
29	Conv. + ReLU	7×7	$32 \times 32 \times 128$
30	Conv. + ReLU	7×7	$32 \times 32 \times 128$
31	Conv.	1×1	$32 \times 32 \times 21$

Table 5: Network architecture of the *PoseNet* network. Except for input every row of the table represents a data tensor of the network and the operations that produced it. Outputs of the network are are predicted score maps $\mathbf{c}$ from layers 17, 24 and 31.

Architecture of HandSegNet on the left, PoseNet on the right copied from [3]

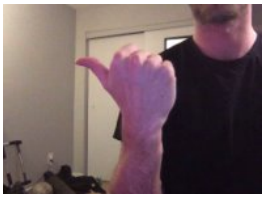


PoseNet produces a score map of 21 “key points” that represent the 2D pose of the hand. After a mean pooling layer, this is the input to Gesture Command Net. Gesture Command Net consists of three layers: two fully connected layers with 32 output channels, and a final fully connected layer with 5 output channels. The loss function is softmax cross-entropy.

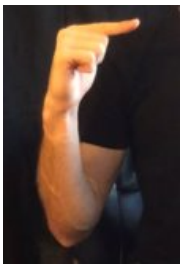
Gesture Command Choices

When designing the commands for controlling the car we wanted to make them as natural as possible as if you were directing a person. Our initial approach was to use a “pull” for backward, a “push” for forward, pointing either left or right for their respective directions and a closed fist for stopping as shown below:

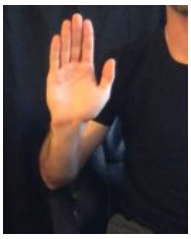
Right:



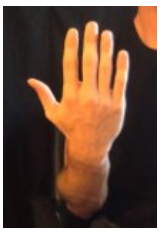
Left:



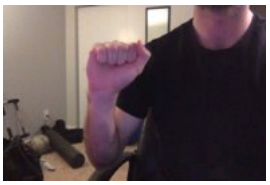
Forward:



Backward:



Stop:

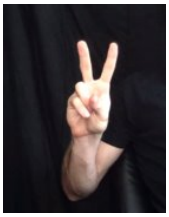


For gathering these gestures each of us filmed 2-minute videos of the gestures where we slightly varied the distance and orientation of the gesture relative to the camera. We then pulled every tenth frame from the videos to gather about 4,000 training samples.

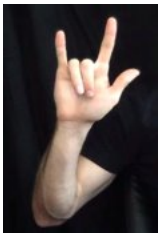
During initial testing with the gestures we noticed the network did not respond well to the closed fist gestures (stop and right) or the flat handed gestures (backward and forward). The network would incorrectly orient our hands or provide a skeleton that was way off for the closed hand gestures. The flat handed gestures were getting a skeleton aligned with the palm if our fingers were too close together. To combat this, we tried a few new gestures to see if we could get something that was more identifiable to the network and increase the gesture identification accuracy. The two gestures we ended up using were the "peace sign" and a "horns" (index, pinky and thumb extended) gesture. Which replaced right and stop respectively. Then to help the gesture recognition on forward and backward we took new videos with our fingers spread wider. This helped the network identify our hand and finger positions drastically.

The new gestures replacing right and stop:

Right:



Stop:



Training Methodology and Results

HandSegNet and PoseNet were trained on the Rendered Hand Pose Dataset [3] following the instructions published at [3]. The RHD provides 41,258 training images and 2728 testing images. Each image includes a 320x320 RGB image, a 320x320 depth map, a 320x320 greyscale segmentation mask, 21 keypoints per hand with coordinates in the image frame, world frame and visibility, and a camera matrix. The mask segments the person, background, fingers and palm.

