Virtual Football Simulations for Predictive Analysis

Patryk Pilichowski
Department of Computer Science
University of Antwerp
Antwerp, Belgium
patryk.pilichowski@student.uantwerpen.be

Chloë Mansibang
Department of Computer Science
University of Antwerp
Antwerp, Belgium
chloe.mansibang@student.uantwerpen.be

***Abstract*—This paper explores methods for virtually simulating football matches to predict possible outcomes, and, potentially manipulate these outcomes. The focus is on identifying computational and data-driven strategies that can generate reliable, actionable simulations. We review several methods, evaluating each for its predictive capability, computational requirements, and viability. Our analysis concludes that the gated recurrent unit (GRU) model is the most suitable due to its ability to process sequential data, capture long-term dependencies, and mitigate the *vanishing gradient problem* while maintaining computational efficiency.**

I. Introduction

Simulating football matches can be significant for various applications, from enhancing team tactics to betting on the team that is most likely to win. The aim is to examine 6 different computational models and evaluate them on their suitability to simulate football matches for predictive analysis and their feasibility of implementation:

1) Monte Carlo method
2) Hidden Markov models (HMMs)
3) Generative adversarial networks (GANs)
4) Recurrent neural networks (RNNs)
5) Reinforcement learning
6) Agent-based modeling

II. Monte Carlo Method

A. Definition

The Monte Carlo method is a stochastic approach that uses repeating random sampling to obtain numerical results [1]. We can formulate our problem as two teams A and B playing a match represented as a sequence of discrete events over time (e.g passes, shots, goals, fouls). Each event e_i has a certain probability of occuring based on historical data, player/team abilities, etc.

Let $t \in [0, T]$ represent the match time, where T is the total match (e.g 90 minutes). In addition X_t represents the state of the match at time t, where each state includes information like current score, possession, player positions.

Each event e_i can be assigned a probability distribution. Let's focus on goal-scoring events, for instance. The *double Poisson distribution* may be an effective choice for predicting this [2]. However, for simplicity, we will use the basic Poisson distribution. Let λ_A and λ_B be the goal-scoring rate for team A and B respectively. Then, the probability that team A scores k goals in time T is given by:

$$P(\text{Goals}_A = k) = \frac{(\lambda_A T)^k e^{-\lambda_A T}}{k!}$$

The probability for team B is analogous.

B. Sampling

A simulation run r involves generating random samples based on the probability distributions. Let G_A^r be the number of goals scored by team A in run r. For each run, we sample G_A^r from the distribution $\text{Poisson}(\lambda_A, T)$ and G_B^r from an analogous distribution.

Each outcome of the simulation r is then:

$$\text{Outcome}_r = \begin{cases} \text{Win for team A}, & G_A^r > G_B^r \\ \text{Win for team B}, & G_A^r < G_B^r \\ \text{Draw}, & G_A^r = G_B^r \end{cases}$$

C. Repeated Sampling

Now we run the simulation N times, where N is large (e.g $N = 10000$). This yields:

- A set of goal outcomes for team A: $\{G_A^1, ..., G_A^N\}$
- A set of goal outcomes for team B: $\{G_B^1, ..., G_B^N\}$

Define:

$$W_A = \frac{1}{N} \cdot |\{G_A^i | G_A^i > G_B^i, i \in \{1, ..., N\}\}|$$
$$W_B = \frac{1}{N} \cdot |\{G_B^i | G_B^i > G_A^i, i \in \{1, ..., N\}\}|$$
$$D = \frac{1}{N} \cdot |\{G_A^i | G_A^i = G_B^i, i \in \{1, ..., N\}\}|$$

D. Aggregating Results

After N runs, we approximate the probabilities of each possible outcome as:

$$P(\text{Team A wins}) \approx W_A$$
$$P(\text{Team B wins}) \approx W_B$$
$$P(\text{Draw}) \approx D$$

E. Implementation

Python can be used with statistical libraries `NumPy`, `Pandas`, and `SciPy` for repeated sampling and different probability distributions.

