

Human-Robot Interaction and Autonomous Systems

AI 616

Master of Science in Artificial intelligence

Prepared by Dr. Khaled Eskaf

Lecture 10





Course Outlines

Module	Content	Simulation Lab
1. Introduction to Robotics	- Fundamentals of Robot Control - Classification of Robotics - Exploration of Robot Types	- Basic Robot Control - Modeling Robot Classes - Simulating Various Robot Types
2. Advanced HRI Studies	- Introductory to Advanced Concepts in HRI	- Basic to Complex HRI Scenarios
3. Social Dynamics in Robotics	- Architectures for Social Interaction - Case Studies and History of HRI	- Social Interaction Simulations - Historical Evolution in HRI
4. Strategic HRI System Design	- Core Principles and Communication Strategies in HRI	- Designing Interfaces - Communication Protocol Simulations
5. HRI Efficiency and Effectiveness	- Metrics for Mission Effectiveness and Human Factors	- Implementing and Analyzing HRI Metrics
6. Innovative HRI System Design	- Exploring Innovative Designs and Real-World Applications	- Testing Innovative Designs - Simulating Diverse Scenarios
7. HRI System Classification	- Developing Comprehensive Classifications	- Classification Application and Case Studies
8. Human Factors in HRI	- Analyzing Ergonomics and Human Factors	- Ergonomic Simulations - Scenario-Based Human Factor Analysis
9. Robotics and AI Integration	- Theoretical Concepts and Practical AI Applications	- AI Algorithm Implementation - AI-Driven HRI Modeling
10. HRI Case Studies	- Mobile Assistive Robot Studies	- Designing and Testing Mobile Assistive Robots

Chapter

09



CHAPTER 9

To integrate the machine learning model training and prediction directly into a Webots controller script, you'll have to prepare a slightly different setup. This **setup involves simulating the training data collection**, training a decision tree model, and using the model for real-time decision-making based on the robot's camera input.

However, it's important to note that for a real-world scenario or more complex simulations, you'd likely perform the model training outside of the Webots environment due to the computational costs and then load the trained model into your Webots controller for predictions.

Step 1: Setting Up Your Environment

```
from controller import Robot
from sklearn.tree import DecisionTreeClassifier
import numpy as np
```

```
robot = Robot()
timestep = int(robot.getBasicTimeStep())
```

NOTE: You're importing necessary libraries (Robot class from Webots, DecisionTreeClassifier from scikit-learn, and numpy) and initializing your robot. The timestep variable stores the simulation timestep, which is essential for synchronizing your control commands with the simulation.

Step 2: Initializing Devices

```
camera = robot.getDevice('camera')
camera.enable(timestep)
left_motor = robot.getDevice('left wheel motor')
right_motor = robot.getDevice('right wheel motor')
left_motor.setPosition(float('inf'))
right_motor.setPosition(float('inf'))
left_motor.setVelocity(0)
right_motor.setVelocity(0)
```

NOTE: You initialize the camera and motors attached to your robot. Setting the motor position to infinity (`float('inf')`) switches the motors to velocity control mode. Initially, the robot's motors are set to 0 speed, meaning the robot will not move.

Step 3: Preparing Training Data for the Decision Tree

```
training_data = np.array([
    [120, 40, 40, 1], # Red objects mean turn right
    [40, 40, 120, 2], # Blue objects mean move backward
    [80, 110, 90, 0] # Neutral colors mean move forward
])
X_train = training_data[:, :3] # RGB values
y_train = training_data[:, 3] # Actions
```

NOTE: This block creates an array of training data, where each row represents a different scenario. The first three numbers in each row are RGB values simulating the color the robot sees, and the fourth number is the action the robot should take (0 for forward, 1 for right, 2 for backward). This data is split into features (X_train) and labels (y_train).

Chapter

09

p 4: Training the Decision Tree Classifier

```
clf = DecisionTreeClassifier()  
clf.fit(X_train, y_train)
```

NOTE: A decision tree classifier is initialized and trained with the RGB values as features and the corresponding actions as labels. The .fit() method adjusts the model parameters to best fit the training data.

Step 5: Defining the Action Function

```
def perform_action(action, left_motor, right_motor):  
    # This function translates the predicted action into motor commands.
```

NOTE: This function maps the predicted action to actual motor speeds, controlling the robot's movement. Depending on the action, both motors may move forward, rotate in opposite directions (to turn), or both move backward.

