ENPH 353 Logbook: Kazu Nakane

Lab 2

Objective

build an algorithm to track the road as the video progress through the road.

Step 1: Extract the center of the road.

Initially, I thought I can use canny edge detection to find the edge. Maybe set a threshold so if the pixel brightness is above the threshold, register the pixel as part of one of the edges.

Next, from this info, extract the center of the path, easiest is take the weighted average of the brightness with index of pixel from the right.

$$ \sum_{i \in row} i * \text{Brightness} / \sum_{i \in row} \text{Brightness}$$

I take the row that is 1/3 from the bottom of the image, since that seemed to capture the sweet spot. Not too close that the output will oscillate crazy, not too far in future that the output is useless for a feedback system for future purposes.

Since the brightness is either 0 or 255, if the weighted average is taken in respect of the index, the average should land between the two clusters of bright pixels, which would be around the center of the road.

This initially worked, but occasionally showed one of the edges as the center, most likely because it Canny did not register the edge for that specific row occasionally. To compensate, I add the Canny result of the row of interest, and each row from above and below. This worked well, since it increased redundancy, and missed edges way less frequently. Good.

Step 2: Smooth the output

Although the output constantly shows the center of the road, it fluctuates quite a lot. This is most likely because of each frame differs in brightness, contrast ...etc and slightly changes the output for Canny.

To smooth this out, I came up with an idea of using a Kalman filter as a noise remover.

What is a Kalman filter: It is a type of filter that reduces the effect of noise with the following steps, according to Chat GPT.

"A Kalman filter continuously predicts the system’s state using a model, then corrects that prediction using noisy measurements, weighting each by how uncertain they are."

In this case, the center detector is the sensor, the state is where the actual center is, and the noise is approximated as a white noise, where the noise does not have a correlation between frames.

I found a library for implementing a Kalman filter, and This is the implementation.

```
from filterpy.Kalman import KalmanFilter
# Kalman object constructor
kalman = KalmanFilter(dim_x=2, dim_z=2)

dt = 1.0 / fps

# init state
kalman.x = np.array([[width // 2], [0]])
# state transition matrix
kalman.F = np.array([[1, dt], [0, 1]])
# measurement function
kalman.H = np.array([[1, 0], [0, 1]])

# how sure of init pos
kalman.P = np.array([[15**2, 0], [0, 10**2]])
# how sure of measurement (positions, velocity)
kalman.R = np.array([[15**2, 0], [0, 10**2]])
# noise in acceleration
kalman.Q = 20**2 * np.array([[dt**4/4, dt**3/2], [dt**3/2, dt**2]])
```

In this case, x is the state, where x[0] is the position of the center, x[1], is the velocity.

I set the initial state as the center located at the center of the image.

F represents the state transition, where

$$\mathbb{x}_{k + 1} = F\mathbb{x}_k$$

H represents the measurement matrix, where the diagonal entries mean that I am measuring both position and velocity for each prediction, that is somewhat independent from each other.

P represents the covariance, where the diagonal entries represents the uncertainty of the initial measurements. In this case, I eyeballed it since these values only mater fot the first few frames.

R represents the measurement error, where the diagonal entries represent the uncertainty in the measured values. This is important, since it is used to determine wether the Kalman filter trust the measured values or the predicted values. I eyeballed the uncertainties based on few frames of measurement data, and these values worked pretty well.

Finally, Q represents the external noise, which represents the fluctuation of the measured center position. Since the variation comes from random changes in brightness, contrast ...etc, it can be approximated as a white noise. The Matrix is taken from Chat GPT, where I asked for a matrix representing brownian motion.

By applying this filter every frame with the measured center position feed to it, the output became very smooth compared to the initial raw outputs. The model successfully removes the random noise with the parameters above. Good

Lab 3

Objective

Use the code from the previous lab to control the robot to follow a line.

Step 1: understand the file structure

What does catkin_make do? ask chat

"catkin_make compiles and registers all ROS packages in your workspace so ROS can find, run, and link them together."

what dose the launce file do?

“Start these nodes, load these parameters, connect these topics, and bring up this world.”

