Final Project

Hand Gesture Controlled Roomba

5/6/2023

Kiwon Sohn

Zachary Biernat

Objective:

The purpose of this project is to control a Roomba using hand gestures. This will be done by having a master computer publish string messages with the movement should do. These messages are forward, backwards, left, right, and stop. Which message is published is determined by a convolution neural network which in real time reads an input from a webcam and classifies what the hand gesture corresponds to. A second computer, also known as the slave, subscribes to these string messages and publishes messages to control the Roomba's movement accordingly.

Background Theory:

When performing the task of image classification, a very common approach is to use deep learning. This is when a neural network, consisting of many layers, performs transformations on data before feeding the transformed data into a final layer. The final layer tends to be a linear classifier, which on its own can only classify between linear relationships of the input variables. As the data gets transformed using non-linear transformation before being input into the classification layer, the neural network can approximate and classify any continuous function in theory. These transformations are learned by training the neural network on large amounts of data, and using a process called backpropagation to update the weights of the neural network, allowing for better and better fits of the data.

The most basic form of neural network layer is called a fully connected layer. It uses weights, which are optimized during training, to apply linear transformations to the layers input. The fully connected layer can be defined with the following equation:

$$fc(\hat{x}) = W\hat{x} + \beta$$

Equation 1:Fully connect layer

$\hat{x}$ is an input vector of size N, representing the input. W is the weight vector. It is of shape N by M, where M is the size of the input. β is the basis term and functions similarly to an intercept in linear equations. It is of size M. The output of the fully connected layer of a vector of size M, which is a linear transformation of $\hat{x}$.

As linear transformations are only able to represent linear relationships of data, non-linearities are required to learn non-linear relationships. A very common non-linearity is the Rectified Linear Unit (ReLU). It can be defined as:

$$ReLU(x) = \max(0, x)$$

Equation 2: ReLU definition

When applied to a tensor or vector, the ReLU will go through each element, and replace every element that is less than 0, with 0. In a simple neural network, the ReLU will be applied after every fully connected layer.

The final layer required for a simple classification network is the SoftMax layer. It takes an input vector, $\vec{x}$, of size K and outputs a vector of size K. Each element of the output vector represents one of K classes, with the value of the element being the probability that the input is of the class. The SoftMax function can be represented with the following equation:

$$\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

Equation 3: Softmax function

Using these layers we could define a simply neural network as the following:

$$Fully\ connected => ReLU => (Fully\ connected => ReLU => \cdots)$$

$$=> Fully\ connected\ with\ output\ size\ of\ K => Softmax$$

This neural network can perform well with tabular data but requires the input to be a flat vector. This causes issues as the network will fail to properly represent the dimensions of images, not recognizing the image has multiple rows instead of being flat. The idea of the convolution is taken from classical image processing to fix this issue. The convolution slides a filter of a set size (i.e. 3x3) over an image, taking the dot product of every position. This is done for every channel (in the case of a RBG image, there would be 3 channels, 1 red, 1 blue, 1 green). As the filter weighs the result is a linear transformation, much like the fully connected layer. This means you can replace most of the fully connected layers in the previously described neural networks with convolution layers. As the output of a 2d convolution layer is still a matrix,

and the input of the SoftMax function must be a vector, you are required to flatten the data and input it to a fully connected layer before the data reaches the classification layer.

Another common type of layer is the pooling layer. This takes a matrix input and reduces the dimensions of it. An example of this would be an average pooling layer. This layer will loop through patches of elements (i.e., 2x2 patches) and for each patch, have a corresponding element on the output matrix which will be the average of the elements in the patch from the input matrix. This has many benefits, such as helping prevent the model learning the training data too well, making it perform well on training data but bad on new data. It also decreases the time it takes to run the network as convolution is expensive and is faster with smaller input sizes.

It has been found that if neural networks are too deep they will underfit and do worse than more shallow neural networks. To fix this problem shortcut connects were created in 2015 for the neural network architecture of the resnet [1]. The resnet breaks the layers of the neural network into residual blocks. The input and output dimensions of a residual block are the same, allowing for shortcut connections. After a block, the blocks input is added to its output then put through a ReLU. This addition is called the shortcut connection. The residual block used for the resnet50 can be seen in figure 1.