Step 6: Main Control Loop

```
while robot.step(timestep) != -1:  
    current_rgb = np.array([[85, 90, 75]]) # Simulated current color  
    predicted_action = clf.predict(current_rgb)[0]  
    perform_action(predicted_action, left_motor, right_motor)
```

NOTE: Within an infinite loop that runs for the simulation's duration, you simulate getting RGB values from the camera and use your trained model to predict the appropriate action. The action is then performed by adjusting the motors' velocities.

- 1) Simulate Color Detection: For demonstration purposes, `current_rgb` is a placeholder for RGB values you might get from processing the camera's image.
- 2) Predict Action: The model predicts the best action (move forward, turn right, move backward) based on the current RGB values.
- 3) Execute Action: The predicted action is translated into motor movements, controlling the robot's behavior in the simulation.

Conclusion

This example demonstrates a basic application of machine learning within a robotics simulation. By training a decision tree to associate colors with movements, you've created a model that can guide a robot's navigation based on visual cues. While simplified, this process lays the foundation for more complex applications involving real-time data processing and decision-making in robotic systems.

```
# Iterate over a 3x3 matrix around the central pixel
for dx in range(-1, 2):
    for dy in range(-1, 2):
        index = ((centerY + dy) * width + (centerX + dx)) * 4
        r = image[index]
        g = image[index + 1]
        b = image[index + 2]
        sum_rgb += np.array([r, g, b])
    print(f"RGB at ({centerX+dx}, {centerY+dy}): ({r}, {g}, {b})") #

# Calculate average RGB values
avg_rgb = sum_rgb / 9
print(f"Average RGB: {avg_rgb}") # Print the average RGB values

# Predict the action based on the current RGB values
predicted_action = clf.predict([avg_rgb])[0]

# Set motor speeds based on the predicted action
speed = 5
if predicted_action == 0: # Move forward
    left_motor.setVelocity(speed)
    right_motor.setVelocity(speed)
elif predicted_action == 1: # Turn right
    left_motor.setVelocity(speed)
    right_motor.setVelocity(-speed)
elif predicted_action == 2: # Move backward
    left_motor.setVelocity(-speed)
    right_motor.setVelocity(-speed)
```

```
"""ML_DecissionTree_3 controller."""  
# Dr. Khaled Eskaf  
#Using ML Decision Tree 3  
from controller import Robot  
import numpy as np  
from sklearn.tree import DecisionTreeClassifier  
# Initialize the robot and simulation timestep  
robot = Robot()  
timestep = int(robot.getBasicTimeStep())  
# Initialize camera and motors  
camera = robot.getDevice('camera')  
camera.enable(timestep)  
left_motor = robot.getDevice('left wheel motor')  
right_motor = robot.getDevice('right wheel motor')  
left_motor.setPosition(float('inf')) # Enable speed control mode  
right_motor.setPosition(float('inf'))  
left_motor.setVelocity(0) # Initially stop the robot  
right_motor.setVelocity(0)
```

```
# Simulated training data for decision making
```

```
training_data = np.array([  
    [120, 40, 40, 1], # Red-ish object  
    [40, 40, 120, 2], # Blue-ish object  
    [80, 110, 90, 0]  # Neutral object  
])
```

```
# Split training data into features and labels
```

```
X_train = training_data[:, :3] # RGB values  
y_train = training_data[:, 3]  # Actions
```

```
# Train the decision tree classifier
```

```
clf = DecisionTreeClassifier()  
clf.fit(X_train, y_train)
```

```
# Main control loop
```

```
while robot.step(timestep) != -1:  
    image = camera.getImage()  
    width = camera.getWidth()  
    height = camera.getHeight()
```

```
# Calculate center pixel coordinates
```

```
centerX = width // 2
```

```
centerY = height // 2
```

```
# Initialize sum of RGB values
```

```
sum_rgb = np.zeros(3)
```

```
# Iterate over a 3x3 matrix around the central pixel
```

```
for dx in range(-1, 2):
```

```
    for dy in range(-1, 2):
```

```
        index = ((centerY + dy) * width + (centerX + dx)) * 4
```

```
        r = image[index]
```

```
        g = image[index + 1]
```

```
        b = image[index + 2]
```

```
        sum_rgb += np.array([r, g, b])
```

```
        print(f"RGB at ({centerX+dx}, {centerY+dy}): ({r}, {g}, {b})") # Print RGB values of each pixel
```

```
# Calculate average RGB values
```

```
avg_rgb = sum_rgb / 9
```

```
print(f"Average RGB: {avg_rgb}") # Print the average RGB values
```

```
# Predict the action based on the current RGB values
predicted_action = clf.predict([avg_rgb])[0]
```

```
# Set motor speeds based on the predicted action
speed = 5
```

```
if predicted_action == 0: # Move forward
```

```
    left_motor.setVelocity(speed)
```

```
    right_motor.setVelocity(speed)
```

```
elif predicted_action == 1: # Turn right
```

```
    left_motor.setVelocity(speed)
```

```
    right_motor.setVelocity(-speed)
```

```
elif predicted_action == 2: # Move backward
```

```
    left_motor.setVelocity(-speed)
```

```
    right_motor.setVelocity(-speed)
```

