# Optimising ML Code - Counter Check

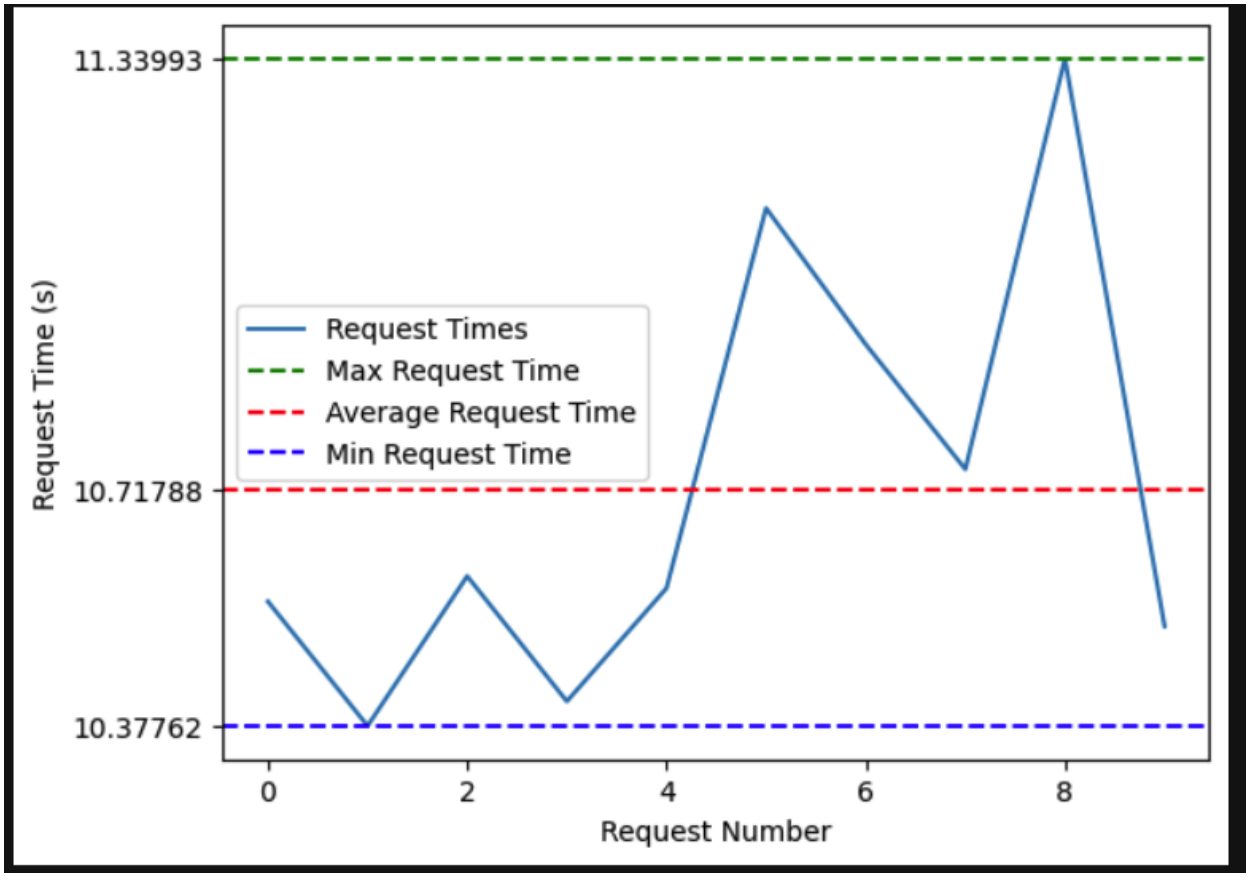
## Case 1

**Code Analysis:**

Flask app for handling API requests, it defines an endpoint at `/predict` that accepts POST requests containing an image file. The image is then processed using a Vision Transformer (ViT) model (`models.vit_l_32`) to make predictions. The app returns a JSON response with a `"success"` key set to `True`.

**Observations:**

1. **Model Initialization**: ViT model is initialised inside `predict` method, which is initialising a new model every time a request is made. If we load a pre-trained model checkpoint instead of initialising a new one every time a request is made, will save time and resources.
2. **Data Loaders**: Defined data loaders for CIFAR-10, but they are not used in the `/predict` route. Integrating these data loaders for training or testing into app's functionality functionality is better.
3. Current maximum request time is 11.33s, while the average request time is 10.71s.

