EECS 4421 Robotics

Assignment 2

Kamaljit Grewal, Student No: 218 159 483

Jaymit Jinjala, Student No: 217 373 465

Question 1: Line following robot

VIDEOS OF ROBOT AUTO FOLLOWING LINE: [Machine 1](https://drive.google.com/file/d/1h1z37_4ApMbzy6rev-UAbiNf0mm2aX4u/view?usp=sharing) & [Machine 2](https://youtu.be/u9c1FusY-bc)

**How far does the robot drive before losing the line?**

The robot kept to the line for an estimated 25 to 30 minutes.

**Can you build other environments that are easier/harder for the robot to learn on?**

Easier environments: straight lines or smoother turns

Harder environments: sharper curves, thinner lines (if the robot’s camera sees all white and no black line, it’s lost), add fake snow or rain, add more or less light.

**How difficult is it to replace the NN with a simple rule-based line follower. Modify the auto driver code so that you compute the appropriate drive command from the inputs. How far does your version of the robot go before losing the line?**

A simple, rule-based line follower could use the same data metrics as the original line follower but with a simple flag to check if the left or right sensor values are above a threshold.

The changes made to the original code to implement the simple rule based line follower are as follows:

if self.\_auto\_driving:

if key == 120:

self.get\_logger().info(f"Auto driving ending")

self.\_auto\_driving = False

threshold = 0.9

if left\_val > threshold:

self.turn\_left()

elif right\_val > threshold:

self.turn\_right()

else:

self.go\_straight()

else:

if key == 106:

self.turn\_left()

elif key == 107:

self.go\_straight()

elif key == 108:

self.turn\_right()

elif key == 32:

self.stop()

elif key == 113:

self.get\_logger().info(f"Closing node")

exit(0)

elif key == 120:

self.get\_logger().info(f"Auto driving starting")

self.\_auto\_driving = True

The robot was able to stay on path for around 3-4 minutes before losing the line

**The network used here is incredibly straightforward. Try at least two changes to the architecture (e.g., more hidden levels with more/fewer elements per level, connection drop out, etc.) and identify better/worse/similar architectures. You will require a better way of estimating total performance. One way of doing this is to build a standard world and then to capture the robot's pose from gazebo (odometry information) for runs both when done by hand (to establish ground truth) and then the same when running under different architectures.**

I’ve made four versions of line follower:

One is the original with 1 hidden layer and loss of 0.5377 and accuracy 0.5992.

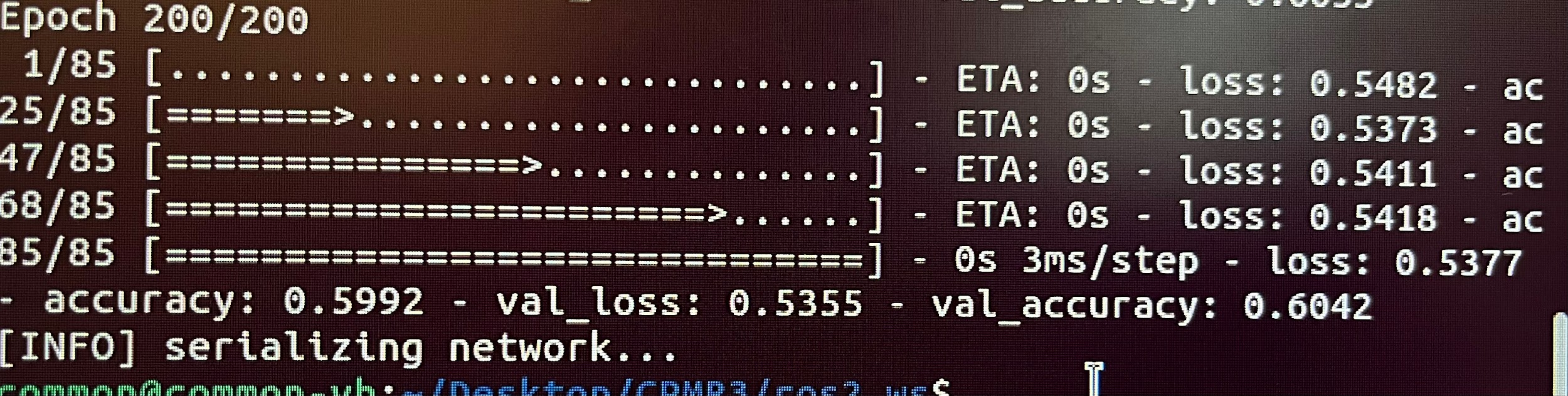
The second has 2 hidden layers and loss 0.5362 and accuracy 0.6036.

