A Universal Robotic Control System using Reinforcement Learning with Limited Feedback

Abstract

Our abstract.

1 Introduction

Introduction here.

2 Neural Network Background

The human brain is composed of billions of neurons, interconnected electrically excitable cells that form the basis of our intelligence. Each neuron has synapses that receive electrical signals from multiple other neurons. If the sum of these inputs is greater than a certain value, the neuron fires, generating a voltage down its axon. The axon, or the output of the neuron cell, is itself connected to a synapse of another neuron. The interconnections of these neurons form a massive network, where a huge number of inputs are processed in parallel to a set of outputs. The basis of artificial neural networks is to simulate the mathematical properties of these neurons in order to perform similar tasks of mass parallel computation.

2.1 Single Artificial Neuron

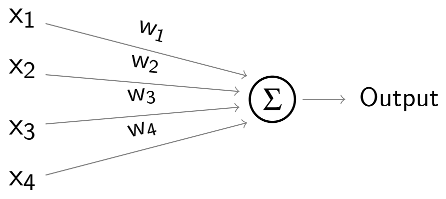

Figure 1: Single Neuron Diagram

An artificial neuron is simply a mathematical model of a biological neuron, and therefore its functionality is very similar. Each artificial neuron has multiple inputs, each one the output of a different neuron. In addition, each neuron has a weight for each input, a bias term, an activation function, and one output. The output of one neuron becomes the input of another, connected neurons. The output of a neuron is expressed mathematically for n inputs x and weights w, a bias term b, and activation function $O(x)$:

$$\text{activation} = \sum_{i=0}^{n} x_i w_i + b$$

$$\text{output} = O(\text{activation})\,.$$

A weight is a positive or negative number that governs the impact of of its respective input on the neuron's single output. The bias term is a number that is added to the summation of weights and inputs. The bias term allows us to make affine transformations on the domain of the activation function.

In biologically inspired neurons, the activation function is a binary step function. If the activation function's input is less than a certain threshold, the output is zero. If it is greater than the threshold, the output is one. However, a binary output can be somewhat limiting for many applications of neural networks. For example, many of Fido's outputs are gradient rather than binary; as an example, Fido can set the brightness of an LED. Because of this, alternate activation functions giving gradient outputs are used. One such function is the sigmoid function, expressed as such:

$$O(a) = \frac{1}{1+e^{-\frac{a}{p}}}, \tag{1}$$

for each output O, activation a, and constant p. The sigmoid activation function can also be graphed as below.

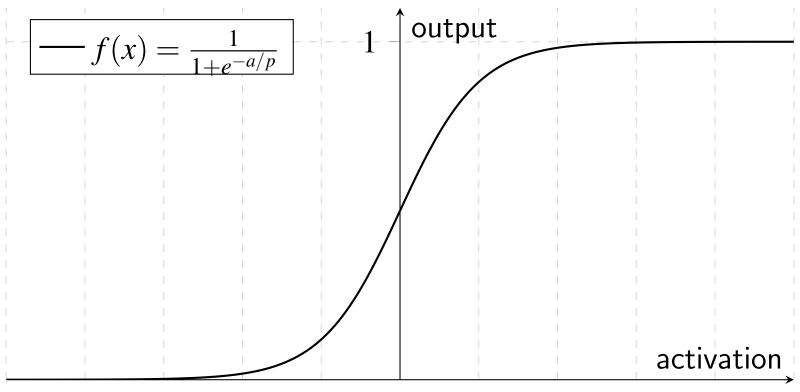

Figure 2: Sigmoid Function Graph

The sigmoid function provides us with a gradient output. However, output is still limited to positive values. An alternative activation function which allows outputs ranging from -1 to +1 is the hyperbolic tangent activation function.