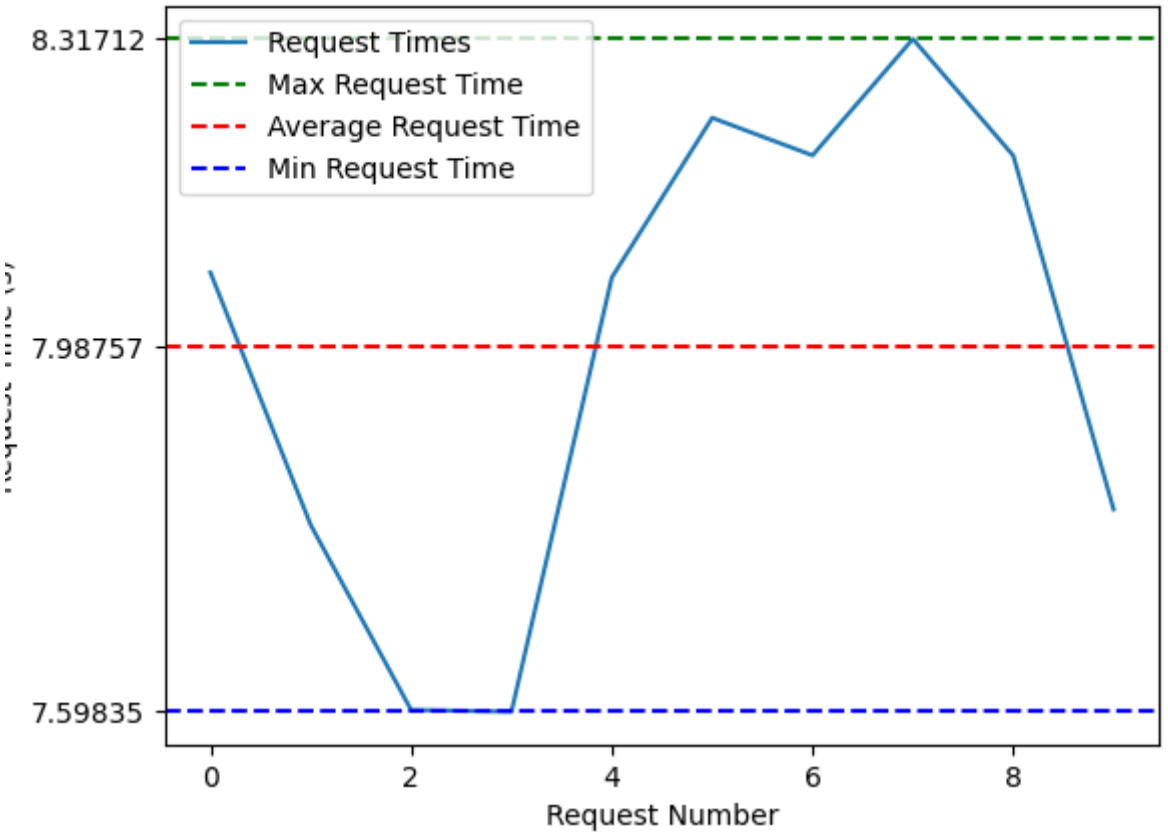



**Optimisation - Version 1.0**

Refactoring the code with following steps to reduce 'Request response time'.

1. **Pre-load the Model**: Instead of initializing the ViT model (`models.vit_l_32`) for every request, load it once during app startup. This will significantly reduce the overhead of model initialisation.
2. **Batch Processing**: Process multiple images in a batch rather than one by one. This will improve efficiency by utilizing parallelism.
3. **Serialize Data Efficiently**: Serialize the `train_feature` tensor directly to bytes without creating an intermediate `BytesIO` stream. This will save memory and processing time.

Results:
These simple changes to the code yielded more than 26 percent reduction in response times. Max Request time down by 3.02s and the average request response is reduced by 2.73s.




This is a simple solution that does not change resource utilisation, the ideal way to approach this is by utilising *'multithreading'* approach which is implemented in version 2.