Example Training images from the Rendered Hand Pose Dataset

The compute environment was prepared on a fresh Ubuntu 20.04 environment on a desktop with an I7 7700k, 16 GB ram and a 8 GB GTX 1080 GPU. Tensorflow was used at version 1.10, which was the closest version to the recommended 1.3.0. HandSegNet was designed from and initialized with the weights of the first 16 layers of the human convolutional pose machine of Wei et al. [2]. We retrained the HandSegNet and PoseNet on the RHD data and achieved similar performance on the evaluation data provided with RHD to the results published in [3]. We measured slightly better performance on the cropped evaluation images, and slightly worse performance on the uncropped evaluation images.

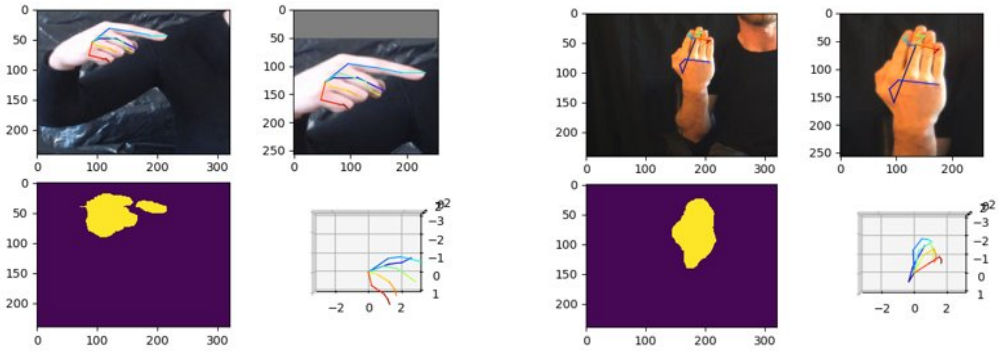
Evaluation Error	*Mean(pixels us/pixels published)*	*Median(pixels us/pixels published)*
Cropped [easier]	7.483/7.630	3.893/3.939
Uncropped	15.960/15.469	4.538/4.374

HandSegNet and PoseNet testing performance

HandSegNet was trained with 40,000 iterations using the tensorflow Adam optimizer minimizing a softmax_cross_entropy_with_logits. The batch size was 8 and the learning rate ramped from 1e-5 to 1e-6 at 20,000 iterations and 1e7 at 30,000 iterations. Their training program modulates the hue of the training images, and randomly crops them to the network used size of 256 x 256.

PoseNet was trained using our trained HandSegNet and the RHD. Posenet was provided with the first 16 layers initialized using Wei et al. [2] with the remaining layers randomly initialized. PoseNet Trained for 30,000 iterations using the Adam optimizer with a loss function of the root mean squared error of the visible hand skeleton keypoints.

GestureNet was trained using the adam optimizer and a softmax_cross_entropy_with_logits loss function to recognize the 5 gestures. The first successful training was on every 10^{th} frame from 2 min videos of Kyle, Alex and Devin demonstrating the gestures.



Example GestureNet Training images, run through HandSegNet and PoseNet. On the left an example of those networks performing well. On the right an example of poor performance of those networks.

This was trained with a learning rate of 0.1 for 20,000 iterations with a batch size of 32. As shown in the examples above there were training images for which HandSegNet and PoseNet performed poorly. After 20,000 iterations the network had likely overfit the training data and achieved perfect training accuracy.

We evaluated this network on the training videos, but every tenth frame offset by five frames from the original training data. With this easy test the network achieved 97.3% Accuracy.

	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	859	1	7	7	9
FORWARD	3	867	2	1	6
LEFT	8	6	762	13	10
RIGHT	9	5	7	694	6
STOP	4	3	0	1	710