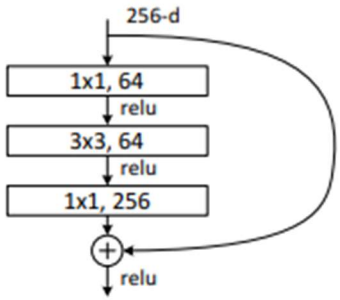

Figure 1:Residual block with skip connection [1]

The resnet is broken up into 4 different stages. Each stage maintains the same input and output size of the other blocks within its stage. Every time the resnet progresses to the next stage, the amount of channels is double, but the resolution is halved. After the fourth stage the resnet is fed into an average pooling layer, then a fully connected layer, and finally the SoftMax layer. This can be seen in figure 2 with a comparison to a normal network architecture, and the VGG-19.

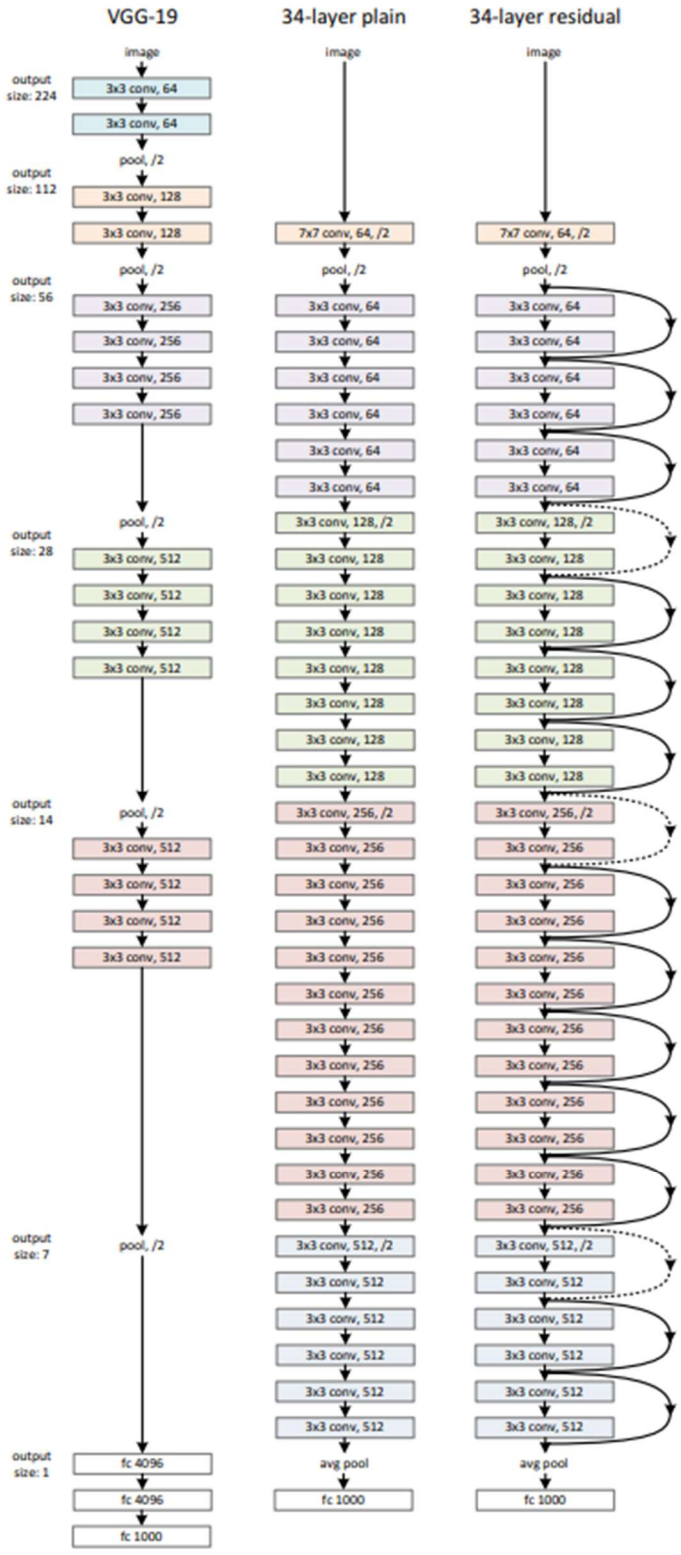

Figure 2: Resnet34 architecture

Training a model like the resnet from scratch requires large amounts of data and computation. Luckily through a process called transfer learning it is possible to use a model like resnet, that has been trained on large amounts of data, such as ImageNet, and retrain it on other tasks. This is done by replacing the SoftMax and final fully connected layer with a new classifier head. The original model is frozen, not allowing for changes in weight. The classifier head is then trained on the new data with the original model extracting important image features. If the new data is very different from ImageNet, it is often work unfreezing the original model and retraining the entire model with a lower learning rate after training just the classification head. This is called finetuning the model.

Results and Analysis:

The Hand Gesture Recognition Image Dataset (HaGRID)[2] was used to train a resnet50 within TensorFlow. The dataset contains images of many people performing many different hand gestures, and bounding box annotations of the hand gestures. Due to the size of the dataset the subsample of 100 images per gesture was used instead, with five gestures being used. These gestures were like, to move forward, dislike, to move backwards, stop, to stop, like, to move left, and fist, to move right (examples of these are omitted from the report and GitHub to avoid copyright violations but can be found on the HaGRID GitHub).

As the images contained the full body, and were intended for object detectors, the hands were cropped to only contain gestures. This was done in the follow code but open an image, reading its bounding box of the hand gesture from the annotation, and crop the image to only be the pixels contained in the bounding box:

```
import cv2
import json

f = open('ann_subsample/forward.json')


def crop(frame, bbox):
    """
    Crops the frame to only give the bounding box
    :param frame: Image
    :param bbox: [x_min,y_min,width,height]
    :return: Cropped image
    """
    '[0.44759481, 0.44759481, 0.16477829, 0.16282391]'
    y_max, x_max, _ = frame.shape
    min_y = int(bbox[1] * y_max)
    height = int(bbox[3] * y_max + min_y)
    min_x = int(bbox[0] * x_max)
    width = int(min_x + bbox[2] * x_max)

    return frame[min_y:height, min_x:width]
```

```python
def get_gesture_index(label_list):
    """
    Gets the index for the correct label form the label_list
    :param label_list: List of labels
    :return: Index in which proper label is at
    """
    x = 0
    for i in label_list:
        if i == 'no_gesture':
            x += 1
        else:
            return x
    return -1


def get_bbox(json):
    """
    Extracts correct bbox from json
    :param json: Single json
    :return: bbox list
    """
    idx = get_gesture_index(json['labels'])
    return json['bboxes'][idx]


def crop_by_json(gesture, json_key, json):
    """
    Saves cropped photo into cropped_data folder
    :param gesture: Gesture being cropped
    :param json_key: Single json_key
    :param json: single json
    :return: None
    """
    frame = cv2.imread(f'data/{gesture}/{json_key}.jpg', cv2.IMREAD_COLOR)
    bbox = get_bbox(json)
    frame = crop(frame, bbox)
    cv2.imwrite(f'cropped_data/{gesture}/{json_key}.jpg', frame)


def save_all_by_gesture(gesture):
    """
    saves all imgs of a gesture
    :param gesture: gesture being saved
    :return: none
    """
```

```
    f = open(f'ann_subsample/{gesture}.json')
    json_list = json.load(f)
    for i in json_list.keys():
        crop_by_json(gesture, i, json_list[i])


label_list = ['backward','forward','stop','left','right']
for i in label_list:
    save_all_by_gesture(i)
```

Figure 3:Script to crop training photos

As this script only extracts the hand making the gesture, some assumptions are added to the model. The image being classified must be a close shot containing mostly the hand gesture. The photo also must contain a hand gesture.