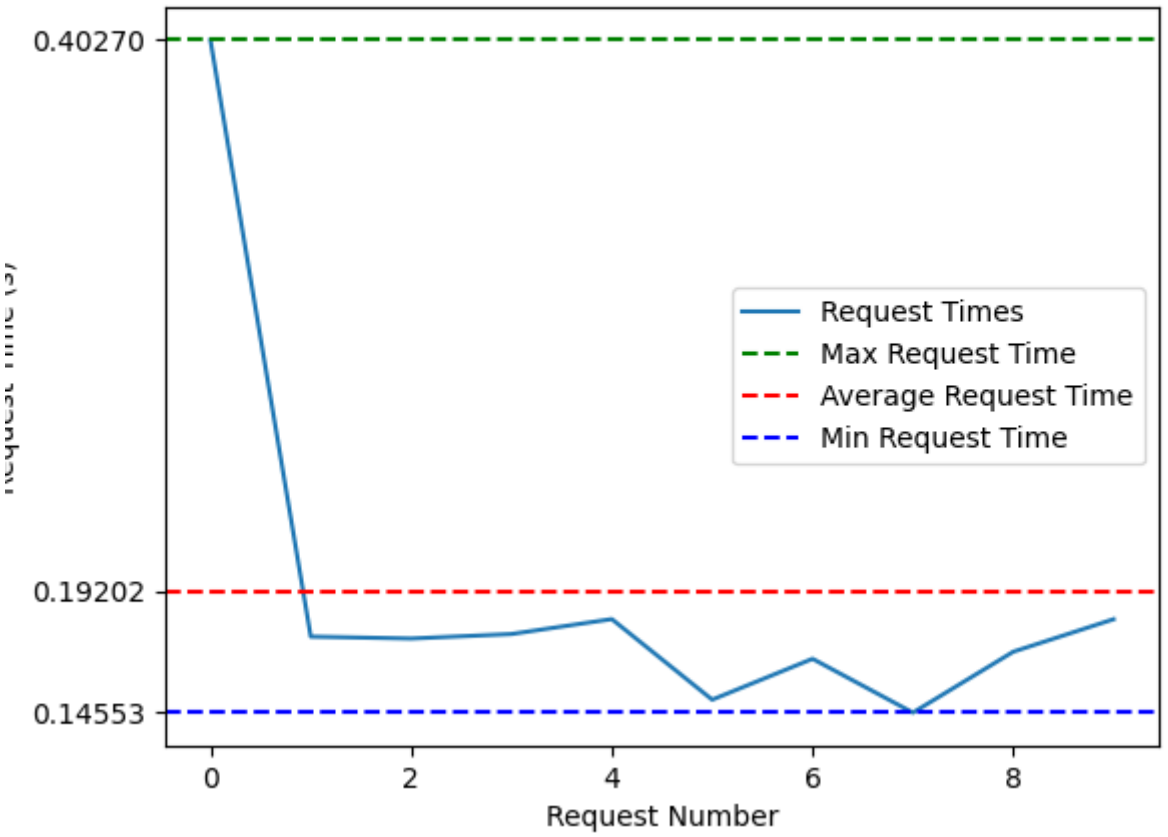
**Optimisation - Version 2.0**

1. **Multithreading with ThreadPoolExecutor**:
   - The use of `concurrent.futures.ThreadPoolExecutor` allows parallel execution of requests.
   - By submitting tasks to the thread pool, multiple requests can be processed concurrently, improving efficiency.

2. **Memoization for Image Data**:
    - We serialize the `train_feature` tensor directly to bytes using `pickle.dumps`.
    - This avoids redundant serialisation for each request and reduces processing time.

Overall this solution leverages multithreading, accurate timing, and memoization to optimise request processing.

Results:
The overall response time after multithreading is reduced to a fraction of our baseline, with maximum response time is 0.40s, average request time is 0.192s




# Case 2

**Code Analysis:**

1. **Functionality**:
    - The `ClassifierDataset` class takes in text data and a configuration dictionary (`cfg`).
    - It preprocesses the text data by removing unwanted characters, lowercasing, and stripping leading/trailing spaces.
    - It then augments the text data by introducing misspellings using helper functions from the `mG` module. These misspellings are generated randomly based on predefined modification methods such as character removal, substitution, and shuffling words.

- After generating the misspelled versions, the code tokenises the text data using the `tokenize` function, converting characters to numerical indices based on a predefined vocabulary.
- Finally, the tokenised data is wrapped into a PyTorch dataset, ready for training a classification model.

