ChatGPT

Hyena-BLT-Genome Technical Guide

Introduction and Motivation

Genomic sequences pose a unique challenge for AI models due to their sheer length and complexity. The human genome, for example, spans roughly 3.2 billion nucleotide "characters" [1], far exceeding the context lengths of typical language models. Capturing long-range dependencies (such as distant regulatory elements affecting gene expression) requires models that can handle contexts of millions of tokens. Recent advances like **HyenaDNA** demonstrated that long-context models (up to 1 million tokens) at single-nucleotide resolution can achieve state-of-the-art results on dozens of genomics tasks [2]. HyenaDNA and its successor **Evo 2** leveraged the **StripedHyena** architecture to enable such extreme context lengths, showing the promise of alternative architectures beyond Transformers for biological sequences [3]. However, these models still process each nucleotide individually, which can be inefficient for repetitive or low-complexity regions commonly found in DNA.

Meanwhile, in the NLP domain, Meta AI's **Byte Latent Transformer (BLT)** introduced a powerful idea: dynamically merge tokens (bytes) into larger "patches" based on content entropy [4]. In BLT, predictable sequences of bytes are grouped into a single token on the fly, allocating more model capacity to parts of the input that are information-rich and compressing those that are not. This entropy-based token merging allows BLT to match the performance of traditional tokenized LLMs while greatly improving efficiency and scaling [4]. The success of BLT suggests a tantalizing opportunity for genomic modeling: DNA sequences often contain repetitive motifs or low-entropy segments (e.g. long runs of `AAAA...` or common repeats) that could be dynamically compressed, focusing computation on the truly complex regions.

Hyena-BLT-Genome is a hybrid approach that combines the best of both worlds: the long-range, high-speed sequence modeling of StripedHyena with the dynamic token merging of BLT. The goal of this project is to create a genomic foundation model that can model sequences at the scale of entire genomes, but with improved efficiency via entropy-based patching. By integrating **entropy-based dynamic patching** with a **StripedHyena (Savanna)** backbone, we aim to handle ultralong DNA sequences more efficiently than ever, without sacrificing resolution where it matters. In the sections below, we detail the architecture, training strategy, repository structure, and development plan for Hyena-BLT-Genome, providing a comprehensive guide for developers and future contributors.

Core Architecture: Entropy-Based Patching Meets StripedHyena

The **Hyena-BLT-Genome architecture** introduces a hierarchical two-level modeling approach, inspired by BLT's design but tailored for genomic data and powered by a StripedHyena core. The key innovation is entropy-based **dynamic patching** of the input sequence, combined with a scalable long-context **StripedHyena** model as the backbone. Below we break down the main components and how they interact:

- **Entropy-Based Dynamic Patching:** Instead of fixed tokenization, the raw DNA sequence (treated at the byte or character level: A, C, G, T, etc.) is segmented on the fly into variable-length **patches**. A

lightweight *entropy model* computes the surprise (uncertainty) of each next nucleotide in context. When the next base is highly predictable (low entropy), it is merged into the current patch; when it is surprising (high entropy), a new patch is started. This results in long patches for repetitive or low-information regions and short patches for complex regions [4]. The dynamic patching mechanism effectively performs **token merging**, compressing the input sequence length without losing important information. In genomic terms, this means common repeats or homopolymers can be represented as single units, while irregular, information-dense segments get detailed representation. An **entropy threshold** hyperparameter controls the sensitivity of patch splitting – e.g. a threshold can be tuned so that patches have an average length (say 8–16 bases) or to ensure entropy per patch falls below a limit. The patching is *dynamic* per sequence and can adapt to any genome region's content.