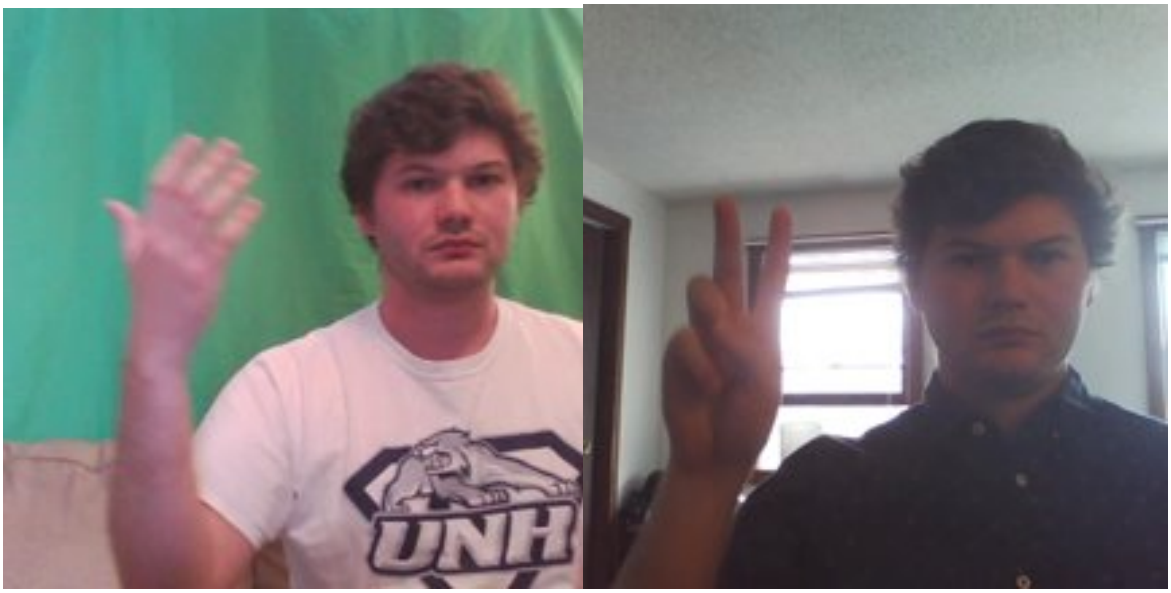
In a testing set of frames taken from extra videos of Kyle and Alex demonstrating the forward and backward gestures the network achieved an accuracy of 94.1%.

	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	416	9	7	9	6
FORWARD	2	412	4	4	11

However tested upon videos of Vinayak demonstrating the backward, forward and left the network performed relatively poorly, only 61.0% accurate.

	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	300	48	32	17	12
FORWARD	94	25	62	15	48
LEFT	8	2	212	2	3

Tested on frames from a video of Alex is poor lighting conditions, and with a more cluttered background the network achieved an accuracy of 79.7%



A training frame of Alex on the left, in contrast with the testing frame on the right, with a more cluttered background much worse lighting.

	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	49	5	3	2	0
FORWARD	3	62	0	2	1
LEFT	3	0	57	5	0
RIGHT	3	3	7	50	3
STOP	4	14	6	1	37

We trained the network a second time, this time including demonstrations from Vinayak. This GestureNet was trained with a ramping learning rate, 0.1 for the first 3000 iterations then 0.01 for another 3000 iterations. Batch size was again 32. In this case every second frame of the videos was used.

After 6000 iterations GestureNet had achieved good, but not perfect accuracy on the training data of 97.55%.

	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	4438	6	17	30	2
FORWARD	25	4451	11	14	7
LEFT	38	3	4026	46	2
RIGHT	51	14	38	3824	17
STOP	69	28	31	65	3739

We then unlocked training the PoseNet and HandSegNet weights, and ran an additional 24,000 iterations at a learning rate of 0.000001. We did this hoping that this would improve the HandSegNet and PoseNet performance in the lighting conditions of our webcam images. This network achieved perfect performance on the training data. We tested the same extra videos of Kyle and Alex demonstrating forward and backward gestures. The 6000 iteration network achieved 95.5% accuracy, the 6000 + 24,000 network achieved 94.2% accuracy.

6000 ITERATION	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	426	8	10	5	1
FORWARD	5	414	3	5	3

6000 + 24000	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	421	2	18	5	4
FORWARD	3	408	2	5	12

We then tested on the Vinayak videos, using different frames from the same videos used in the training. We achieved slightly better but still poor performance 63.2% for the 6000 iteration network and 64.2%, implying that the good training performance on Vinayak's training samples was overfitting not actual learning. Only barely outperforming the network not trained on Vinayak.