- 1) Initializes a simulated robot in Webots, equipping it with a camera and motors for navigation.
- 2) Sets up training data linking specific RGB values to actions (forward, right, backward) using a decision tree classifier.
- 3) Trains the classifier on this data, enabling the robot to make decisions based on color detected by the camera.
- 4) During operation, captures real-time camera images, extracting average RGB values from a central 3x3 pixel matrix.
- 5) Uses the trained model to predict actions based on these RGB values, guiding the robot's movement.
- 6) Demonstrates basic machine learning application in robotics for environment-responsive navigation based on visual cues.

1) Import Libraries:

- Robot from Webots allows you to control the simulated robot.
- numpy is used for handling arrays efficiently.
- DecisionTreeClassifier from scikit-learn is the machine learning model the robot will use to make decisions.

2) Initialize the Robot and Simulation Timestep:

- Create a Robot instance to interact with the simulation.
- timestep determines how often the robot's sensors are updated and actions are taken.

3) Set Up Camera and Motors:

- Access the camera and motors by their names, then enable the camera and set the motors to velocity control mode by assigning their positions to infinity. Initially, the motors are stopped.

4) Prepare Training Data:

- The training data simulates situations where different RGB color values are associated with specific actions (e.g., moving forward, turning, or moving backward). This data is represented in an array where each row contains RGB values followed by an action label.

5) Train the Decision Tree Classifier:

- The features (RGB values) and labels (actions) are extracted from the training data. The decision tree classifier is then trained on these features and labels, learning to associate specific color patterns with actions.

6) Main Control Loop:

- In an infinite loop, the robot continually captures images from its camera, calculates the average RGB values from a central 3x3 matrix, and uses the trained classifier to predict an action based on these values.

7) Calculate Average RGB Values:

- For each pixel in the 3x3 matrix centered around the camera's focal point, extract the RGB values, sum them, and then divide by 9 to get the average RGB values. This step involves iterating over the pixels, extracting their color, and accumulating the values.

8) Predict and Perform Actions:

The average RGB values are fed into the trained classifier, which predicts the most appropriate action (move forward, turn right, or move backward). The predicted action is then translated into motor velocities to move the robot accordingly.

9) Print Statements:

Throughout the process, print statements output the RGB values of each pixel in the 3x3 matrix and the overall average RGB values. This helps in understanding how the robot perceives its environment and decides on its actions.

```
# Import necessary libraries:
# - Robot: to control the simulated robot in Webots.
# - numpy: for efficient array operations, crucial for handling data.
# - DecisionTreeClassifier: a simple machine learning model from scikit-learn.
from controller import Robot
import numpy as np
from sklearn.tree import DecisionTreeClassifier

# Create an instance of the Robot class to interact with the simulation.
robot = Robot()

# Get the basic time step of the current world, which determines
# how often the simulation updates the robot's sensors and actuators.
timestep = int(robot.getBasicTimeStep())

# Access the camera and motor devices by their names in the robot's definition.
# Then, enable the camera to start capturing images at the simulation's timestep.
camera = robot.getDevice('camera')
camera.enable(timestep)
```

Chapter

09

```
# For the motors, set their positions to infinity, which switches them  
# to velocity control mode, allowing us to specify speed rather than position.
```

```
left_motor = robot.getDevice('left wheel motor')  
right_motor = robot.getDevice('right wheel motor')  
left_motor.setPosition(float('inf'))  
right_motor.setPosition(float('inf'))
```

```
# Initially, stop the robot by setting both motors' speeds to 0.
```

```
left_motor.setVelocity(0)  
right_motor.setVelocity(0)
```

```
# Define the training data for the decision tree classifier.
```

```
# This data simulates scenarios where certain RGB color values of objects  
# seen by the robot are associated with specific actions (turning or moving).
```

```
training_data = np.array([  
    [120, 40, 40, 1], # Red objects make the robot turn right.  
    [40, 40, 120, 2], # Blue objects make the robot move backward.  
    [80, 110, 90, 0]  # Neutral (less distinctly colored) objects make the robot move forward.  
])
```