- **Local Encoder & Decoder Modules:** Following the BLT architecture, Hyena-BLT-Genome uses two small transformer-based modules at the byte level – a **Local Encoder** and a **Local Decoder** [5]. The Local Encoder takes a sequence of raw bytes (nucleotides) within a patch and produces a latent **patch representation** (embedding). Think of this as compressing a short sequence of, say, 10 nucleotides into a single vector that represents that entire patch's information. The Local Encoder is designed to be lightweight (with only a few layers) to keep the patching overhead minimal [6]. It may incorporate *n-gram embedding* techniques (similar to BLT's hashed byte n-gram features) to efficiently encode common nucleotide motifs or k-mers into the patch representation [5]. For example, a patch containing "ATGCGGG" might benefit from 3-mer embeddings for "ATG", "TGC", etc., to inform its representation.

The Local Decoder performs the inverse operation when generating output: it takes a latent patch representation (from the global model's output) and decodes it back into actual nucleotide sequences [7] [8]. During training (which is autoregressive next-sequence prediction), the Local Decoder uses the predicted patch representation to reconstruct the next patch's sequence of nucleotides and is trained to match the ground-truth sequence for that patch. Both the encoder and decoder use **multi-headed cross-attention** mechanisms to interface with the global model: the encoder uses patch-level queries attending to byte-level keys/values to produce the patch embedding, while the decoder does the opposite (byte-level queries attending to patch-level context) to expand a patch embedding into actual bytes [9] [8]. These cross-attention layers ensure that detailed byte-level information flows between the local and global levels, preserving high-resolution data even though the global model operates on merged tokens [5].

- **StripedHyena Global Backbone:** The compressed sequence of patch embeddings (each representing a variable-length chunk of DNA) is fed into the **Global Model**, which is based on the StripedHyena architecture. StripedHyena is a **convolution- and state-space-based** sequence model that forgoes standard self-attention in favor of more efficient long-range operators [10]. It uses techniques from the Hyena family of models (such as structured convolutions, gating, and state-space models) to achieve excellent scaling with sequence length. In practice, the Hyena-based global model can handle input sequences of hundreds of thousands to millions of tokens with far less memory and compute than a Transformer, while maintaining competitive accuracy [10] [2]. This makes it ideal as the backbone for genomic modeling. In Hyena-BLT-Genome, the global model treats each patch embedding as a "token" in a reduced sequence. Because patching has collapsed many raw bases into one token, the sequence length for the global model is dramatically reduced (potentially by an order of magnitude or more, depending on content entropy). The StripedHyena model then processes this patch sequence to capture long-range dependencies **between distant**

patches – for example, linking a regulatory DNA element 100kb upstream to a gene's promoter patch. Importantly, StripedHyena (especially in its updated **StripedHyena 2** form) has been validated on genomic data up to 1M context [3], so we leverage its proven scalability. The Savanna framework (which powered Evo's pretraining) provides distributed training support for such multi-hybrid models, ensuring we can train this large model efficiently [11] [12].

- **Hybrid Integration via Cross-Attention:** To integrate the local and global components, Hyena-BLT-Genome uses BLT's strategy of *cross-attention bridges*. The Local Encoder's output for each patch is passed to the global model; conversely, when the global model is predicting the next patch, its latent output is passed into the Local Decoder. At these junctions, cross-attention layers allow the patch-level representations to attend back to the raw bytes (in the encoder) or allow the raw byte generation to attend to the patch context (in the decoder) [5] [9]. This design ensures that while the global StripedHyena operates on a high-level representation (patches), the fine-grained nucleotide details are not lost – the local modules can always query the detailed sequence if needed. Effectively, the architecture forms a **U-shape** information flow: bytes -> patch (via local encoder), patch -> global Hyena -> patch (predicted) -> bytes (via local decoder), with skip connections in the form of cross-attention between corresponding byte and patch representations.

- **Handling of Outputs:** In an **autoregressive training regime**, the model proceeds patch by patch. Suppose the input sequence up to position *t* has been processed into patches; the global model produces an output embedding for the next patch *t+1*. The Local Decoder takes that embedding and generates the actual nucleotide sequence for patch *t+1* (one byte at a time, internally) [7]. The training loss is computed at the byte level by comparing the decoder's output sequence to the ground truth nucleotides of that patch (e.g., via cross-entropy on each predicted base). This way, the model learns both to form meaningful patch representations and to predict coherent long-range sequence structure. During generation or inference, the entropy-based patching mechanism would be applied iteratively: as the model generates bytes, we would also dynamically determine when a patch should end (likely using the same entropy model to detect a boundary in generation).