So if I make a node for line following and include it in the launch file, I can launch everything at once with the right parameters.

Different RosTopics -> /image_raw publishes camrea feed, not in CV format so must use CVVbridge to convert to cv style for line following

/Twist Publish the robot movement here, and it will control the tires to reach this motion. Individual control is abstracted away, so I only have to worry about x velocity and Yaw rate.

Step 2: Implement line following

Since the Kalman filter from last time works pretty well, I'll just use that.

under /node, I made 2 python files, one is `LineFollowClass.py`, the other is `linefollow.py`.

LineFollowClass implements the Kalman filter, where the class constructor builds the Kalman filter, and the `get_center()` method runs the center detecter and outputs the filtered center so it can be used as feedback to control the robot.

I changed the output format for `get_center()` so that x = 0 corresponds to the center of the frame, instead of being the far left side of the frame. This makes more sense because I want the PID to make sure the center of the road is aligned with the center of the frame at all time.

`linefollow.py` implements the PID controller, and submits the required motion to /Twist.

```
def line_follower(msg):

    cv_image = bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
    center_offset = lf.get_center(cv_image)

    # Use center_offset to control the robot
    twist = Twist()
    kp = 0.01
    forward_speed = 20
    twist.linear.x = forward_speed
    twist.angular.z = -center_offset * kp
    pub.publish(twist)

rospy.init_node('line_follower')
```

```
pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
sub = rospy.Subscriber('/image_raw', Image, line_follower)

rospy.spin()
```

`line_follower()` takes msg, which is an image, then publisher the output to /Twist. I decided to only do P control since the Kalman filter already removes the noise, which is conventionally removed using the D term. I also removed I term, since it worked perfectly with only P.

I keep the x velocity constant to remove complexity to controlling 2 degrees of freedom.

When this node is executed, it initializes the node, set /Twist as a publisher, subscribe `line_follower()` to the image stream. I didn't know what `rospy.spin()` does, so I asked chat.

"rospy.spin() blocks the program so your ROS node stays alive and can keep responding to callbacks."

So If I don't put `rospy.spin()` there, the node terminates right after setting everything up.

I played around with different k_p values, and 0.01 seemed to work the best.

Lab 4

Objective

Learn how to use SIFT, and to build GUI

Step 1: Build GUI with Qt

what is Qt?

"Qt is a cross-platform C++ application framework used to build professional GUIs, heavily used in Linux, ROS, and engineering tools."

Use Qt designer to build a GUI with GUI. Each element on a QtGUI has an object name, that points to that object, so there cannot be duplicates.

QLabel is a simple label where various things can be displayed. In this case, we use it to display the images.

A button is a button. You can "Subscribe" piece of code to it so the code gets executed on a button push.

Step 2: Understand how to construct the backend

Example code from Miti

```
#!/usr/bin/env python3

from PyQt5 import QtCore, QtGui, QtWidgets
from python_qt_binding import loadUi

import cv2
import sys
```

```
class My_App(QtWidgets.QMainWindow):

    def __init__(self):
        super(My_App, self).__init__()
        loadUi("./SIFT_app.ui", self)

if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    myApp = My_App()
    myApp.show()
    sys.exit(app.exec_())
```

`My_APP` extends a QMainwindow class, which represents the frontend of our app. `loadUI()` finds our ui file, then loads all components inside the UI as part of the class.

under main, the app is executed, and ran until it is closed. `sys.exit(app.exec_())` is almost like `rospy.spin()`, where it blocks the app from terminating until the app window is closed.

Step 3: Build the actual app.

Copy code from Miti.

Constructor loads the UI, initializes the parameters for the camera, attach code to corresponding buttons, setup "timer interrupts" to set intervals to run SIFT on camera feed.

`def SLOT_browse_button(self):` is used to upload the reference image to the UI. It calls a dialogue in file mode, so user can select the image file using a similar GUI as a file viewer. It converts the image to pixmap, then plots it on a label located on the UI. I added code so it runs SIFT on the loaded image, and stores it as a variable in the class. THis way, it only has to run SIFT on reference once, and I will not forget to run SIFT after loading a new image in the first place.

what is PixMap?

