Alexander Hay

alexanderhay2020@u.northwestern.edu

ME 469 - Machine Learning and AI in Robotics

Assignment #2 - Machine Learning

Algorithm: Neural Networks
Learning Aim: Motion Model
Input Vector: [t, x, y, Θ, v, ω]
Output Vector: [Δx, Δy, $\Delta\Theta$]

11/19/2019

Note: please refer to the readme.txt for specific attribute and function definitions.

Part A

The *how* and *what* of the assignment were determined in the following way. The *how* was already decided due to personal interest in learning neural networks. Determining *what* was more difficult and required coordination with Shangzhou Ye, my partner for the assignment, and ultimately decided to have the algorithm learn the change in linear and angular displacement via the motion model.

Dataset 1 (ds1) was chosen because it contained the most data (longer collection time). The data was sorted by timestamp and any gaps in the data caused by the sort was interpolated. The data was then split into three distinct sets; a training set, a validation set, and a test set, in a 45/45/10 ratio. The training and test sets were manipulated the most, while the validation set was set aside to validate once the network had been properly developed.

The motion model is borrowed from hw0 with some minor tweaks, such that rather than calculating the robot's new position it just calculates the difference:

$$\Delta\theta = \omega * t \qquad \text{Eq. 1}$$

$$\Delta x = v * t * cos(\theta) \qquad \text{Eq. 2}$$

$$\Delta y = v * t * sin(\theta) \qquad \text{Eq. 3}$$

Where $\Delta\theta$ is the heading [rad], ω is angular velocity [rad/s], t is time [s], v is linear velocity [m/s], Δx and Δy are their respective displacements. The input for the algorithm should then include the odometry data. The groundtruth data was included as a means for verification.

Neural nets are built on units called neurons, and for this exercise a special neuron called a perceptron is used. Perceptrons are special in that they can represent fundamental logic functions: AND, OR, NAND, NOR. Though a perceptron can't represent XAND or XOR, layered perceptrons can, thus all logic functions can potentially be built using a layered network structure.

Perceptrons take a vector of inputs $[x_1, x_2, \ldots, x_n]$ and calculate a linear combination of the inputs, then outputs a 1 or -1 based on a given threshold. To do that a weighted sum is calculated using the input vector and a set of randomized weights:

$$\Sigma(x_i w_i) = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n \qquad \text{Eq. 4}$$

Weights are [nxm] matrices, where n is the dimension of the input and m is the dimension of the output. The sum is then passed through an activation function. In this case a sigmoid function is used to normalize the result:

$$\sigma = \frac{1}{1+e^x} \qquad \text{Eq. 5}$$

Because the weights were randomly generated it's almost assured that the output is inaccurate. Adjustments need to be made to the weights. Adjustments are calculated by finding the error:

$$error = output_{expected} - \sigma \quad \text{Eq. 6}$$

Then finally calculated by multiplying the error by the sigmoid derivative, which makes an adjustment that is proportional to the size of the error:

$$\sigma' = \sigma * (1 - \sigma) \quad \text{Eq. 7}$$

$$adjustments = error * \sigma' \quad \text{Eq. 8}$$

$$w_i = w_i + \hat{x}^T \bullet adjustments \quad \text{Eq. 9}$$

These new weights wouldn't have changed much, but over many iterations they converge to their proper values of minimizing error. This method of adjusting the weights is called backpropagation.

To test the algorithm a small, simple sample set was used to provide easy-to-interpret results. The table below shows this set:

	Variable 1	Variable 2	Variable 3	Output
Input 1	0	0	1	0
Input 2	1	1	1	1
Input 3	1	0	1	1
Input 4	0	1	1	1

The system is that if variable 1 or variable 2 contained a 1, then the output will be a 1, ignoring whatever variable 3 is.

perceptron.py demonstrates the algorithm and predicted output. Given the input array and initial weights adjusted 20,000* times, the predicted results are as follows:

```
Input:          Starting Weights:          Weights after training:    Output:
[[0 0 1]        [[  4.17022005e-01]        [[ 9.38838295]             [[ 0.0114106 ]
 [1 1 1]         [  7.20324493e-01]         [ 9.38838305]              [ 0.99999939]
 [1 0 1]         [  1.14374817e-04]]        [-4.46176172]]             [ 0.99280105]
 [0 1 1]]                                                              [ 0.99280105]]
```

*this is a tunable parameter

Given an infinite amount of time and iterations the output would eventually converge to 0 or 1. Instead, the algorithm iterates. More iterations means a more precise convergence but at the cost of computational speed. The non-binary property of the sigmoid function produces results *very close* to 0 or 1. Also of note are the adjusted weights. The algorithm clearly recognized that the first and second column were important while the third column was to be ignored. What's also interesting is that the algorithm placed nearly *identical* weights for the first and second columns, effectively creating an OR logic gate.

Part B

A test was developed with the same scale in mind as the perceptron example, one that can be done by hand. The first 10 entries of the training data (input.tsv) were taken and passed through the motion model, generating a [10x3] matrix of [Δx, Δy, $\Delta\theta$] entries. Using this method we should expect the algorithm to ignore the x and y position data since it is not used in calculating the displacement.

run_single.py executes the neural network and displays the initial and final weights. Executing run.py showed that the weights didn't change. This suggests that the data is not linearly related or separable (which is true). For the sake of confirmation the network was executed with the entire training input set and produced the same, unchanged weights.