Figure 1: Hybrid Architecture of Hyena-BLT-Genome. The genomic input sequence (e.g., DNA bases) is dynamically segmented into variable-length patches *using an entropy-based algorithm (green hexagons). A lightweight Local Encoder compresses each patch of raw nucleotides into a latent vector representation (blue). These patch embeddings are then fed into the Global StripedHyena Model (orange) which can efficiently capture long-range dependencies across the entire sequence of patches. When predicting the next segment, the Local Decoder takes the global model's output for a patch and reconstructs the actual nucleotide sequence (red). Cross-attention connections (dashed arrows) between the local and global modules ensure information flows from the byte level to patch level and back, preserving fine-grained details [5]. This two-level approach allows the model to allocate more capacity to complex regions while compressing redundant sequence stretches [4].*

Repository Structure and Module Guide

The project's code is organized to clearly separate the concerns of patching, modeling, and training. Below is the planned repository structure, along with the purpose of each major module:

- `hyena_blt/` – *(Python package directory)* Main source code for the Hyena-BLT-Genome model.
- `entropy_patch.py` – Implementation of the entropy-based patching algorithm. This module contains the logic for the *dynamic patch segmentation*. It likely includes a class or function to compute

next-byte entropy given a context (using a small pretrained model or statistical estimator) and to merge bytes into patches accordingly. Key functions: `compute_entropy(sequence_chunk)` and `segment_sequence(sequence)` returning a list of patches with their boundaries.

- `encoder.py` – Definition of the **Local Encoder** model. This includes the lightweight transformer (or alternative) that takes a sequence of byte embeddings (e.g., one-hot or learned base embeddings) and produces a fixed-size patch embedding. It will also implement the encoder-side cross-attention: bytes as keys/values, patch token as query [9]. Likely provided as a PyTorch `nn.Module` class, e.g. `LocalEncoder`, with a configurable number of layers/heads.
- `decoder.py` – Definition of the **Local Decoder** model. This is the counterpart to the encoder, taking a patch embedding (plus, internally, previously generated bytes in a patch for auto-regression) and generating the raw nucleotide sequence of that patch. It uses cross-attention with the patch embedding as key/value and byte representations as queries (the "roles reversed" compared to encoder) [9] [8]. Provided as `LocalDecoder` class. It will output a sequence of nucleotides (during training, this is forced to match the ground truth patch via teacher forcing).
- `hyena_model.py` – (or `global_model.py`) Integration of the **StripedHyena** global model. This may either wrap an existing StripedHyena implementation or build it from primitives (e.g., using the H3 or Hyena library). The global model will likely be configured with the desired number of layers, hidden dimensions, etc., possibly loading predefined kernels (like FlashFFTConv). This module exposes the global model as `GlobalModel` class which accepts a sequence of patch embeddings and returns outputs for the next patch. If using an existing library for Hyena, this could simply initialize that model architecture.
- `model.py` – High-level model definition assembling the encoder, global, and decoder into one end-to-end **HyenaBLTModel**. The `forward` method here orchestrates the full forward pass: given an input sequence, call `entropy_patch.segment_sequence` to get patches, encode patches with `LocalEncoder`, pass through `GlobalModel`, and decode the next patch with `LocalDecoder`. This is also where loss calculation might occur (comparing decoder output with target sequence). It ties everything together for training or inference.
- `train.py` – Training script or module. Handles dataset loading, model instantiation, training loop, and optimization. It likely uses PyTorch Lightning or a custom loop with DeepSpeed (especially if integrating with Savanna or large-scale training). It will parse config files or arguments (like learning rate, batch size, number of GPUs, etc.), prepare the data pipeline, and coordinate the forward/backward passes. For example, `train.py` may define an `Trainer` class or just a `if __name__ == "__main__":` block that sets up training.
- `utils/` – Utility functions (data preprocessing, metrics, etc.). For instance, a `data.py` for loading genomic data (could handle reading FASTA files or encoded genome data, perhaps sharding the OpenGenome dataset if used), or `metrics.py` for computing validation metrics like bits-per-byte (BPB) [13].
- `config/` – Configuration files for experiments (if using a config system). This might include default hyperparameters, model dimension settings, etc., possibly in YAML or JSON.
- `tests/` – Unit tests for components to ensure patching, encoding, decoding, etc., work as expected.
- `docs/` – Documentation files (including this guide, diagrams, etc.). This could contain additional design notes or usage instructions aside from the README.