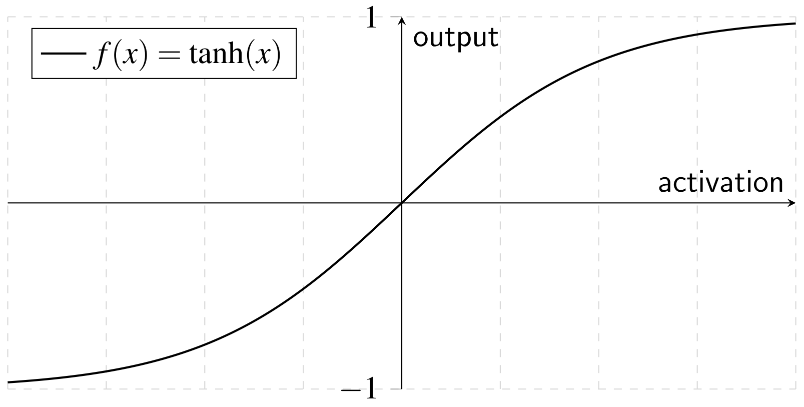

Figure 3: Hyperbolic Tangent Function Graph

2.2 Feed-forward Neural Network

A traditional method of arranging artificial neurons in a neural network is called a feed-forward network. Neurons are connected as previously described: the output of each neuron is connected to one of the inputs of another neuron. These neurons are organized into layers, as described in the following figure:

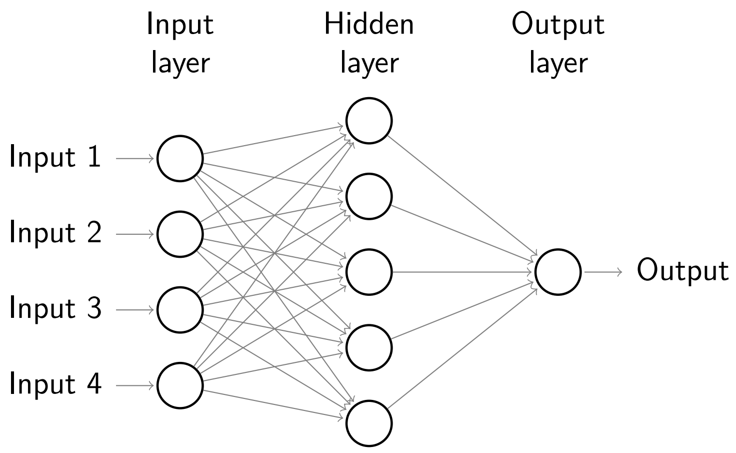

Figure 4: Single Output Feed-forward Network

The output of each neuron in a layer becomes an input of every neuron in the next layer. In this way, the original inputs of the neural network are "fed forward" layer by layer, starting from the first layer (the input layer), passing through any number of hidden layers, ending with the last layer (the output layer). The outputs of the neurons of the output layer are the outputs of the whole network.

A concrete example of a feed-forward network is that of Fido. Sensor inputs such as light and sound are sent into the input layer. The outputs of the output layer's neurons are used to change the speed of Fido's motors, the color of Fido's LEDs, and the sounds of Fido's buzzers.

2.3 Back Propagation

Supervised learning is a type of neural network training that modifies neuron input weights in order to reach a specific output from a specific set of inputs. Initially the neuron weights are randomly generated, but over time the weights are adjusted to converge on the correct output. One such method of adjusting neural weights is called back propagation (Werbos, 1974).

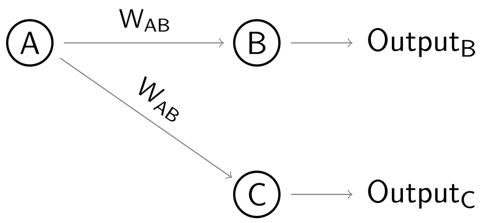

Figure 5: Single Branch in a Back Propagation Neural Network

In the most abstract sense, back propagation neural networks learn through example. Example inputs (known as training sets) are linked to a particular target output, forming a training pair. The first step of training is to pass in an arbitrary training set to a neural network with randomly generated weights. The error of the network for a certain connection δ is defined as the difference between the actual output and the target output for that training set. As an example we will use the single connection described in figure 5. We first determine the neuron B error δ_B as $\text{Target}_B - \text{Output}_B$. Next we modify weight W_{AB} as follows, defining W_{AB}^{+} as the newly trained weight:

$$W_{AB}^{+} = W_{AB} + (\delta_B \text{Output}_A)\,. \tag{2}$$

The same approach could be taken for neuron C. However, this method cannot calculate an improved weight for hidden layer neurons such as neuron A, as there is no predefined target. Therefore we must calculate the error of A indirectly by back propagating from the two output layer neurons, for which the error is already known.

$$\delta_A = W_{AB}\text{Error}_B + W_{AC}\text{Error}_C \tag{3}$$

Once the error for neuron A is obtained, its trained weights can be calculated in the same fashion as done for neuron B. This pattern can be repeated throughout a larger neural network in order to converge upon the target output.