F. Strengths & Limitations

A Monte Carlo simulation is easy to implement due to its model simplicity. It is also scalable as running more simulations may improve robustness of its predictions.

On the other hand, the model is dependent on accurate data to create fitting probability distributions. Poor-quality data, or lack thereof, may lead to nonsensical predictions. More critically however, the model oversimplifies the complex nature of football. It lacks nuanced analyses because it cannot account for real-time tactical changes or momentum shifts. Consequently, while useful for broad predictions, Monte Carlo may be too simplistic for accurately simulating a sport as intricate as football.

III. Hidden Markov Model

A. Definition

A Markov model is a stochastic model that assumes the Markov property: future states depend only on the current state and not on states that occurred before. In statistics and probability this is also called ”memorylessness”.

A hidden Markov model is a Markov model where there are observable events that depend on underlying, hidden Markov chains. We will base the definition on [3]:

Let HMM $= (N, M, A, B, \pi)$ such that:

$Q = \{q_1, q_2, ..., q_N\}$ a set of hidden states, where $Q_t \in Q$ is state at time t

$V = \{v_1, v_2, ..., v_M\}$ the set of all possible observations, also called the vocabulary set

$A = a_{1,1}...a_{i,j}...a_{N,N}$ transition probability matrix, where $a_{i,j}$ represents the probability of moving from state i to state j: $a_{i,j} = P(Q_{t+1} = q_j | Q_t = q_i)$

$\pi = \pi_1, \pi_2, ..., \pi_N$ initial probability distribution vector, where π_i is the probability that the Markov chain will start in state i: $\pi_i = P(Q_1 = q_i)$

$B = b_{1,1}...b_{j,k}...b_{N,M}$ emission probability matrix, where $b_{j,k}$ represents the probability of observation v_k being generated from state q_j: $b_{j,k} = P(O_t = v_k | Q_t = q_j)$

B. Simulation

Choose the number of observations T that are to be generated, then we can generate the sequence of observations with the following algorithm:

Algorithm 1 Generating observations

```
Choose $Q_1 = q_i$ according to $\pi_i$
Set $t = 1$
**repeat**
    Choose $O_t = v_k$ according to $b_{i,k}$
    Transit to a new state $Q_{t+1} = q_j$ according to $a_{i,j}$
**until** $t > T$
```

We now have the sequence of observations $o_1 o_2 ... o_T$ with $o_i \in V$

C. Example

A concrete football example of a discrete hidden Markov model can be shown with a single graph:

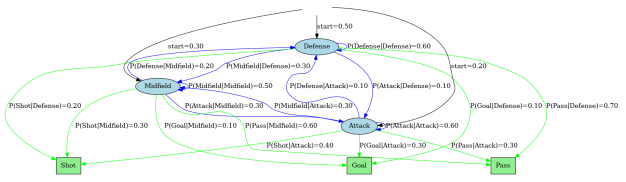

Fig. 1. DHMM Football Example

The blue, circular nodes represent the hidden states, while the green, box-shaped nodes correspond to the possible observations. Edges from blue nodes to green nodes indicate elements of the emission probability matrix, whereas edges between blue nodes represent elements of the transition probability matrix. The transition and emission probabilities would be chosen based on the available historical data of teams and players.

Note that this example is very simple, and the possible observations can, for instance, be extended to all 11 positions (left fullback, center back, goalkeeper, etc.) per team as well as other events. Additionally, the hidden states can be extended to the different tactics in use (e.g 4-4-2, tiki-taka, etc.). This could make the simulations more nuanced and the predictions of observations more accurate.

D. Inference

Given the parameters of the model, we can apply the forward algorithm to calculate the probability of a particular sequence of observations $P(o_1 ... o_T)$.

On the other hand, the Viterbi algorithm can be used to determine the most probable sequence of hidden states that would have generated that observed sequence.

This can be useful in football for calculating probabilities of specific sequences of events occurring during a match.

E. Generalizing The Model

Hidden Markov models can be generalized to allow for continuous observations using continuous probability distributions, such as the normal distribution. If we represent the observation at time t as O_t and the hidden state as S_t, then:

$$O_t | Q_t = q \sim \mathcal{N}(\mu_q, \sigma_q^2)$$

Continuous observations in the model enable the simulation of more events in football, such as the trajectories of the ball and players, their speed, etc.

The model can be generalized even further by allowing the hidden states themselves to exist in a continuous state space. However, such a model requires more sophisticated inference techniques like Kalman filtering or particle filtering [4]. As a result, we will not delve into this generalization here.

F. Implementation

Python can be used with the combination of libraries `scikit-learn`, `hmmlearn`, and `NumPy`.The `hmmlearn` library supports HMMs with various emission probabilities, including discrete and continuous types.

G. Strengths & Limitations

Hidden Markov models may be useful for recognizing recurring patterns that precede key events (e.g goals or fouls). However, the assumption of Markov property, that each state depends only on the previous one, is perhaps the strongest limitation with this model. Realistically, there are significantly more dependencies in a match of football, which this model inherently oversimplifies.

IV. Generative Adversarial Network

A. Definition

A GAN can be defined as a two-player, minimax game. Let's define the components of GAN formally according to [5], [6]:

- $(\Omega, \mu_{\text{ref}})$ probability space, with Ω the sample space and μ_{ref} the reference distribution over Ω (the distribution with real data, such as actual football tracking and event data).
- The generator G, a neural network which takes random noise as input and generates synthetic data samples according to the distribution μ_G, it's task is to achieve $\mu_G \approx \mu_{\text{ref}}$ and ultimately "fool" the discriminator D that the samples come from μ_{ref}
- The discriminator D, a neural network that evaluates whether a given data sample comes from μ_{ref} or μ_{G} and produces output $y \sim \mu_D(x)$, with y close to 1 if input seems to come from μ_{ref}, otherwise close to 0.
- $\underset{G}{\min}\underset{D}{\max} L(\mu_G, \mu_D) = \left(\mathbb{E}_{x\sim\mu_{\text{ref}}, y\sim\mu_D(x)}\left[\ln(y)\right] + \mathbb{E}_{x\sim\mu_{\text{G}}, y\sim\mu_D(x)}\left[\ln(1-y)\right]\right)$

  is the objective function which the generator wants to minimize and the discriminator wants to maximize

The generator and discriminator are engaged in a feedback loop where the generator improves its output to better fool the discriminator, while the discriminator gets better at distinguishing real from fake samples.

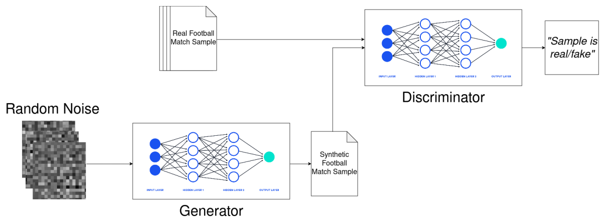

Fig. 2. GAN high-level football example

B. Implementation

`PyTorch` and `TensorFlow` are the most popular tools because of their extensive support for machine learning applications. `Pytorch` is favored for its flexiblity, ease of debugging, and small-scale projects, while `TensorFlow` is considered better for large-scale projects with high-performance and scalable requirements.

C. Strengths & Limitations

General adversarial networks are able to produce high-quality synthetic data, such as realistic simulated football match data. This can be helpful for teams to analyze opponent strategies and prepare for them. A large limitation, however, is that GANs require a large amount of data in order to actually produce high-quality results. Additionally, due to the feedback loop between the generator and the discriminator, training GANs can be unstable and slow [18].

V. Recurrent Neural Networks

A. Definition

An RNN is a neural network designed for processing sequential data across multiple time steps. The main difference between RNNs and traditional neural networks is that RNNs can access and process information from previous inputs while the traditional nets have input that is independent from each other.

There are many variants of RNNs, here we focus for the most part on the basic, abstract definition based on [8].

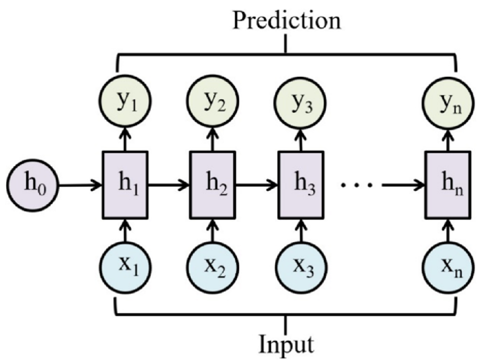

Fig. 3. Basic RNN [8]

At each time step $t \in \{1, ..., n\}$, the RNN takes an input vector $x_t \in \mathbb{R}^k$ and updates its hidden layer vector $h_t \in \mathbb{R}^m$ using the following equation:

$$h_t = \sigma_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

With:

- $W_{xh} \in \mathbb{R}^{m\times k}$ the weight matrix between the input and the hidden layer
- $W_{hh} \in \mathbb{R}^{m\times m}$ the weight matrix between the hidden layers, also called the recurrent connection
- $b_h \in \mathbb{R}^m$ the bias vector for the hidden state

- σ_h the activation function

Additionally the output is also computed at each time step t using:

$$y_t = \sigma_y(W_{hy}h_t + b_y)$$

Where:

- $W_{hy} \in \mathbb{R}^{k\times m}$ the weight matrix between the hidden and the output layer
- $b_y \in \mathbb{R}^k$ the bias vector for the output
- σ_y the activation function for the output layer

Activation functions that are most commonly used are [7]:

- Sigmoid function $\sigma(x) = \dfrac{1}{1+e^{-x}}$, used when its output needs to be interpreted as a probability. It can be used in the hidden or the output layer. However it suffers from the *vanishing gradient problem*.
- $\sigma(x) = \tanh(x) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$ used for the hidden layer. It centers its output to 0 by defining its output range $[-1, 1]$. For many problems it is chosen over the sigmoid because of the broader output range. However, it still may suffer from the *vanishing gradient problem*.
- ReLU function $\sigma(x) = \max(0, x)$ used for the hidden layer. It is less susceptible to the *vanishing gradient problem*. However it can suffer from the *dying ReLU problem* (0 is returned for each negative input).
- Softmax function $\sigma(\vec{x})_i = \dfrac{e^{x_i}}{\sum_j e^{x_j}}$, used in the output layer where it converts raw data into probabilities. Primarily used for multi-class classification problems.

For our problem, $\tanh$ may be a suitable activation function for the hidden layer because it supports negative inputs. Furthermore, $\mathrm{softmax}$ would be the most suitable output-layer activation function for predicting the likelihood of discrete events (e.g next pass, shot, goal, interception) in our problem.

B. Vanishing Gradient Problem

The main challenge with a standard RNN that uses the *backpropagation through time* (BPTT) algorithm is the *vanishing gradient problem*. During each training iteration, the weights are updated proportional to the gradient, which is the vector of partial derivatives of the loss function (a function that measures model performance) with respect to the weights. As this loss is propagated backward in time, the gradient diminishes exponentially due to repeated multiplication by weight matrices and activation function derivatives. This issue becomes more severe with longer input sequences, and consequently the network struggles to learn long-term dependencies.

Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRU) are RNNs specifically designed to mitigate this problem by using multiple gates that regulate flow of information. Such architectures are able to choose to either remember or discard information over time, which allows for more stable gradients.

C. Implementation

`PyTorch` and `TensorFlow` are again the most popular choices for the same reasons as stated in section IV-B. Additionally, `Keras` is a Python library specialized in artificial neural networks. See [9] for actual implementation examples.

