Chapter 5

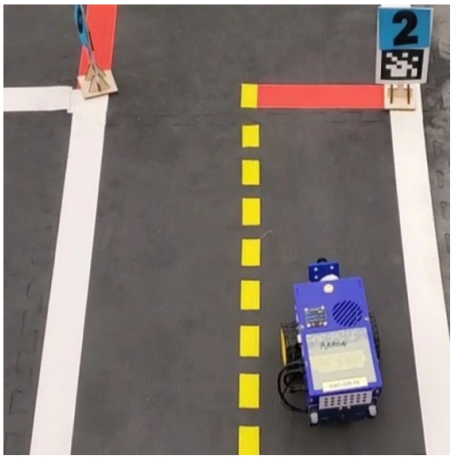


Image 1 - Duckiebot doing detection

Duckiebot learns its numbers

In this lab we were tasked to learn in greater depth about the machine learning by applying several concepts including back propagation for updating our weights and multilayered perceptron, which is one approach in machine learning to execute image classification or this case digit recognition. Ultimately the goal here is to execute what we learned in this lab and apply it to our duckiebot such that it can do number recognition on the large digits that were placed on sticky notes on top of the preexisting sticky notes that were a part of the duckie town.

So in part one, we learn about backpropagation which is a method of updating the weights in a neural network such that we reach a minimized form of our loss function, leading to a reduced prediction error. This updating strategy utilizes what is known as gradient descent and works by going backwards through the network. In the next bit, we will demonstrate a second pass of our backpropagation based on the elements presented in the tutorial that the TA selected.

1. This is the weight update formula based on gradient descent

Gradient descent is an iterative optimization algorithm for finding the minimum of a function; in our case we want to minimize th error function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point.

$$'W_x = W_x - a\left(\frac{\partial Error}{\partial W_x}\right)$$

New weight · Old weight · Learning rate · Derivative of Error with respect to weight

2. This is the application of the chain rule for deriving the error function to obtain the updated weights between the hidden layer and output layer.

$$\frac{\partial Error}{\partial W_x} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial W_x}$$

$$\frac{\partial Error}{\partial W_x} = \frac{\frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial\left((i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6\right)}{\partial W_x}$$

$$\frac{\partial Error}{\partial W_x} = 2 * \frac{1}{2}(\text{predictoin} - \text{actula}) \frac{\partial(predictoin - actula)}{\partial prediciton} * (i_1 w_3 + i_2 w_4)$$

$$\frac{\partial Error}{\partial W_x} = (\text{predictoin} - \text{actula}) * (h_2)$$

$$\frac{\partial Error}{\partial W_x} = \Delta h_2$$

Error = $\frac{1}{2}$(prediction − actual)² · prediction = $(i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6$ · $h_2 = i_1 w_3 + i_2 w_4$ · Δ = prediction − actual ← delta · chain rule

3. This is the application of the chain rule for deriving the error function to obtain the updated weights between the input layer and the hidden layer.

$$\frac{\partial Error}{\partial W_x} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial h_x} * \frac{\partial h_x}{\partial W_x}$$

$$\frac{\partial Error}{\partial W_x} = \frac{\partial \frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial\left((h_1) w_5 + (h_2) w_6\right)}{\partial h_x} * \frac{\partial\left(i_1 w_1 + i_2 w_2\right)}{\partial w_x}$$

$$\frac{\partial Error}{\partial W_x} = 2 * \frac{1}{2}(\text{predictoin} - \text{actula}) \frac{\partial(predictoin - actula)}{\partial prediciton} * (w_5) * (i_1)$$

$$\frac{\partial Error}{\partial W_x} = (\text{predictoin} - \text{actula}) * (w_5 i_1)$$

$$\frac{\partial Error}{\partial W_x} = \Delta w_5 i_1$$

Error = $\frac{1}{2}$(prediction − actual)² · prediction = $(h_1) w_5 + (h_2) w_6$ · $h_x = i_1 w_1 + i_2 w_2$ · Δ = prediction − actual ← delta · chain rule

4. This is the weight update formula in matrix form for all of the weights in our example neural network, which is present down below.

