Brute-Forcing Evolution:
Optimizing Win Distributions with Iterative Weighted Sampling

Carrot Gaming
Declan Armstrong

1 Discrete Outcome Random Draw

In traditional online slot machines, the back-end code is responsible for generating results for the front-end in real time. The Carrot Gaming solution uses a different approach. This is often referred to as a 'lookup table' or pregenerated results approach, which uses a predefined set of outcomes stored in a database. The game result selection process is analogous to a replacement random draw.

Using the random draw approach, a large number of possible outcomes are generated (typically between 500,000 and 1,000,000) independently for each mode within a game. A database stores the instructions for all possible game rounds, along with a lookup table, which is a single file containing a simulation number, weight, and final payout multiplier. Each time a bet is placed, the Remote Game Server (RGS) selects a specific simulation number with a probability proportional to its assigned weight. The game logic for this simulation is then returned in JSON format by querying a database for the given simulation number, which is then passed to the front-end for visualizing gameplay.

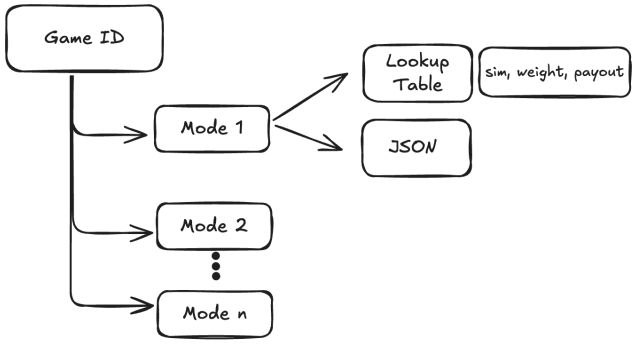

Figure 1: Game information structure

In order to select a result, a random integer value is selected using Golang's *crypto/rand* standard library package to generate a value on the closed interval $[0, \texttt{sum(LUT weights)} - 1]$. This weighted draw method means that we are able to explicitly calculate the Return To Player (RTP) for a game by computing the dot-product of the payout and weight,

$$\mathcal{RTP} = \frac{1}{W} \sum_{i=1}^{N} w_i p_i, \tag{1}$$

where w_i is the weight of a specific simulation, p_i is the payout multiplier for this simulation number and W is the sum of the weights for all possible outcomes.

2 Optimization Algorithm

The algorithm described here is capable of balancing win distributions of finitely constructed mathematical models. When results are initially simulated, all possible outcomes are assigned a weight of '1' such that any unique simulation is equally likely to be chosen. The purpose of this algorithm is to reassign weights for each unique payout such that the discrete set of solutions result in a weighted distribution with the correct RTP, while biasing outcomes to make payouts within a specified range more desirable. Because of this assignment, the initial lookup-table does not have to be balanced (you can more or less pass any RTP lookup table to it and it will still output the desired RTP). In practice, the idea is that the starting point is relatively close to the desired RTP, in order to avoid assigning excessive weighting to any single outcome. If the RTP is far too high, when playing the game the player may notice situations where a board appears 'primed' but does not result in a win. For example, seeing very high multipliers or prizes that then fail to connect.

2.1 Evolutionary Process

The core of this optimization process is the balancing of over- and under-paying distributions, ensuring that the final combined distribution meets the desired Return to Player (RTP). The process can be visualized as blending two opposing forces to reach equilibrium. Ultimately, this process is an evolutionary algorithm that takes in two inputs, one where the RTP is greater than the target value and one where the output is lower than desired. Given these two distributions with the same payout values, a balanced distribution can always be generated by performing a weighted average of the 'over' and 'under' distributions. Given a balanced win distribution ($\mathcal{T}$), a 'positive' distribution ($\mathcal{P}$) where the average win satisfies $\mathcal{P} > \mathcal{T}$, and a 'negative' distribution, $\mathcal{N} < T$ we can solve,

$$\mathcal{T} = x \cdot \mathcal{N} + (1 - x)\mathcal{P}, \tag{2}$$

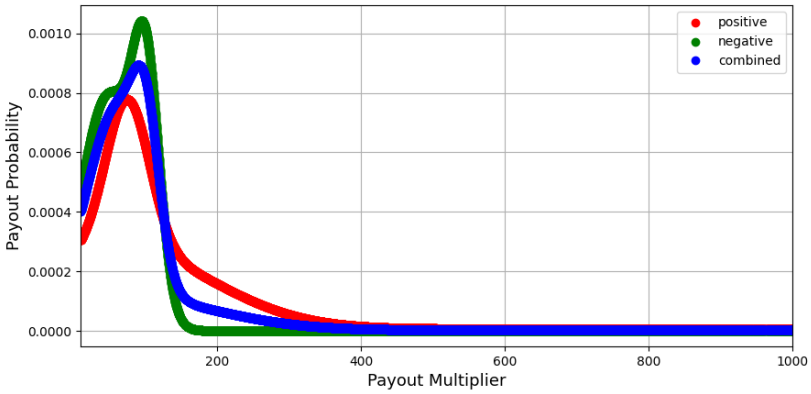

Figure 2: Win distribution balanced to 97% Return To Player (RTP), generated by combining distributions with greater than (positive) and less than (negative) the target RTP for a game with a bet-cost of 100x.

where the weighting, $x \ \exists \ (0, 1)$.

$$x = \frac{\mathcal{T} - \mathcal{N}}{\mathcal{P} - \mathcal{N}}. \tag{3}$$

Weighting the sum of the positive and negative distributions by this factor effectively solves the RTP., as demonstrated in Figure 2.

The real crux of the problem comes with how these positive and negative distributions are generated. This is done by brute-forcing random weights for win distributions which satisfy certain statistical properties that well-performing slot games tend to abide by. But generally speaking, we do this by generating many trial distributions which consist of summing Gaussian curves. Recalling the structure of a Gaussian is of the form,

$$\mathcal{G}(x) = \mathcal{A}\frac{1}{\sqrt{2\pi\sigma}} * \exp\big(-\frac{1}{2}\big(\frac{x-\mu}{\sigma}\big)^2\big), \tag{4}$$

where σ is the standard deviation, μ is the distribution mean, x is the coordinate on the distribution axis (the payout amount here for an ordered win distribution) and $\mathcal{A}$ is some scaling factor.

We start by selecting a random number of Gaussian curves (N) we want to combine (typically between 5 and 15). For each of these distributions we also assign random values for the amplitudes, means and standard deviations. We do this because Gaussians can be conveniently added together to form another

unique curve described by the parameter set:

$$\begin{aligned}\boldsymbol{\mathcal{A}_i} &= [\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_N],\\ \boldsymbol{\sigma_i} &= [\sigma_1, \sigma_2, \ldots, \sigma_N],\\ \boldsymbol{\mu_i} &= [\mu_1, \mu_2, \ldots, \mu_N].\end{aligned}$$

Once these parameters are selected, we write the form of our trial distribution, with a minimum integer weight of '1' assigned to each payout, as

$$\boldsymbol{\mathcal{G}}(x) = 1 + \sum_{i=1}^{N} \boldsymbol{\mathcal{A}_i} \frac{1}{\sqrt{2\pi\boldsymbol{\sigma}_i}} * \exp\big(-\frac{1}{2}\big(\frac{x-\boldsymbol{\mu}_i}{\boldsymbol{\sigma_i}}\big)^2\big). \tag{5}$$

Leaving out distribution fitness criteria (which will be discussed later), we could simply ask the question of whether $\boldsymbol{\mathcal{G}}(\bar{x})$ has an expected return of $>$ $||$ $<$ the target RTP, where $\bar{x}$ is the set of all possible game payouts. We can continuously create these distributions until we have sufficiently many positive and negative distributions which can iteratively combine. By taking $\boldsymbol{\mathcal{G}_N}(\bar{x})$ as the function producing weights which satisfy, $E\langle\boldsymbol{\mathcal{G}}(\bar{x})\rangle < \mathcal{T}$ and $\boldsymbol{\mathcal{G}_P}(\bar{x})$ as the function resulting in $E\langle\boldsymbol{\mathcal{G}}(\bar{x})\rangle > \mathcal{T}$ with $E\langle\rangle$ is the expectation value of the distribution. We create a balanced distribution by concatenating the Gaussian parameters with amplitudes weighted by Eq. 3 to obtain the new parameter set used to create an optimised distribution,

$$\begin{aligned}\boldsymbol{\mathcal{A}_T} &= \left[\frac{x}{\sum_i \boldsymbol{\mathcal{A}}_{\mathcal{P}i}}[\mathcal{A}_{\mathcal{P}1}, \mathcal{A}_{\mathcal{P}2}, \ldots, \mathcal{A}_N]\right] + \left[\frac{(1-x)}{\sum_i \boldsymbol{\mathcal{A}}_{\mathcal{N}i}}[\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_N]\right]\\ \boldsymbol{\sigma_T} &= [\sigma_{\mathcal{P}1}, \sigma_{\mathcal{P}2}, \ldots, \sigma_{\mathcal{P}N}] + [\sigma_{\mathcal{N}1}, \sigma_{\mathcal{N}2}, \ldots, \sigma_{\mathcal{N}N}],\\ \boldsymbol{\mu_T} &= [\mu_{\mathcal{P}1}, \mu_{\mathcal{P}2}, \ldots, \mu_{\mathcal{P}N}] + [\mu_{\mathcal{N}1}, \mu_{\mathcal{N}2}, \ldots, \mu_{\mathcal{N}N}].\end{aligned}$$

In practice, these parameters should be selected to correspond to reasonable solutions to slot game mechanics. Such as ensuring that the probability of selecting a value for μ decreases proportionality as the value strays further from the target mean. All we really want to start checking for at this stage is if the distribution generated is above or below the target RTP (as a minimum condition).

By deconstructing the above balanced, combined distributions we can see the individual Gaussian curves which constitute the final win distribution in Figure 3.

2.2 Acceptance Criteria

If we carefully select our Gaussian parameters such that we tend to generate equal numbers of positive and negative distributions, we can efficiently generate many possible trial solutions. Though we want to put some constraints on which are acceptable. The list of requirements can become quite substantial, such as enforcing hit-rate ranges for particular payouts or performing a Mean Square Error analysis on some *ideal* or distribution. To keep things general, we

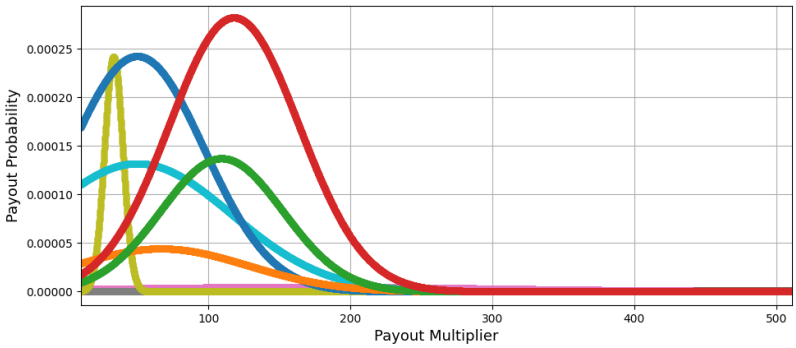

Figure 3: All Gaussian curves constituting the balanced distribution are shown, each with unique amplitudes, standard deviations and means.

will look at a simple measure of volatility by comparing the distribution **mean** and **median** ratios.

The user should define the limits on these values which will depend on how they want the game to *feel*. One method we have utilised is by taking the ratio of the mean payout with the median payout. Where we define the median payout to be the payout multiplier within an ordered win distribution where the cumulative probability is $>= 0.5$. Generally speaking, the larger the value of mean/median, the more volatile the game will feel. As large ratios are indicative of *right-skewed* distributions with more probability being assigned to extreme payouts. Conversely the smaller this value, the more probability is shifted to lower payouts leading to *left-skewed* distributions where more of the distribution probability is clustered around smaller, but more frequent payout amounts closer to the distribution's expected value.

2.3 Ranking Methods

A consideration when selecting win distributions is the macro-scale payout probabilities. We can run simulations for many test players to study the expected player balances after a given number of spins. One strategy discussed here considers distribution volatility by maximising the chances of a player's final balance exceeding some predefined after a fixed number of spins.

To do this we can setup simulated players, typically several thousand, with a large arbitrary starting balance. We then perform a weighted random draw for N spins. After each draw, the players bank balance is updated to reflect the cost of the bet and payout multiplier. The value of N should be selected to represent a realistic number of spins a player is likely to perform. Say, for example, we investigate a game and determine that the average number of spins a player

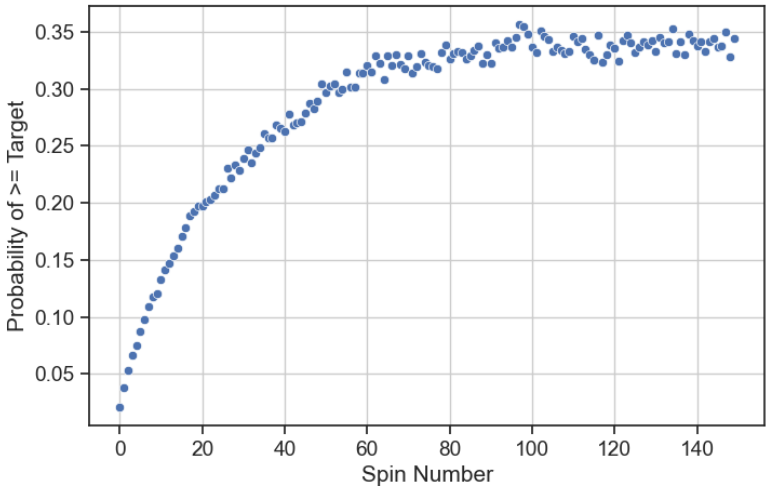

Figure 4: Average probability of a given player being >= 1.0 RTP for a 1x base-game spin using a 0.97 RTP distribution.