- `scripts/` – Auxiliary scripts for training on clusters, evaluation, etc. For example, `launch_distributed.sh` or Slurm submission scripts if training on HPC (some of these might come from Savanna if reused).

- `examples/` – Example notebooks or scripts demonstrating how to use the trained model for tasks like generating a DNA sequence or predicting a masked segment.

- `README.md` – High-level overview (we may use this technical guide as `GUIDE.md` and have a shorter README that links to it).

Each module is designed with separation of concerns: e.g., you can test `entropy_patch.py` on its own with some synthetic sequence to see the patches, or test `encoder.py` + `decoder.py` together to ensure you can encode and decode a patch without loss. The integration in `model.py` ensures that all parts talk to each other correctly.

Training Strategy and Flow

Training the Hyena-BLT-Genome model requires coordinating the learning of the local modules (encoder/decoder and entropy model) and the global model. The strategy we adopt is a **modular two-stage training process** followed by joint fine-tuning:

1. **Pre-training the Entropy Model for Patching:** First, we train a small language model on genomic sequences to serve as our entropy estimator for patching. This could be a proportional 5-10 layer Transformer or a small Hyena RNN that reads bytes and predicts the next nucleotide, thereby learning the distribution of genomic sequences. We train this on a large corpus (e.g., the OpenGenome dataset used in Evo 2 [14] or other assembled genomes) to ensure it has a good sense of what comes next in DNA sequences. The result is a model that can output a probability distribution for the next base given preceding context; we use this to compute the **next-byte entropy**. Once this entropy model is trained (or we could initialize it from an existing model like a compressed version of Evo if available), we **freeze it** for use in patch segmentation. This avoids having to update patch boundaries during main model training – the patching acts like a fixed preprocessing guided by a pretrained model. (In future research, one could consider updating this model or training it jointly, but that adds complexity since patch boundaries would shift during training.)

2. **Stage 1 – Training Local Modules Independently (Optional):** An optional step is to pretrain the Local Encoder and Decoder on a reconstruction task. For instance, feed the Local Encoder some random patches and train the Local Decoder to reconstruct them from the encoder's output (essentially an autoencoder on short sequences). Another pretraining idea: train encoder+decoder together to predict the next few nucleotides given a short context, as a way to initialize them with some understanding of nucleotide composition. This stage can ensure that the local modules at least learn basic nucleotide representations (like that "ATG" is a common motif, etc.) before coupling with the global model. However, this step may not be strictly necessary; BLT results indicated even a very small encoder can learn adequately in the full context [6].

3. **Stage 2 – End-to-End Pretraining of Hybrid Model:** With the entropy patching mechanism in place and possibly initialized local modules, we proceed to train the full hybrid model (Local Encoder + Hyena Global + Local Decoder) on next-token prediction across massive genomic data. We use an **autoregressive language modeling objective**: the model is given a long sequence of DNA and is trained to predict the next base (or next patch of bases). Concretely, for each position in the training sequence (or each patch boundary), the model (with current hidden state) predicts the next patch of nucleotides. During training, we generate patches on-the-fly for each training example using the fixed entropy model. The Local Encoder compresses each encountered patch, the global model processes patch representations, and the Local Decoder outputs the next patch's nucleotides, which is compared to the actual next patch in the data via cross-entropy loss. We accumulate the loss over all bytes in the output patch so that longer patches (which are low-entropy easy parts) and shorter patches (high-entropy parts) both contribute appropriately. The training uses standard backpropagation; note that the entropy-based segmentation is not learned via gradient (since the entropy model is fixed), which simplifies training dynamics.

4. **Joint Fine-Tuning:** After the main pretraining, we fine-tune the entire model jointly on specific tasks or to refine the interface between local and global modules. In this phase, we can unfreeze the entropy model if we want the model to possibly adjust patching decisions for the domain of interest (though doing so would require a differentiable or reinforcement learning approach to adjust boundaries, which is advanced and may not be done in initial stages). Fine-tuning could include training on supervised genomic tasks (like variant effect prediction, regulatory region classification, etc.) by adding task-specific heads or prompts while still using the core model weights.

During training, we pay special attention to a few important aspects:

- **Loss Functions:** The primary loss is the autoregressive next-token (next-patch) loss. We compute it at the byte level to properly supervise the Local Decoder's output. Specifically, if the next patch in the sequence is, say, "ACGT", the model must output those four characters exactly. The decoder will produce a probability for each position (with cross-attention to the global context), and we sum the cross-entropy loss over those four positions. This ensures the model not only predicts that some patch should occur but also gets the internal content right. We do not have an explicit loss on the patch representations themselves (they are learned implicitly), but one could consider an auxiliary loss like reconstructing the entire input sequence from patch embeddings to regularize the encoder (however, this might be redundant). Another possible auxiliary objective is to predict patch lengths or entropy values to align the global model's understanding with the patching mechanism, but this is experimental.

- **Patching Threshold and Schedule:** The entropy threshold that determines patch breaks is a crucial hyperparameter. We might start with a relatively low threshold to generate many small patches during initial training (ensuring the model sees fine-grained detail early on). As training progresses or in later curriculum stages, we could raise the threshold to allow longer patches (compressing more) – effectively teaching the model to handle more compression. An alternative approach is to sample patching strategies: occasionally use pure random or fixed-length patches for data augmentation, or vary the threshold per batch, so the model doesn't overfit to one particular segmentation pattern. The BLT paper suggests a procedure to identify an entropy threshold that yields a target average patch length [15] [16]; we can use a similar calibration on a validation set to choose our threshold so that (for example) ~50% of the bases end up in patches of length >1.

Ultimately, the threshold may be tuned to balance performance and efficiency (longer patches = more compression, faster but potentially harder to model precisely).

- **Optimization and Scaling:** We plan to train on powerful GPU clusters, leveraging **Savanna** for distributed training across many GPUs [11]. Savanna provides integration with DeepSpeed (for ZeRO sharding of optimizer states) and efficient parallelism strategies suited for multi-hybrid models (including **a2a/p2p context parallelism** for extremely long contexts) [17]. This will allow us to scale to context lengths of 1 million or more and billions of training tokens. For instance, Evo 2 (a similar architecture without token merging) was trained on 8.8 trillion bases across diverse species [14]; our training will likely also require trillions of bases to capture genomic diversity. We might start with smaller-scale experiments (e.g., training on the human genome or a subset of chromosomes) to validate the model, then scale up to large multi-genome datasets.

- **Training Flowchart:** The overall training workflow is illustrated below. It shows how data flows from raw DNA sequences through patching, into the model, and out to loss computation during training.