D. Strengths & Limitations

RNNs are effective models for learning sequential data which makes it suitable for modeling football matches for predictions. However they may struggle with interpretability and depend on abundant training data to avoid overfitting. Furthermore, RNNs may also be computationally expensive to train.

It is not always clear when to choose LSTM and GRU. GRUs are preferred for short input sequences because they are computationally more efficient. However, LSTMs may be better at capturing long-term dependencies. See [19] as it explores this question in detail.

Using GRU over LSTM may be desirable for our goal as it is computationally efficient without sacrificing the general quality of the results.

VI. Reinforcement Learning

The following overview on Reinforcement Learning is based on the work of Richard S. Sutton and Andrew G. Barto [10].

A. Definition

Reinforcement learning is a machine learning technique where an agent learns to make decisions in a certain environment to maximize its reward signal. The environment consists of everything the agent interacts with outside itself. The environment would be modeled as a Markov Decision Process (MDP):

$$(S, A, P, R, \lambda)$$

With:

- S: the state space, which consists of player positions, ball location and game context
- A: the action space, which consists all of the movements that players are allowed to perform
- P: the transition probabilities from one state to another
- R: the expected rewards
- λ: the discount for future rewards

The transition probabilities to a next-state given a state and an action can be computed by:

$$\begin{aligned} p(s'|s,a) &= Pr\{S_{t+1} = s' | S_t = s, A_t = a\} \\ &= \sum_r p(s', r|s,a). \end{aligned}$$

And the expected reward given a state, action and next-state can be computed by:

$$\begin{aligned} r(s,a,s') &= \mathbb{E}[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s'] \\ &= \sum_{r\in\mathcal{R}} r p(s', r|s,a) \end{aligned}$$

Besides the agent and the environment, a reinforcement learning system has four key components: a policy, a reward signal, a value function and a model of the environment The policy maps a state to an action. It describes the behavior of an agent given the current state of the game. A multi-agent policy should be used if there are multiple agents that interact with teammates and opponents. A stochastic policy λ represents, given a state, the probability that an action will be performed, is modeled as:

$$\pi(a|s) = P(A_t = a|S_t = s)$$

The goal of the agent is to maximize the cumulative reward in the long run, not immediate reward. So we want to maximize the expected return G_t, computed by:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Where t is the time step and λ is the discount rate $(0 \leq \lambda \leq 1)$.

The state-value function quantifies how good it is to be in a certain state or how good it is to perform a certain action in the long run. The value of a state s, given a certain policy π, is denoted as $v_\pi(s)$. For MDPs, the value can be computed by the state-value function for policy π:

$$v_\pi(s) = E_\pi[G_t|S_t = s] \quad = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right]$$

Where $\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable given a policy π and time step t. We can also assign a value to an action given a state and a policy.

The action-value function quantifies how good it is to take a certain action, given a state and a policy. The value of action a, given a certain policy λ, is denoted as $q_\pi(s, a)$. For MDPs, the value can be computed by the action-value function for policy λ:

$$\begin{aligned} q_\pi(s, a) &= E_\pi[G_t|S_t = s, A_t = a] \\ &= E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \end{aligned}$$

We can compute the optimal state-value function and action-value functions as:

$$v^*(s) = \max_v \pi(s),$$

$$q^*(s, a) = \max_\pi q_\pi(s, a),$$

for all $s \in S$ and $a \in A(s)$.