3 Learning Implementation

Intro text.

3.1 Q-Learning

Reinforcement learning seeks to find the optimal action to be undertaken for a given state through trial and error. In the context of Fido, an action could be the playing of a note or driving straight forward, while the state could the amount of light detected by the robot or how near the robot is to another object. Once an action is performed, a reward and a new state are given back to the reinforcement learning algorithm. As actions are performed over time a reinforcement learning algorithm sharpens its ability to receive reward.

Q-Learning (Watkins, 1989) is a popular reinforcement learning algorithm that works by learning an action-value function Q that takes an state-action pair as an input and outputs an expected utility value of performing that action in that state. This utility value is know as the Q-value. The

Q-value is a combination of immediate reward and expected future reward. Every reward iteration, the Q-value of an state-action pair is updated as such:

$$Q(s,a) := Q(s,a)(1-\alpha) + \alpha(R + \gamma \max Q(s_{t+1},a)), \tag{4}$$

Where a is the action carried out, s is the initial state, R is the reward received, and s_{t+1} is the new state. α is the learning rate of the algorithm. The learning rate determines the rate of convergence by diminishing or amplifying the changes made to the Q-value each reward iteration. γ is the devaluation factor, which determines the weight given to future rewards. A devaluation factor approaching $\gamma = 0$ will force the algorithm to only value immediate reward, while a devaluation factor approaching $\gamma = 1$ will make it focused on high long term reward.

The Q-Learning algorithm had to be immediately modified in two ways to make it suitable for Fido. Its scalability had to be improved, and it had to be able to work in continuous state-action spaces.

Q-learning in its simplest form uses a table to model the Q function, storing past state-action pairs and each pair's respective Q-value. However, this strategy lacks scalability. In large state-action spaces, such as those Fido will have to perform in, the amount of data and computation needed to maintain such a table renders such a strategy impractical. Since Fido will be working in large state-action space, it is necessary to use a function approximator to model Q. A feed-forward neural networks were chosen for this task for their ability to model non-linear functions, lightweight computational footprint, high trainablility.

Conventional Q-Learning is discrete. No relation is made between states or actions, and every action for each state must be performed individually repeatedly in a noisy feedback system to determine its Q-value. However, Fido will work in a large, continuous state-action spaces where relations made between (state-action, Q-value) pairs can drastically reduce the number of reward iterations needed for convergence. An example of a task that would benefit from continuity is teaching Fido to adjust the speed of its motors based on the intensity of light that the robot detects. There is an obvious gradient relation between the (state-action, Q-value) pairs in this task and with a limited number of Q-values known, it is possible to correctly model Q.

3.2 Wire-Fitted Q-Learning

To accommodate continuous action-spaces, we coupled a wire-fitted moving least squares interpolator with our feed-forward neural network as described in (Gaskett, Wettergreen, & Zelinsky, 1999).

Feed-forward neural networks can generalize between states in Q-Learning problems with discrete actions as described in (Rummery, 1995). To extend this implementation to a continuous action space, our feed-forward neural network outputs discrete "wires" when given a state. Each wire consists of an action with its respective Q-value for the state given to the neural network. These wires may be interpolated to model Q, allowing us to get the Q-value of any action performed in state given as an input to the network. The interpolator used in Fido is a wire-fitted moving least squares interpolator used in the context of a memory-based learning system in (Baird & Klopf, 1993).