6000	BACKWARD	FORWARD	LEFT	RIGHT	STOP
TESTING ERROR					
BACKWARD	168	11	12	17	6
FORWARD	46	12	24	35	10
LEFT	4	0	108	2	1

6000 + 24000	Backward	Forward	Left	Right	Stop
TESTING ERROR					
BACKWARD	171	6	17	14	6
FORWARD	40	18	31	25	13
LEFT	4	2	104	3	2

In contrast the performance on the extra Alex videos was substantially better than the first GestureNet.

6000 TESTING ERROR	BACKWARD	FORWARD	LEFT	RIGHT	STOP
BACKWARD	74	5	2	7	0
FORWARD	0	85	1	4	4
LEFT	2	1	82	6	4
RIGHT	14	4	1	71	0
STOP	5	17	6	3	58

6000 + 24000 TESTING ERROR	BACKWARD	FORWARD	LEFT	RIGHT	STOP
BACKWARD	76	2	3	4	3
FORWARD	2	82	0	8	2
LEFT	2	0	89	4	0
RIGHT	0	0	2	83	5
STOP	2	12	1	6	68

Webcam Image Capturing

To facilitate the input feed into the crafted neural network we implemented a webcam image capturing module. The initial plan involved getting the raw video data and converting it into stream of images, however, this approach was slow and required more processing time than what was expected.

With the help of python's *cv2* modlule we modified the implementation and instead of using "cv2.VideoWriter_fourcc()" we utilized the "cv2.VideoCapture()" with "cv2.imwrite()" to capture images using webcam directly. Further, the code was modified to capture images until user interrupt is encountered and the design allowed only 100 recent images to be stored in the directory path to optimize disk space. Python *time* module was used to keep track of elapsed time and can be used to alter frame rates when needed.

Example Webcam image capture from Webcam_Capture.py file

Simulated Environment

Understanding the constraints involved due to COVID-19 we adapted our approach for experiments and created a simulated environment instead of a using robot. For our implementation we chose *pygame* module in python. The pygame module allows us to create a 2D environment to simulate our experiments.

The pygame simulated environment snippet is as below:

The game starts with the red car placed roughly in the middle and towards the left of the window. The objective of the game is to reach and park the car at one of the two targets which lie on the path. The output from the dense layer feeds into the game and produces the movements according to the kinematics defined. All the gestures i.e. backward, forward, left, right and stop are fed into the prediction network and its output drives the car.

We had to manually perform adjustments and tuning for the velocity, acceleration, chassis length and steering parameters to produce smoother kinematics justified to the map being used. We found that these parameters can be scaled according to the car to map size ratio. For example, if the car is small then we can have control over faster speed and acceleration due to the smaller chassis length steering angle. For a larger car vice-versa is true.

The target objects introduced in the scene utilizes collision detection approach [4] to terminate the game. After the collision event the experiment stops when car reaches one of the two targets and waits for user key input to exit game.

```
def update(self, dt):
    self.velocity += (self.acceleration * dt, 0)
    if self.steering == 0.0:
        angular_velocity = 0
    else:
        turning_radius = (self.length / np.sin(np.deg2rad(self.steering)))
        angular_velocity = (self.velocity.x / turning_radius )
    self.position += self.velocity.rotate(-self.angle) * dt
    self.angle += (np.rad2deg(angular_velocity) * dt)
```

Update method under class *Car* defines the kinematics [5] used to update the car position as per the input provided by the user.

User Study

We wanted to perform an HRI experiment to test the interaction between the user and our network. Unfortunately, we were only able to get the image capture through zoom working late and could only get 3 participants. There are also difficulties with zoom as there is delay between the person gesturing and the video being received, further compounded by our networks response time to sample the video and predict a gesture.