These optimal functions are linked by writing q* in terms of v*: $q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1})|S_t = s, A_t = a]$

The Bellman optimality equation expresses that given the optimal policy, the value of a state is equal to the expected return for the best action that can be performed from that state. The Bellman optimality equations for v* and q* are denoted as:

$$\begin{aligned} v^*(s) &= \max_a \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s, A_t = a\right] \end{aligned}$$

$$\begin{aligned} q^*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q^*(s', a')\right] \end{aligned}$$

B. Learning Algorithms

Q-learning is a value-based off-policy method. The agent learns a policy based on experiences it has by following a different policy, with the goal of updating or optimizing the policy by using the Q-learning update rule:

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha h R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ Where:

- s_t: state at time t
- a_t: action taken at time t
- R_{t+1}: reward after performing action a_t
- s_{t+1}: next state after performing action a_t
- a': all actions that can be performed in the next state
- γ: discount

State-Action-Reward-State-Action (SARSA) is a value based on-policy method. This method updates or optimizes a policy by updating the Q-value based on actions it takes while following the policy it wants to update or optimize: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha h R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$ Where:

- s_t: current state
- a_t: current action
- R_{t+1}: reward
- s_{t+1}: next state
- a_{t+1}: action performed in the next state (according to the policy)
- γ: discount

C. Implementation

Python can be used. rSoccer [11] and OpenAI Gym are also very helpful tools for implementation.

D. Strengths & Limitations

Reinforcement learning is useful for exploring different solutions for a game and gaining insights in new strategies.

Despite this big advantage, some limitations must be considered. Training RL models requires a high number of resources, time and data. Training also impacts learning, which makes defining an effective reward function a crucial step. It can be a challenge due to the complexity of balancing short-term and long-term rewards.

VII. Agent-Based Modeling

A. Definition

Agent-Based modeling (ABM) is a computational method that is used to simulate how agents interact with other agents in a certain environment, and how they interact with the environment according to defined rules. The agents are distinct entities with different characteristics and behavior, meaning that each agent makes decisions differently. [21]

In ABMs, there are two main components [21]: agents and environment. The agents are distinct entities with different characteristics and behavior, meaning that each agent makes decisions differently. There can be one or more populations of agents. The environment is a virtual world where the agents interact with each other. It can have zero effect on the agents, or the environment can have defined rules.

The agents are usually described as entities with four key attributes [21]:

- Perception: agents know what's happening around them
- Performance: agents can perform certain actions within their environment
- Memory: agents can store their past actions
- Policy: agents have defined rules that affect their decision making that are based on past states and the current state.

B. Computational Models

The following models can be used for the decision-making of a player agent: expected goals (xG) Model and Pitch Control Model [13].

The expected goals model is a statistical model that quantifies the probability that a shot will result in a goal.

The pitch control model is used for off-the-ball actions that affect offensive and defensive play. It indicates how the different teams control different areas of the field.

C. Implementation

Python can be used alongside the package AgentPy. It's a package for agent-based modeling in Python. It's highly flexible and allows for unique attribute assignments. The framework can be expanded more by adding interactions based on player skills, fatigue and team strategies.

D. Strengths & Limitations

Agent-based modelling allows us to model individual players with different behaviours and skill sets. This is useful for simulating realistic team dynamics and interactions between players. However, highly detailed and numerous interactions would ask for a large amount of computational power.

It's also important to consider the lack of a learning component. The agents don't learn from experiences over time, unless we pair it with a learning framework.

VIII. Model Evaluation & Comparison

In this section we will evaluate and compare the six models based on four criteria we believe are important for the choice of the suitable model. Furthermore, each criterion c_i has a specific weight w_i to denote its level of importance, with $c_i, w_i \in \{1, 2, 3\}$:

- Accuracy: The quality and reliability of the model's predictions. We want the model to be as accurate as possible.
- Feasibility: How convenient it is to implement the model. Ease of implementation is preferred, however we accept a reasonable challenge.
- Flexibility: The model's ability to perform effectively regardless of the abundance or quality of datasets. We wish for a balance between a model's reliance on data and its accuracy even when the data is less abundant or noisy
- Efficiency: The amount of computation required for training or running the model. We accept a reasonable degree of computational complexity.