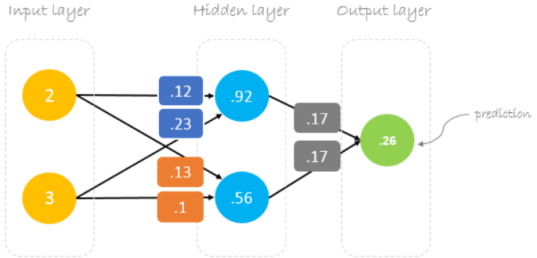
$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - a\,\Delta \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - \begin{bmatrix} a h_1 \Delta \\ a h_2 \Delta \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - a\,\Delta \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} . [w_5 \quad w_6] = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \begin{bmatrix} a\, i_1 \Delta w_5 & a\, i_1 \Delta w_6 \\ a\, i_2 \Delta w_5 & a\, i_2 \Delta w_6 \end{bmatrix}$$

5. Then apply the updated weights for another forward pass

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - 0.05(-0.809) \begin{bmatrix} 0.85 \\ 0.48 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - \begin{bmatrix} -0.034 \\ -0.019 \end{bmatrix} = \begin{bmatrix} 0.17 \\ 0.17 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - 0.05(-0.809) \begin{bmatrix} 2 \\ 3 \end{bmatrix} . [0.14 \quad 0.15] = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - \begin{bmatrix} -0.011 & -0.012 \\ -0.017 & -0.018 \end{bmatrix} = \begin{bmatrix} .12 & .13 \\ .23 & .10 \end{bmatrix}$$



- Note: All of the images supplied comes from this link: https://hmkcode.com/ai/backpropagation-step-by-step/

Work for Deliverable 1: Second backpropagation of these same weights variables

Second backprop:

Delta term:

$$\Delta = (0.26 - 1) = -0.74$$

Assuming learning rate $a = 0.05$

$$^*w_6 = w_6 - a \cdot (h_2 \cdot \Delta) = 0.17 - 0.05(0.56 \cdot (-0.74)) = 0.19072$$
$$^*w_5 = w_5 - a \cdot (h_1 \cdot \Delta) = 0.17 - 0.05(0.92 \cdot (-0.74)) = 0.20404$$
$$^*w_4 = w_4 - a \cdot (i_2 \cdot \Delta w_6) = 0.10 - 0.05(3 \cdot (-0.74) \cdot 0.17) = 0.11887$$
$$^*w_3 = w_3 - a \cdot (i_1 \cdot \Delta w_6) = 0.13 - 0.05(2 \cdot (-0.74) \cdot 0.17) = 0.14258$$
$$^*w_2 = w_2 - a \cdot (i_2 \cdot \Delta w_5) = 0.23 - 0.05(3 \cdot (-0.74) \cdot 0.17) = 0.24887$$
$$^*w_1 = w_1 - a \cdot (i_1 \cdot \Delta w_5) = 0.12 - 0.05(2 \cdot (-0.74) \cdot 0.17) = 0.13258$$

Prediction:

$$\begin{bmatrix} 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 0.13258 & 0.14258 \\ 0.24887 & 0.11887 \end{bmatrix} = \begin{bmatrix} 1.01177 & 0.64177 \end{bmatrix}$$

$$\begin{bmatrix} 1.01177 & 0.64177 \end{bmatrix} \cdot \begin{bmatrix} 0.20404 \\ 0.19072 \end{bmatrix} = [0.32883993]$$

New error:

$$^*Error = \frac{1}{2}(0.32883993 - 1)^2 \approx 0.225$$

We see that the prediction error is reduced.

For the next section, we learned more about multilayer perceptron, another supervised learning model of a fully connected, feed-forward artificial neural network. This type of neural network is just a rudimentary solution of many solutions available to solve the problem of image recognition. To train our model, we will use the MNIST dataset, which is a dataset of images on hand-written numbers which is crucial to provide many training and testing examples (with extra examples generated through data augmentation) such that the model generalizes to any hand-written digit input we feed it in the future. Below, we can see the analysis of such a neural network and make several changes to see what happens to the performance of our model.

Questions for the Deliverable 2:

1. What data augmentation is used in training? Please delete the data augmentation and rerun the code to compare.

Answer: Data augmentation is used to artificially generate more training examples by manipulating the existing data of our training set. In this particular exercise, the data augmentation strategy used was applying `transforms.Compose` where we randomly rotated and cropped the image, we converted an image from a PIL into a Pytorch tensor and normalized the images. The use of data augmentation helps mitigate the overfitting of our model, therefore leading to the generalization of the model. As seen in these two screen shoots, our model seems to train on our data faster and more accurately in the first few epochs when we remove the data augmentation training examples since there are fewer data to train on. Still, when we observe our validation accuracy, there is a different situation occurring; once we remove the process of data augmentation, there is a slight dip in accuracy which indicates that our model will be slightly more prone to error without that extra data to train on. This is an indication of overfitting of our model when we don't add these extra training examples.

```
Epoch: 01 | Epoch Time: 0m 29s
        Train Loss: 0.405 | Train Acc: 87.54%
         Val. Loss: 0.146 |  Val. Acc: 95.64%
Epoch: 02 | Epoch Time: 0m 29s
        Train Loss: 0.173 | Train Acc: 94.74%
         Val. Loss: 0.114 |  Val. Acc: 96.66%
Epoch: 03 | Epoch Time: 0m 28s
        Train Loss: 0.141 | Train Acc: 95.58%
         Val. Loss: 0.112 |  Val. Acc: 96.35%
Epoch: 04 | Epoch Time: 0m 29s
        Train Loss: 0.120 | Train Acc: 96.32%
         Val. Loss: 0.087 |  Val. Acc: 97.29%
Epoch: 05 | Epoch Time: 0m 30s
        Train Loss: 0.109 | Train Acc: 96.54%
         Val. Loss: 0.079 |  Val. Acc: 97.47%
Epoch: 06 | Epoch Time: 0m 29s
        Train Loss: 0.096 | Train Acc: 97.03%
         Val. Loss: 0.074 |  Val. Acc: 97.68%
Epoch: 07 | Epoch Time: 0m 29s
        Train Loss: 0.093 | Train Acc: 97.13%
         Val. Loss: 0.077 |  Val. Acc: 97.73%
Epoch: 08 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.36%
         Val. Loss: 0.066 |  Val. Acc: 97.96%
Epoch: 09 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.43%
         Val. Loss: 0.071 |  Val. Acc: 97.88%
Epoch: 10 | Epoch Time: 0m 29s
        Train Loss: 0.080 | Train Acc: 97.46%
         Val. Loss: 0.077 |  Val. Acc: 97.71%
```

Image 2. Train/ Validation loss and accuracy of our original model

```
Epoch: 01 | Epoch Time: 0m 18s
        Train Loss: 0.246 | Train Acc: 92.61%
         Val. Loss: 0.149 |  Val. Acc: 95.21%
Epoch: 02 | Epoch Time: 0m 17s
        Train Loss: 0.100 | Train Acc: 96.88%
         Val. Loss: 0.121 |  Val. Acc: 96.40%
Epoch: 03 | Epoch Time: 0m 17s
        Train Loss: 0.067 | Train Acc: 97.87%
         Val. Loss: 0.100 |  Val. Acc: 97.04%
Epoch: 04 | Epoch Time: 0m 17s
        Train Loss: 0.053 | Train Acc: 98.27%
         Val. Loss: 0.083 |  Val. Acc: 97.58%
Epoch: 05 | Epoch Time: 0m 19s
        Train Loss: 0.042 | Train Acc: 98.66%
         Val. Loss: 0.094 |  Val. Acc: 97.41%
Epoch: 06 | Epoch Time: 0m 17s
        Train Loss: 0.036 | Train Acc: 98.74%
         Val. Loss: 0.091 |  Val. Acc: 97.59%
Epoch: 07 | Epoch Time: 0m 17s
        Train Loss: 0.027 | Train Acc: 99.14%
         Val. Loss: 0.094 |  Val. Acc: 97.61%
Epoch: 08 | Epoch Time: 0m 17s
        Train Loss: 0.025 | Train Acc: 99.14%
         Val. Loss: 0.101 |  Val. Acc: 97.55%
Epoch: 09 | Epoch Time: 0m 17s
        Train Loss: 0.022 | Train Acc: 99.25%
         Val. Loss: 0.092 |  Val. Acc: 97.77%
Epoch: 10 | Epoch Time: 0m 18s
        Train Loss: 0.022 | Train Acc: 99.26%
         Val. Loss: 0.116 |  Val. Acc: 97.55%
```

Image 3. Train/ Validation loss and accuracy of our model without data augmentation