We first walk the user through each of the commands and how to correctly use the gesture to get optimal performance. We then give them an open space to test the gestures and how they interact with the car in terms of acceleration and turning speed. Next, we load the participant with a map that has obstacles and goals. The user is then instructed to navigate to one goal at a time, coming in contact with as few obstacles as possible. We kept track of the time the users took to reach the obstacle and the number of obstacles hit.

The testing took a more informal approach as it was through zoom and only done with roommates available in our respective apartments. The users were allowed to reset the environment if they felt they were too far off course or getting frustrated in their attempt to reach the goal. Because they were able to retry multiple times it gave them the ability to reduce the number of obstacles, they hit on their best run this makes obstacles hit a somewhat unreliable statistic. Attempts taken is also only a semi-useful statistic as an attempt can be as long or as short as the user determines meaning you can choose to stick with it for longer and get additional practice or continually retry depending on frustration. If we had done the testing again, keeping a record of the time it took for each attempt and the obstacles hit for each attempt would have useful statistics in order to look at on average per participant. The information can be found in the following table:

Participant	Goal 1 Time	G1 Obstacles Hit	G1 Attempts	Goal 2 Time	G2 Obstacles Hit	G2 Attempts
1	51s	3	3	57s	3	3
2	68s	2	2	125s	7	2
3	40s	0	1	66s	0	1

The more interesting information was in the responses to our questions:

On a scale of 1-5, 5 being the most natural how natural did the hand commands feel?

On a scale of 1-5, 5 being the most confident how confident were you in the networks ability to understand your hand gestures?

On a scale of 1-5, 5 being the most anxious how anxious did you feel while navigation the car using hand commands?

On a scale of 1-5, 5 being the most likely how likely is it that you would use this technology controlling your car in real life?

On a scale of 1-5, 5 being most frustrated how frustrated were you if you could not reach the goal?

Participant	Naturalness of Hand Gestures	Confidence in network	Anxiety	How Likely to use in real life	Frustration level
1	2	4	3	3(with collision detection)	3
2	3	2	4	2	3
3	3	5	1	1	1

The participants mentioned that the gestures became more natural as they experimented with them more and more during the testing. Some of the participants in the study had a much easier time having their gestures be picked up by the network which contributes to such varying levels of confidence. It also shows the higher the confidence generally correlates to lower anxiety and frustration. The most telling question about the technology as it stands now is the likeliness to use it in real life as the responses are very low.

Discussion and Future Work

We had initially planned to adapt HandSegNet and PoseNet to be trained on the FreiHAND dataset, which consists of images of real hands and was compiled by the same Uni Freiburg group as the RHD. Naïvely we assumed the datasets would be in the same, or nearly the same format and would just require rescaling the images and masks, but we found that they did not contain the same data in the same format. There were no depth images, so we made tried making a fake depth layer with a constant depth. The main issue was the difference in mask format: the FreiHAND was provided as black for the background and white for the hand, palm and wrist, and the RHD has individual finger and palm layers. This is visually very hard to detect; unless you are looking closely at the image it is not obvious that the hands are not a single brightness. Unaware of this difference in structure, we applied a simple

conversion to make the FreiHAND masks the same shape as the RHD ones. Upon discovering that the RHD has different layers for different hand parts it is clear why training HandSegNet on FreiHAND failed. We do not see a robust way to convert the FreiHAND mask to have all the layers of the RHD, but it should be possible to reimplement HandSegNet and PoseNet to train on the FreiHand format. Both datasets provide the type of mask output by HandSegNet and the key points output by PoseNet. For the sake of time we decided to focus on our Gesture Network, and to use the RHD-trained HandSegNet and PoseNet. We suspect that training the entire network on a dataset of real images, such as FreiHAND, would affect great performance gains.

We encountered several other challenges during the training process. The delays caused trying to adapt FreiHAND were compounded by having only one computer between us able to run the training on a GPU. We came up with an initial design of the gesture network fairly quickly, and then collected training data and debugged syntax errors that caused TensorFlow to fail. However, this created a substantial bottleneck when training, and more importantly when debugging semantic errors, and tuning the learning rate.