"A pixmap is how Qt stores and draws images efficiently inside GUI widgets."

ok its nothing crazy.

`def SLOT_query_camera(self)` is called every time the "timer interrupt" is triggered. It reads the image from the webcam, converts it to gray scale, and finds matching key points with the reference image. The matching is done with the `def match_features(self, threshold=0.75):` method, which I implemented later in the code. Also it tiles the live feed and reference image with lines connecting the key points so the result can be seen easily. The result is plotted on a label to be seen on the UI.

```
def match_features(self, threshold=0.75):
    matches = self.bf.knnMatch(self._descriptors_template,
self._descriptors_frame, k=2)
    good_matches = []

    for m, n in matches:
        if m.distance < threshold * n.distance:
            good_matches.append(m)
    return good_matches
```

This code finds the best matches between the live feed and reference image. The idea is taken mainly from ChatGPT, since I had no idea how to solve this problem.
`self.bf.knnMatch(self._descriptors_template, self._descriptors_frame, k=2)` finds the 2 best matches for each key points. This is done with brute force. the output is formatted as,

```

matches = [
  [m1, n1],   # best and second-best match for descriptor 1
  [m2, n2],   # for descriptor 2
  ...
]
```

according to ChatGPT.

However, these contain bad matches where one descriptor is tied to a completely different descriptor in the image that somehow have very high similarities. To remove this, we run the ratio test. The basic concept is that we compare the descriptor distance between the best and second best match, and we only take the matches where the best match has way higher "confidence" than the second best match. This way, we can eliminate random matches that has loose correlation between the matches.

I thought instead of the L2 norm of the descriptor, we can use cosine similarity to see if the key points are similar, but according to one of the TA, there is no use because of how the vector components are normalized.

At the end, this method outputs only the high quality matches between the two images.

`def create_tiled_matches_view(self, frame, good_matches)` creates a tiled image where the reference and live image is tiled side by side, with lines connecting matching key points. I initially tried to write this method from scratch, but after finding a function that does exactly this in cv, I decided to just use this for convenience.

After implementing these, I ran the app with FPS = 3, which was the recommended value by Miti. It seemed to run no problem, so I increased it to 20 fps. My computer handled it well.

SIFT seem to have a hard time matching key points for the Android logo. I think this is because the logo is very blend where the geometry are mainly straight lines or circles. When I present the Logo in front of the camera, it recognizes some feature points at close distance, but struggles when the logo is held further away.

When I use a image that is more "Feature rich", such as an image of Mt. Fuji, it seems to match features at even greater distances.

I tried with an image of the robot summer "Pets" since our robot used computer vision to detect plusies during the summer. Considering how "Feature rich" this object looks, it had a hard time matching key points. It only seems to be able to match points when the plushie is held straight to the camera. Any bit of rotation of the object along an axis parallel to camera FOV seems to throw off the output of SIFT quite a lot. Also it might have to do with the plushie being fluffy, and not having a clear boundary at its edges.

Lab 5

Objective

Build a NN from scratch, and learn how micrograd works.

Step 1: Value class

The Value class represents a numerical value that computation can be done with. It contains the numerical values, its gradient in respect of the operation done, and its children in terms of operation.

`__add__`, `__mul__`, ... `__radd__` all replaces the operation done for standard arithmetics. This way, for example Value + Value will actually compute the addition of the two values instead of throwing error that two classes cannot be added together.

Every time an arithmetic is done between two values, an topological map is built, where it records its input, operation inside its self.

When `backward()` is called, the parents of a value is back propagated so a tree of operations that leads to the value s built. This tree respects the order of operations and relationship between nodes. It sets the gradient of its self as 1, corresponding to $$\frac{d\text{self}}{d\text{self}} = 1$$, then adds the gradients of its parents recursively to mimic the chain rule.

`def trace(root):` builds a computation graph just like how the topological tree is built when back propagation is done. This helps visualizing the process.

