ChatGPT

# OptiFormer-Lite End-to-End Smoke Test Plan

**Objective:** Validate the full OptiFormer-Lite pipeline on a minimal scale (15–30 minute run on a single RTX 4090) before committing to full-scale experiments. This smoke test covers all key components – from synthetic data generation to Optuna integration – ensuring each part works correctly in isolation and in concert.

## Step 1: Synthetic Data Generation (GP + Symbolic Regression)

**Goal:** Generate a small, diverse synthetic HPO dataset (~1,000 total trials) using Gaussian Process (GP) priors and symbolic regression. This simulates a variety of optimization landscapes for the model to learn from [1] [2].

**Procedure:**

- **1.1 GP-Based Functions (Smooth Landscapes):** Use GPyTorch to sample several random functions from GP priors with varied kernels (e.g. RBF, Matérn, Periodic) [3] [4]. Each sampled function represents a synthetic objective surface. For each function:
- Select a random kernel and hyperparameters (e.g. lengthscale) to vary smoothness [5].
- **Simulate Optimization Trajectory:** Run an existing optimizer (e.g. Optuna's TPE or Random search) on the function for a limited number of trials (e.g. 50 trials) to produce a sequence of hyperparameter suggestions and observed objective values [6]. Record each trial's parameters and outcome as one optimization trajectory.

- Inject observation noise into objective values to mimic real experimental noise (e.g. add small Gaussian noise to f(x)) [7].

- **1.2 Symbolic Functions (Jagged Landscapes):** Use PySR or a similar symbolic expression generator to create a set of random mathematical expressions (with operators like +, -, ×, sin, etc.) [8]. These functions introduce irregular, non-smooth behavior. For each symbolic function:

- Evaluate it on random hyperparameter inputs to simulate an objective surface.

- Run the same optimizer for ~50 trials to obtain a trajectory of trials (these will be more "rugged" optimization paths, complementing the smooth GP ones).

- **1.3 Combine and Format:** Collect ~20 trajectories (half from GP-sampled functions, half from symbolic functions) to total roughly 1,000 trials [1]. Ensure each trajectory is stored in a consistent format (e.g. list of trials, where each trial contains hyperparameter values and an objective/result). This will form the training dataset for the smoke test.

**Expected Output & Criteria:**

- A **diverse synthetic dataset** of ~1,000 trials covering both smooth and jagged optimization behaviors (approximately 10 GP-based trajectories and 10 symbolic trajectories) [1] [2]. Each trajectory should resemble a plausible hyperparameter optimization log (i.e. a sequence of parameter configs and their performance).
- **Diversity Check:** Confirm that some trajectories show smooth, gradually improving performance (from GP functions) while others have more erratic behavior (from symbolic functions). This ensures the model will learn both exploitation and exploration patterns.
- **Integrity Check:** No function should produce degenerate data (e.g. all trials with identical outcomes). If any trajectory is trivial or constant, regenerate with different seeds or parameters.
- **Resource Use:** Synthetic data generation should complete within ~1–2 minutes on CPU (GP sampling can use GPU but is lightweight at this scale). Memory usage is negligible. If runtime is excessive, reduce the number of trials or complexity of generated functions.

**Safety Checks:** If either sub-component fails (e.g., PySR not installed or GP sampling errors), adjust or install necessary packages. Verify that the total number of trials and their value ranges match expectations (e.g. parameter values are within [0,1] if normalized). This step ensures a correct and varied dataset to train on, reducing risk of model overfitting to a single pattern.

## Step 2: Tokenization Pipeline Verification (QuantizedNumericalTokenizer)

**Goal:** Ensure the **QuantizedNumericalTokenizer** can accurately convert continuous hyperparameter values to tokens and back. Since the tokenizer is a single point of failure for the pipeline, we perform rigorous unit tests on float-to-token encoding and decoding [2].

**Procedure:**

- **2.1 Initialize Tokenizer:** Configure the QuantizedNumericalTokenizer with the expected hyperparameter domains and precision. For example, define the quantization scheme (e.g. number of bins or codebook size for each continuous parameter) and include any special tokens (such as parameter identifiers or a `<target_regret_0>` prompt token used for inference) [9] [10]. Ensure the vocabulary size is known for later reference (vocab_size).

- **2.2 Round-Trip Tests:** For each hyperparameter type in the search space:

- Generate a set of test values covering the range (e.g. min, max, median, random samples within bounds).
- Encode each float value into token ID(s), then immediately decode it back to a float.
- Compare the decoded float to the original. **Expected:** The values should match exactly or be within the quantization tolerance (no significant loss of precision). For example, if a value 0.1234 was quantized, decoding should return 0.1234 (or very close, within one quantization step).

- Repeat the test for edge cases (boundary values, extremely small/large values within the allowed range, etc.). Also test multi-parameter encoding if the tokenizer handles vectorized inputs (each parameter should round-trip correctly independently).

- **2.3 Special Token Checks:** Verify that special tokens (e.g., tokens indicating parameter names or the special `<target_regret_0>` used to prompt the model for the next suggestion) are properly recognized by the tokenizer. For instance, calling `tokenizer.token_to_id('<target_regret_0>')` should return a valid token ID [10]. Also ensure that decoding such tokens (if applicable) yields the expected symbolic value or placeholder (these might not round-trip to floats, but should remain consistent).

**Expected Output & Criteria:**

- **High Fidelity Encoding:** All test floats should successfully round-trip through the tokenizer with negligible error. The maximum error should be on the order of the quantization resolution (ideally zero for representable values) [2].
- **Deterministic Behavior:** Encoding the same value multiple times yields the same token, and decoding that token yields the same float every time. This consistency is crucial for the model to learn meaningful patterns.
- **Vocabulary Confirmation:** Record the final `vocab_size` and ensure it aligns with expectations (e.g., if 100 bins per parameter and some special tokens were planned, confirm the count). This will be used to interpret the random baseline entropy in training.

**Safety Checks:** If any value fails to round-trip accurately, refine the tokenizer (e.g., increase number of quantization buckets or correct normalization). A common failure could be floats not mapping to the correct integer token (which would later manifest as flat training loss) [11]. This test is a pass if **all** sampled values round-trip without significant error. Only proceed to model training once the tokenizer passes, since downstream learning depends on this accuracy.

## Step 3: Miniature Transformer Model Setup (50M–80M Params)

**Goal:** Instantiate a small Transformer-based model (~50–80 million parameters) that will serve as the "OptiFormer-Lite." Validate that the model fits in memory and is configured for our tokenization scheme (e.g., correct vocab size and context length).

**Procedure:**

- **3.1 Model Configuration:** Define a decoder-only Transformer architecture suitable for sequence modeling (e.g., a causal language model). For instance, use ~8 layers, hidden size ~512, ~8 attention heads, with a context window length that can accommodate the longest trajectory in our dataset (e.g., 256–512 tokens context) [12]. If using a library (such as Hugging Face Transformers), one could start from a small GPT-2 or LLaMA configuration and adjust dimensions to reach ~50M parameters [12].

- **3.2 Vocabulary Integration:** Set the model's embedding layer to size `vocab_size` (from Step 2). Ensure that positional embeddings (or Rotary Positional Embeddings, as LLaMA uses) are configured for the max context length of the optimization history sequences.

- **3.3 Instantiate Model:** Initialize the model weights (randomly). Confirm the total parameter count. **Expected:** The count should be in the tens of millions (e.g., around 50–80M) as planned [12]. If it's much higher, double-check layer sizes or number of layers.

- **3.4 Device Placement:** Move the model to the GPU (single RTX 4090). Use `fp16` (mixed-precision) for efficiency if possible. At this size, the model's memory footprint is very small relative to GPU capacity – roughly 200MB in FP16 for 100M params, well under 2GB even including optimizer states and activations [13]. This leaves the majority of VRAM free for training data batches.

**Expected Output & Criteria:**

- **Successful Initialization:** The model should be created without errors and properly reflect the intended architecture (verify layer count, hidden size, etc.). There should be no shape mismatches with the tokenizer (embedding matrices and output logits size must equal the tokenizer's vocab size).
- **Memory Check:** GPU memory usage for the model (weights) should be on the order of a few hundred MB. Confirm via `nvidia-smi` or programmatically that memory allocated is well below 24GB (expected <2GB for the model itself) [13]. This ensures we can use large batches in training.
- **Parameter Count Verification:** Log the number of parameters. It should match the design (~50–80M). If it's off, adjust the configuration (e.g., layers or width) to hit the target size. This size is chosen to be *feasible* on the hardware while still capturing HPO dynamics [12].

**Safety Checks:** If the model instantiation fails (e.g., out-of-memory or configuration error), reduce the model size or check for mis-specified arguments (such as a vocab size mismatch). Ensure to enable gradient checkpointing or other memory-saving techniques if early experiments show high activation memory usage, but for this scale it shouldn't be necessary. Passing this step means the model is ready to train, with correct dimensions and manageable memory footprint.

## Step 4: Short Training Loop & Monitoring

**Goal:** Train the mini-Transformer on the synthetic dataset for a brief period (~15 minutes) to ensure the training pipeline works and the model starts learning. We will monitor the training loss curve for expected signs of learning and catch any anomalies early [14].

**Procedure:**

- **4.1 Data Loader Setup:** Encode the synthetic trajectories (from Step 1) into token sequences using the tokenizer. Each trajectory becomes a sequence of token IDs. Prepare a PyTorch `Dataset` / `DataLoader` that yields batches of sequences for training. Use a **large batch size** to utilize available VRAM (e.g., 256 or 512 sequences per batch) [1]. If the total dataset is small (1000 trials), you might reuse sequences or train for multiple epochs within 15 minutes.

- **4.2 Training Configuration:** Use a standard language-model training loop:

- Loss function: Cross-entropy on next-token prediction (causal language modeling objective).
- Optimizer: AdamW (with appropriate weight decay).
- Learning rate: Start with a moderate LR (e.g., 1e-3 or 1e-4) since the model is small; use linear warmup for a few hundred steps to avoid instability.

- Mixed precision (FP16/BF16) training to speed up computation on RTX 4090 Tensor Cores [15].

- **4.3 Run Mini Training Loop:** Train for a fixed short duration. **Time-box the run to ~15 minutes** of wall-clock time (or an equivalent fixed number of steps that typically fit in that time, e.g., a few hundred iterations) [1]. Print or log the training loss periodically (e.g., every 10 or 20 steps) to observe the trend.

- **4.4 Monitor Learning Curve:** Pay close attention to the first ~100 training steps:

- **Expected Behavior:** The loss should start high (around the entropy of a random predictor) and **drop rapidly in the first 100 or so steps**, indicating the model is beginning to learn patterns in the data [16]. By the end of 15 minutes, the loss should stabilize to a value *below* the entropy of random guessing (~ln(vocab_size), if using natural log) [16]. For example, if `vocab_size ≈ 1000`, random-choice loss would be ~6.9 nats; we expect the model's loss to fall below this (meaning it's doing better than random predictions).

- **Potential Failure:** If the **loss remains flat** (or declines extremely slowly) and stays near the random baseline, it's a red flag [11]. The most common cause is a tokenization issue – e.g., the model isn't actually seeing meaningful differences between tokens (floats not mapped correctly to tokens, so it can't learn). Other causes could be an overly high learning rate causing instability or a bug in data feeding.

- **4.5 Terminate Training:** Stop training after ~15 minutes of running. Save a checkpoint of the model weights (optional for inspection). This short run is only to verify learning behavior, not to fully train the model.

**Expected Output & Criteria:**

- **Loss Trend:** A clear downward trend in the training loss within the short run. Ideally, an initial loss drop of at least ~20-30% from the starting value in the first 100 steps, and continuing to decrease or flatten out at a lower value by the end of the test. For instance, if starting loss is ~7 nats, it might drop below ~5 nats after the mini-run (exact numbers will vary) [16].
- **Stability:** The training process should run without crashes (no out-of-memory errors, no NaNs in loss). The large batch size should be sustainable given the 24GB VRAM (expect well under 20GB usage even with batch 512 and context ~256) [13]. If using FP16, ensure no overflow (use gradient scaling if necessary to prevent NaNs).

- **Pass Criteria:** The model shows signs of learning (loss < random baseline, noticeably decreasing). This validates that the data format and tokenization are learnable and the model architecture is sound.

- **Fail Indicators & Diagnostics:**

- **Flat or No Loss Decrease:** If loss stays flat around the starting value, investigate the tokenizer encoding (Step 2). A float-to-token mapping bug can make the sequence prediction essentially random [11]. Verify the data pipeline is shuffling and providing varied sequences (a non-shuffling data loader could also cause weird plateaus if the same sequence repeats).
- **Diverging Loss/NaNs:** If loss spikes or becomes NaN, lower the learning rate and retry, or check for unstable gradients (reduce batch size or use gradient clipping).

- **Performance Note:** If the model learns *too* slowly (barely any drop in 15 min), consider that 50M params on 1000 trials is a very high capacity vs. data size – it should overfit easily. Little movement might indicate a bug. Fix issues and re-run this mini-training until the criteria are met before proceeding.

## Step 5: Inference Logic Check (Exploitation vs. Exploration)

**Goal:** After the short training, verify that the model's **inference mechanism** produces reasonable hyperparameter suggestions, demonstrating both exploitation of good regions and exploration of new regions. This ensures the forward pass and decoding logic work as intended.

**Procedure:**

- **5.1 Prepare Test Histories:** Construct two small hypothetical optimization histories (sequences of past trials) to probe the model:
- **Exploration Scenario:** A history of "bad" trials – e.g., 5 trials whose objective values are all poor (far from the optimum). For realism, use one of the synthetic functions from Step 1: take a set of hyperparameter values that yielded high error/low score. Encode this history into tokens via the tokenizer (including any necessary context tokens like parameter identifiers and observed objective values if those are part of the sequence).

- **Exploitation Scenario:** A history of "good" trials – e.g., 5 trials where one or more configurations were very close to the optimum (low error or high score). Encode this history similarly into tokens. In this context, the best previous trial should clearly stand out as superior.

- **5.2 Model Inference:** For each scenario:

- Append the special prompt token indicating we want the model's suggestion for the next hyperparameter (for example, prepend `<target_regret_0>` or a similar meta token as used in the training format) [10] .
- Feed the tokenized sequence into the trained model and perform a **forward pass** to obtain the logits for the next token prediction. Use greedy selection or a sampling strategy (e.g., `argmax` for deterministic output) to select the next token as the model's suggestion for the next hyperparameter value [17] .

- Decode the predicted token back to a float hyperparameter value using the tokenizer's decode method [18] . Then **denormalize** if needed to get the value in the original parameter scale (as the Optuna sampler would do) [19] .

- **5.3 Analyze Suggestions:** Compare the model's suggested hyperparameter value to the history:

- In the **Exploration scenario**, check that the suggested value is significantly different from those in the history. For example, if the history trials all had a particular hyperparameter around 0.2 (with poor outcomes), the model might suggest a value around, say, 0.8 or a different region of the space. A large deviation indicates the model is attempting to explore away from the failing region [20] .

- In the **Exploitation scenario**, verify that the suggestion is close to the best-known value from the history. If the best trial had a hyperparameter value 0.75, the model might suggest something like 0.70–0.80. This would indicate the model is focusing (exploiting) near the known good region [20].
- In both cases, ensure the suggested token decodes to a valid value within the allowed range of that hyperparameter (no out-of-bounds or illogical values).

**Expected Output & Criteria:**

- **Meaningful Suggestions:** The model's next-point proposals should make intuitive sense given the context:
- **Exploration test:** The output value should not simply mimic the bad history values. A clear shift signifies the model "recognizes" that those values performed poorly and tries something different (even with its limited training) [20]. This could be considered a pass if the suggestion differs by a substantial margin from all historical values.

- **Exploitation test:** The output should be near the top-performing historical value, showing the model leans into what looks promising [21]. A pass here is a suggestion in the neighborhood of the best trial's parameters (or at least not a wild outlier when good data is present).

- **Consistency:** If the model includes randomness in generation (e.g. if using non-greedy sampling), run the inference a few times. The exploitation case should consistently yield similar suggestions (clustered around the optimum region), whereas the exploration case might yield varied suggestions (anywhere away from the poor region, which is fine).

**Safety Checks:** This qualitative check ensures the **inference pipeline** (tokenize -> model forward -> decode) is working. If the suggestions in both scenarios are nonsensical (e.g., in the bad history the model suggests a value *within* the same bad region, or outputs the exact same value for both scenarios), it may indicate insufficient training or a flaw in how the history is encoded. Double-check that the history encoding includes performance outcomes, so the model knows which trials were "bad" vs "good." If performance data wasn't included in context, the model might not distinguish them – consider encoding the objective values or using a "regret" token scheme as planned in the full design. At this stage, minor oddities are acceptable given minimal training, but the model should at least produce **valid, in-range outputs** and show a *tendency* towards exploring or exploiting appropriately. Once this is observed, we proceed to the final integration test.

## Step 6: Optuna Integration Test (OptiFormerSampler)

**Goal:** Validate the end-to-end integration by using the trained model as a drop-in sampler within Optuna. This step will ensure the `OptiFormerSampler` class can generate suggestions during an actual optimization loop and that all components (history tracking, encoding, model inference, decoding) work in a real Optuna study [22].

**Procedure:**

- **6.1 Implement OptiFormerSampler:** Following the design, subclass `optuna.samplers.BaseSampler` to create an `OptiFormerSampler` that wraps our model and tokenizer [22] [23]. Key methods:

- `sample_relative(study, trial, search_space)`: Use the model to suggest a value for a given parameter, given the study's history.
    - Serialize the study's past trials (e.g., `study.trials`) into the token context using the tokenizer (this will involve normalizing and tokenizing each past trial's parameters and outcomes) [24] [25].
    - Append any required prompt token (e.g. a target/regret token or the parameter name token) to the context [10].
    - Do a forward pass as in Step 5 to predict the next token, then decode it to a float value [26].
    - Return that float as the suggestion for the current parameter.
- `sample_independent(study, trial, param_name, param_distribution)`: Fallback for when no history is available. Implement this to simply return a random suggestion (Optuna's default RandomSampler) for robustness [27]. This covers the first trial or any parameter not handled by the relative sampler.