Chapter

09

```
# Split the training data into features (X) and labels (y).
# Features are the RGB values, and labels are the actions.
X_train = training_data[:, :3] # Extract the first three columns as features.
y_train = training_data[:, 3]  # Extract the last column as labels.

# Initialize the decision tree classifier and train (fit) it on the provided data.
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# The main control loop of the robot.
while robot.step(timestep) != -1:
    # Retrieve the current image captured by the camera.
    image = camera.getImage()
    # Get the width and height of the camera image to calculate the center pixel.
    width = camera.getWidth()
    height = camera.getHeight()

    # Calculate the coordinates of the center pixel.
    centerX = width // 2
    centerY = height // 2
```

Prepare to calculate the average RGB values of the 3x3 area around the center pixel.

```
sum_rgb = np.zeros(3) # Initialize a sum for RGB values.
```

Loop over the pixels in a 3x3 area around the center.

```
for dx in range(-1, 2):
```

```
    for dy in range(-1, 2):
```

```
        # Calculate the index for each pixel in the flattened image array.
```

```
        index = ((centerY + dy) * width + (centerX + dx)) * 4
```

```
        # Extract the RGB values of each pixel and add them to the sum.
```

```
        r = image[index]
```

```
        g = image[index + 1]
```

```
        b = image[index + 2]
```

```
        sum_rgb += np.array([r, g, b])
```

```
        # Print the RGB values for educational purposes.
```

```
        print(f"RGB at ({centerX+dx}, {centerY+dy}): ({r}, {g}, {b})")
```

```
# Calculate the average RGB values by dividing the sum by the number of pixels (9).
avg_rgb = sum_rgb / 9
# Print the average RGB values for insight into the decision-making process.
print(f"Average RGB: {avg_rgb}")

# Use the trained classifier to predict the action based on the average RGB values.
predicted_action = clf.predict([avg_rgb])[0]

# Based on the predicted action, set the speeds of the motors to move the robot.
speed = 5 # Define a base speed.
if predicted_action == 0: # If the action is to move forward.
    left_motor.setVelocity(speed)
    right_motor.setVelocity(speed)
elif predicted_action == 1: # If the action is to turn right.
    left_motor.setVelocity(speed)
```

```
# If the action is to turn right, the left motor moves forward and the right motor moves backward.  
    left_motor.setVelocity(speed)  
    right_motor.setVelocity(-speed)  
elif predicted_action == 2: # If the action is to move backward.  
    # If the action is to move backward, both motors move in reverse.  
    left_motor.setVelocity(-speed)  
    right_motor.setVelocity(-speed)
```

Chapter

09

**Robotics & AI**

Integrations between robotics and AI are shaping the future of numerous industries, transforming how tasks are performed, from manufacturing and healthcare to agriculture and domestic chores. This synergy enhances the capabilities of robots, allowing them to operate more autonomously, make decisions in real-time, and adapt to new or changing environments without human intervention.

In the manufacturing sector, AI-enhanced robots are being used for a variety of tasks including assembly, painting, and quality control. AI algorithms can analyze data from the manufacturing process in real-time, enabling robots to identify and correct errors, optimize production processes, and predict maintenance needs.



Robots equipped with AI are revolutionizing the healthcare industry by assisting in surgeries with precision beyond human capability, managing patient care, and even providing companionship. These robots can learn from data collected during medical procedures, improving their performance over time.



AI-driven robotic systems are being deployed in agriculture for tasks such as planting seeds, harvesting crops, and monitoring field conditions. These robots can analyze data from various sensors and satellite images to optimize crop yield and reduce waste.



Robots, combined with AI, play a crucial role in search and rescue missions, especially in environments that are dangerous or inaccessible to humans. They can navigate through rubble, identify victims using heat sensors, and assess structures' stability.



In the domestic sphere, robotic vacuum cleaners and lawn mowers are becoming smarter through AI, learning the layout of homes and gardens to optimize their cleaning paths. Future developments may lead to more complex domestic robots capable of performing a wide range of household tasks.