2. What is the batch size in the code? Please change the batch size to 16 and 1024 and explain the variation in results.

Answer: The current batch size in the code is 64. From the training/validation loss and accuracy analysis of the three different batch sizes, we can see that for a batch size of 16, our model takes longer to train based on the epoch time of around 36 seconds but does not show any significant change in training/validation accuracy and loss, only showing a slight improvement in the earlier epochs. When it comes to a batch size of 1024, we see a significant increase in model training speed with an epoch time of about 26 seconds. Still, we do see a slightly worse performance in training/validation loss and accuracy with our 10 epochs. It's very notable in the first few epochs. In theory, we would see a benefit in accuracy with a larger number of epochs. This indicates that in terms of batch size, we want a reasonable number because too small of a batch size, and our model will take longer to train even in higher epochs, training our model won't have much benefit; while too large of a batch size, our model will lose some of the initial accuracies of training/validation with fewer epochs, but with more epochs, it will minimize our loss over the long run.

```
Epoch: 01 | Epoch Time: 0m 29s
        Train Loss: 0.405 | Train Acc: 87.54%
         Val. Loss: 0.146 |  Val. Acc: 95.64%
Epoch: 02 | Epoch Time: 0m 29s
        Train Loss: 0.173 | Train Acc: 94.74%
         Val. Loss: 0.114 |  Val. Acc: 96.66%
Epoch: 03 | Epoch Time: 0m 28s
        Train Loss: 0.141 | Train Acc: 95.58%
         Val. Loss: 0.112 |  Val. Acc: 96.35%
Epoch: 04 | Epoch Time: 0m 29s
        Train Loss: 0.120 | Train Acc: 96.32%
         Val. Loss: 0.087 |  Val. Acc: 97.29%
Epoch: 05 | Epoch Time: 0m 30s
        Train Loss: 0.109 | Train Acc: 96.54%
         Val. Loss: 0.079 |  Val. Acc: 97.47%
Epoch: 06 | Epoch Time: 0m 29s
        Train Loss: 0.096 | Train Acc: 97.03%
         Val. Loss: 0.074 |  Val. Acc: 97.68%
Epoch: 07 | Epoch Time: 0m 29s
        Train Loss: 0.093 | Train Acc: 97.13%
         Val. Loss: 0.077 |  Val. Acc: 97.73%
Epoch: 08 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.36%
         Val. Loss: 0.066 |  Val. Acc: 97.96%
Epoch: 09 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.43%
         Val. Loss: 0.071 |  Val. Acc: 97.88%
Epoch: 10 | Epoch Time: 0m 29s
        Train Loss: 0.080 | Train Acc: 97.46%
         Val. Loss: 0.077 |  Val. Acc: 97.71%
```

Image 4 - Train/ Validation loss and accuracy of our model with batch_size = 64

```
Epoch: 01 | Epoch Time: 0m 36s
        Train Loss: 0.348 | Train Acc: 89.05%
         Val. Loss: 0.148 |  Val. Acc: 95.58%
Epoch: 02 | Epoch Time: 0m 35s
        Train Loss: 0.172 | Train Acc: 94.64%
         Val. Loss: 0.115 |  Val. Acc: 96.52%
Epoch: 03 | Epoch Time: 0m 36s
        Train Loss: 0.146 | Train Acc: 95.51%
         Val. Loss: 0.119 |  Val. Acc: 96.32%
Epoch: 04 | Epoch Time: 0m 35s
        Train Loss: 0.125 | Train Acc: 96.13%
         Val. Loss: 0.096 |  Val. Acc: 97.40%
Epoch: 05 | Epoch Time: 0m 35s
        Train Loss: 0.119 | Train Acc: 96.30%
         Val. Loss: 0.101 |  Val. Acc: 96.78%
Epoch: 06 | Epoch Time: 0m 36s
        Train Loss: 0.112 | Train Acc: 96.71%
         Val. Loss: 0.092 |  Val. Acc: 97.33%
Epoch: 07 | Epoch Time: 0m 36s
        Train Loss: 0.110 | Train Acc: 96.79%
         Val. Loss: 0.097 |  Val. Acc: 97.23%
Epoch: 08 | Epoch Time: 0m 36s
        Train Loss: 0.102 | Train Acc: 96.94%
         Val. Loss: 0.077 |  Val. Acc: 97.75%
Epoch: 09 | Epoch Time: 0m 36s
        Train Loss: 0.101 | Train Acc: 96.98%
         Val. Loss: 0.079 |  Val. Acc: 97.63%
Epoch: 10 | Epoch Time: 0m 35s
        Train Loss: 0.098 | Train Acc: 97.11%
         Val. Loss: 0.075 |  Val. Acc: 97.52%
```