Step 2: Neuron, Layer, MLP class

The neuron class implements a simple artificial neuron. This class wraps the Value class specifically so it acts like an neuron. It takes an input vector x, then computes $$x* w + b$$. Tanh is the default activation function, but I added a special version of tanh where a vertical scaling and vertical offset can be learned so it can fit a wider variation of values.

The weights are set to random values so it breaks symmetry.

The Layer class bunches the neurons to form a single layout for a NN. Each neuron has the same input dimensions since it is meant to see the same input. `parameters()` return the learnable parameters so that the builtin `backward()` method can be used to execute back propagation on each learnable parameter.

The MLP class is a wrapper for the layer class, where it implements a whole NN with one class. For example, `MLP(5, [2, 2, 1])` will create a fully connected NN with 5 -> 2 -> 2 -> 1 neurons on each layer.

I added a code to test this NN class, such as,

```
x = [2.0, 3.0, -1.0]
n = MLP(3, [4, 4, 1])
n(x)
```

```
draw_dot(n(x))

xs = [
  [2.0, 3.0, -1.0],
  [3.0, -1.0, 0.5],
  [0.5, 1.0, 1.0],
  [1.0, 1.0, -1.0],
]
ys = [1.0, -1.0, -1.0, 1.0]


for k in range(20):

  # forward pass
  ypred = [n(x) for x in xs]
  loss = sum((yout - ygt)**2 for ygt, yout in zip(ys, ypred))

  # backward pass
  for p in n.parameters():
    p.grad = 0.0
  loss.backward()

  # update
  for p in n.parameters():
    p.data += -0.1 * p.grad

  print(k, loss.data)


  xs = [
  [2.0, 3.0, -1.0],
  [3.0, -1.0, 0.5],
  [0.5, 1.0, 1.0],
  [1.0, 1.0, -1.0],
]
ys = [1.0, -1.0, -1.0, 1.0] # desired targets

ypred = [n(x) for x in xs]
ypred
```

So the NN is 3 -> 4 -> 4 -> 1 neurons for the layers. For each training loop, its forward pass is calculated, L2 norm is used to calculate the loss. Next, all the grads are zeroed. This is necessary to remove the old grad used in the previous loop. Here, I used a learning rate of 0.1, which was suggested by the tutorial video. There is a negative sign when updating the values, and this is so that the values are nudged in the direction that reduces the loss. Don't forget this one....

The lab portion

I want to fit

```
xs = [[2], [3], [4]]
ys = [2, 1, -2]
```

y is neither a linear , increasing nor decreasing function in terms of the input x, so a conventional activation function cannot be used.

I first came up with a quadratic activation function. Since quadratic can have a both increasing and decreasing part, it can fit this weird data. Given that we have to fit 3 datapoints, we only need 3 variables to tune (Ideally). So I initially used,

$$y = A(wx+b)^2$$

where A, w, b is the learned variables. Mathematically this works, but in fact, it did not work well. It usually goes into a false minima, and staggers with a certain loss. I assumed this is because the constraints are so tight, since we are fitting 3 values with 3 parameters.

Next, I asked Chat GPT what can possibly be the alternative. It suggested to use,

$$y = A \tanh{(wx+b)} + B$$

where A, w, b, B are learned parameters. This is a more loose constraint, since we are using 4 parameters to fit 3 data points. This seemed to work well, and my loss did not stagger like the quadratics.

However, for higher learning rates than 0.02 seemed to not work, and it often had the same issue of going into a false minima. This is probably because many local minima are pretty close together that high learning rate can nudge the parameters into a separate minima, like tunneling.

Lab 6

Objective

Make a CNN that reads characters off car plates

Disclaimer

I don't have much time for this lab, so it will be jank...

Step 1: Generate training data

Miti had the code to generate the car plates with a specific font and color, so I will take advantage of that. I think that is the most consistent way to generate a homogeneous dataset.

```
def draw_char(char):

  h, w = 135, 106
  background = Image.new("RGB", (w, h), (255, 255, 255))

  draw = ImageDraw.Draw(background)
  monospace =
ImageFont.truetype(font="/usr/share/fonts/truetype/liberation/LiberationMon
o-Regular.ttf",size=165)

  draw.text(xy = (0, -10), text=char, fill=(255, 0, 0), font=monospace)

  return np.array(background)

test = draw_char('0')
cv2_imshow(test)
```

This will be a function that can generate an image of a character with the font and color.