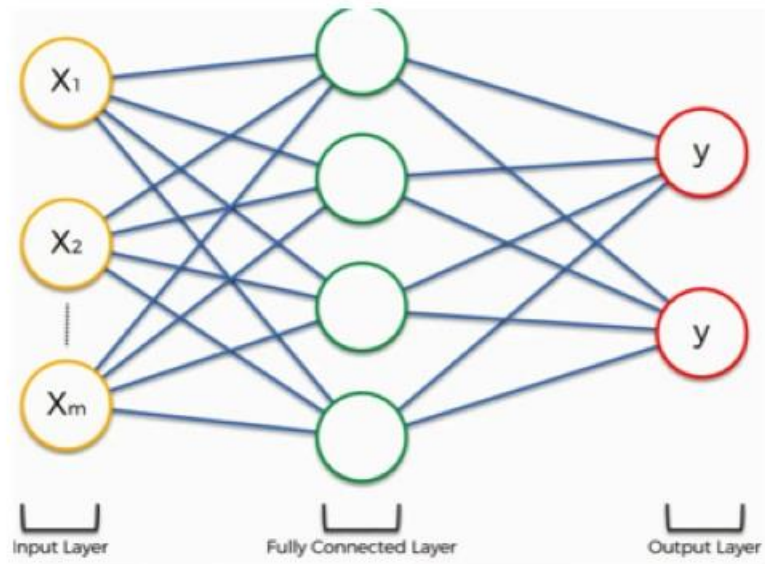
Autonomous vehicles, essentially robots on wheels, rely heavily on AI for navigating and making decisions on the road. This integration is crucial for interpreting sensor data, predicting the actions of other road users, and choosing the safest course of action.



AI-powered robots are streamlining logistics and supply chain operations by automating warehousing tasks, such as sorting, packing, and shipping goods. These robots can adapt to changes in inventory levels or demand patterns, improving efficiency and reducing operational costs.



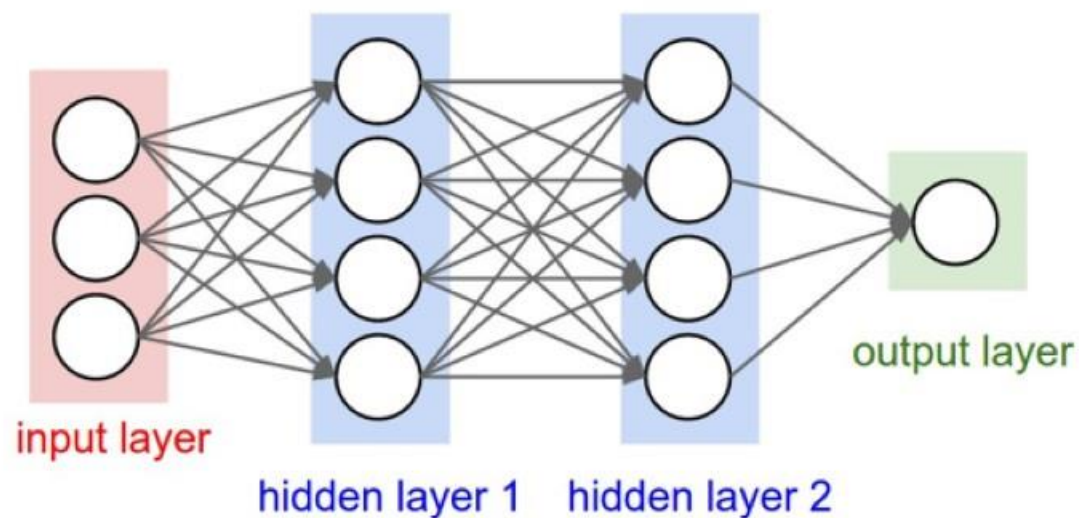
CHAPTER 10



we use a simple **Fully Connected Neural Network (FCNN)**, also known as a **Dense Neural Network**.

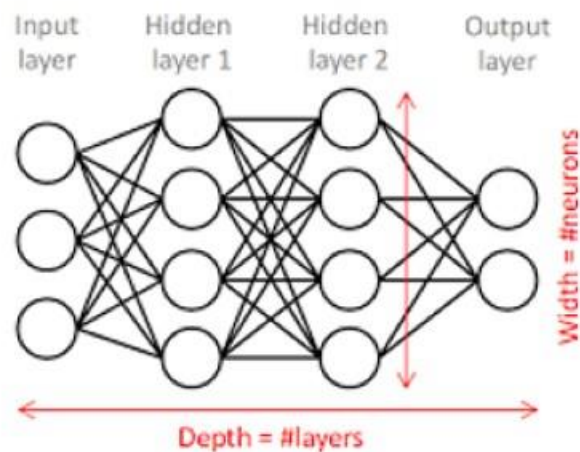
What is a Fully Connected Neural Network?

A Fully Connected Neural Network consists of multiple layers of neurons, where each neuron in a layer is connected to every neuron in the next layer. The "fully connected" term comes from this characteristic—each neuron receives input from all neurons in the previous layer, processes it, and passes its output to every neuron in the next layer.