performs is 150 spins. We are interested in looking at what the probability is of a player ending an average playing session with a bank balance >= their starting balance (100% RTP). We could investigate any final balance, say 1.5 or 2.0 times their starting balance depending on how we want the game to play. Generally larger values will preference distributions with larger, more sporadic payouts.

```
final_balance = []

for player in range(n_players):
    balance = starting_balance
    spin_balance = []

    for spin in range(N_spins):
        balance -= bet_cost
        payout = weighted_random_draw(payouts, weights)
        balance += payout
        spin_balance.append(balance)

    final_balance.append(spin_balance)
```

For every spin, we can now look at the ratio of players who have a balance >= their starting balance over the total number of players, such as in Figure 4.

2.4 Sub-Game-Type Optimization

Much of the power from this optimization algorithm comes from being able to accurately assign hit-rates, probabilities, or average wins to specific game properties. Given that

$$M(x) = R(x) \cdot \frac{1}{p(x)},$$

Where $M(x)$ is the average payout of some event x, R is the RTP contribution of this event occurring and p is the probability of this event. Given two of these three variables, we apply the evolutionary algorithm to these isolated events and recombine the assigned probabilities to create a configurable win distribution.

For example, if we want to consider a game where we have 60% RTP allocated to the *basegame* and 37% RTP allocation to the *freegame*, we can identify which simulations result in a *freegame* entry and apply the evolutionary algorithm separately to both sets of simulations. These simulation events must be mutually exclusive and the problem essentially reduces to optimizing the *basegame* simulations (where there are no *freegame* triggers) to 0.60 RTP and the free-game simulations to 0.37 RTP. Then by enforcing that the sum of all weights are normalised by $\sum_{i=0}^{N-1} w_i = 1$ we can retrieve a lookup table with well defined RTP, hit-rates or average wins for specific events.

In this example we have only considered a base-game and free-game split, but in practice we may want to enforce a specific hit-rate for 0-wins, max-wins or multiple bonus conditions. By applying the process described in Section 2.1 many times, we end up with many balanced *basegame* distributions and equally as many *freegame* distributions. Since all of these sections are balanced, we can iteratively (or randomly) select balanced distributions for each of these sections and test the overall statistical properties after combination. This provides us with many unique payout-weight distributions to test.

2.5 Forced Parameter Scaling

Another powerful feature of this optimization algorithm is the ability to bias selected distributions to preference certain payout ranges (p_{lower}, p_{upper}). We can define a payout range, scaling factor, and probability of applying this factor to a randomly chosen sum of Gaussian curves. Once the combined Gaussian has been constructed and before the distributions are split into the *positive* and *negative* categories we can scale the probabilities within a specified payout range by a constant factor,

$$w_p^* = \mathcal{C} * w_p \quad \text{if } p_{lower} <= p <= p_{upper}$$

where $\mathcal{C}$ is a user defined scaling parameter and w_p is the weight associated with the unique payout multiplier (p). This scaling can either be > 1 to increase the assigned probability with the specified range or < 1 to decrease the likelihood of payouts within the chosen range.

2.6 Pseudo-random parameter estimation

Given the brute-force approach there is generally a random-noise term (ξ) added to each parameter to ensure a diverse range of possible solutions,

$$\mu \to \mu + \xi_1,$$
$$\sigma \to \sigma + \xi_2,$$

The principle behind how Gaussian parameters are chosen is outlined briefly below.

2.6.1 Mean Distribution Values: μ

We would like to predominantly select functions whose mean is near that of the target mean, where the likelihood of selecting a μ is proportional to the relative distance $|\mathcal{T} - \mu|$. The mean will be constrained to values >= 0 and <= Max(x).

2.6.2 Distribution Standard Deviations σ

The variation in the distribution about mean should not be overly narrow such that the assigned weight is disproportionality applied to a small win range (which would manifest as a narrow spike in the payout-probability distribution plot). We are interested in having an equal number of positive and negative distributions. Since the RTP is particularly sensitive to σ, we generally select a starting value for σ and if we observe significantly more distributions to be positive compared to negative (or *vice-verse*), we vary this value until the number of viable positive and negative distributions balance out.

2.6.3 Distribution Amplitudes $\mathcal{A}$

The selected amplitude should again be proportional to the mean distribution value such that we expect that more significant weighting is assigned to Gaussians whose mean is close to that of the target distribution.