The wire-fitting function calculates Q-value of an action $\hat{a}$ for a state s given a set of n actions a each with a respective Q-value q as such:

$$Q(a,s) = \frac{\sum_{i=0}^{n} \dfrac{q_i}{||\hat{a}-a_i||^2 + c(q_{max}-q_i)+k}}{\sum_{i=0}^{n} \dfrac{1}{||\hat{a}-a_i||^2 + c(q_{max}-q_i)+k}}, \tag{5}$$

where q_{max} is the greatest Q-value among the set of Q-values q, and k is a small constant that avoids division by 0. c is the smoothing factor. The greater the smoothing factor, the smoother the interpolated function.

Figure 6 is an example of interpolation on a set of wires. The graph shows the value of one-dimensional actions plotted against their respective Q-values.

The wire-fitting function has few properties that make it especially suited for Fido.

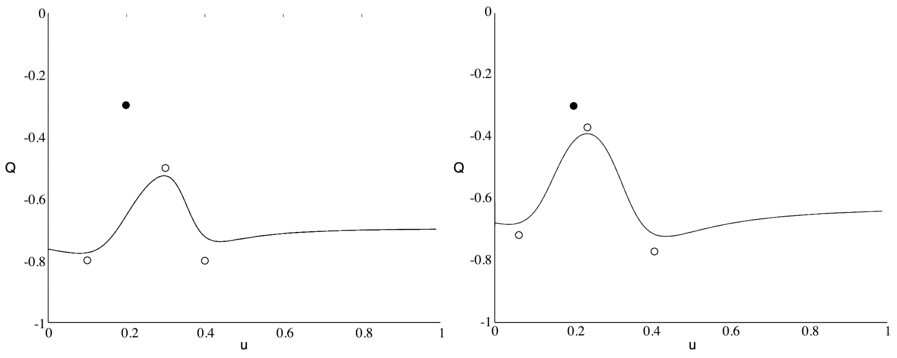

Figure 6: Moving Least Squares Interpolator (adapted from Gaskett, Wettergreen, & Zelinsky, 1999)

Every update to the Q-value requires that Q-max is computed and the action for Q-max is often needed for action selection. As proved in (Baird & Klopf, 1993), the wire with the greatest Q-value is the interpolation point with the greatest Q-value, and so, maximum Q-value outputted by the wire-fitting function is the maximum Q-value out of the set of wires that the function takes as an input. This makes it extremely computationally cheap to compute Q-value, allowing Fido's latency to stay minimal.

The wire-fitting function is derivable, allowing us to update our estimation of the Q. Once the reward and new state for an action-state pair is received and an updated Q-value is calculated using equation **??**, the error in previous Q-learn value can be calculated as such:

$$(\hat{Q} - Q)^2, \tag{6}$$

where $\hat{Q}$ is our new Q-value and Q is our old Q-value. Using equation 6 as our cost function and the partial derivative of the wire-fitting function with respect to each action vector and each q-value, new wires that model Q so that it correctly outputs $\hat{Q}$ may be calculated for our current state using gradient descent. Our neural network may then be trained to output these new wires using back propagation, thus updating our estimation of Q.

3.3 Boltzmann Soft-Max Distribution

4 Simulation

The robot model chosen for simulation was modeled after easily producible robots on the same scale. The software driving the simulation was intended to be portable enough to work on a hardware implementation, and the model facilitates this goal as well. Additionally, the robot model would have to be easily trainable and debuggable when implemented in hardware; use of a Geiger counter as an input would be unfavorable. Lastly the sensors and design chosen had to facilitate the concept of natural learning, modeling after nature to some degree.

4.1 Robot Inputs and Outputs

Multiple inputs were modeled for simulation with outlets for control both by a human operator using sliders and by programmed handlers using a bridge class. A microphone and light sensor were chosen as clear, human modifiable inputs that model after nature and could easily be used for reinforcement training. An infrared light sensor was added as another easily controller variable in a testing setup: a human operator could easily bring closer and farther an IR LED for purposes of training. Additionally sensors for battery level, three axes of accelerometers, and three axes of gyroscopes were added as more complex inputs for Fido to master. The last sensor added was a three axis radio receiver. The purpose of the receiver was to allow location based training of the robot relative to a radio beacon, such as following the beacon or avoiding it. The radio sensor is a stand-in for Bluetooth, Wi-Fi, or any other radio technology; it is common practice in many areas of robotics to use radio beacons for localization.