Image 5 - Train/ Validation loss and accuracy of our model with batch_size = 16

```
Epoch: 01 | Epoch Time: 0m 26s
        Train Loss: 1.043 | Train Acc: 68.64%
         Val. Loss: 0.412 |  Val. Acc: 87.68%
Epoch: 02 | Epoch Time: 0m 26s
        Train Loss: 0.464 | Train Acc: 86.29%
         Val. Loss: 0.246 |  Val. Acc: 93.47%
Epoch: 03 | Epoch Time: 0m 26s
        Train Loss: 0.289 | Train Acc: 91.65%
         Val. Loss: 0.177 |  Val. Acc: 94.95%
Epoch: 04 | Epoch Time: 0m 26s
        Train Loss: 0.221 | Train Acc: 93.51%
         Val. Loss: 0.141 |  Val. Acc: 95.95%
Epoch: 05 | Epoch Time: 0m 26s
        Train Loss: 0.180 | Train Acc: 94.65%
         Val. Loss: 0.124 |  Val. Acc: 96.45%
Epoch: 06 | Epoch Time: 0m 26s
        Train Loss: 0.160 | Train Acc: 95.24%
         Val. Loss: 0.115 |  Val. Acc: 96.70%
Epoch: 07 | Epoch Time: 0m 26s
        Train Loss: 0.144 | Train Acc: 95.71%
         Val. Loss: 0.106 |  Val. Acc: 96.82%
Epoch: 08 | Epoch Time: 0m 25s
        Train Loss: 0.130 | Train Acc: 96.00%
         Val. Loss: 0.093 |  Val. Acc: 97.10%
Epoch: 09 | Epoch Time: 0m 26s
        Train Loss: 0.120 | Train Acc: 96.39%
         Val. Loss: 0.089 |  Val. Acc: 97.31%
Epoch: 10 | Epoch Time: 0m 26s
        Train Loss: 0.114 | Train Acc: 96.58%
         Val. Loss: 0.088 |  Val. Acc: 97.48%
```

Image 6- Train/ Validation loss and accuracy of our model with batch_size = 1024

3. What activation function is used in the hidden layer? Please replace it with the linear activation function and see how the training output differs. Show your results before and after changing the activation function in your written report.

Answer: The activation function utilized in the hidden layers is called rectified linear unit function. I then substituted it with a sigmoid activation function. These are the results before and after.

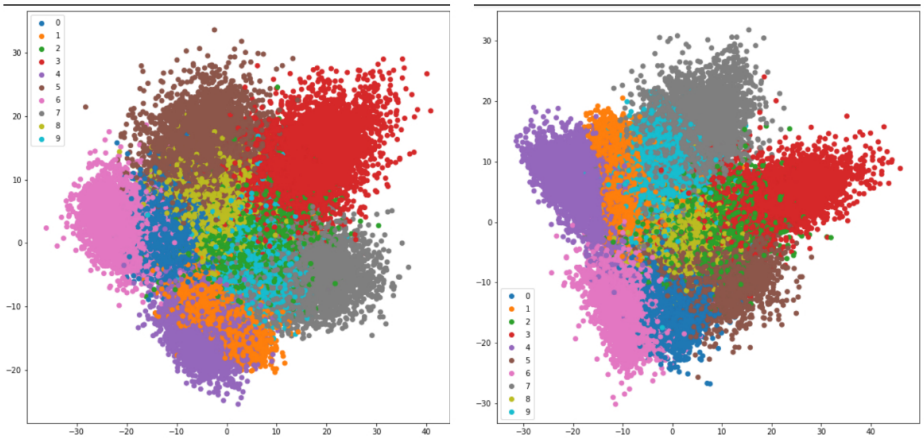
Before using the sigmoid activation function

```
Epoch: 01 | Epoch Time: 0m 29s
        Train Loss: 0.405 | Train Acc: 87.54%
         Val. Loss: 0.146 |  Val. Acc: 95.64%
Epoch: 02 | Epoch Time: 0m 29s
        Train Loss: 0.173 | Train Acc: 94.74%
         Val. Loss: 0.114 |  Val. Acc: 96.66%
Epoch: 03 | Epoch Time: 0m 28s
        Train Loss: 0.141 | Train Acc: 95.58%
         Val. Loss: 0.112 |  Val. Acc: 96.35%
Epoch: 04 | Epoch Time: 0m 29s
        Train Loss: 0.120 | Train Acc: 96.32%
         Val. Loss: 0.087 |  Val. Acc: 97.29%
Epoch: 05 | Epoch Time: 0m 30s
        Train Loss: 0.109 | Train Acc: 96.54%
         Val. Loss: 0.079 |  Val. Acc: 97.47%
Epoch: 06 | Epoch Time: 0m 29s
        Train Loss: 0.096 | Train Acc: 97.03%
         Val. Loss: 0.074 |  Val. Acc: 97.68%
Epoch: 07 | Epoch Time: 0m 29s
        Train Loss: 0.093 | Train Acc: 97.13%
         Val. Loss: 0.077 |  Val. Acc: 97.73%
Epoch: 08 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.36%
         Val. Loss: 0.066 |  Val. Acc: 97.96%
Epoch: 09 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.43%
         Val. Loss: 0.071 |  Val. Acc: 97.88%
Epoch: 10 | Epoch Time: 0m 29s
        Train Loss: 0.080 | Train Acc: 97.46%
         Val. Loss: 0.077 |  Val. Acc: 97.71%
```

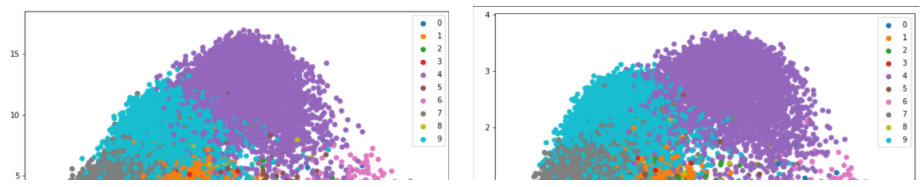
After using the sigmoid activation function

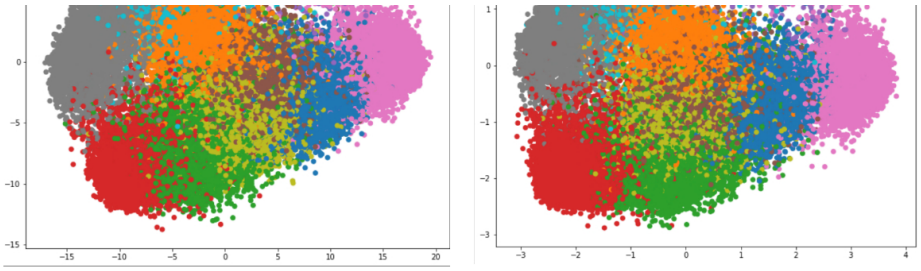
```
Epoch: 01 | Epoch Time: 0m 28s
        Train Loss: 0.776 | Train Acc: 77.82%
         Val. Loss: 0.238 |  Val. Acc: 93.93%
Epoch: 02 | Epoch Time: 0m 28s
        Train Loss: 0.248 | Train Acc: 92.87%
         Val. Loss: 0.134 |  Val. Acc: 96.01%
Epoch: 03 | Epoch Time: 0m 29s
        Train Loss: 0.165 | Train Acc: 95.13%
         Val. Loss: 0.115 |  Val. Acc: 96.39%
Epoch: 04 | Epoch Time: 0m 30s
        Train Loss: 0.133 | Train Acc: 95.99%
         Val. Loss: 0.102 |  Val. Acc: 96.99%
Epoch: 05 | Epoch Time: 0m 29s
        Train Loss: 0.120 | Train Acc: 96.20%
         Val. Loss: 0.083 |  Val. Acc: 97.37%
Epoch: 06 | Epoch Time: 0m 29s
        Train Loss: 0.108 | Train Acc: 96.62%
         Val. Loss: 0.085 |  Val. Acc: 97.26%
Epoch: 07 | Epoch Time: 0m 29s
        Train Loss: 0.101 | Train Acc: 96.86%
         Val. Loss: 0.086 |  Val. Acc: 97.19%
Epoch: 08 | Epoch Time: 0m 29s
        Train Loss: 0.095 | Train Acc: 96.99%
         Val. Loss: 0.069 |  Val. Acc: 97.93%
Epoch: 09 | Epoch Time: 0m 30s
        Train Loss: 0.091 | Train Acc: 97.15%
         Val. Loss: 0.069 |  Val. Acc: 97.74%
Epoch: 10 | Epoch Time: 0m 29s
        Train Loss: 0.087 | Train Acc: 97.30%
         Val. Loss: 0.066 |  Val. Acc: 97.95%
```

Before using the sigmoid activation function



After using the sigmoid activation function





4. What is the optimization algorithm in the code? Explain the role of the optimization algorithm in the training process.

Answer: The algorithm used in the code is Adam, an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based on the training data. In the training process, we want to learn the weights of our neural networks such that we can, in our case, minimize the error of our prediction. This led to the use of a stochastic gradient descent update, which presents itself as a single learning rate that assists in updating those previously mentioned weights with each epoch. What Adam does is that it takes advantage of the extensions of this stochastic gradient descent, giving an adaptive step size from the AdaGrad algorithm but also maintaining per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight from the Root Mean Square Propagation. This significantly improves the way we learn our weights on the training data.

5. Add dropout in the training and explain how the dropout layer helps in training.

Answer: The dropout layer helps minimize a common issue that occurs within neural networks, overfitting. Dropout is a regularization method that approximates our training by randomly ignoring or "dropping out" a certain number of layer outputs. This makes the layer look like and is treated like a layer with a different number of nodes and connectivity within the hidden layer architecture. This tends to simulate the sparse activation to a given layer, similar to what convolution neural networks simulated but with more or fewer connections between nodes. This can help in training by making the training process noisy by forcing nodes to probabilistically take more or less responsibility for the inputs, generating scenarios where the network layers can correct prior layers mistakes, therefore turning our model more robust in the process.

```
Epoch: 01 | Epoch Time: 0m 29s
        Train Loss: 0.405 | Train Acc: 87.54%
         Val. Loss: 0.146 |  Val. Acc: 95.64%
Epoch: 02 | Epoch Time: 0m 29s
        Train Loss: 0.173 | Train Acc: 94.74%
         Val. Loss: 0.114 |  Val. Acc: 96.66%
Epoch: 03 | Epoch Time: 0m 28s
        Train Loss: 0.141 | Train Acc: 95.58%
         Val. Loss: 0.112 |  Val. Acc: 96.35%
Epoch: 04 | Epoch Time: 0m 29s
        Train Loss: 0.120 | Train Acc: 96.32%
         Val. Loss: 0.087 |  Val. Acc: 97.29%
Epoch: 05 | Epoch Time: 0m 30s
        Train Loss: 0.109 | Train Acc: 96.54%
         Val. Loss: 0.079 |  Val. Acc: 97.47%
Epoch: 06 | Epoch Time: 0m 29s
        Train Loss: 0.096 | Train Acc: 97.03%
         Val. Loss: 0.074 |  Val. Acc: 97.68%
Epoch: 07 | Epoch Time: 0m 29s
        Train Loss: 0.093 | Train Acc: 97.13%
         Val. Loss: 0.077 |  Val. Acc: 97.73%
Epoch: 08 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.36%
         Val. Loss: 0.066 |  Val. Acc: 97.96%
Epoch: 09 | Epoch Time: 0m 29s
        Train Loss: 0.085 | Train Acc: 97.43%
         Val. Loss: 0.071 |  Val. Acc: 97.88%
Epoch: 10 | Epoch Time: 0m 29s
        Train Loss: 0.080 | Train Acc: 97.46%
         Val. Loss: 0.077 |  Val. Acc: 97.71%
```

Image 7 - Train/ Validation loss and accuracy of our model (control)

```
Epoch: 01 | Epoch Time: 0m 31s
        Train Loss: 0.627 | Train Acc: 80.46%
         Val. Loss: 0.214 |  Val. Acc: 93.92%
Epoch: 02 | Epoch Time: 0m 30s
        Train Loss: 0.285 | Train Acc: 91.90%
         Val. Loss: 0.162 |  Val. Acc: 95.09%
Epoch: 03 | Epoch Time: 0m 30s
        Train Loss: 0.225 | Train Acc: 93.54%
         Val. Loss: 0.138 |  Val. Acc: 95.96%
Epoch: 04 | Epoch Time: 0m 30s
        Train Loss: 0.199 | Train Acc: 94.37%
         Val. Loss: 0.117 |  Val. Acc: 96.52%
Epoch: 05 | Epoch Time: 0m 30s
        Train Loss: 0.180 | Train Acc: 94.84%
         Val. Loss: 0.114 |  Val. Acc: 96.56%
Epoch: 06 | Epoch Time: 0m 30s
        Train Loss: 0.172 | Train Acc: 95.16%
         Val. Loss: 0.104 |  Val. Acc: 97.01%
Epoch: 07 | Epoch Time: 0m 30s
        Train Loss: 0.161 | Train Acc: 95.40%
         Val. Loss: 0.107 |  Val. Acc: 96.85%
Epoch: 08 | Epoch Time: 0m 30s
        Train Loss: 0.154 | Train Acc: 95.56%
         Val. Loss: 0.101 |  Val. Acc: 97.11%
Epoch: 09 | Epoch Time: 0m 31s
        Train Loss: 0.150 | Train Acc: 95.79%
         Val. Loss: 0.097 |  Val. Acc: 97.16%
Epoch: 10 | Epoch Time: 0m 30s
        Train Loss: 0.140 | Train Acc: 96.02%
         Val. Loss: 0.093 |  Val. Acc: 97.37%
```

Image 8- Train/ Validation loss and accuracy of our model (dropout layer (25%))

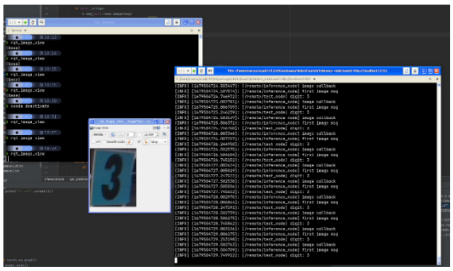
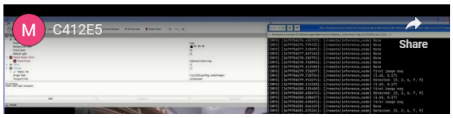


