Artificial Intelligence Project

Genetic Programming for Inverted Pendulum

by Salih Karagollu

GDAŃSK UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

Purpose: To make the inverted pendulum stay in balance.

Evaluation Method: Training an Artificial Neural Network (ANN) **or** a mathematical model of the pendulum with genetic algorithms to find the optimal weights that will allow us to control the system.

Creation of ANN:

The ANN consists of 4 input neurons (theta, theta_dot, x, x_dot), 2 hidden layers with 8 neurons and one output neuron (force).

```python
def preprocess_state(state):
    return state.reshape((1, input_size))

def neural_controller(model, preprocessed_state):
    force = model.predict(preprocessed_state)
    return force

input_size = 4  # Number of state variables
output_size = 1  # Number of force outputs

model = tf.keras.Sequential()
model.add(Dense(units=8, input_shape=(4,) , activation='relu'))
model.add(Dense(units=8, activation='relu'))
model.add(Dense(units=output_size, input_dim=8, activation='linear'))
model.compile(loss='mean_absolute_error', optimizer='adam', metrics=['accuracy'])
```

Training:

Initializing the training, applying the control function and accorsing to that giving each individual a reward, ranking the highest-rewarded (population_size * selection_factor) individuals and reproducing with mutation and crossover to get the new population.

```python
p_cross = 0.85
    p_mut = 0.01
    mutation_scale= 4.0
    selection_factor = 1.0
    controller = genetic_controller


# controller parameter vector initialization:
    population_size = 100
    population = []
    for _ in range(population_size):
        individual = Sequential.from_config(model.get_config())
# Create a copy of the model
        individual.set_weights(model.get_weights())
# Set the weights of the model
        population.append(individual)

    temp_population = population[:int(selection_factor * population_size)]
# Reproduce the population to create randomness
    population = neuro_reproduction(temp_population, p_cross, p_mut, mutation_scale)
    for episode in range(num_of_episodes):
        initial_state_no = episode % num_of_initial_states
        state = initial_states[initial_state_no, :]

        step = 0
        if_pendulum_fall = 0
        rewards = []
        rewarded_solutions = []
        for individual in population:
            state = initial_states[initial_state_no, :]
            while (step < 1000) & (if_pendulum_fall == 0):
                step += 1
                F = controller(individual, state)
                # F = 200  # for now


# new state determination:
                new_state = next_state(state, F)

                if_pendulum_fall = (abs(new_state[0]) >= np.pi / 2)
                state = new_state

# R = reward(state[0], state[1])
                R = reward(state, new_state, F, step)
                rewards.append(R)
                rewarded_solutions.append((R, individual, state))


        rewarded_solutions.sort(key=lambda x: x[0], reverse=True)

        best_solutions = rewarded_solutions[:int(selection_factor * population_size)]
        best_solutions = [x[1] for x in best_solutions]
        best_solutions = np.array(best_solutions)

        #reproduction:
        population = neuro_reproduction(best_solutions, p_cross, p_mut, mutation_scale)

        # test + history file generation:
        if episode % (num_of_episodes / 3) == 0:
            inv_pendulum_test(initial_states, controller, best_solutions[0])

    inv_pendulum_test(initial_states, controller, best_solutions[0])

```

Controller Functions:

```python
#NEURAL NETWORK
def genetic_controller(model, state):
    # Extract the state variables
    theta, theta_dot, x, x_dot = state

    # Define the system parameters
    Fmax, time_step, g, friction, cart_weight, pend_weight, drw = global_variables()
    m = pend_weight  # Mass of the pendulum
    M = cart_weight  # Mass of the cart
    l = drw  # Length of the pendulum

    dense_input = tf.keras.layers.Dense(units=state.shape[0], input_shape=(state.shape[0],))

    dense_state = dense_input(tf.expand_dims(state, axis=0))

    original_stdout = sys.stdout
    sys.stdout = open(os.devnull, 'w')

    force = model.predict(dense_state)

    sys.stdout = original_stdout

    return force
```

```python
#MATHEMATICAL MODEL
def genetic_controller(weights, state):
    # Extract the state variables
    theta, theta_dot, x, x_dot = state

    # Define the system parameters
    Fmax, time_step, g, friction, cart_weight, pend_weight, drw = global_variables()
    m = pend_weight  # Mass of the pendulum
    M = cart_weight  # Mass of the cart
    l = drw  # Length of the pendulum


    # Calculate the state derivatives using the equations of motion
    x_ddot = (m * l * theta_dot**2 * np.sin(theta) - m * g * np.sin(theta) * np.cos(theta)) / (M + m - m * np.cos(theta)**2)
    theta_ddot = (g * np.sin(theta) - np.cos(theta) * x_ddot) / l


    # Calculate the force using a non-linear combination of the state variables and the weight vector
    force = weights[0] * x_ddot + weights[1] * theta_ddot + weights[2] * np.sin(theta) + weights[3] * np.sin(x)

    return force
```

Controller function helps the program to guess the force value based on the current weights.

