ДОДАТОК А

Коефіцієнти впливу маси пального на час

Таблиця А.1 – Результати обчислення коефіцієнтів впливу маси пального на час для трас «Формули-1»

Назва траси	**Коефіцієнту впливу маси пального на час (γ)**
Yas Marina Circuit	0,0408
Albert Park Circuit	0,0411
Red Bull Ring	0,0263
Baku City Circuit	0,0424
Bahrain International Circuit	0,0327
Circuit de Spa-Francorchamps	0,0583
Silverstone Circuit	0,0665
Circuit Gilles-Villeneuve	0,0261
Circuit Zandvoort	0,04
Emilia Romagna Grand Prix	0,0503
French Grand Prix	0,0395
Hungaroring	0,0381
Autodromo Nazionale Monza	0,003
Suzuka Circuit	0,0102
Autódromo Hermanos Rodríguez	0,0306
Miami International Autodrome	0,0693
Circuit de Monaco	0,0092
Jeddah Cornich Circuit built	0,0781
Marina Bay Street Circuit	0,0815
Circuit de Barcelona-Catalunya	0,0397
Autódromo José Carlos Pace	0,1124
Circuit of The Americas	0,0299
Las Vegas Strip Circuit	0,0425
Lusail International Circuit	0,0876
Shanghai International Circuit	0,0338

ДОДАТОК Б

Вихідний код програми

```
def create_sequence_tensor(df, features, target, sequence_length=5):
  X_list = []
  y_list = []

  grouped_stints = df.groupby(['Year', 'EventName', 'Driver', 'Stint'])
  for _, stint_data in grouped_stints:
    stint_data = stint_data.sort_values('LapNumber').reset_index(drop=True)
    if len(stint_data) <= sequence_length: continue

    feature_array = stint_data[features].values
    target_array = stint_data[target].values

    for i in range(len(stint_data) - sequence_length):
      X_window = feature_array[i : i + sequence_length]
      y_target = target_array[i + sequence_length]
      X_list.append(X_window)
      y_list.append(y_target)

  X_tensor = np.array(X_list)
  y_tensor = np.array(y_list)

  return X_tensor, y_tensor

features = [
      'TyreLife', 'TyreLife_Squared', 'TrackTemp', 'Grip_Evolution_State',
'Longitudinal_Energy',
      'Lateral', 'Thermal_Degradation_Risk', 'Slip_Ratio', 'Asphalt Grip'
  ] + comp_cols
target = 'Degradation_Delta'

train_data = data[data['Year'].isin([2022, 2023])].copy()
val_data = data[data['Year'] == 2024].copy()
test_data = data[data['Year'] == 2025].copy()

scaler = MinMaxScaler()

train_data[features] = scaler.fit_transform(train_data[features])
val_data[features] = scaler.transform(val_data[features])
test_data[features] = scaler.transform(test_data[features])

stds = []
for name, values in train_data[features].items():
  std = np.std(values)
  stds.append(std)

delta = np.median(stds)
```

```
sequence_length = 10
X_train, y_train = create_sequence_tensor(train_data, features, target,
sequence_length)
X_val, y_val = create_sequence_tensor(val_data, features, target,
sequence_length)
X_test, y_test = create_sequence_tensor(test_data, features, target,
sequence_length)

def build_model(timesteps, features):
  inputs = Input(shape=(timesteps, features))
  lstm_1 = LSTM(32, return_sequences=True)(inputs)
  batch_norm_1 = BatchNormalization()(lstm_1)
  dropout_1 = Dropout(0.3)(batch_norm_1)

  lstm_2 = LSTM(16, return_sequences=False)(dropout_1)
  batch_norm_2 = BatchNormalization()(lstm_2)
  dropout_2 = Dropout(0.3)(batch_norm_2)

  dense_1 = Dense(8, activation='relu')(dropout_2)
  outputs = Dense(1, activation='linear')(dense_1)

  model = Model(inputs=inputs, outputs=outputs)

  model.compile(optimizer=Adam(learning_rate=0.001),
loss=Huber(delta=delta), metrics=['mae'])
  return model

timesteps = X_train.shape[1]
features = X_train.shape[2]

model_lstm = build_model(timesteps, features)
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
patience=3, min_lr=1e-5, verbose=1)
early_stop = EarlyStopping(monitor='val_loss', patience=12,
restore_best_weights=True)
history_lstm = model_lstm.fit(X_train, y_train_scaled, epochs=100,
batch_size=32, validation_data=[X_val, y_val_scaled],
callbacks=[early_stop, lr_scheduler], verbose=1)