Next, I will make a one-hot encode to map the output vector to its corresponding character.

Step 2: randomization for minimizing overfitting

```
import random

def random_rotate_scale(img, max_angle=20, scale_range=(0.8, 1.1)):
    h, w = img.shape[:2]

    # Pick random rotation angle and scale factor
    angle = random.uniform(-max_angle, max_angle)
    scale = random.uniform(*scale_range)

    # Compute rotation matrix about the image center
    center = (w / 2, h / 2)
    M = cv2.getRotationMatrix2D(center, angle, scale)

    # Apply the affine transform (keep size same as input)
    rotated_scaled = cv2.warpAffine(img, M, (w, h),
                                    flags=cv2.INTER_LINEAR,
                                    borderMode=cv2.BORDER_CONSTANT,
                                    borderValue=(255, 255, 255))
    return rotated_scaled
```

I wrote this code to rotate, and scale the image randomly within the threshold, os that the training data varies a little even for the same character. This is not necessary for this lab, but will help in future cases when versatility is needed.

Step 3: generate a crap ton of training data

Next, I wrote a script to quickly generate 100 images per character as training data. Each images are scaled and rotated randomly with the function give above.

I do not save the images in disk, and I have it all stored in RAM. This is because each images are relatively small, and 3600 images (ish) could probably stored in RAM without causing overflow. It is not ideal though, since it is hard to keep track of the training data when it gets swiped and regenerated for each training cycle. This is not the best practice.

I shuffle the training data so they are not correlated, and split it to training and val data with a ratio of 8:2. This number was suggested by chat GPT. The smaller the validation is, the the more correlated the val becomes, and it will not be a good metric for detecting over fitting.

Step 4: Construct the CNN

I used the CNN from the cats vs dogs example, since I did not have much time.

```
#heavy one

conv_model = models.Sequential()
conv_model.add(layers.Conv2D(32, (3, 3), activation='relu',
                         input_shape=(135, 106, 3)))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(64, (3, 3), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(128, (3, 3), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(128, (3, 3), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Flatten())
conv_model.add(layers.Dropout(0.5))
conv_model.add(layers.Dense(512, activation='relu'))
conv_model.add(layers.Dense(36, activation='softmax'))
```