After the cropping was run, we were left with 500 photos, 100 of each gesture. The data was loaded into a pandas dataframe, containing a column for image path, and a column for the image labels. A split of 80% train, 10% test, and 10% validation was generated with the following code:

```
from sklearn.model_selection import train_test_split
x_train,x_val,y_train,y_val = train_test_split(unsplit_data['Image'],
                                               unsplit_data['Label'],
                                               test_size=.2,random_state=42)
unsplit_data = {'Image':x_val,'Label':y_val} #puts data into dictionary
unsplit_data = pd.DataFrame(unsplit_data) #puts dict into df
x_val,x_test,y_val,y_test = train_test_split(unsplit_data['Image'],
                                               unsplit_data['Label'],
                                               test_size=.5,random_state=42)
```

Figure 4:Splitting the dataset

An ImageDataGenerator was used for inputting the data into the model. The data generator applied preprocessing of the image, such as resizing and normalization. For the training data it applied data augmentation of width and height shifts, rotations, and horizontal flips. As the image gets resized based off of bounding boxes, and the shapes of the bounding box differ based on the hand gesture (i.e., stop is taller than rock), having both width and height

augmentation is very important to ensure the model does not learn based off of height and width alone. The rotation helps ensure the hand does not have to be perfectly aligned to the camera. The horizontal flip helps make the model less hand dependent. The code to create the data generators can be seen below:

```
from tensorflow.keras.applications.resnet50 import preprocess_input
#Create data generators
train_data_gen = ImageDataGenerator(
                                preprocessing_function=preprocess_input,
                                rotation_range=15,
                                width_shift_range=.2,
                                height_shift_range=.2,
                                horizontal_flip=True)
val_data_gen =  ImageDataGenerator(preprocessing_function=preprocess_input)
test_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
test_generator = test_datagen.flow_from_dataframe(
    test_gen,
    x_col='Image',
    y_col = 'Label',
    target_size = (224,244),
    batch_size = 64,
    shuffle = False,
    class_mode='sparse',
    validate_filenames=True
) train_generator = train_data_gen.flow_from_dataframe(

    train_gen,
    x_col='Image',
    y_col = 'Label',
    target_size = (224,244),
    batch_size = 64,
    shuffle = True,
    class_mode='sparse',
    validate_filenames=True
)
val_generator = val_data_gen.flow_from_dataframe(
    val_gen,
    x_col='Image',
    y_col = 'Label',
    target_size = (224,244),
    batch_size = 64,
    shuffle = True,
    class_mode='sparse',
```

```
    validate_filenames=True
)
```

Figure 5: Data Generator

The model was first trained with a resnet50 that was frozen. It was used as a feature extractor and went into a dropout layer with a dropout of 40%. This then went into a 5 neuron SoftMax function. The code to create the model can be seen below.

```
def create_model(drop_out=.1):
    resnet = ResNet50(include_top=False,weights='imagenet', pooling='avg')
    model = Sequential()
    model.add(resnet)
    model.add(Dropout(drop_out))
    model.add(Dense(5,activation='softmax'))
    return model

model = create_model(drop_out=.4) #Create model
model.compile(optimizer=Adam(learning_rate=0.0001,decay=0.01), #compile model
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Figure 6: Model Creation

Adam was used as the optimizer with a learning rate of 0.0001, and a weight decay of 0.01. When the model was run with a learning rate of 0.001, the model would fail to improve validation loss, having the loss explode and become a nan value. Due to this the learning rate was lowered. The resnet50 was picked as a model as it has been a very useful model historical and is small enough to run on a Jetson Xavier. It also is included in base TensorFlow, which was originally a requirement for the model to be able to run on the given hardware. Adam was picked for the optimizer as it tends to be the best preforming optimizer with little hyper parameter tuning. AdamW is better in cases where weight decay is used, however it is not included within base TensorFlow, therefore was not used.

The model was saved every epoch that its validation accuracy improved. It also had early stopping which would stop training the model if the model failed to improve after 3 epochs of training. The code this this is scene below:

```
filepath = 'checkpoints/frozen.hdf5'

checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath=filepath,
                                                monitor='val_accuracy',
                                                save_best_only=True,
                                                verbose=1,
                                                mode='max')
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_accuracy',
    min_delta=0,
    patience=3,
    verbose=0,
    mode='auto',
    baseline=None,
    restore_best_weights=True,
)
callbacks = [checkpoint,early_stop]
```

Figure 7:Callbacks

Afterwards the model was trained for a maximum of 50 epochs, with the model converging at 100% training accuracy, and 98% validation accuracy after 8 epochs with the following code:

```
history = model.fit(
    train_generator,
    epochs=50,
    validation_data=val_generator,
    callbacks=callbacks)
```

Figure 8: Training the model

Afterwards the best weights were restored, and the resnet model was made trainable with the following code:

```
from keras.models import load_model
model = load_model('checkpoints/frozen.hdf5')
#Unfreeze layers
for layer in model.layers:
  layer.trainable =  True
```

Figure 9: Unfreezing the layers

This model was then trained in a similar fashion, with the only difference being the learning rate being decreased to 0.00001. The accuracy from model these models were the

same. The final test accuracy of the model was 96%. The model only mistakenly predicted stop as right twice and was correct for every other image.

This model was then sent over to a Jetson Xavier. The Jetson Xavier was connected to the same network as the laptop that was to control the Roomba. The Jetson Xaiver was the ROS master, therefore roscore was run on it. It then used OpenCV to access the webcam and used the model to predict the hand gesture. These gestures where published to the chatter topic with a string message. The code for this can be found below:

```
import rospy
from std_msgs.msg import String
import cv2
import tensorflow as tf
from tensorflow.keras.applications.resnet50 import preprocess_input
from keras.models import load_model
import numpy as np




def talker():
    model =
load_model('/home/ece5489/catkin_ws/src/beginner_tutorials/scripts/finetuned2.hdf
5') #Load pretrained model
    vidcap = cv2.VideoCapture(0) #turn on camera
    label_list = ['backward','forward','left','right','stop'] #list of labels
    #define publisher
    pub = rospy.Publisher('chatter', String, queue_size=1)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    #check if video open
    if vidcap.isOpened():
        while not rospy.is_shutdown():
        #read crop and show frame
            ret, frame = vidcap.read()
            frame = frame[:255,:255]
            cv2.imshow('frame', frame)
            #model prediction
            frame = preprocess_input(frame) #preprocess frame
            frame = tf.expand_dims(frame, axis=0) #expand dim to mimic batch size
            results = model.predict(frame) #predict image
```

```
            string = label_list[np.argmax(results)] #get label of result
            rospy.loginfo(string) #print
            pub.publish(string) #publish
            rate.sleep()
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break


if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Figure 10: Publisher

By adding the line “export ROS_MASTER_URI=http://IP_OF_MASTER:PORT_OF_MASTER/” to the .bashrc file, the laptop was able to receive the messages published by the publisher. Using the following code it was able to use these messages to move the robot accordingly.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "std_msgs/String.h"
#include "string.h"

ros::Publisher pub;
std::string gesture = "";
//so i dont have to cast types
std::string forward = "forward";
std::string backward = "backward";
std::string left = "left";
std::string right = "right";
geometry_msgs::Twist makeTwist(int x,int z){
/**
creates twistMsg with x and z set
x is forward/back
z is left/right
*/
    geometry_msgs::Twist twistMsg;
    twistMsg.linear.x = x;   //Linear Values
    twistMsg.linear.y = 0;
    twistMsg.linear.z = 0;
```