- Ensure the sampler maintains any required state (the model, tokenizer, device) in its constructor [28].

- **6.2 Dry Run on Dummy Objective:** Define a simple test objective function for Optuna (e.g., a known 1-D function like $f(x) = (x-2)^2$ where the optimum is at 2, within [0,5] range). This is just to observe the sampler's behavior.

- Create an Optuna `Study` with our `OptiFormerSampler`. Set a small number of trials (e.g., 10 trials) for this test.

- Run `study.optimize(objective, n_trials=10)`. The expectation is that:

    - On the first trial, `sample_independent` will trigger (no history yet), producing a random value.
    - On subsequent trials, `sample_relative` will be called, using the model's suggestions based on the accumulated history.

- **6.3 Monitor the Suggestions:** As the study runs, log each suggested parameter and the resulting objective value:

- **Validity:** Check that each suggested value is within the defined search space bounds (the sampler's decode/denormalize should ensure this [19]). There should be no out-of-range suggestions or errors.
- **Non-trivial Behavior:** Even with a lightly trained model, see if the suggestions avoid obvious repetition. For example, if the first random trial was very bad, does the second trial suggested by the model differ (indicating it's trying something else)? If a trial achieves a notably better result, do subsequent suggestions cluster around that region (even roughly)? This would mirror the exploitation/exploration behavior in a real scenario.
- **Optuna Integration:** Confirm that the study completes all trials without exceptions. The integration is successful if Optuna seamlessly accepted the sampler's suggestions and the optimization loop ran to completion.

**Expected Output & Criteria:**

- **Successful Run:** The Optuna study finishes 10 trials using the `OptiFormerSampler` without any runtime errors. This proves that our sampler implementation is compatible with Optuna's API and that our model+tokenizer can plug into an actual HPO workflow [22].
- **Correct Suggestion Formatting:** All suggested parameter values were valid floats in the expected range, meaning the encode->model->decode pipeline produced legitimate outputs each time (no decoding failures or type errors).
- **Basic Pattern Reflection:** While the model is not fully trained, it's encouraging (though not required for the smoke test) if the sequence of suggestions shows some adaptation. For instance, if our dummy objective's optimum is at 2 and the first random guess was 4 (with a high objective value), perhaps the model suggests something different like 1 or 3 next (as opposed to repeating 4). After a lower objective value is observed near 2, the model might suggest another value near 2. Any such rudimentary adjustment indicates the model is functioning logically as a sampler. However, given minimal training, the primary **pass criterion** is that the sampler executes and returns *some* reasonable values, not necessarily optimal ones.

**Safety Checks:** If the study fails (exception thrown), examine the stack trace for where in `OptiFormerSampler` the error occurred. Common issues could include: - Mis-serialization of the study history (e.g., our `encode_history` function might have a bug). Ensure it properly handles an empty history (maybe by calling `sample_independent` as needed). - Model inference issues (e.g., if the model wasn't on the correct device or input tensor shape issues). Make sure the model is on CUDA and in eval mode with `torch.no_grad()` during inference to avoid needless grad overhead [17]. - Decode/denormalize issues (e.g., decoding a token that wasn't expected, leading to KeyError). If the model outputs a token ID outside the known range (shouldn't happen if vocab is correct and model is trained at least a bit), clamp or handle unknown tokens gracefully. - After fixing any issues, re-run the short Optuna optimization. This step is passed when the sampler can drive a study without errors and produces plausible suggestions.

---

**Conclusion:** By following this step-by-step smoke test, we validate that each component of OptiFormer-Lite functions correctly on a small scale. We will have confirmed that synthetic data generation produces learnable patterns, the tokenizer reliably maps data to/from tokens, the mini Transformer can be trained (with loss dropping) in a short run, and the model can generate sensible suggestions integrated within Optuna. Meeting the diagnostics criteria (e.g. loss drop, correct tokenization, logical inference behavior) indicates the project is **feasible** on our hardware and ready to scale. If all tests pass, proceed to the full 24-hour pre-training on the large synthetic dataset, followed by fine-tuning on real data, with high confidence in the pipeline's integrity [29].

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

Steps improved.txt

file://file_0000000060ec720c986914ff6f83b233