Reproduction, Crossover and Mutation:

There are also two versions for this: ANN and mathematical.

```python
#NEURAL NETWORK
def reward(state,new_state,F, step):
    penalty_diff = abs(new_state[0])**2 + 0.25*(abs(new_state[1]))**2 + 0.0025*abs(new_state[2])**2 + 0.0025* abs(new_state[3])**2
    penalty_fall = (abs(new_state[0]) >= np.pi / 2) * (200 / step) * 1000

# ........................................

# ........................................
    return -(penalty_diff + penalty_fall)

def neuro_mutation(individual, p_mut, mutation_scale):

# Add random noise to the individual's weights
    weights = individual.get_weights()
    for i in range(len(weights)):
        mask = np.random.choice([0, 1], size=weights[i].shape, p=[1 - p_mut, p_mut])
        noise = np.random.normal(mutation_scale, size=weights[i].shape)
        weights[i] += mask * noise
    individual.set_weights(weights)

def neuro_crossover(parent1, parent2, p_cross):

# Perform crossover by averaging the weights of the parents
    if np.random.rand() < p_cross:
        parent1_weights = parent1.get_weights()
        parent2_weights = parent2.get_weights()
        child_weights = tf.keras.models.clone_model(model).get_weights()
        for i in range(len(child_weights)):
            child_weights[i] = (parent1_weights[i] + parent2_weights[i]) / 2
        child = tf.keras.models.clone_model(model)
        child.set_weights(child_weights)
        return child
    else:

# If crossover doesn't occur, return a random parent
        return np.random.choice([parent1, parent2])

def neuro_reproduction(population, p_cross, p_mut, mutation_scale):
    new_population = []
    for _ in range(len(population)):
        parent1 = np.random.choice(population)
        parent2 = np.random.choice(population)
        child = neuro_crossover(parent1, parent2, p_cross)
        neuro_mutation(child, p_mut, mutation_scale)
        new_population.append(child)
    return new_population
```

```python
#MATHEMATICAL MODEL
# reward for transition from state to new state with action F
def reward(state,new_state,F, step):
    penalty_diff = abs(new_state[0])**2 + 0.25*(abs(new_state[1]))**2 + 0.0025*abs(new_state[2])**2 + 0.0025* abs(new_state[3])**2
    penalty_fall = (abs(new_state[0]) >= np.pi / 2) * (200 / step) * 1000

# ........................................

# ........................................
    return -(penalty_diff + penalty_fall)


def reproduction(best_population, population_size, p_mutation, p_crossover, mutation_scale=0.1):
    new_population = []
    for _ in range(population_size):
        random_row = np.random.choice(best_population.shape[0])
        parent1 = best_population[random_row]
        random_row = np.random.choice(best_population.shape[0])
        parent2 = best_population[random_row]
        child = crossover(parent1, parent2, p_crossover)
        mutated_child = mutation(child, p_mutation, mutation_scale)
        new_population.append(mutated_child)
    return new_population

def crossover(parent1, parent2, p_crossover):
    child = np.zeros(parent1.shape)
    for i in range(parent1.shape[0]):
        if np.random.rand() < p_crossover:
            child[i] = (parent1[i] + parent2[i]) / 2.0
        else:
            child[i] = parent1[i]
    return child


def mutation(individual, p_mutation, mutation_scale=0.1):
    mutated_individual = individual.copy()
    for i in range(mutated_individual.shape[0]):
        if np.random.uniform() < p_mutation:

# Add random noise to the parameter with a scale defined by mutation_scale
            mutated_individual[i] += np.random.normal(scale=mutation_scale)
    return mutated_individual
```

Conclusion

The program tries to obtain better weights at each iteration in both methods. To obtain better results, training with more unique initial_state's, increasing the number of episodes, widening the weights or optimizing the parameters of genetic algorithm is necessary.

Better results means that the pendulum can stay upright as long as possible and with the least amount of change in location and speed.

Overall, this is an effective way of applying genetic algorithms and ANN's to a problem. Inverted pendulum is a classical control problem which makes it suitable for our purpose.