def build_model_gru(timesteps, features):
  inputs = Input(shape=(timesteps, features))
  gru_1 = GRU(32, return_sequences=True)(inputs)
  batch_norm_1 = BatchNormalization()(gru_1)
  dropout_1 = Dropout(0.3)(batch_norm_1)

  gru_2 = GRU(16, return_sequences=False)(dropout_1)
  batch_norm_2 = BatchNormalization()(gru_2)
  dropout_2 = Dropout(0.2)(batch_norm_2)

  dense_1 = Dense(8, activation='relu')(dropout_2)
```

```
  outputs = Dense(1, activation='linear')(dense_1)

  model = Model(inputs=inputs, outputs=outputs)

  model.compile(optimizer=Adam(learning_rate=0.001),
loss=Huber(delta=delta), metrics=['mae'])
  return model

timesteps = X_train.shape[1]
features = X_train.shape[2]

model_gru = build_model_gru(timesteps, features)
history_gru = model_gru.fit(X_train, y_train_scaled, epochs=100,
batch_size=32, validation_data=[X_val, y_val_scaled],
callbacks=[early_stop, lr_scheduler], verbose=1)

def build_model_rnn(timesteps, features):
  inputs = Input(shape=(timesteps, features))
  gru_1 = SimpleRNN(32, return_sequences=True)(inputs)
  batch_norm_1 = BatchNormalization()(gru_1)
  dropout_1 = Dropout(0.3)(batch_norm_1)

  gru_2 = SimpleRNN(16, return_sequences=False)(dropout_1)
  batch_norm_2 = BatchNormalization()(gru_2)
  dropout_2 = Dropout(0.3)(batch_norm_2)

  dense_1 = Dense(8, activation='relu')(dropout_2)
  outputs = Dense(1, activation='linear')(dense_1)

  model = Model(inputs=inputs, outputs=outputs)

  model.compile(optimizer=Adam(learning_rate=0.001),
loss=Huber(delta=delta), metrics=['mae'])
  return model

timesteps = X_train.shape[1]
features = X_train.shape[2]

model_rnn = build_model_rnn(timesteps, features)
history_rnn = model_rnn.fit(X_train, y_train_scaled, epochs=100,
batch_size=32, validation_data=[X_val, y_val_scaled],
callbacks=[early_stop, lr_scheduler], verbose=1)


class RaceState:
  def __init__(self, lap, max_laps, total_time, current_compound, tyre_age,
               compounds_used, track_profile, fuel_load_proxy,
num_pit_stops,
               stint_base_pace, model, onnx_input_name, X_scaler, y_scaler,
prediction_cache, sequence_length,
```

```
                feature_history, tyre_inventory=None, track_status="GREEN",
sc_laps_remaining=0, dirty_air_laps=0):
    self.lap = lap
    self.max_laps = max_laps
    self.total_time = total_time

    self.current_compound = current_compound
    self.tyre_age = tyre_age
    self.compounds_used = set(compounds_used)
    self.fuel_load_proxy = fuel_load_proxy
    self.num_pit_stops = num_pit_stops
    self.stint_base_pace = stint_base_pace

    self.track_profile = track_profile

    self.track_status = track_status
    self.sc_laps_remaining = sc_laps_remaining

    self.model = model
    self.onnx_input_name = onnx_input_name
    self.X_scaler = X_scaler
    self.y_scaler = y_scaler
    if X_scaler is not None:
      self.x_min = X_scaler.min_
      self.x_scale = X_scaler.scale_
    self.prediction_cache = prediction_cache if prediction_cache is not
None else {}
    self.sequence_length = sequence_length
    self.feature_history = feature_history

    self.dirty_air_laps = dirty_air_laps
    if tyre_inventory is None:
      self.tyre_inventory = {
          "SOFT": 1,
          "MEDIUM": 2,
          "HARD": 1
      }
    else:
      self.tyre_inventory = tyre_inventory.copy()


  def get_legal_actions(self):
    if self.is_terminal(): return []

    actions = []
    unique_compounds = set(self.compounds_used)
    needs_mandatory_stop = len(unique_compounds) < 2

    cliff_limit = 35
    if self.current_compound == "SOFT": cliff_limit = 12
    elif self.current_compound == "MEDIUM": cliff_limit = 20
```

```
    elif self.current_compound == "HARD": cliff_limit = 30

    if self.tyre_age < cliff_limit: actions.append('STAY_OUT')

    min_tyre_age = 12
    is_slow_track = self.track_status in ["SC", "VSC"]

    available_pits = []