Figure 2: Training Pipeline for Hyena-BLT-Genome. In this flowchart, the training process begins with raw genomic sequences sampled from the dataset. The Entropy-Based Patcher (left) processes the input stream of nucleotides, deciding patch boundaries (indicated by segmented blocks) based on the entropy of upcoming bases. These patches are encoded by the Local Encoder into latent vectors (colored squares) which form a much shorter sequence. The Global Hyena Model then processes this sequence of patch embeddings, capturing long-range interactions across the genome. To predict the next portion of sequence, the global output for the next patch is fed into the Local Decoder, which generates the actual nucleotide sequence for that patch (right). During training, the predicted patch sequence is compared against the true sequence (ground truth) for that segment, and a loss is computed (red star) on each predicted nucleotide. The error is backpropagated through the decoder, global model, and encoder, while the entropy-based patching (as a fixed preprocessing step) does not directly receive gradients. This modular training approach allows the local and global models to be trained end-to-end on the next-token prediction objective, after which joint fine-tuning or task-specific training can be applied.

Local Encoder/Decoder Design Choices and Rationale

Designing the encoder and decoder modules involves trade-offs between complexity, capacity, and speed. We explored several candidate architectures for the **Local Encoder/Decoder**:

- **Transformer-Based (Chosen):** We selected a **lightweight Transformer** design for both encoder and decoder, as used in the original BLT. Each is a mini-Transformer with a small number of layers (e.g., 1–4 layers for the encoder, and a slightly larger decoder). The Transformer architecture is naturally suited for sequence-to-sequence mapping and works well with cross-attention integration. This design also allows reuse of well-tested transformer implementations and optimization kernels. BLT's authors found that surprisingly, an extremely small encoder (even just **one layer**) can suffice when augmented with n-gram embeddings, and that it's beneficial to allocate more layers to the decoder for better generation fidelity [6]. Following this insight, our Local Encoder is kept minimal – essentially just enough capacity to pack the local context information into the patch embedding. The Local Decoder is given a bit more depth to adequately reconstruct possibly longer patches (especially because a patch could be, say, 20 bases of a repeat – the decoder needs to correctly output all 20 `A`s, which can be non-trivial if the representation is too compressed). The transformer-based modules handle these requirements with multi-head self-attention (for internal context of a

patch) and cross-attention (to interface with global context), which we found necessary to maintain accuracy.

- **Convolutional or CNN-Based:** We considered using a small 1-D convolutional network for the encoder/decoder, given that CNNs are fast and can capture local motifs (for example, a 1-D CNN with kernel size 3 could easily learn to detect common 3-mer patterns in DNA). A CNN encoder could slide over the patch and produce a combined representation via pooling. However, a purely convolutional encoder/decoder would struggle to interface with the global model unless we added a custom mechanism for injecting the global context. It's simpler to use the standard cross-attention approach with transformers than to hack a CNN to attend to global information. Moreover, a CNN decoder generating one base at a time would need an iterative approach, complicating the design. We ultimately opted not to use CNNs for the main modules, though convolutional filters might still be employed internally (e.g., within Hyena's layers or as part of a hybrid transformer).

- **Recurrent or LSTM-Based:** Another alternative was a recurrent network (like an LSTM) as the local encoder/decoder. RNNs are lightweight and naturally process sequences of arbitrary length, which could be useful for variable patch lengths. An LSTM encoder could compress a patch by reading it sequentially into its final hidden state. However, RNNs have limitations in representational power and parallelizability. Given modern transformer superiority and the need to handle cross-attention with the global model, we steered away from RNNs for the core design. Nonetheless, the *entropy model* that drives patching could potentially be an LSTM or small RNN, since it only needs to estimate next-token probabilities and is not in the main gradient path.

- **Hash Embeddings & K-mer Features:** Regardless of the above choices, one trick from BLT we adopt is the use of **byte-level n-gram hashing** to enrich the input embedding [18] [19]. In genomics, this is analogous to providing k-mer context. For example, we can hash subsequences of length 3-8 (k-mers) within a patch to generate additional embedding features that are fed into the Local Encoder. This helps the encoder quickly identify common sequences (like known promoter motifs or repeats) without needing huge dimensionality. The BLT paper showed that with hashed n-gram vocabularies, even a very light encoder can perform well [6]. We plan to incorporate a similar mechanism: e.g., before the encoder's self-attention, we concatenate or add the embeddings of several hashed k-mers present in the patch to the sequence representation. This gives the local encoder a rich starting point.