Now let's assume the following values for each model based on the information in the previous sections:

Table	**Evaluation**			
Method	***Accuracy***	***Feasibility***	***Flexibility***	***Efficiency***
Weight	3	2	2	2
MC	1	3	2	1
HMMs	2	2	1	2
GANs	3	2	1	1
RNNs	3	2	2	2
RL	3	2	2	1
ABM	2	2	2	1

The maximizer for $\sum_{i=1}^{4} c_i w_i$ in this particular setting of weights is RNNs. Furthermore, it appears that RNNs is also the unique maximizer if weights are ignored completely. Now let's justify the values that have been picked for the criteria more in detail:

- Monte Carlo method
  - Low Accuracy: Fails to capture the complex nature of football to reliably be able to predict match outcomes.
  - High Feasibility: Convenient to implement the model due to its simplicity.
  - Moderate Flexibility: The model is generally dependent on historical data of different teams and players to create fitting probability distributions. However, increasing the number of iterations may compensate for the missing or poor-quality data.
  - Low Efficiency: The method may require many samples to reach more accurate approximations, especially in the context of simulation.
- Hidden Markov models
  - Moderate Accuracy: May be able to predict certain events, however the Markov property may be too strong of an assumption to make HMMs suitable as football's most effective model in terms of predictive accuracy.
  - Moderate Feasibility: The model is generally simple to implement, but modeling transitions based on

historical data and configuring the state-space can be challenging.
  - Low Flexibility: HMMs are heavily dependent on high-quality, abundant data to effectively estimate the transition probabilities between states.
  - Moderate Efficiency: Training and inference are computationally reasonable but scale poorly with large state-spaces.
- Generative adversarial networks
  - High Accuracy: GANs can create synthetic samples that are high quality because of the feedback loop between the two neural networks.
  - Moderate Feasibility: GANs are relatively convenient to implement with the existing tools but they are known to be very difficult to train [18].
  - Low Flexibility: An abundance of data is necessary for the generator to create realistic samples and avoid overfitting.
  - Low Efficiency: Due to the adversarial optimization, training GANs is known to be computationally expensive [18].
- Recurrent neural networks
  - High Accuracy: LSTM and GRU architectures are known to be effective RNNs because they capture long-term dependencies better than standard RNNs. [14].
  - Moderate Feasibility: The existing tools make the implementation convenient, however tuning the model may be challenging.
  - Moderate Flexibility: RNNs require high-quality data for optimal performance, but they can still function with smaller or noisier datasets.
  - Moderate Efficiency: The output of an RNN can be obtained in linear time [15], and the computational complexity per time step, which involves training and inference, is $\mathcal{O}(W)$ with W the number of weights in the network [16], [17]. GANs on the other hand intuitively have a greater computational complexity as it involves training two competing neural networks.
- Reinforcement learning
  - High Accuracy: There exist reinforcement learning models that are accurate in predictions of football matches [20].
  - Moderate Feasibility: Defining an environment, state-action spaces, and reward functions, can be challenging. However, available libraries and frameworks can make implementation more convenient.
  - Moderate Flexibility: While it benefits from abundant and high-quality data, it can still operate with less data by learning through simulations. Noisy data may require more training iterations to compensate.
  - Low Efficiency: It is computationally expensive due to iterative learning processes and numerous simulations.
- Agent-based modeling
  - Moderate Accuracy: It can capture individual player behaviors and interactions effectively. However, its accuracy is limited by the complexity and realism of the agents' rules and behaviors.
  - Moderate Feasibility: ABM requires designing detailed agents and rules for interactions, which may be labor-intensive.
  - Moderate Flexibility: Synthetic, randomly generated data may be used to compensate for the lack of data.
  - Low Efficiency: Simulating multi-agent interactions over a full football match is computationally expensive.

IX. Conclusion

According to our analysis, recurrent neural networks, particularly the gated recurrent unit, is the most promising model for simulating football matches in a way that yields realistic, actionable outcomes. Unlike Monte Carlo and Markov models, it does not oversimplify football simulations. Additionally it isn't as computationally expensive as GANs, agent-based modeling and reinforcement learning and mitigates the *vanishing gradient problem.*

However, we emphasize that this paper has limitations. It focused mainly on theoretical comparisons, without original implementations or empirical testing. Moreover, time constraints prevented deeper exploration of, for instance, hybrid models.