We discovered some of the areas in which HandSegNet and PoseNet did not produce good results. Our initial choice of gestures specified fingers being tightly together, but the RHD contained hands in more natural positions, so the predicted keypoints were incorrect most of the time. This was compounded by the HandSegNet having trouble with shadows on the hands, and to a lesser extent overexposed sections of the hand. Qualitatively this seems to have led to poorer performance by HandSegNet and PoseNet with darker skin tones. This is hopefully where a more realistic dataset, like FreiHAND would come in use.

During our initial training attempts Gesture Command Net failed to learn. We had used a learning rate from the 3DPoseNetwork which was designed to be trained on the RHD, which has orders of magnitude more unique training images than we were training on. The network appeared to be able to train, however upon evaluating the network on the training data it was clear that it was not learning, but had fallen into a local optimum of classifying every gesture as a single gesture. We believe that this behavior was related to a misunderstanding of how the shuffle_batch in TensorFlow worked. Before this we expected that the code was shuffling our training data, but it appears to have been shuffling the order of batches, but the batches were pulled contiguously from our training data. We created the training data by converting video frames to still images, so our training data was highly locally correlated, with a single batch likely containing only one person demonstrating one gesture and the frames from the same ~3 seconds of video. Our hypothesis is that these highly correlated samples yanked the gradient towards performing well on a single gesture, by classifying everything as that gesture and then the optimization was unable to climb out of this behavior. Upon realizing our misunderstanding of shuffle_batch, we manually shuffle the image order when creating the database, and increased the batch size from 8 to 64. As our training data is nearly evenly distributed for people and gestures these two steps should ensure that each gradient sees a relatively representative set of images. The sum of these changes finally resulted in a network which successfully learned.

We then moved to testing the performance of the network. The initial network trained on Alex, Kyle and Devin's images worked relatively well for them, but displayed poor performance when tested on Vinayak. It also subjectively was not robust, with small changes to the gestures causing them to be mis identified. We tried to address both of these in the second and third networks. Hoping that using more

frames from the videos would make the gestures more stable, and that training on Vinayak would significantly improve the networks performance on his images. We also thought that training the third network with the weights for HandSegNet and PoseNet unlocked might help it overcome the issues those networks were having with Vinayak. These steps did seem to improve the performance somewhat in most cases, particularly in the test of Alex's poorly lit video. We however saw only a small performance improvement with these methods on Vinayak's test images.

The network and simulation are capable of good, predictable performance. Both on those who it was trained on, as subjectively observed by Devin, Kyle and Alex and on someone who the network had never seen before, HRI subject 3. It can also perform poorly on someone it was trained on in Vinayak, and those it was not such as the other two HRI subjects. This suggests to us the more realistic, and diverse training data would likely help. And that the network seems to require relatively good lighting, and image quality. An issue we ran into when running the program remotely over zoom.

On the basis of responses obtained from the HRI experiment we observed the need of a reset command in the pygame implementation. Participants also demanded collision detection for car off-roading. We seek to implement these changes as future work.

Conclusion

We developed a system to control a simulated car through hand gestures. We trained a neural network to classify the gestures, captured images from a webcam, and fed the output to a game where the user controls a simple car. In a small study, naïve users were able to drive the car to two targets with little to moderate frustration. We feel this demonstrates the feasibility and potential of this approach.

References

[1] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, p. 1097–1105.

[2] S.-E. Wei, V. Ramakrishna, T. Kanade and Y. Sheikh, *Convolutional Pose Machines,* 2016.

[3] C. Zimmermann and T. Brox, "Learning to Estimate 3D Hand Pose from Single RGB Images," in *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[4] http://rmgi.blog./pygame-2d-car-tutorial.html : RMGI blog on game development.

[5] https://github.com/Rion5/Pygame_2D_Game/blob/master/game.py : Reference site for collision detection.