Fido's outputs were chosen similarly. Two motors allow a drive mechanism known as differential drive, which will be discussed further in the following section. A buzzer of varying tone and frequency and a multicolor LED complete the set of outputs.

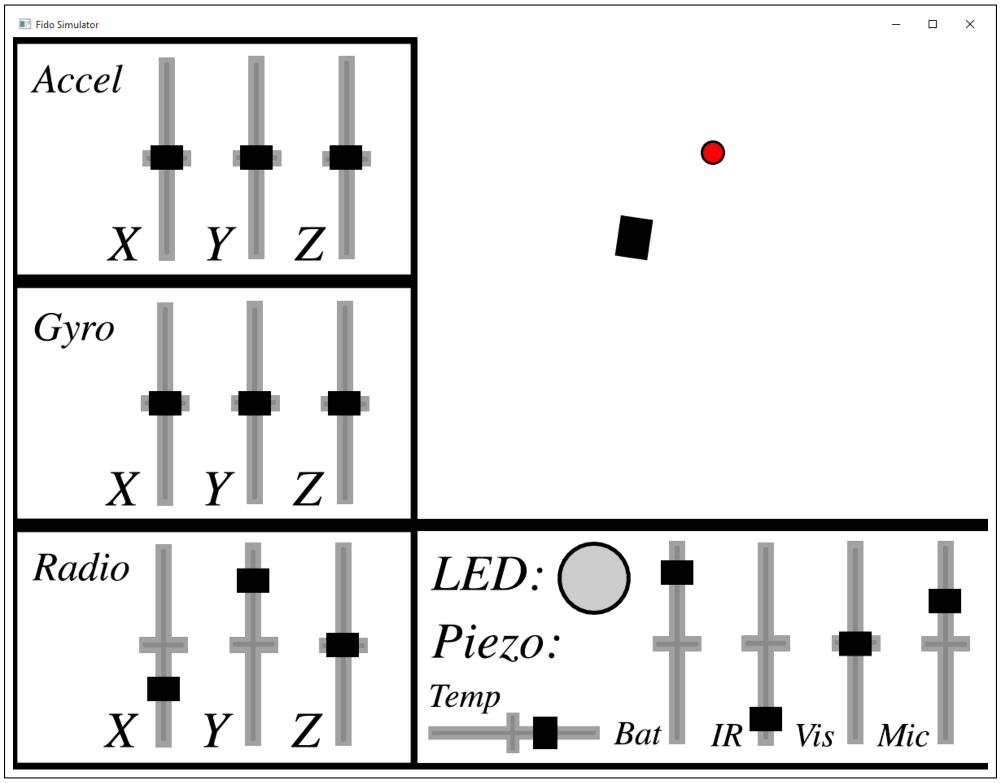

Figure 7: Screenshot of Fido Simulator GUI

4.2 Implementation and Kinematics

A graphical user interface was made using the SFML multimedia library for C++. Sliders manually adjust inputs such as accelerometer and gyroscope axes, battery life, and temperature. A colored circle displays the output of the multicolor LED, while the frequency and volume of the piezoelectric speaker are displayed on the bottom bar. Initially vectors of motor values were displayed graphically in the top right of the window. This made sense for initial testing purposes: both competition entrants are experienced with differential drive robots, and can easily visualize robot movement from robot vectors. However, as we decided to do more development on the simulator we wanted to allow more complex training on the simulator, such as path following. Such training requires a visual kinematic simulation of the robot model.

Fido's two motors are arranged in a differential drive arrangement. Driving the left motor acts as a force vector on the left side of the robot, creating a torque around the right wheel. The same applies with the right wheel. When both motors are activated together, the motor drives straight.