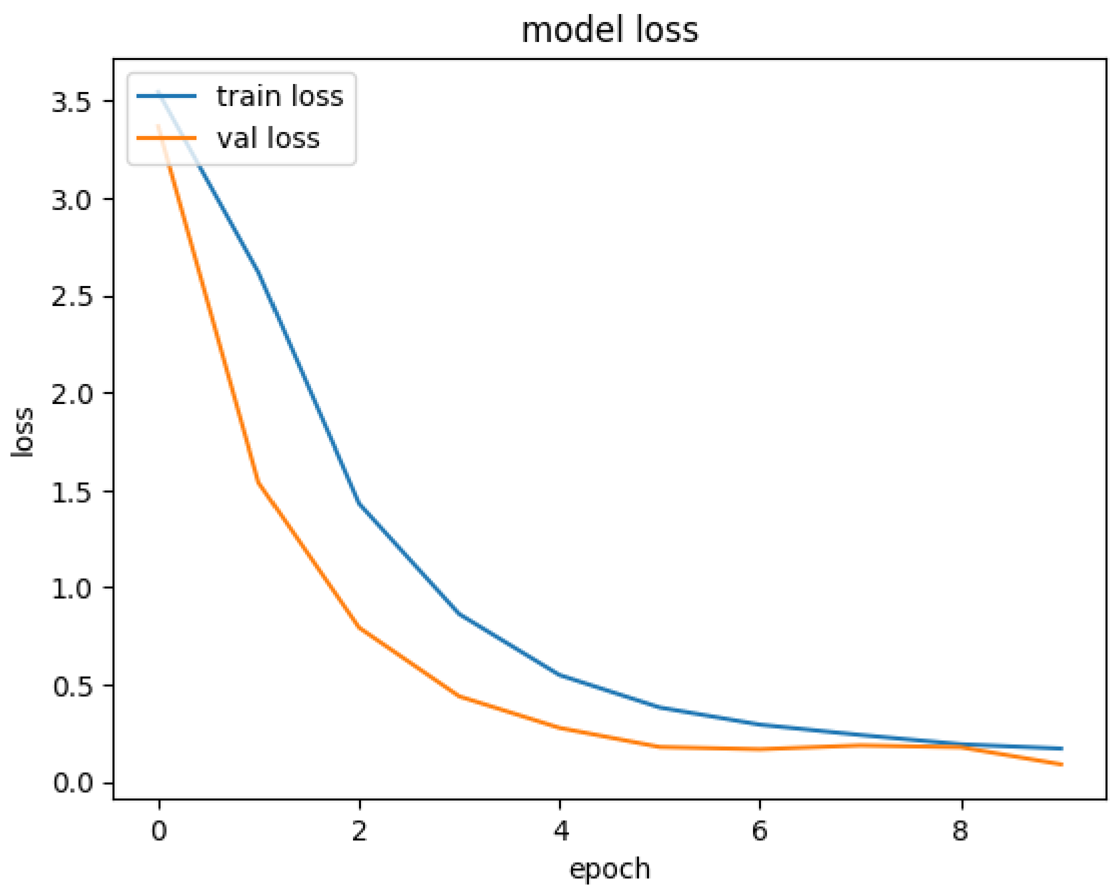
This is huge, and it is overkill since it will output 512 feature maps for a relatively simple input, but it works fine, so I'll take it.

Step 5: Training

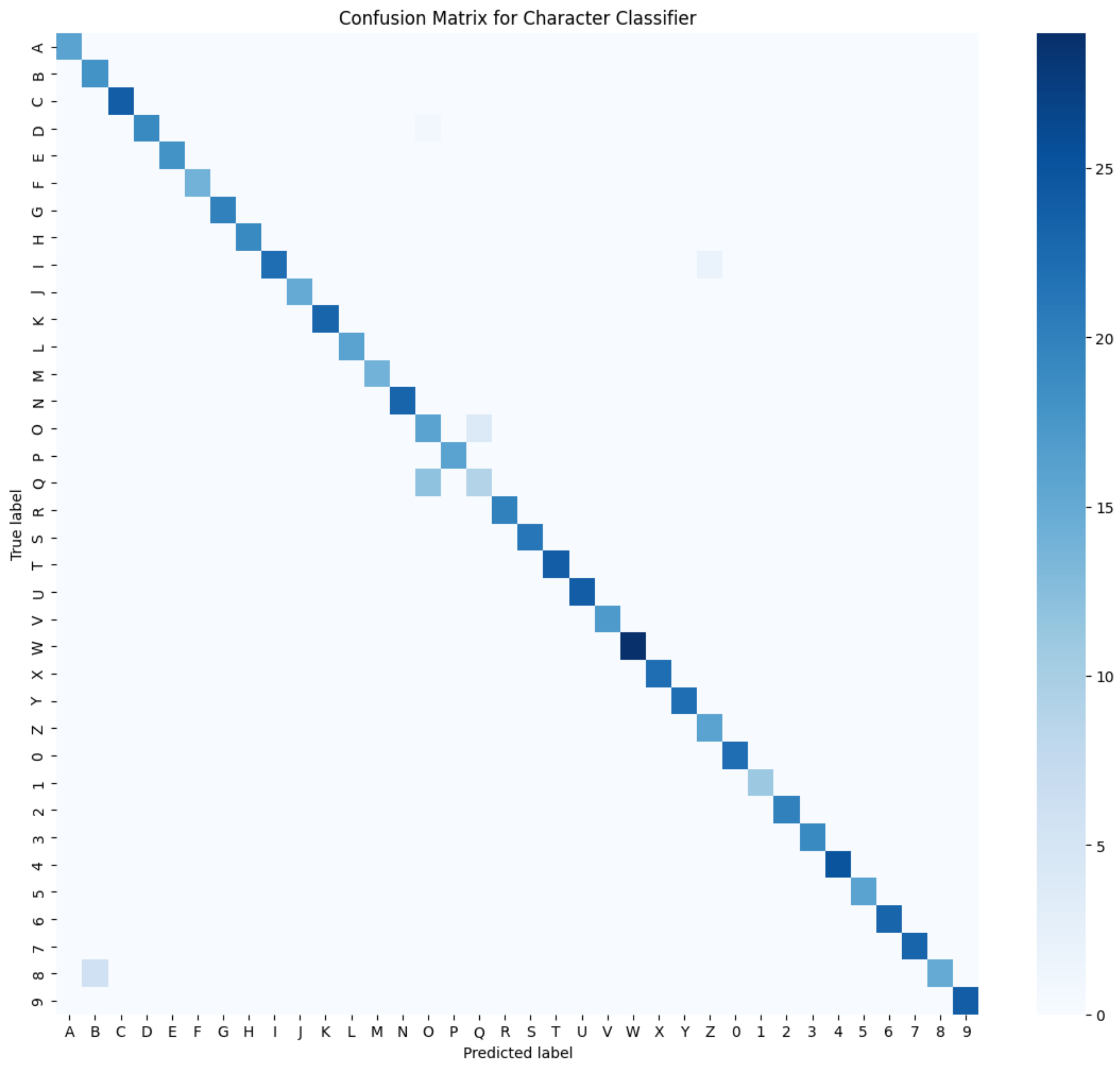
Training in Tensorflow is a one-liner

```
history = conv_model.fit(X_train_dataset,
                     Y_train_dataset,
                     epochs=10,
                     batch_size=32,
                     validation_data=(X_val_dataset, Y_val_dataset))
```