```
    twistMsg.angular.x = 0;  //Angular Values
    twistMsg.angular.y = 0;
    twistMsg.angular.z = z;

    return twistMsg;
}



//Calculate Loop Stop Time
ros::Time getEndTime(double dur){
/*
calcuates how long to move robot for
*/
  ros::Time startTime = ros::Time::now();
  ros::Duration seconds = ros::Duration(dur);
  ros::Time endTime = startTime + seconds;
  return endTime;
}
void move(double dur,int x, int z){
/**
moves robot for dur seconds, in (x,z) direction
*/
    geometry_msgs::Twist twistMsg = makeTwist(x,z);
    ros::Time endTime = getEndTime(dur);
    while (ros::Time::now() < endTime)
    {
      pub.publish(twistMsg);
      ros::Duration(.01).sleep();
    }
}

void move2(double x, double z){

    geometry_msgs::Twist twistMsg = makeTwist(x,z);
    pub.publish(twistMsg);
    ros::Duration(.01).sleep();
}

void turn(int z){
/**
backs up robtot then moves robot in z direction
*/
    //move(1.0, -1, 0);
    move(.1, 0, z);
```

```
}

void get_str(const std_msgs::String::ConstPtr& msg)
{
 gesture = msg->data; //save string

}

int main(int argc, char **argv)
{
  const double turnTime = 1.5;    //When roomba rotates
  ros::init(argc, argv, "example1_a");
  ros::NodeHandle n;
  ros::Subscriber sub2 = n.subscribe("chatter", 1, get_str);
  pub = n.advertise<geometry_msgs::Twist>("cmd_vel", 1);

  while(true)
  {
    ros::spinOnce();
  //check the gesture then move accordingly
  if(forward.compare(gesture) == 0){
  move(.1,1,0);
    }
   else if(right.compare(gesture) == 0){
  turn(-1);
  }
  else if(left.compare(gesture) == 0){
  turn(1);
}
  else if(backward.compare(gesture) == 0){
  move(.1,-1,0);
  }
else
  move(.1,0,0);
}
  return 0;
}
```

Figure 11: Subscriber

Conclusion:

1. a. Electrical engineers can create control systems for robots. This allows the I-robot to be able to move and turn according to the sent message. They also help computer engineers create routers for networks which allow the master and slave computers to communicate. In addition, they can create the cameras used to capture images.

b. Computer engineers/scientists work on the program side creating the algorithms, such as the deep learning algorithms, for image processing. They also design tools like ROS to be able to send messages to robots. In addition, they work with electrical engineers to create the computer networks.

c. Mechanical engineers work with the kinematics and movement of the robot. They are responsible for the creation of things such as the wheels of the i-robot allowing it to move.

2. One real world problem that could benefit from hand gestures recognition is sign language to text translation. A vision-language model could attribute certain hand movements with words, and in real time translate what is being signed to text. This would better allow non-verbal people to communicate with those who do not know sign language.

3. The project was overall successful. The model was much more accurate than I was expecting in real world performance, only misclassifying hand gestures when the hand was mostly out of frame. The model was also able to predict an image in less than a tenth of a second on the hardware, making the model faster to publish messages than the subscriber code needed.

As the top corner of the input frame was all the was being used, it was awkward to line up the hand for hand gestures at times. On top of this the hand had to be close to the camera making the line up feel too exact. This could be improved with more time by training an object detector to identify hands and detect the hand gesture based off that. This would make it that as long as the hand was in frame from the entire pixels, instead of a subsection of the webcam, the

model would recognize it. The project could also be expanded to use hand motion to control other types of robots.

In addition, better models, both speed and accuracy wise, can be used when not limited to just base TensorFlow. In addition, the AdamW optimizer would be able to be used without this restriction.

Work Cited

1. K. He, J. Sun, S. Ren, and X. Zhang, “Deep residual learning for image recognition - arxiv.” [Online]. Available: https://arxiv.org/pdf/1512.03385.pdf. [Accessed: 08-May-2023].

2. K. Alexander, M. Andrew, and K. Karina, “Hagrid — hand gesture recognition image dataset - arxiv.” [Online]. Available: https://arxiv.org/pdf/2206.08219.pdf. [Accessed: 08-May-2023].

Additional Information and Helpful Resources

Link to GitHub of project: https://github.com/zbiernat2000/ECE549-model-training

Link to Dataset: https://github.com/hukenovs/hagrid

Link to useful course for understanding deep learning with computer vision:

https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2019/ (Lecture 8 covers resnets)

Jetpack Version: 5.1

TensorFlow Version:2.11

ROS version on Xavier: noetic