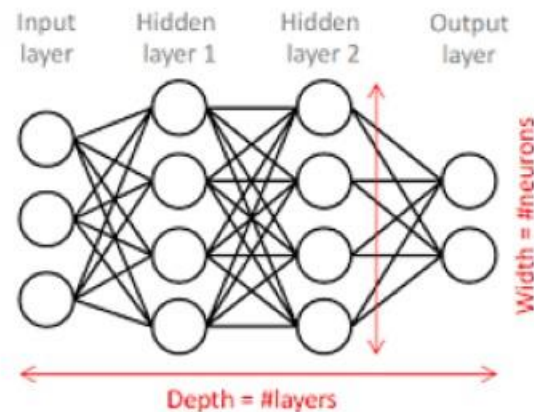
Input Layer:

- The input layer is the first layer of the neural network. In our example, it consists of **3 neurons**.
- These neurons correspond to the normalized RGB (Red, Green, Blue) values of the input (a simplified representation of what the robot's camera "sees").
- Since we're working with RGB values, the input dimension is 3 (one for each color channel).



Hidden Layers:

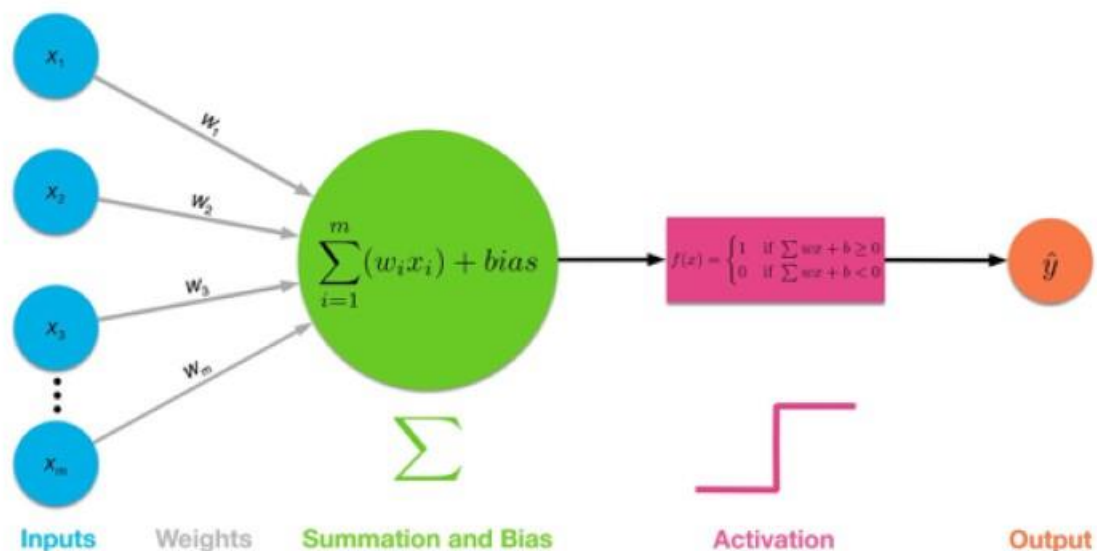
- Following the input layer, we have **hidden layers** that allow the neural network to learn complex patterns and relationships in the data.
- In the provided example, there is **one hidden layer** with **64 neurons**. The choice of 64 neurons is somewhat arbitrary for this simple example and could be adjusted based on the complexity of the problem.
- Each neuron in the hidden layer takes the weighted sum of all inputs, applies an activation function (in this case, **ReLU**, or Rectified Linear Unit), and passes the result to the next layer. The ReLU activation function is commonly used in neural networks for its simplicity and efficiency. It simply outputs the input if it's positive and zero otherwise.



- The final layer is the **output layer**, which in our example, has **3 neurons**. Each neuron represents one **possible action the robot can take based on the input image**: Move Forward, Turn Right, or Move Backward.
- The activation function used in the output layer is **softmax**, which is standard for classification problems. The softmax function converts the outputs of the last layer into probabilities by taking the exponential of each output and then normalizing these values by dividing by the sum of all exponentials. This ensures that the output values are between 0 and 1 and sum up to 1.
- The neuron with the highest probability determines the action the robot will take.