The third has 3 hidden layers and loss 0.5427 and accuracy 0.6007.

The fourth has 15 hidden layers and a loss of 0.6172 and accuracy of 0.4659.

Of the four, we see the best accuracy and performance from 2 hidden layers and worst from 15 hidden layers.

Original:



2 hidden layers:

class LineFollower:

def build():

# initialize the model

model = Sequential()

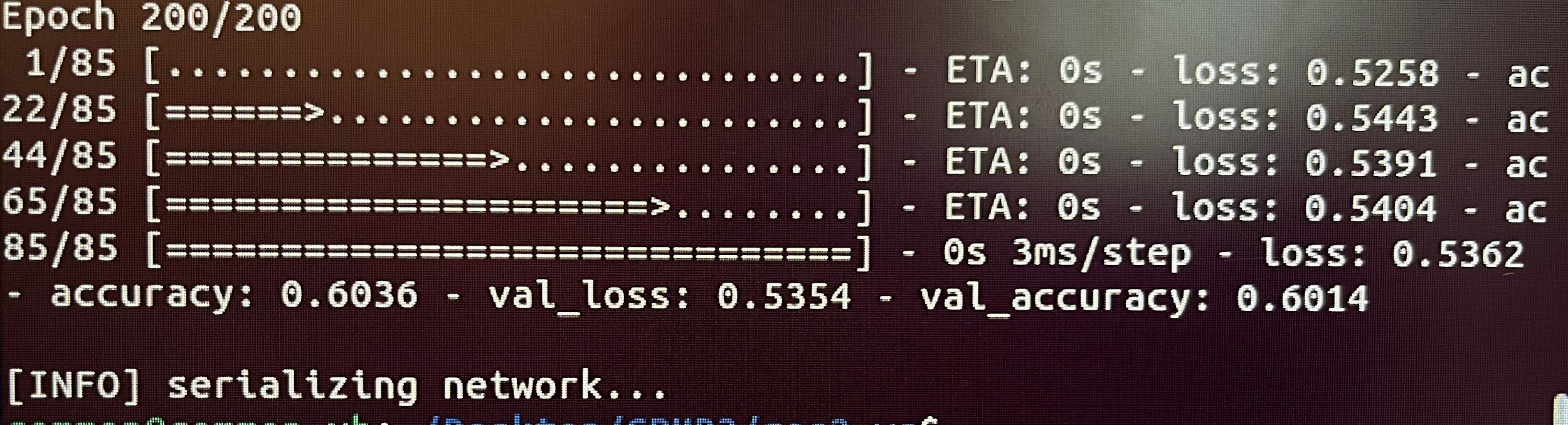
model.add(Dense(8, input\_shape=(2,), activation='relu'))

model.add(Dense(8, activation='relu'))

model.add(Dense(6, activation='relu')) # added one new hidden layer with 6 nodes

model.add(Dense(3, activation='softmax'))

return model



3 hidden layers:

class LineFollower:

def build():

# initialize the model

model = Sequential()

model.add(Dense(8, input\_shape=(2,), activation='relu'))

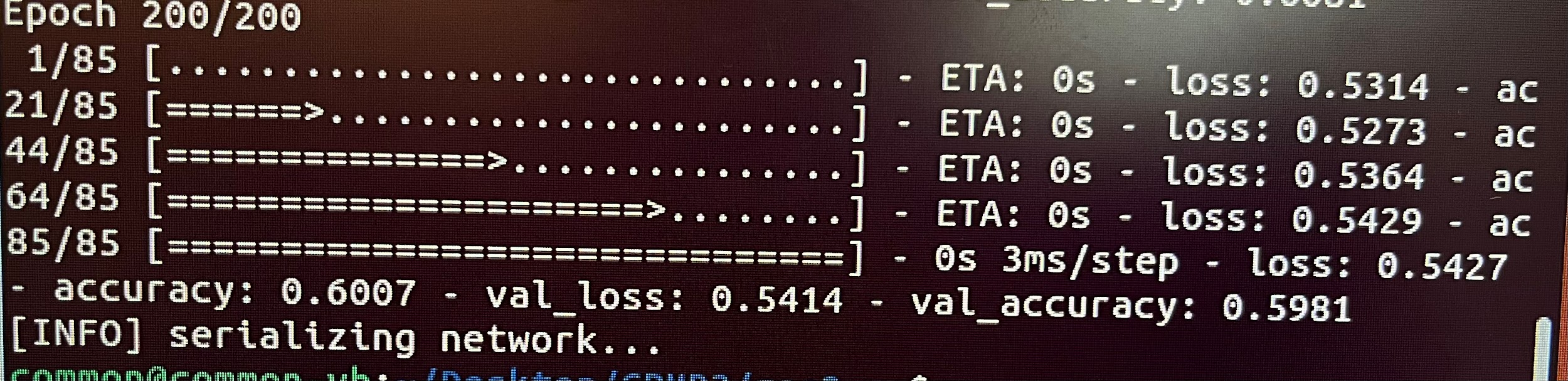
model.add(Dense(8, activation='relu'))

model.add(Dense(6, activation='relu')) # added 2 new hidden layers

model.add(Dense(4, activation='relu'))

model.add(Dense(3, activation='softmax'))

return model



15 hidden layers:

class LineFollower:

def build():