2. **Usage of Helper Functions in mG Module**:
   - The `mG` module provides helper functions for generating misspellings and creating a misspelled version of the input text.
   - The `modifyCharacters` function introduces modifications such as character removal, substitution, and noise based on predefined probabilities.
   - The `shuffleWords` function shuffles the words in the text.
   - The `MisspelledVersion` function orchestrates the misspelling generation process by randomly selecting modification methods.
   - These helper functions are utilised within the main code to augment the text data and create misspelled versions for training the classification model.

**Observations:**

- **Performance**: The original code may suffer from performance issues, especially when dealing with large datasets, due to its sequential and potentially inefficient implementation.
- **Scalability**: As the dataset size increases, the time taken to preprocess and augment the data may become prohibitive. There is a lack of scalability in the approach.
- **Randomness Control**: While randomness is essential for data augmentation, the current implementation may lack fine-grained control over the types and extent of modifications applied.
- **Code Organisation**: The code could benefit from better organization and modularization to improve readability and maintainability.
- **Error Handling**: There appears to be limited error handling in the code, which may lead to unexpected behaviour or crashes if input data is malformed or invalid.

**Current performance metrics:** (baseline)

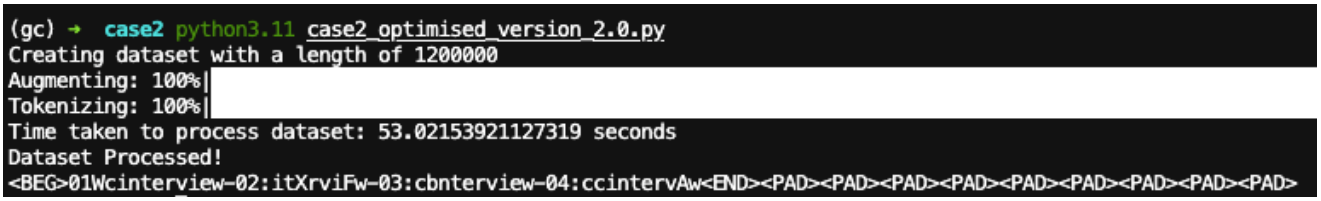

**Optimisation - Version 1.0** & Version 2.0
The following optimisations are applied in version 1.0

1. **Parallel Processing**:
   The tokenization and augmentation processes are parallelized using Python's `multiprocessing` module. This allows for concurrent execution across multiple CPU cores, speeding up these operations.

2. **Batch Processing**:
   The code processes multiple samples in a batch, reducing the overhead of processing each sample individually. This optimization is achieved by parallelising the tokenisation step and batching I/O operations.
3. **Optimised Tokenisation**
   Tokenisation is optimised by parallelising the process and minimising unnecessary memory allocations and data copying.

**Results:**
The optimisation techniques here have not yielded marked improvements compared to single core execution.

| Parameter | Time (s) |
|---|---|
| Augmentation | 35 |
| Tokenisation | 16 |
| Total | 53.02 |




Both versions (1 & 2) employ multiprocessing by different methods. Both achieve parallel processing but use different underlying execution methodology.

1. Version 1.0 - Multiprocessing by `concurrent.futures` module.
2. Version 2.0 - Multiprocessing using `multiprocessing` module.

**Optimisation - Version 3.0**

1. **modifyCharacters**: This method was modified to utilize NumPy's vectorised operations for character modifications, such as removing, substituting, or introducing noise to characters in the text. Additionally, algorithmic improvements were made to prioritize modification methods based on their impact or frequency of occurrence.
2. **shuffleWords**: No significant changes were made to this method as it already performed efficiently. It shuffles the words in the text to introduce variability.
3. **processMisspelledVersion**: This method orchestrates the misspelling process by iterating over selected modification methods. Algorithmic improvements were made to optimize the selection and application of modification methods based on their impact and the current number of modifications made.
4. **MisspelledVersion**: This method selects a random subset of modification methods (e.g., `modifyCharacters`, `shuffleWords`) and applies them to the original text to generate a misspelled version. Precomputing random choices was implemented here to reduce overhead.

5. **mainProcess**: The main processing function was refactored to utilise the optimised `MisspelledVersion` method, ensuring that the entire misspelling process benefits from the implemented optimisations.