    if self.tyre_inventory.get("SOFT", 0) > 0:
available_pits.append('PIT_SOFT')
    if self.tyre_inventory.get("MEDIUM", 0) > 0:
available_pits.append('PIT_MEDIUM')
    if self.tyre_inventory.get("HARD", 0) > 0:
available_pits.append('PIT_HARD')

    can_pit = (self.num_pit_stops < 3) or needs_mandatory_stop
    laps_remaining = self.max_laps - self.lap
    min_stint_length = 8
    is_tyre_fresh = self.tyre_age < min_stint_length
    forced_to_pit_early = needs_mandatory_stop and (laps_remaining <=
min_stint_length)

    if self.lap < self.max_laps - 1:
      if (not is_tyre_fresh or forced_to_pit_early) and can_pit and
len(available_pits) > 0:
        if self.tyre_age >= min_tyre_age or is_slow_track or self.tyre_age
>= cliff_limit:
          if needs_mandatory_stop and self.lap > 30:
            if "SOFT" not in unique_compounds and "PIT_SOFT" in
available_pits: actions.append("PIT_SOFT")
            if "MEDIUM" not in unique_compounds and "PIT_MEDIUM" in
available_pits: actions.append("PIT_MEDIUM")
            if "HARD" not in unique_compounds and "PIT_HARD" in
available_pits: actions.append("PIT_HARD")
          else:
            actions.extend(available_pits)

    if needs_mandatory_stop and laps_remaining <= 3:
      if 'STAY_OUT' in actions:
        actions.remove('STAY_OUT')
      if len(actions) == 0:
        if "SOFT" not in unique_compounds and "PIT_SOFT" in available_pits:
actions.append("PIT_SOFT")
        if "MEDIUM" not in unique_compounds and "PIT_MEDIUM" in
available_pits: actions.append("PIT_MEDIUM")
        if "HARD" not in unique_compounds and "PIT_HARD" in available_pits:
actions.append("PIT_HARD")

    if len(actions) == 0:
      actions.append("STAY_OUT")
```

```
    if "STAY_OUT" in actions and self.num_pit_stops >= 1:
      return ["STAY_OUT"]

    return actions


  def get_next_state(self, action, is_simulation=False):
    next_status = self.track_status
    next_sc_laps = self.sc_laps_remaining
    next_dirty_air_laps = self.dirty_air_laps

    next_fuel_proxy = self.fuel_load_proxy - 1

    if action == "STAY_OUT":
      next_num_pit_stops = self.num_pit_stops
      next_tyre_age = self.tyre_age + 1
      next_compound = self.current_compound
      physical_comp =
self.track_profile["compound_map"][self.current_compound]
      next_compounds_used = set(self.compounds_used)
      next_stint_base_pace = self.stint_base_pace
      next_inventory = self.tyre_inventory.copy()

      current_features = self._build_feature_array(
          tyre_age=next_tyre_age,
          physical_comp=physical_comp,
          lap=self.lap + 1
      )

      if len(self.feature_history) >= self.sequence_length:
        next_history = self.feature_history[-(self.sequence_length - 1):] +
[current_features]
      else:
        next_history = self.feature_history + [current_features]

      if len(next_history) == self.sequence_length:
        history2d = np.array(next_history)
        cache_key = np.round(history2d, decimals=4).tobytes()
        if cache_key in self.prediction_cache:
          predicted_delta = self.prediction_cache[cache_key]
        else:
          scaled_history2d = (history2d * self.x_scale) + self.x_min
          input_tensor = np.array([scaled_history2d]).astype(np.float32)
          ort_inputs = {self.onnx_input_name: input_tensor}

          scaled_prediction = self.model.run(None, ort_inputs)[0]
          real_seconds_delta =
self.y_scaler.inverse_transform(scaled_prediction)[0][0]
          predicted_delta = real_seconds_delta
          self.prediction_cache[cache_key] = predicted_delta
      else:
```

```
        predicted_delta = 0.04 * next_tyre_age

      target_lap_time = next_stint_base_pace + predicted_delta
      lap_time = target_lap_time + (self.track_profile["fuel_coefficient"]
* next_fuel_proxy)
      if next_dirty_air_laps > 0:
        next_dirty_air_laps -= 1

      if next_dirty_air_laps > 0:
        traffic_penalty = np.random.uniform(0.5, 2.5)
        lap_time += traffic_penalty

    else:
      next_num_pit_stops = self.num_pit_stops + 1
      next_compound = action.split('_')[1]
      next_stint_base_pace =
self.track_profile["base_paces"][next_compound]
      next_compounds_used = set(self.compounds_used)
      next_compounds_used.add(next_compound)
      next_tyre_age = 1

      physical_comp = self.track_profile["compound_map"][next_compound]
      out_lap_features = self._build_feature_array(
          tyre_age=1,
          physical_comp=physical_comp,
          lap=self.lap + 1
      )

      next_history = [out_lap_features] * self.sequence_length

      next_inventory = self.tyre_inventory.copy()
      next_inventory[next_compound] -= 1

      if next_status == "SC" or next_status == "VSC":
        pit_loss = self.track_profile["pit_loss_sc"]
      else:
        traffic_tax = 6.0
        pit_loss = self.track_profile["pit_loss_green"] + traffic_tax

      history2d = np.array(next_history)
      cache_key = np.round(history2d, decimals=4).tobytes()
      if cache_key in self.prediction_cache:
        predicted_delta = self.prediction_cache[cache_key]
      else:
        scaled_history2d = (history2d * self.x_scale) + self.x_min
        input_tensor = np.array([scaled_history2d]).astype(np.float32)
        ort_inputs = {self.onnx_input_name: input_tensor}
        scaled_prediction = self.model.run(None, ort_inputs)[0]
        real_seconds_delta =
self.y_scaler.inverse_transform(scaled_prediction)[0][0]
        predicted_delta = real_seconds_delta
```

```
        self.prediction_cache[cache_key] = predicted_delta
      if next_num_pit_stops <= 2: risk_tax = 5.0
      else: risk_tax = 200.0
      pit_loss += risk_tax

      target_lap_time = next_stint_base_pace + pit_loss + predicted_delta
      lap_time = target_lap_time + (self.track_profile["fuel_coefficient"]
* next_fuel_proxy)

      if self.lap < 25: next_dirty_air_laps = 3
      if next_dirty_air_laps > 0:
        traffic_penalty = np.random.uniform(0.5, 2.5)
        lap_time += traffic_penalty

      cliff_limit = 35
      if self.current_compound == "SOFT": cliff_limit = 12
      elif self.current_compound == "MEDIUM": cliff_limit = 20
      elif self.current_compound == "HARD": cliff_limit = 30

      if self.tyre_age > cliff_limit:
        laps_over_limit = self.tyre_age - cliff_limit
        cliff_penalty = 0.5 * (laps_over_limit ** 2)
        safe_cliff_penalty = min(cliff_penalty, 20.0)
        lap_time += cliff_penalty

    if self.sc_laps_remaining > 0:
      next_sc_laps -= 1
      if next_sc_laps == 0:
        next_status = "GREEN"
    elif next_status == "GREEN":
      if not is_simulation:
        roll = random.random()
        if roll < self.track_profile["p_sc_trigger"]:
          next_status = "SC"
          next_sc_laps = random.choices(
            population=self.track_profile["sc_duration_laps"],
            weights=self.track_profile["sc_duration_weights"],
            k=1
          )[0]
        elif roll < (self.track_profile["p_sc_trigger"] +
self.track_profile["p_vsc_trigger"]):
          next_status = "VSC"
          next_sc_laps = random.randint(1, 2)

    if next_status == "SC": lap_time *= 1.45
    elif next_status == "VSC": lap_time *= 1.3

    return RaceState(
        lap=self.lap + 1,
        max_laps=self.max_laps,
        total_time=self.total_time + lap_time,
```

```
            current_compound=next_compound,
            tyre_age=next_tyre_age,
            compounds_used=next_compounds_used,
            track_profile=self.track_profile,
            fuel_load_proxy=next_fuel_proxy,
            num_pit_stops=next_num_pit_stops,
            stint_base_pace=next_stint_base_pace,
            model=self.model,
            onnx_input_name=self.onnx_input_name,
            X_scaler=self.X_scaler,
            y_scaler=self.y_scaler,
            prediction_cache=self.prediction_cache,
            sequence_length=self.sequence_length,
            feature_history=next_history,
            track_status=next_status,
            sc_laps_remaining=next_sc_laps,
            tyre_inventory=next_inventory,
            dirty_air_laps=next_dirty_air_laps
        )

    def _build_feature_array(self, tyre_age, physical_comp, lap):
        current_grip = (self.track_profile["track_evolution"] / 5.0) *
np.log(lap + 1)
        current_fuel_proxy = self.max_laps - lap
        current_softness = self.track_profile["softness_scores"][physical_comp]
        current_thermal_risk = self.track_profile["asphalt_abrasion"] *
current_softness
        features = {
            'TyreLife': tyre_age,
            'TyreLife_Squared': tyre_age ** 2,
            'TrackTemp': self.track_profile["track_temp"],
            'Grip_Evolution_State': current_grip,
            'Longitudinal_Energy': self.track_profile["longitudinal_energy"],
            'Lateral': self.track_profile["lateral"],
            'Thermal_Degradation_Risk': current_thermal_risk,
            'Slip_Ratio': self.track_profile["slip_ratio"],
            'Asphalt Grip': self.track_profile["asphalt_grip"],
            'Comp_C1': 0.0, 'Comp_C2': 0.0, 'Comp_C3': 0.0, 'Comp_C4': 0.0,
'Comp_C5': 0.0, 'Comp_C6': 0.0
        }
        features[f'Comp_{physical_comp}'] = 1.0
        ordered_keys = self.track_profile["ordered_feature_keys"]
        return [features[k] for k in ordered_keys]

    def is_terminal(self):
        return self.lap > self.max_laps

    def get_reward(self):
        penalty = 0.0

        unique_compounds = set(self.compounds_used)
```

```
    if self.is_terminal() and len(unique_compounds) < 2:
      penalty = 100.0

    return -(self.total_time + penalty)

class Node:
  def __init__(self, state, parent=None, action_taken=None):
    self.state = state
    self.parent = parent
    self.action_taken = action_taken
    self.children = []
    self.untried_actions = state.get_legal_actions()
    random.shuffle(self.untried_actions)
    self.visits = 0
    self.value = 0.0
    self.max_value = float('-inf')

  def is_fully_expanded(self):
    return len(self.untried_actions) == 0

  def best_child(self, c=50.0):
    choices_weights = []
    for child in self.children:
      exploit = child.max_value
      explore = c * np.sqrt((2 * np.log(self.visits) / child.visits))
      score = exploit + explore
      choices_weights.append(score)

    return self.children[np.argmax(choices_weights)]

class Strategist:
  def __init__(self, iterations=1000):
    self.iterations = iterations

  def simulate_random_playout(self, state):
    current_state = state

    while not current_state.is_terminal():
      actions = current_state.get_legal_actions()

      if "STAY_OUT" in actions:
        chosen_action = "STAY_OUT"
      elif "PIT_HARD" in actions:
        chosen_action = "PIT_HARD"
      elif "PIT_MEDIUM" in actions:
        chosen_action = "PIT_MEDIUM"
      elif "PIT_SOFT" in actions:
        chosen_action = "PIT_SOFT"
      else:
        chosen_action = actions[0]
```

```
      current_state = current_state.get_next_state(chosen_action,
is_simulation=True)

    return current_state.get_reward()

  def get_best_action(self, initial_state):
    root = Node(initial_state)

    for _ in range(self.iterations):
      node = root
      while not node.state.is_terminal() and node.is_fully_expanded():
        node = node.best_child()

      if not node.state.is_terminal() and not node.is_fully_expanded():
        action = node.untried_actions.pop()
        next_state = node.state.get_next_state(action, is_simulation=True)
        child_node = Node(next_state, parent=node, action_taken=action)
        node.children.append(child_node)
        node = child_node

      reward = self.simulate_random_playout(node.state)

      while node is not None:
        node.visits += 1
        if reward > node.max_value:
          node.max_value = reward
        node = node.parent

    best_node = max(root.children, key=lambda c: c.max_value)
    return best_node.action_taken


class StrategyEnv(gym.Env):
  metadata = {'render.modes': ['console']}

  def __init__(self, model, X_scaler, y_scaler, track_consts):
    super(StrategyEnv, self).__init__()

    self.model = model
    self.fast_predict = tf.function(self.model)
    self.y_scaler = y_scaler
    self.X_scale = X_scaler.scale_
    self.y_scale = y_scaler.scale_
    self.X_min = X_scaler.min_
    self.y_min = y_scaler.min_
    self.track = track_consts
    self.max_laps = self.track['max_laps']

    self.action_space = spaces.Discrete(4)
    self.observation_space = spaces.Box(
        low=0.0,
```

```
        high=1.0,
        shape=(13,),
        dtype=np.float32
    )


def reset(self, seed=None, options=None):
    super().reset(seed=seed)
    self.current_lap = 1
    self.tyre_life = 0
    if options is not None and 'starting_compound' in options:
      self.current_compound = options['starting_compound']
    else:
      self.current_compound = np.random.choice([0, 1, 2])
    self.compounds_used = {self.current_compound}
    self.inventory = self.track['tyre_inventory'].copy()
    self.inventory[self.current_compound] -= 1
    self.pit_stop_count = 0
    self.dirty_air_laps = 0
    self.total_time = 0.0
    self.feature_history = deque(maxlen=10)
    initial_features = self._build_feature_array()
    for _ in range(10):
      self.feature_history.append(initial_features)
    self.clean_race = False
    if options is not None and options.get('clean_race') == True:
      self.clean_race = True
    self.sc_laps = 0
    self.vsc_laps = 0
    observation = self._get_obs()
    info = {}
    return observation, info


def _build_feature_array(self):
    physical_comp = self.track['compound_mapping'][self.current_compound]
    softness_score = self.track['softness_scores'][physical_comp]
    thermal_risk = self.track['Asphalt_Abrasion'] * softness_score
    track_evolution = self.track['Track_Evolution']
    grip_state = (track_evolution / 5.0) * np.log(self.current_lap + 1)

    comp_cols = [1.0 if i == physical_comp else 0.0 for i in range(1, 7)]
    features = [
        self.tyre_life,
        self.tyre_life ** 2,
        self.track['TrackTemp'],
        grip_state,
        self.track['Longitudinal_Energy'],
        self.track['Lateral'],
        thermal_risk,
        self.track['Slip_Ratio'],
```

```
            self.track['Asphalt Grip']
        ]

        return features + comp_cols


    def _get_obs(self):
        expected_lap_time = self._predict_lap_time()

        is_soft = 1.0 if self.current_compound == 0 else 0.0
        is_medium = 1.0 if self.current_compound == 1 else 0.0
        is_hard = 1.0 if self.current_compound == 2 else 0.0

        return np.array([
            self.current_lap / self.max_laps,
            self.tyre_life / self.max_laps,
            is_soft,
            is_medium,
            is_hard,
            min(expected_lap_time / 200.0, 1.0),
            len(self.compounds_used) / 3.0,
            self.dirty_air_laps / 3.0,
            self.inventory[0] / max(1,
sum(self.track['tyre_inventory'].values())),
            self.inventory[1] / max(1,
sum(self.track['tyre_inventory'].values())),
            self.inventory[2] / max(1,
sum(self.track['tyre_inventory'].values())),
            1.0 if self.sc_laps > 0 else 0.0,
            1.0 if self.vsc_laps > 0 else 0.0
            ], dtype=np.float32)


    def _predict_lap_time(self):
        current_features = self._build_feature_array()
        self.feature_history.append(current_features)
        sequence_array = np.array(self.feature_history)
        scaled_features = (sequence_array * self.X_scale) + self.X_min
        input = np.reshape(scaled_features, (1, 10, scaled_features.shape[1]))

        with tf.device('/CPU:0'):
            scaled_pred = self.fast_predict(input).numpy()
        delta = (scaled_pred[0][0] - self.y_min[0]) / self.y_scale[0]

        fuel_correction = self.track['Fuel_Coefficient'] * (self.max_laps -
self.current_lap)
        lap_time = self.track['base_lap_time'][self.current_compound] + delta +
fuel_correction

        if self.dirty_air_laps > 0:
            traffic_penalty = np.random.uniform(0.5, 2.5)
```

```
    lap_time += traffic_penalty

  return lap_time


def step(self, action):
  pit_action = False
  earned_reward = False
  invalid_action_penalty = 0.0
  was_legal = len(self.compounds_used) >= 2

  if self.sc_laps == 0 and self.vsc_laps == 0 and not self.clean_race:
    crash_roll = np.random.random()
    if crash_roll < self.track['sc_prob']:
      self.sc_laps = np.random.randint(3, 6)
    elif crash_roll < self.track['sc_prob'] + self.track['vsc_prob']:
      self.vsc_laps = np.random.randint(1, 3)

  track_status = 'green'
  if self.sc_laps > 0:
    track_status = 'sc'
    self.sc_laps -= 1
  elif self.vsc_laps > 0:
    track_status = 'vsc'
    self.vsc_laps -= 1

  pit_stop_loss = self.track['pit_loss'][track_status]

  action_val = int(np.squeeze(action))

  if action_val != 0:
    target_compound = action_val - 1
    if self.pit_stop_count >= 3 or self.inventory[target_compound] <= 0:
      invalid_action_penalty = 50.0
    else:
      pit_action = True
      self.pit_stop_count += 1
      self.inventory[target_compound] -= 1
      self.current_compound = target_compound
      self.compounds_used.add(self.current_compound)
      self.tyre_life = 0
      self.has_pitted = True

      if self.current_lap < 25:
        self.dirty_air_laps = 3
  if not pit_action:
    self.tyre_life += 1
    if self.dirty_air_laps > 0:
      self.dirty_air_laps -= 1

  lap_time = self._predict_lap_time()
```

```
    if pit_action:
      lap_time += pit_stop_loss

    max_safe_laps = self.track['cliff_laps'][self.current_compound]
    if self.tyre_life > max_safe_laps:
      laps_over_limit = self.tyre_life - max_safe_laps
      cliff_penalty = 0.5 * (laps_over_limit ** 2)
      safe_cliff_penalty = min(cliff_penalty, 20.0)
      lap_time += cliff_penalty

    reward = (float(-lap_time) - invalid_action_penalty) / 100.0

    if track_status != 'green':
      lap_time *= self.track['pace_modifier'][track_status]

    self.total_time += lap_time

    is_legal = len(self.compounds_used) >= 2
    if not was_legal and is_legal:
      reward += 1.0

    self.current_lap += 1

    terminated = bool(self.current_lap > self.max_laps)
    truncated = False
    if terminated and not is_legal:
      reward -= 10.0

    observation = self._get_obs()
    info = {"lap_time": lap_time, "total_time": self.total_time,
"track_status": track_status}
    return observation, reward, terminated, truncated, info


env = StrategyEnv(model=lstm_model, X_scaler=X_scaler, y_scaler=y_scaler,
track_consts=bahrain_const)
policy_kwargs = dict(activation_fn=th.nn.ReLU, net_arch=[256, 256])
model = PPO("MlpPolicy", env, verbose=1, learning_rate=scheduler(0.0003),
ent_coef=0.05, gamma=0.99, policy_kwargs=policy_kwargs)
model.learn(total_timesteps=5000000)
```

ДОДАТОК В

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	102 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Гітіс І.В.			Дослідження методів штучного інтелекту для формування перегонової стратегії в умовах стохастичної невизначеності	Шифр групи	Код напр./спец.
Перевірив		Філатов В.О.				СШІм-24-2	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Чала Л.Е.					