1- Forward Pass:

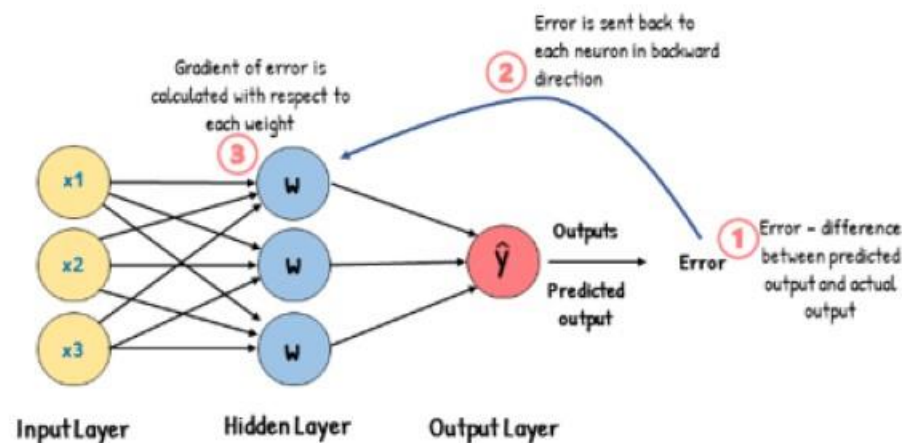
- When the neural network makes a decision, it performs a "forward pass." This means it takes the input (RGB values), processes it through each layer of the network by computing the weighted sums and applying the activation functions, and finally produces the probabilities of each action in the output layer.



2- Training:

- During training, the network adjusts the weights and biases (parameters) within its neurons to minimize the difference between its predicted output and the actual desired output (known as the loss or cost). This process involves backpropagation, where the network calculates the gradient of the loss function with respect to each parameter and updates the parameters in the direction that reduces the loss.
- The model uses the **Adam optimizer** for updating the weights and biases, and the **categorical_crossentropy** loss function, which is typical for multi-class classification problems.

Backpropagation



you'll need to install several Python libraries:

- **NumPy**: For numerical operations, especially handling arrays (used extensively for data manipulation in machine learning).
- **Keras (with TensorFlow backend)**: For creating and training the neural network model.

You can install these libraries using pip (Python's package installer). Open your command line or terminal and run:

```
pip install numpy tensorflow keras
```

Creating a neural network that learns to **associate specific colors with actions**, like turning right, moving backward, or moving forward.

Key Points:

- **Normalization:** We normalize the RGB values by dividing by 255. This is because machine learning models generally work better with smaller input values.
- **One-hot Encoding:** This is a way of representing categorical data, like our actions, as binary vectors. It's essential for classification tasks.
- **Sequential Model:** This is one of the simplest types of models in Keras, where you define a stack of layers in sequence. Here, we use three layers: an input layer, one hidden layer, and an output layer.
- **Activation Functions:** **ReLU** (Rectified Linear Unit) is used for the hidden layers to add non-linearity to the model, making it capable of learning more complex patterns. **Softmax** is used in the output layer to convert the network's output to probabilities for each class (action).
- **Model Compilation:** This step prepares the model for training, specifying the optimizer, loss function, and metrics to monitor.
- **Model Training:** The model is trained with the **fit** method on our training data. Training involves adjusting the model's weights to minimize the loss function, improving the model's predictions over epochs.
- **Saving the Model:** After training, the model is saved to disk. This allows us to reuse the trained model without retraining.

Training a simple neural network model to recognize colors and decide actions based on those colors.

```
# First, we import necessary libraries.
```

```
from controller import Robot
```

```
import numpy as np
```

```
from keras.models import Sequential, load_model
```

```
from keras.layers import Dense
```

```
import os #is used to check if the neural network model file (my_model.h5) already exists in the filesystem.
```

We define a function called 'train_model' that will handle the training of our neural network.

```
def train_model():
```

```
    """
```

This function trains a simple neural network on some predefined data.

```
    """
```

We have some example training data. Each item in this list represents the average

color (in RGB) of some objects our robot might see. The colors are in the format [R, G, B].

The "/ 255.0" part is to normalize these values, so they are between 0 and 1, which is

a format neural networks work well with.

```
X_train = np.array([[120, 40, 40], [40, 40, 120], [80, 110, 90]]) / 255.0
```

The labels for our training data. These tell the network what **action to take for each color**:

[1, 0, 0] means "Turn Right" for the first color, [0, 0, 1] means "Move Backward" for the second color,

and [0, 1, 0] means "Move Forward" for the third color. This is called "one-hot encoding."

```
y_train = np.array([[1, 0, 0], [0, 0, 1], [0, 1, 0]])
```