# initialize the model

model = Sequential()

model.add(Dense(8, input\_shape=(2,), activation='relu'))

model.add(Dense(8, activation='relu'))

model.add(Dense(8, activation='relu'))

model.add(Dense(8, activation='relu'))

model.add(Dense(7, activation='relu'))

model.add(Dense(7, activation='relu'))

model.add(Dense(7, activation='relu'))

model.add(Dense(6, activation='relu'))

model.add(Dense(6, activation='relu'))

model.add(Dense(6, activation='relu'))

model.add(Dense(5, activation='relu'))

model.add(Dense(5, activation='relu'))

model.add(Dense(5, activation='relu'))

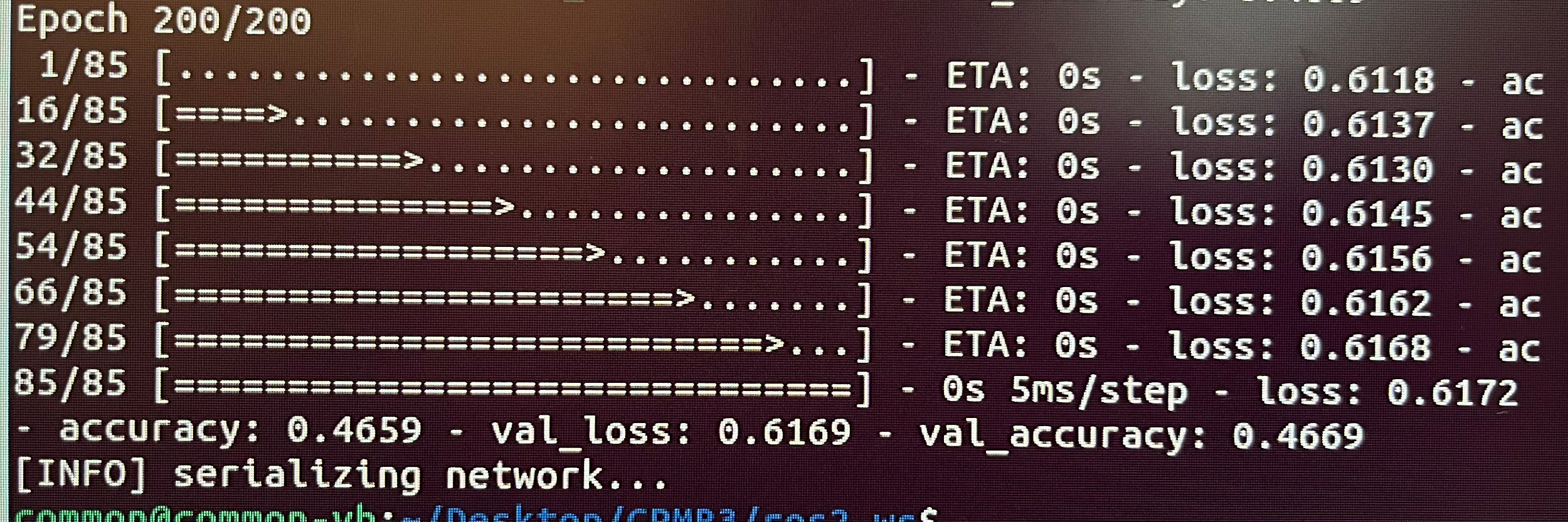
model.add(Dense(4, activation='relu'))

model.add(Dense(4, activation='relu'))

model.add(Dense(4, activation='relu'))

model.add(Dense(3, activation='softmax'))

return model



Adding additional hidden layers and more nodes per layer improves performance, to an extent. More nodes and layers have a greater capacity for picking up more complex relationships and functions in the data.

But, if we have too many layers and nodes per layer relative to the data, then the neural network may start to overfit. It’ll learn the noise in the data instead of just the underlying patterns. It’ll become specialised to the training data and be unable to generalise. Resultantly, the robot won’t be able to follow other lines. As we add more and more layers and nodes, we’ll eventually reach a point where we see a decrease in model accuracy.