References

[1] Wikipedia contributors. (2024b, October 3). Monte Carlo method. Wikipedia.
https://en.wikipedia.org/wiki/Monte_Carlo_method

[2] Penn, M. J., & Donnelly, C. A. (2022). Analysis of a double Poisson model for predicting football results in Euro 2020. PloS One, 17(5), e0268511.
https://doi.org/10.1371/journal.pone.0268511

[3] Jurafsky, D., & Martin, J. H. (2000, Chapter A). Speech and Language processing: An introduction to natural language processing, computational linguistics, and speech recognition.

[4] Wikipedia contributors. (2024, September 23). Hidden Markov model. Wikipedia.
https://en.wikipedia.org/wiki/Hidden_Markov_model

[5] Wikipedia contributors. (2024c, October 19). Generative adversarial network. Wikipedia.
https://en.wikipedia.org/wiki/Generative_adversarial_network

[6] Goodfellow, I., et al. Generative Adversarial Nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, & K. Q. Weinberger (Eds.), Advances in Neural Information Processing Systems (Vol. 27)

[7] Brownlee, J. (2021, January 21). How to choose an activation function for deep learning. MachineLearningMastery.com.
www.machinelearningmastery.com/choose-an-activation-function-for-deep-learning/

[8] Mienye, I. D., Swart, T. G., & Obaido, G. (2024). Recurrent neural networks: A comprehensive review of architectures, variants, and applications. Information (Basel), 15(9), 517.
https://doi.org/10.3390/info15090517

[9] 10. Modern Recurrent Neural Networks — Dive into Deep Learning 1.0.3 documentation. (n.d.).
https://d2l.ai/chapter_recurrent-modern/index.html

[10] Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed.). MIT Press.

[11] Martins, F. B., Machado, M. G., Bassani, H. F., Braga, P. H. M., & Barros, E. S. (2021). RSoccer: A framework for studying reinforcement learning in small and very small size robot soccer. In arXiv [cs.LG].
https://doi.org/10.48550/ARXIV.2106.12895

[12] Armano Srbljinovic, & Ognjen Skunca. (2004). An Introduction to Agent Based Modelling and Simulation of Social Processes.
https://doi.org/10.48550/arXiv.cond-mat/0409312

[13] Alvarez-Bran, C.-J., & Cortes-Poza, Y. (2024). Simulation of counterattack plays in soccer using advanced agent-based modeling techniques.
https://doi.org/10.2139/ssrn.4983391

[14] Nyquist, R.A., & Pettersson, D. (2017). Football match prediction using deep learning.
https://api.semanticscholar.org/CorpusID:67298626

[15] Ian Goodfellow, Yoshua Bengio, & Aaron Courville (2016). Deep Learning. MIT Press.

[16] Graves, A. (2012). Supervised Sequence Labelling with Recurrent Neural Networks. In Studies in computational intelligence.
https://doi.org/10.1007/978-3-642-24797-2

[17] Sak, H., W, A., Senior, & Beaufays, F. (2014). Long Short-Term memory based recurrent neural network architectures for large vocabulary speech recognition.
https://doi.org/10.48550/arxiv.1402.1128

[18] Saxena, D., & Cao, J. (2020). Generative Adversarial Networks (GANS Survey): Challenges, solutions, and future Directions.
https://doi.org/10.48550/arxiv.2005.00065

[19] Chung, J., Gülçehre, Ç., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling.
https://doi.org/10.48550/arxiv.1412.3555

[20] Xu, N. C. (2024). Tactical intelligent decision modelling in sports competitions based on reinforcement learning algorithms.
https://doi.org/10.52783/jes.3124

[21] Salgado, M., & Gilbert, N. (2013). Agent Based Modelling.
https://doi.org/10.1007/978-94-6209-404-8_12