Next, we define our neural network's structure. It's a simple one with one input layer,

one hidden layer, and one output layer.

```
model = Sequential([
```

```
    Dense(64, activation='relu', input_dim=3), # The input layer has 3 neurons (for R, G, B), and uses ReLU activation.
```

```
    Dense(64, activation='relu'), # The hidden layer also has 64 neurons and uses ReLU activation.
```

```
    Dense(3, activation='softmax') # The output layer has 3 neurons (for our actions) and uses softmax activation.
```

```
])
```

Now we compile our model. This step configures the model for training.

```
model.compile(optimizer='adam',          # The Adam optimizer is a popular choice.  
              loss='categorical_crossentropy', # This loss function is good for classification tasks like ours.  
              metrics=['accuracy'])      # We'll track accuracy during training.
```

**We train (fit) our model on our dataset. We do this for 100 "epochs," which means
the model will see the entire dataset 100 times. We don't print out progress (verbose=0).**

```
model.fit(X_train, y_train, epochs=100, verbose=0)
```

After training, we **save our model** to a file so we can use it later without having
to retrain it every time we start our Webots simulation.

```
model.save('my_model.h5')
```

Focus on the integration of the trained model with the Webots robot for making decisions based on the robot's camera input.

Step 1: Initialize the Webots Robot

Create an instance of the Robot class. This is necessary to interact with the robot and its devices.

```
robot = Robot()
```

Step 2: Check for an Existing Model

Before we use the neural network, we check if we've already trained and saved it.

This is to avoid retraining the network every time we start the simulation, saving time and resources.

```
if not os.path.exists('my_model.h5'):
```

If the model doesn't exist, we call the function `train\_model()` to train and save our model.

```
train_model()
```

Step 3: Load the Trained Model

Once we have our model (either by just training it or because it was already saved), we load it.

This model will be used to make decisions based on the robot's camera input.

```
model = load_model('my_model.h5')
```

Step 4: Initialize Camera and Motors

To make decisions based on what the robot "sees," we need to initialize and enable the camera.

```
camera = robot.getDevice('camera')
```

```
camera.enable(64) # The camera is enabled with a time step of 64 milliseconds.
```

We also initialize the motors. These motors control the robot's wheels, allowing it to move.

```
left_motor = robot.getDevice('left wheel motor')
```

```
right_motor = robot.getDevice('right wheel motor')
```

To control the robot's movement through velocity, we set the motor position to infinity.

```
left_motor.setPosition(float('inf'))
```

```
right_motor.setPosition(float('inf'))
```

Initially, we don't want the robot to move, so we set the velocity of both motors to 0.

```
left_motor.setVelocity(0)
```

```
right_motor.setVelocity(0)
```

Step 5: Define a Function to Process Camera Images and Make Decisions

```
def process_image_and_decide():
```

```
    """
```

This function grabs the current view from the robot's camera, processes it to understand the colors it sees, and decides on an action based on those colors.

```
    """
```

```
# We get the RGB array from the camera. This is a 3D array with dimensions corresponding to the width,
# height of the image, and the RGB color channels.
```

```
image = camera.getImageArray()
```

```
width = camera.getWidth()
```

```
height = camera.getHeight()
```

```
# Calculate the average RGB values across the entire image.
```

```
avg_r = avg_g = avg_b = 0
```

```
for x in range(width):
```

```
    for y in range(height):
```

```
        avg_r += image[x][y][0]
```

```
        avg_g += image[x][y][1]
```

```
        avg_b += image[x][y][2]
```

```
avg_r /= (width * height)
```

```
avg_g /= (width * height)
```

```
avg_b /= (width * height)
```

Normalize these averages to be between 0 and 1.

```
input_data = np.array([[avg_r, avg_g, avg_b]]) / 255.0
```

Use the neural network model to predict the best action based on these colors.

```
prediction = model.predict(input_data)
```

```
action = np.argmax(prediction) # We choose the action with the highest predicted score.
```

```
return action
```

```
# This loop runs continuously for the duration of the simulation. In each iteration, the robot
# processes its camera input, makes a decision on how to move, and then executes that movement.
while robot.step(64) != -1:
    action = process_image_and_decide()

    # Depending on the action decided by the neural network, we set the velocities of the motors.
    # This causes the robot to move forward, turn right, or move backward.
    if action == 0: # Move Forward
        left_motor.setVelocity(5.0)
        right_motor.setVelocity(5.0)
    elif action == 1: # Turn Right
        left_motor.setVelocity(5.0)
        right_motor.setVelocity(-5.0)
    elif action == 2: # Move Backward
        left_motor.setVelocity(-5.0)
        right_motor.setVelocity(-5.0)
```



Thank You