Starting Weights:
[[4.17022005e-01 7.20324493e-01 1.14374817e-04]
[3.02332573e-01 1.46755891e-01 9.23385948e-02]
[1.86260211e-01 3.45560727e-01 3.96767474e-01]
[5.38816734e-01 4.19194514e-01 6.85219500e-01]
[2.04452250e-01 8.78117436e-01 2.73875932e-02]
[6.70467510e-01 4.17304802e-01 5.58689828e-01]]

Weights after training:
[[4.17022005e-01 7.20324493e-01 1.14374817e-04]
[3.02332573e-01 1.46755891e-01 9.23385948e-02]
[1.86260211e-01 3.45560727e-01 3.96767474e-01]
[5.38816734e-01 4.19194514e-01 6.85219500e-01]
[2.04452250e-01 8.78117436e-01 2.73875932e-02]
[6.70467510e-01 4.17304802e-01 5.58689828e-01]]

Furthering confirmation, a [10x6] array was created of random integers, similar to the 10 entries of training data. Executing run.py with that random integer set also produced unchanging weights, suggesting that the data itself is not the cause.

The strength of perceptrons lies in their ability to be constructed together to create a computational network. Executing run.py shows the intermediate weights, the synapses, between perceptron (neuron). Each weight linearly correlates what the algorithm thinks the relationship is between an input node to an output node. This suggests that more layers would create more connections, creating more computationally complex connections. This comes at great computing cost, so parameters such as the number of layers, number of iterations to converge, and activation function need to be considered to ensure a reasonable runtime. In this exercise, the algorithm has 2 layers and iterates 20,000 times.

Looking at the results run.py uses the same array of random integers as before. Here are the results

Training Output:
```
[[  0.          -0.         5.        ]
 [  4.78580264  34.67125745  14.      ]
 [-18.30202138 -21.19046987   4.      ]
 [ 31.66389468  27.59343715   7.      ]
 [  0.          0.         0.        ]
 [ -3.83410482   5.85658947  21.      ]
 [  0.          -0.        12.       ]
 [  1.91432105  13.86850298  14.      ]
 [ 46.71650819  30.87989415  32.      ]
 [ -0.          -0.        10.       ]]
```

Layer 1 Weights:
```
[[-0.40213978  0.16877824  0.13182405  0.22787664  0.91307131 -0.47804204]
 [-0.53796916  0.06689698  0.89987628 -0.01388081  0.08120102  0.53097021]
 [-0.90930854 -0.72006776  0.58480717 -0.94039728  0.76625095  0.08157638]
 [-0.10403964  0.78427174 -0.24483132  0.07684939  0.30459776 -0.27747796]
 [ 0.14201712  0.27567297 -0.74737023  0.38040918  0.2954988  -0.29212182]
 [ 0.52646611 -0.28693655  0.50557671  0.76268366 -0.97666161 -0.00378186]]
```

Layer 2 Weights:
```
[[-0.85241598  0.57390295 -0.87186534]
 [-0.28937928  0.8836739  -0.24039343]
 [ 0.52584015  0.54319    -0.39727901]
 [ 0.54547829 -0.69414035  0.15726786]
 [-0.98198401  0.41808504 -0.05871838]
 [ 0.52918514 -0.06521463 -0.4619979 ]]
```

Output:
```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Layer 1 Weights:
```
[[-1.15662913  1.03703291  6.56835633  0.72820868  1.24721281 -3.30242924]
 [-0.68297684 -0.23871361  1.27933284  0.07582685  0.07182823 -2.39737598]
 [-2.14530402  0.45607781  5.06619345 -0.05015285  1.51724076 -1.91836265]
 [-0.8294075   1.28595832  3.95880621  1.74195555  0.70507952 -2.68959217]
 [-2.03651036  3.33831469  3.43890351  1.74942975  1.52159175 -1.1674609 ]
```

[-1.61358575 2.08319982 2.91258875 0.80069782 0.28454031 -1.62655026]]

Layer 2 Weights:
[[-0.66905998   2.17437721   0.71710193]
 [13.6305082   13.84978578 20.56721486]
 [12.17231422 14.61489307 22.30583226]
 [12.29455663 12.44736352 18.50306279]
 [12.6666522   15.02003651 25.14292233]
 [0.43028128   0.28996622   3.12892022]]

Crude, but it's clear that adding another layer addressed the issue of the weights not changing. Here they change as expected. However the output as it is, being all ones, is unexpected. A few things may address this; adding another layer may provide more insight for the algorithm into any hidden structures within the data. An additional layer can provide another "step" the algorithm can use to determine a function between the input and output data.Another consideration is that the data itself could be improper for the purpose of the algorithm, but that seems unlikely as the model is relatively simple.

References:

Arnx, Arthur. “First Neural Network for Beginners Explained (with Code).” Medium, Towards Data Science, 11 Aug. 2019, towardsdatascience.com/first-neural-network-for-beginners-explained-with-code-4cfd37e06eaf.

“Artificial Neural Networks.” *Machine Learning*, by Thomas M. Mitchell, McGraw-Hill, 1997, pp. 81–112.

Fried, Charles. “Let's Code a Neural Network From Scratch.” Medium, TypeMe, 6 Apr. 2017, medium.com/typeme/lets-code-a-neural-network-from-scratch-part-1-24f0a30d7d62.

Spencer-Harper, Milo, director. Create a Simple Neural Network in Python from Scratch. YouTube, PolyCode, 31 Mar. 2018, www.youtube.com/watch?v=kft1AJ9WVDk.

Spencer-Harper, Milo. “How to Build a Simple Neural Network in 9 Lines of Python Code.” Medium, 8 Apr. 2019, medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1.

Trask, Andrew. “A Neural Network in 11 Lines of Python (Part 1).” *A Neural Network in 11 Lines of Python (Part 1)*, Github.io, 12 June 2015, iamtrask.github.io/2015/07/12/basic-python-network/.