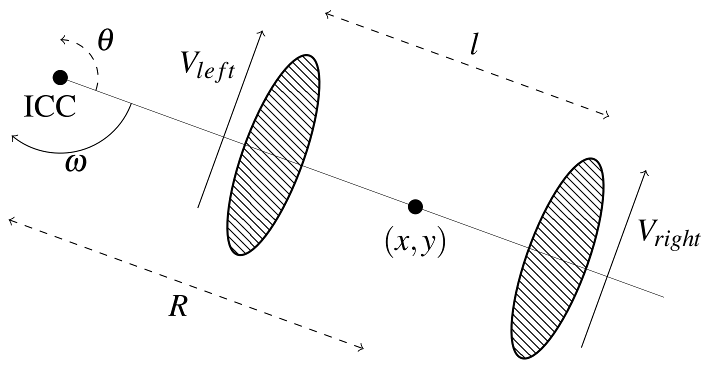

Figure 8: Differential Drive Kinematics Diagram

Each motor has a value ranging from -100 to 100, where -100 is full power backwards, 0 is stopped, and 100 is full power forwards. In order to transform motor values into a plottable transformation, we must first model the movement that our robot will take.

Every movement the robot takes can be interpreted as a rotation of radius R around the ICC, or instantaneous center of curvature. If the robot is moving straight, this radius is simply infinite. The robot travels at an angular velocity ω around the circle, and at any given moment is at the coordinates (x,y) and orientation θ. The length of the robot from wheel-center to wheel-center is defined as l, while the velocity vectors of each motor are defined as V_l and V_r respectively. The passing of time from the last call of motor values is defined as Δt. We can solve for R at any point using the following equation:

$$R = \frac{l}{2} \times \frac{V_l + V_r}{V_r - V_l}. \tag{7}$$

We can solve for ω using the following equation:

$$\omega = \frac{V_r - V_l}{l} \tag{8}$$

And the ICC location using the following equation:

$$ICC = \Big[x - R\sin\theta\,, \quad y + R\cos\theta \Big] \tag{9}$$

We can then use these values to solve for the robot's new position, defined as coordinates (x', y') and orientation θ'.

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos(\omega\Delta t) & -\sin(\omega\Delta t) & 0 \\ \sin(\omega\Delta t) & \cos(\omega\Delta t) & 0 \\ 0 & & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\Delta t \end{bmatrix} \tag{10}$$

A brief inspection of these equations verify their performance in certain use cases. If $V_r = -V_l$ R becomes zero, as the robot turns around it's center. If $V_r = V_l$ R becomes infinite, as the robot is traveling in a straight line. However the methodology by which the equation simplifies in the case of $V_r = V_l$ involves division of zero, which can be problematic in computer programming. Therefore we must first check if $V_r = V_l$, and if so substitute an alternate equation as a simplification:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + V_l\Delta t \cos\Theta \\ y + V_l\Delta t \sin\Theta \\ \theta \end{bmatrix} \tag{11}$$

The use of V_l in this equation rather than V_r is unimportant, as the values are equal. Using these equations were were able to fully simulate Fido's kinematic model, and attach simulator motor inputs to Fido's outputs to visualize learning taking place. The black rectangle in the upper right corner of the simulator is the robot, having been moved as part of training. The red dot near the rectangle is a graphical representation of a radio beacon. As adjusting sliders to represent the location of a radio beacon relative to the robot would be impractical, we decided to implement a beacon that could be placed and dragged by right clicking on the simulator. Simulated sensor readings of beacon strength on two axis are gathered using an inverse square law, as applies to radio waves in general. These readings are then displayed in the sliders and can be manually altered as well. The radio beacon can be removed by a human operator by pressing the P key. This has been especially helpful in the task of training Fido to follow a radio beacon.

5 Results

Results, testing, and applications go here.

5.1 Training Methods

5.2 Findings

5.3 Further Applications

6 Discussion

7 Conclusion

Restate, discuss further study, improving experimentation, etc.

References

Baird, L. C., & Klopf, A. H. (1993). Reinforcement learning with high-dimensional, continuous actions. *Wright Laboratory, Wright-Patterson Air Force Base, Tech. Rep. WL-TR-93-1147*.

Gaskett, C., Wettergreen, D., & Zelinsky, A. (1999). Q-learning in continuous state and action spaces. In *Australian joint conference on artificial intelligence* (pp. 417–428).

Rummery, G. A. (1995). *Problem solving with reinforcement learning* (Unpublished doctoral dissertation). University of Cambridge Ph. D. dissertation.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Unpublished doctoral dissertation). University of Cambridge England.

Werbos, P. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences.