Turbo Codes :

Introduction to Turbo Codes

Turbo codes are an essential advancement in error correction technology, designed to enhance the reliability of data transmission over noisy channels. Their primary purpose is to reduce bit error rates (BER) in digital communications, making them crucial in various applications, including satellite communications, mobile networks, and data storage systems.

Historical Background

The inception of turbo codes dates back to the 1990s, credited largely to Claude Berrou, Alain Glavieux, and Olivier Puri. Their seminal work introduced these coding schemes at a time when conventional error-correcting techniques were nearing their limits in performance. Turbo codes utilize a parallel concatenation of two or more convolutional codes separated by an interleaver, which rearranges the input data, thereby exposing the underlying redundancy. This innovation led to the realization of near-Shannon-limit performance, fundamentally reshaping the landscape of coding theory.

Key Features and Significance

Turbo codes exhibit several important characteristics:

Iterative Decoding: Turbo codes employ a unique decoding approach, where multiple iterations refine the estimated transmitted message. Each iteration utilizes soft information, enhancing accuracy and efficiency.

High Error Correction Capability: Capable of correcting a significant number of errors, turbo codes dramatically lower BER, even under adverse conditions, making them highly effective in real-world applications.

Flexibility: These codes can be adapted to different operational environments and system requirements, allowing for tailored solutions in telecommunications.

Usage in Modern Communication Systems

In the context of modern communication systems, turbo codes are instrumental for maintaining data integrity. Their efficiency in error correction is paramount for:

- **Wireless Communications**: Turbo codes are widely adopted in standards such as 3G, 4G, and upcoming 5G technologies.
- **Deep-Space Communications**: Their ability to handle long-distance transmissions with minimal error rates is crucial in interplanetary missions.

Through continuous enhancements and innovations, turbo codes remain a pivotal element in achieving robust data transmission in increasingly demanding communication environments.

Understanding the Turbo Coding Process

Turbo coding consists of several integral steps that work in unison to minimize errors during data transmission. This section delves into the encoding, interleaving, and puncturing processes, explaining how each step contributes to effective error correction.

Encoding

The first step in turbo coding is the encoding process, where data bits are transformed into a coded sequence that can withstand errors during transmission. This process involves two or more convolutional encoders operating in parallel. The key features of encoding are:

- **Redundancy Creation**: The encoding process introduces redundancy into the data stream, allowing for error detection and correction during decoding.

The encoded output consists of the original data bits along with parity bits, which are crucial for the error correction mechanisms to function properly.

Interleaving

Interleaving is a critical step that rearranges the encoded bits. It transforms the coded sequence into a different order before transmission, further optimizing error correction capabilities. The main benefits of interleaving include:

- **Error Dispersion**: By spreading errors across the coded bit stream, interleaving ensures that burst errors (where multiple consecutive bits are errored) can be corrected by the iterative decoding processes.
- **Enhancement of Decoding Performance**: Low weight messages would also give moderate to high weight encoded message due to interleaving and rsc encoding, this improves error correction.

Puncturing(Not Implemented here)

Puncturing is the process of selectively omitting certain bits from the encoded output, which is essential for optimizing the transmission rate. The significance of puncturing lies in:

- **Efficiency**: Puncturing allows for reduced transmission overhead, meaning that more useful data can be sent in a finite interval with slight compromise on error-correction.

Summary of Steps

The processes of encoding, interleaving, and puncturing collectively enhance the robustness of turbo codes against errors. Here's a quick summary of how these steps contribute to minimizing errors:

Step	Contribution to Error Minimization
Encoding	Introduces redundancy and diversity in code sequences.
Interleaving	Disperses errors across the data stream for better correction.
Puncturing	Improves Code Rate, not implemented here.

By integrating these steps, turbo coding offers a powerful solution for maintaining data integrity in various communication scenarios, ultimately reducing bit error rates and improving overall system reliability.

Implementation Overview

The provided code offers a comprehensive framework for implementing turbo codes utilizing the Sionna library's BCJR decoder function. This code is crucial in simulating and processing data communication over simulated Gaussian channels, addressing the need for robust error correction in real-world applications.

Key Functions Implemented

Turbo Encoding:

- This function implements parallel concatenation of convolutional codes. It takes input data, applies two encoders, and generates redundancy through additional parity bits. This is essential for ensuring that even if some bits are lost or erroneous during transmission, the original message can still be recovered.

Interleaving:

- The interleavier reshuffles the encoded data to enhance error correction. By spreading out consecutive bits, it mitigates the risk of burst errors, allowing for more effective iterative decoding.

Turbo Decoding:

- At the core of the decoding process is the BCJR algorithm, which processes the received signal by estimating the likelihood of transmitted bits using soft information. Iterative decoding improves accuracy with each pass, as it leverages the redundancy introduced during encoding.

Error Checking:

- This functionality evaluates the bit error rates (BER) resulting from transmission, comparing the decoded output to the original inputs. It provides critical insights into the performance of the turbo coding system under different channel conditions.

Dependencies

The implementation relies heavily on the **Sionna library and TensorFlow library**, which is specifically designed for simulation and analysis tasks in the domain of communications. Essential features included in Sionna facilitate high-performance decoding and support diverse communication channel models, ensuring that the turbo codes function optimally under various conditions.

Functions Breakdown

In this section, we will break down the core functions involved in the turbo coding implementation. Each function plays a crucial role in the overall system by contributing to the error correction capabilities of turbo codes. Understanding these functions is vital for software developers and engineers who wish to leverage this implementation effectively.

Gaussian Channel Simulator

The gaussian_channel_simulator function simulates the effects of a Gaussian noise channel on the transmitted signal. This simulation mimics real-world scenarios where data transmitted over a channel is subject to noise, thereby impacting the integrity of that data.

Input Parameters:

- **encoded_message_list (list):** The encoded signal produced by the encoding process.
- **std_dev_of_error (float):** Determines the level of noise added to the signal, which significantly affects the bit error rate.

Expected Outputs:

- **Noisy Signal (array):** The output is the original signal corrupted by Gaussian noise, thus preparing it for decoding. This noisy signal is crucial for assessing the performance of the turbo decoding algorithms.

zero_padding

This function pads zero to the end of message list if length of message list is not divisible by no_of_info_bits_in_block.

RS Encoder

The rs_encoder function is responsible for applying the Recursive systematic convolutional encoding to the input data. This is particularly useful as a preliminary step before the turbo encoding process.

(7,5) RSC encoding has been used.

Input Parameters:

message_list : Message list that has to be encoded.

no_of_info_bits_in_block : It specifies the number of bit that contain information bits in the message. Each block would have 2 extra bits to make the final state (0,0). Also, if the length of message list is not divisible by no_of_info_bits_in_block, it adds some extra zeros at the end of the last block. Due to the type of interleaver used here, no_of_info_bits_in_block + 2 should be a prime.

extra_bits_added_to_make_final_state_00 (default = False) : It specifies if extra bits have to be added to make the final state (0,0). By default, it is false, i.e, the function would add two extra bits to each block, when that is not required, set it to zero. For the second set of parity bits, final state being (0,0) is not required, it doesn't affect performance significantly.

Expected Outputs:

- **Encoded Data (array):** After applying the rs code, this output contains the original data interspersed with redundancy, enhancing error correction applied later in the turbo encoding process.First half would be the message bits while the second half is the parity bits.

Interleaver

The interleaver function takes the encoded data and rearranges it to improve the performance of the iterative decoding process.Working : The number gets multiplied by values from 1 to no_of_info_bits_per_block_plus_two, then, then answers are divided by no_of_info_bits_per_block_plus_two and the remainders are taken, resulting in a permutation of numbers from 0 to no_of_info_bits_per_block_plus_two - 1.

Input Parameters:

- **number :** This is a number should be such that there exists some other number (say inv), (inv*number) % no_of_info_bits_per_block_plus_two = 1.
- **no_of_info_bits_per_block_plus_two :** This must be equal to no_of_info_bits_per_block + 2, and must be a prime number.
- **list_or_array :** The list or array that gets permuted.

Expected Outputs:

- **Interleaved list/tuple**

Turbo Encoder Without Puncturing

The turbo_encoder_without_puncturing function performs the encoding operation using two convolutional encoders without applying any puncturing.It calls the functions rs_encoder and interleaver from inside.

Input Parameters:

message_list : The message to be encoded.

no_of_info_bits_per_block : Number of information bits in each block. Same as in the rs_encoder.

number : Same as in the interleaver.

Expected Outputs:

- **Turbo Encoded Output (list):** The result of the encoding process, including both the original data and generated parity bits from the two convolutional encoders.

mix_sys_parity

This function generates a tensor alternating sys and parity data , taking them as inputs.

Turbo Decode

The turbo_decode function utilizes the BCJR algorithm for decoding the received signals. This iterative decoding process refines the estimate of the original data using the redundancy provided in the encoded stream.

Input Parameters:

systematic : The ndarray of message bits after passing through the channel

parity1 : The ndarray of parity bits from the first encoder after passing through the channel

parity2 : The ndarray of parity bits from the second encoder after passing through the channel

interleaver_number : Same as in interleaver

inverse_of_interleaver_number : The other number which gives remainder one when multiplied by interleaver_number and then divided by block_len (or) no_of_info_bits_per_block_plus_two

block_len : Same as no_of_info_bits_per_block_plus_two

num_iter (default=6) : No of iterations to run the decoder for, greater the number of iterations greater the accuracy, but it takes a lot more time.

Expected Outputs:

- **output_llrs (list):** The output after the decoding process represents the estimated original data. Over multiple iterations, the refined estimate should closely match the input data.

hard_message_output

This function takes llrs as input and uses it to give hard output(0 or 1). It also removes the two extra bits added to each block to make its final state (0,0), but it doesn't remove the padded zeros.

no_of_bit_errors_finder

Takes in two lists as inputs and returns the number of positions the at which both lists differ in value.

Summary of Major Functions :

Function Name	Purpose
gaussian_channel_simulator	Simulates transmission noise effect on the signal
rs_encoder	Applies Recursive Systematic encoding to the data
interleaver	Rearranges encoded data for better error correction
turbo_encoder_without_puncturing	Performs turbo encoding with full data redundancy
turbo_decode	Decodes received signals using iterative BCJR decoding

Together, these functions create a comprehensive framework for implementing turbo codes, each contributing uniquely to the error-correcting capabilities required for reliable data transmission across varying communication environments. Understanding these functions prepares developers to optimize, modify, or further enhance the turbo coding system according to specific project needs.

Results and Outputs

In this section, we present the expected outputs derived from our turbo coding implementation. This will include the encoded message, the hard channel output, the original message, the decoded message, and the number of bit errors detected during the process. We will illustrate these results using sample data, providing insights into their significance.

Sample Run Output :

Encoded list: [1, 0, 1, 0, 0, , …. , 0, 0, 0, 0, 0, 0, 0]
Only hard channel output : [1, 0, 1, 0, 0, …. 0, 0, 1, 0, 0, 0, 0, 1, 0]

Original message with 0 padding : [1, 0, 1, 0, 0, …. 0, 0, 0, 0, 0, 0, 0, 0, 0]
Decoded message list : [1, 0, 1, 0, 0, …. 0, 0, 0, 0, 0, 0, 0, 0, 0]
No of bit errors : 6
No of bit errors without any encoding : 173

The Impact of Errors

The implications of bit errors can vary significantly across application domains:

- **Wireless Communications**: For applications like mobile networks, high BER could lead to dropped calls or data corruption, impacting user experience. Turbo codes significantly reduce BER, aiding in maintaining connection quality.
- **Data Storage**: In systems where data integrity is critical, such as hard drives or solid-state drives, even a single error can lead to data loss. Turbo codes offer robust error correction mechanisms to mitigate such risks.

From these examples, it is evident that turbo codes significantly lower the BER, leading to improved reliability in data transmission. Understanding these metrics is essential for software developers and engineers seeking to implement effective error-correction solutions in telecommunications and data transmission systems.

References:

- https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://en.wikipedia.org/wiki/Turbo_code&ved=2ahUKEwiisMe1quSMAxXZxDgGHbI3CigQFnoECBYQAQ&usg=AOvVaw2gKqseE84QMtO9wN6HrY5u
- https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://faculty.uml.edu/jweitzen/16.548/ClassNotes/art_sklar3_turbocodes.pdf&ved=2ahUKEwiisMe1quSMAxXZxDgGHbI3CigQFnoECDgQAQ&usg=AOvVaw2lrPfH53q6ikvx7cOMuoGh
- https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.youtube.com/watch%3Fv%3DG9G9iwlrf_o&ved=2ahUKEwjZ0uTZquSMAxU-jGMGHY_PC74QtwJ6BAgTEAI&usg=AOvVaw00iNs4e3vNH0X-hHCLdSxM