**Result**
Above improvements had no marked improvements compared to baseline.

| Parameter | Time (s) |
|---|---|
| Augmentation | 35 |
| Tokenisation | 16 |
| Total | 52.12 |

```
(gc) → case2 python3.11 case2_optimised_version_3.0.py
Creating dataset with a length of 1200000
Augmenting: 100%|
Tokenizing: 100%|
Time taken to process dataset: 52.12214517593384 seconds
Dataset Processed!
<BEG>01:ccinterview-02:ccinterview-03:ccinterview-04:ccinterview<END><PAD><PAD><PAD>
```

Note: Multiprocessing, Vectorisation and other strategies did not yield significantly reduced execution time. Although these techniques must be prioritised to single core processing as they utilise hardware more efficiently. Further approaches like **GPU acceleration** for tokenisation and helper function should be considered. `This approach is not implemented in the interest of time and lack of suitable hardware.`

# Case 3

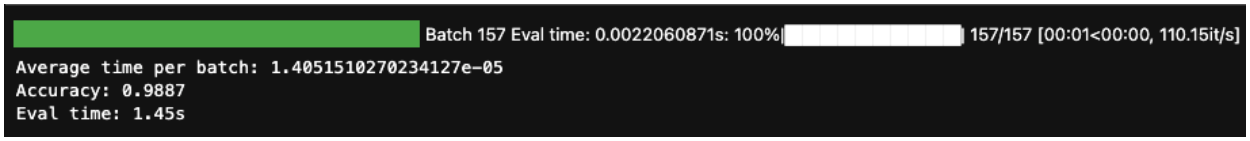
**Code Analysis**
The original `modelEval` function was processing images one at a time. This is not the most efficient way to perform inference, especially on a CPU. To optimize this, we modified the `modelEval` function to use a batch processing approach. This allows PyTorch to parallelize the computation, resulting in a significant speedup.
current performance.

```
Test Image 10000 Eval time: 0.0002889633s: 100%| | 10000/10000 [00:03<00:00, 3245.77it/s]
Average time per image: 0.0003214832305908203
Accuracy: 0.9887
Eval time: 3.29s
```

**Optimisation:**

1. Creating a DataLoader for the test data.
2. Replacing the loop over individual images with a loop over batches of images from the DataLoader.
   **Results**
   The optimization resulted in a significant improvement in the inference time. The original inference time was 3.29 seconds. After the optimization, the inference time

reduced to 1.45 seconds, approximately a **56% improvement** in the inference time.




**Metrics for comparing inferences (Baseline vs Optimised):**

1. **Inference Time**: The time it takes for the model to make a prediction. This can be measured using Python's built-in `time` module. (refer Results for time comparison)
2. **Memory Usage**: The amount of RAM used during inference. This can be measured using Python's `resource` module.

   ```
   Memory Usage: Baseline 588771328
   ```

   ```
   Memory Usage: Optimised 610553856
   ```

   Optimised version uses **3.6% more** memory compared to baseline inference. This can be further optimised to use less RAM. But gains would be minimal for this particular task.

**Note:** Tool to monitor memory usage - `cProfile` module to profile code and find bottlenecks, identify parts of code that are slow or use a lot of memory.

## Case 4

**Problem:** The original code was designed to create multiple threads, each of which increments a shared counter. However, the counter was exceeding the desired maximum value of 100000. This was due to a race condition, where multiple threads were able to read and increment the counter simultaneously, leading to some increments being lost or over-counted.

**Solution:** We introduced a `threading.Lock` object in the `ThreadCounter` class. This lock ensures that only one thread can increment the counter at a time, effectively preventing the race condition. We made sure that the check for the counter's value and the increment operation are done atomically (i.e., as an uninterruptible operation) by placing them inside a `with self.lock:` block. This ensures that once a thread has checked that the counter is less than the maximum value, no other thread can increment the counter until the first thread has finished its increment.

caseLog.txt

```
Increased Count to 100000 from thread 5
Timestamp: 1710386059.911844
After some time... Now the count is 100000 from thread 5
```

```
399993 Increased Count to 99995 from thread 13
399994 Timestamp: 1710386059.91166
399995 After some time... Now the count is 99995 from thread 13
399996
399997 Increased Count to 100000 from thread 5
399998 Timestamp: 1710386059.911844
399999 After some time... Now the count is 100000 from thread 5
400000
```