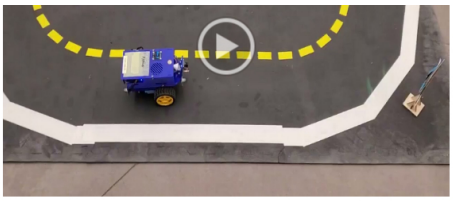
Image 9 - Screenshot of the CNN recognizing an image of the number 3.

As the final task of this lab, we tried to apply what we learned from the previous two exercises and applied it to our robot. So the first thing we did was design our neural network model; after testing the use of some algorithms, we decided to go with a version of a convolutional neural network called resnet-18, a CNN developed by Microsoft, which introduces residual connections to a conventional CNN, allowing us to train the network with an unseen number of layers. We were both familiar with convolutional neural networks and their efficiency in classifying images. My partner Marcus started by collecting data on the numbers in duckie town by creating a script that would collect multiple images of all the numbers in our duckietown and separate them into their own folders. We must have collected around 100 images for each number. We decided not to go with the MNIST dataset because the numbers placed in the duckietown were specifically of a particular computer font, meaning our input, when given to our CNN, had a high chance of not recognizing the numbers based on the training with the MNIST dataset.

Then in our model, we applied some data augmentation to obtain more examples for both training and testing. Once our model had been trained with our generated dataset, we tested it with a cropped image that would be generated once the robot detected the number. The model identified the number with great success. This lab's more challenging aspect was getting our robot to traverse the duckietown at any given position. This is due to the fact that in the previous lab, we run into issues in getting a turn to occur without any detection happening. So after many attempts, we came up with a very unconventional approach where each apriltag would be associated with a particular action (Left, Right and Straight). What I mean by this is that after we stop on the stop line will ignore any stopping detection until we successfully turn and start lane following once more. This was quite a brute-force approach since we had to "patch" the lane following detection so that it couldn't detect anything from the right side of the frame up to a certain extent. Also, to close all the nodes, we had a few issues but managed to find a solution but not an elegant one. The final result is shown below.



Video 1 - The execution of our detection and path following through RViz



Video 2 - Human eye-view of the detection and path following

Questions:

1. **How well did your implemented strategy work?**

   Overall, the implemented strategy worked very well, recognizing all the numbers in one pass through the town.

2. **Was it reliable?**

   The full implementation is mostly reliable at best, but it can sometimes get stuck on apriltags when turning right. Also, it may really rarely miss a detection.

3. **In what situations did it perform poorly?**

   We did not do a thorough testing process to check for any particular scenario that might not work, but I would say if there is an extreme change of lighting in the room, the detection may not occur.

Issues that occurred: The biggest challenge that we faced was with the automation of our motion through duckietown so that we could do the detections. This is because we weren't sure how to execute the turning. Some students in the prior attempted to execute the turning based on time, which wasn't very successful. Then one of the TAs suggested a combination of dead reckoning and apriltags which was the approach we kind of leaned on. Another issue that occurred is that how we implemented everything, we were creating subscribers that were trying to subscribe to a topic that its publisher hadn't initialized yet. The way we used to circumvent the delay was to make sure the subscriber started strictly after the publisher was initialized. So we held the robot for a few minutes, which solved the communication issue at the start but didn't solve the issue at shutdown. We had to set a terminal command in our code to execute the shutdown once our inference node terminated.

To improve: Never underestimate a task. The task can appear easy, and it may be easy, but if you can't find that "easy" solution, that easy task can ultimately become a challenge.

References :

- OpenCV:
  1. https://docs.opencv.org/3.3.0/dc/dbb/tutorial_py_calibration.html
  2. https://www.tutorialspoint.com/detection-of-a-specific-color-blue-here-using-opencv-with-python
  3. https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html (Others from the OpenCV documentation as well).
- A.I related websites:
  1. https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/
  2. https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/
  3. https://zablo.net/blog/post/pytorch-resnet-mnist-jupyter-notebook-2021/index.html
  4. https://www.kaggle.com/datasets/pytorch/resnet18
  5. https://huggingface.co/microsoft/resnet-18
  6. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
- Other links:
  1. https://answers.ros.org/question/294069/shutdown-a-node-from-another-node-in-ros-cpp/

Received help from: Justin (TA), Rozwan (TA), Marcus (Partner & student)