I trained for over 30 Epochs, which seemed to be good practice.



I plotted the confusion matrix, which turned out to be,



Lab 7

Objective

Implement Q-learning for Reinforcement learning

Step 1: Understand Q-learning

What is Q-learning?

"Q-learning learns an action-value function Q(s,a) that estimates the expected future reward for taking action a in state s, assuming optimal behavior afterward."

So basically it is a huge table of state, action from state, and its quality. I update the Quality in the training stage so it is optimized.

Step 2: Start implementing the Q-learning class

`loadQ` and `saveQ` basically stores a python object by serializing it. In this case, it saves Q table.

`getQ` fetches the Q value for a given state and action. The q table is a simple tuple, where the value can be searched with the (state, action) pair.

`chooseAction` chooses the action, given a state. The most important parameter here is `self.epsilon`. This parameter decides how frequently this model takes random actions. It is set high (~0.95) for the initial stage of learning, so the robot can wander around, and explore different states and actions. This helps generalize the model without bias. In other cases, the action with the highest Q value is taken. `self.epsilon` can be set to a low value later on in the training stage, so the robot chooses the most optimal path almost all the time, and only occasionally it will take a random action, so the model gets some perturbation that it has to adapt to.

`learn` implements the updating of Q-values. Each time an action is taken, this method is called to update the Q-value. For each step, It chooses an action based on the criteria above, executes it, and gets the updated state after the action. Based on this information, it computes the updated Q value using the Bellman-eq.

```
new_q = current_q + self.alpha * (reward + self.gamma * max_q_state2 - current_q)
```

`reward` is given by the environment, `max_q_states2` is the maximum earnable Q-value from the new updated state, `self.alpha` as the learning rate. This means the Q-values are updated based on how much Q it can earn in the future, plus how much short term reward it got.

Step 3: Setup training environment

I chose the most basic state and action space. The state space is 10-dimensional vector, where the camera feed is divided into 10 columns, and depending on which column the center of the road falls in, Ths state was assigned. `[1, 0, ..., 0]` means that the center is at the far left.

The action is 3-dimensional, which is turn left, straight, or turn right. Each action is associated with an reward, where turning is a reward of 2, and going straight is a 4. I planned to change this so it gives reward based on how close the center is to the center of the frame, but ended up not having the time to do so.