How to Resolve RuntimeWarning: Invalid Value Encountered in Double Scalars

Authored by

stats writer

December 3, 2025

The RuntimeWarning: invalid value encountered in double_scalars is a common notification encountered when working with heavy numerical processing in Python, particularly within libraries like NumPy or SciPy. Unlike a critical error that halts program execution, a RuntimeWarning signifies that a mathematical operation yielded an invalid result, such as an infinity, a value too large to represent, or NaN (Not a Number), but the program continued running. This specific warning focuses on operations involving 64-bit floating-point numbers, which Python often handles as standard double-precision types, hence the term "double_scalars."

This warning is frequently triggered when attempting complex calculations that push the limits of standard computer arithmetic, such as division where the denominator is extremely close to zero, or exponentiation resulting in vastly disproportionate numbers. While the immediate cause might seem like incompatible data types, the core issue usually relates to precision and the catastrophic loss of information when the result exceeds the maximum representable value for a 64-bit float. Failing to address this warning can lead to highly inaccurate model outputs or unexpected behavior further down the computational pipeline, especially in scientific research or machine learning applications where precision is paramount.

To effectively resolve this issue, one must understand the underlying principles of floating-point representation and identify the specific calculation that is causing the numerical instability. General fixes involve utilizing specialized functions designed to handle mathematically challenging calculations robustly, thereby ensuring both the stability and accuracy of our numerical results.

One critical numerical issue frequently encountered in scientific Python environments is:

runtimewarning: invalid value encountered in double_scalars

This occurs when a mathematical operation involving extremely large or extremely small intermediate numbers causes numerical overflow or underflow, forcing Python's numerical libraries to output an indeterminate NaN value or infinity as the final result, signifying a loss of precision.

Understanding Floating-Point Arithmetic and Precision

To truly grasp why the RuntimeWarning appears, we must first review how computers handle non-integer numbers. Most modern systems use the IEEE 754 standard for representing floating-point numbers. Double precision, referenced by the term "double_scalars," uses 64 bits to store a number. This allocation includes a sign bit, an exponent, and a significand. While this system offers an impressive range and precision, it is finite. The maximum finite value representable by a 64-bit float is approximately $1.8 times 10^{308}$, and the smallest positive normal number is around $2.2 times 10^{-308}$.

When a calculation produces a result that falls outside this representable range, we encounter

either **overflow** (result is too large, often represented as infinity) or **underflow** (result is too close to zero, often converted to zero). Numerical instability arises when intermediate results in complex calculations--especially those involving exponential functions and subsequent division--cause these limits to be breached. In data science contexts, such instability often occurs when dealing with probability distributions or statistical models where weights or intermediate values might become extremely large or vanishingly small, leading to the creation of an invalid value (NaN).

Furthermore, standard arithmetic operations in Python, particularly those vectorized operations provided by NumPy, are optimized for speed but do not inherently include numerical safeguards against these extreme cases. If the calculation involves an expression like A/B, and B is calculated via an exponentiation that underflows to zero, Python attempts the division by zero, resulting in infinity and triggering the RuntimeWarning. Recognizing that this is not simply a data type mismatch, but rather a limit-of-precision issue, is the key to selecting the correct mathematical solution.

How to Reproduce the Error in Python/NumPy

We will now demonstrate how this error manifests in a typical numerical calculation involving NumPy arrays, highlighting the scenario where the exponential function causes catastrophic underflow.

import numpy as np

#define two NumPy arrays with large inputs
array1 = np.array([])
array2 = np.array([])

#perform complex mathematical operation: Ratio of sums of negative exponentials
np.exp(-3*array1).sum() / np.exp(-3*array2).sum()

RuntimeWarning: invalid value encountered in double_scalars

We receive a **RuntimeWarning** because the terms inside the denominator, $exp(-3 times 1200)$ and $exp(-3 times 4000)$, become so close to zero that they underflow to exactly 0.0 in 64-bit precision.

This means the result of the division problem is mathematically indeterminate or extremely large, causing an overflow. Since the denominator evaluates to zero, the attempt at division results in an invalid numerical value, triggering the warning and typically yielding `inf` or NaN.

Analyzing Numerical Instability and Log-Space

The core issue here is not that the overall result is large, but that the intermediate step--the calculation of the exponential sum in the denominator--loses all meaningful precision. When the input to the negative exponential is large (e.g., -3600 or -12000), the output is incredibly small, far below the 10^{-308} threshold for double_scalars. The hardware automatically flushes these values to zero, a process known as gradual underflow, which corrupts the subsequent division.

In highly precise scientific computations, we often encounter these ratios of extremely small probabilities. If we calculate these ratios directly using the standard formula, numerical instability is guaranteed whenever the exponents become sufficiently large. The mathematical community solved this problem by shifting the entire calculation into log-space. Instead of computing A/B, we focus on $log(A) - log(B)$, thereby operating on the exponents themselves rather than the vanishingly small probabilities.

By transforming the problem, we avoid the explicit calculation of $exp(X)$ where X is large and negative, thus preventing the underflow that leads to division by zero. This method is fundamental to robust numerical computing when dealing with highly skewed probability distributions or complex likelihood functions where precision must be maintained across many orders of magnitude.

Implementing the Fix: Leveraging logsumexp()

The typical approach to fixing this type of error is to use a special function from a specialized library in Python, such as SciPy, that is capable of handling these extreme values within calculations.

In this specific case, where we are calculating the logarithm of a sum of exponentials, we utilize the powerful logsumexp() function from the `scipy.special` module. This function implements the numerically stable calculation of $log(sum e^x)$.

```
import numpy as np
from scipy.special import logsumexp

#define two NumPy arrays
array1 = np.array([])
array2 = np.array([])

#perform stable mathematical operation
# Original: exp(A).sum() / exp(B).sum()
# Stable form: exp( logsumexp(A) - logsumexp(B) )
np.exp(logsumexp(-3*array1) - logsumexp(-3*array2))
```

2.7071782767869983e+195

Notice that the resulting value, $2.707 times 10^{195}$, is still extremely large, but we do not receive any error or warning because we used the logsumexp() function. This function was designed specifically for scientific computing to handle these types of unstable mathematical operations by performing the calculation in a logarithmically transformed space.

Best Practices for Stable Numerical Computing

When dealing with algorithms that frequently involve exponentiation of large negative numbers, it is essential to look up and utilize specialized functions from libraries like SciPy or NumPy. These functions are highly optimized and engineered to maintain numerical stability far beyond what standard Python arithmetic can achieve.

General best practices include:

Data Scaling: Ensure input data is normalized or standardized to prevent features from taking on excessively large magnitudes that would later result in extreme exponents.
Log-Space Implementation: Whenever possible, formulate probability or likelihood calculations entirely in log-space (e.g., using log-probabilities instead of probabilities).
Library Dependence: Rely on functions specifically designed for stability, such as `logaddexp`, `logsumexp`, or specialized statistical methods, rather than building custom combinations of `exp`, `sum`, and `log`.

These practices ensure that calculations remain within the numerical bounds of double_scalars, avoiding the precision loss and warnings associated with numerical overflow and underflow.

Note: You can find the complete online documentation for the logsumexp() function on the SciPy official website.

Conclusion

The `RuntimeWarning: invalid value encountered in double_scalars` is a crucial indicator of numerical instability rooted in the finite precision of 64-bit floating-point arithmetic. While the warning does not crash the program, it flags an unreliable result. By understanding the causes--primarily catastrophic underflow or overflow stemming from intermediate exponential calculations--developers can implement robust solutions, typically by restructuring the mathematical expression and utilizing specialized log-space functions provided by established scientific libraries like SciPy.