Rationale for Retraining & Tuning: The Local Encoder and Decoder are trained from scratch on genomic data, as their weights will specialize to the nucleotide distribution and the kind of patterns present in DNA (which differ from natural language). If we had started from a BLT model trained on text, we would still need to retrain these modules because genomics has a tiny vocabulary (4 bases) but very different higher-order statistics (e.g., palindromic sequences, GC-rich vs AT-rich regions, etc.). During pretraining, the local modules learn to compress and reconstruct those patterns in tandem with the global model learning long-range structure. We anticipate iterating on these modules' design as we progress: - We might experiment with increasing the decoder's depth if we find the model struggling to regenerate long patches accurately. - If the encoder is too shallow and fails to capture slightly larger motifs, we could add an extra layer or a wider feed-forward dimension. - Conversely, if the encoder is overfitting or too slow, we might prune it down further. The modular design allows swapping implementations: for instance, trying a single-layer transformer vs. a two-layer one easily. - Retraining the local modules (with the global fixed) can be a useful strategy in later fine-tuning. For example, if we fine-tune the global model on a species-specific genome, we

might also fine-tune the encoder/decoder so that the patch representations adjust to any shift in local distribution (say, a particular species might have more frequent repeats, so perhaps adjusting how those are encoded could help). - We will carefully monitor the **patch reconstruction accuracy** (i.e., how often the decoder perfectly reconstructs the true patch) during training. If this lags behind the global modeling accuracy, it indicates the decoder might need more capacity or the encoder is losing information.

In summary, our design favors a minimal but effective transformer-based approach for local modules, taking cues from BLT's findings and adapting them to the DNA domain. This should provide a strong starting point, and we remain open to revising the architecture as we validate performance on real genomic data.

Development Roadmap

Building Hyena-BLT-Genome is an ambitious project. We outline a development roadmap with milestones to organize the work and track progress:

1. **Milestone 1: Patching Prototype and Base Infrastructure** – *Timeline: Week 0-2.* Set up the repository structure and implement the entropy-based patching module (`entropy_patch.py`). This includes training or integrating a simple entropy model for bytes (initially, we might use a pre-trained character-level language model on DNA or train a small one on a subset of genome data). Test the patching on sample sequences (e.g., synthetic DNA with known patterns) to ensure it produces sensible variable-length patches (e.g., a long run of `N`'s merges, a random sequence breaks into many small patches). Also, define data loading utilities for genomic data (reading FASTA, etc.) and ensure we can stream sequences for training. Milestone 1 is achieved when we can take an input DNA sequence and get a list of patch tokens out, reliably and efficiently.

2. **Milestone 2: Local Encoder & Decoder Implementation** – *Timeline: Week 2-4.* Implement the `LocalEncoder` and `LocalDecoder` classes (transformer-based). Write unit tests for these: for example, ensure that encoding followed by decoding of a patch (with the same input fed) can reconstruct the input (when the global model is identity or not involved). This tests the basic autoencoding capability. Experiment with simple training of encoder-decoder on short sequences (like autoencoding 10-base sequences) to initialize their weights. By the end of this stage, we should have working local modules that can be integrated.

3. **Milestone 3: Global Model Integration (StripedHyena)** – *Timeline: Week 3-6.* Integrate the StripedHyena architecture. This could involve using the existing StripedHyena 2 code (possibly as a Git submodule or a library) or implementing a simplified version if needed for prototyping. Ensure that the global model can accept an arbitrary sequence length of patch embeddings (we might start with modest lengths for debugging). At this point, assemble the full pipeline in `model.py`: the forward pass that goes from bytes -> patches -> patch embeddings -> global model -> next patch embedding -> decoded bytes. Run a few end-to-end forward passes with dummy data to verify shapes and data flow. Milestone 3 is complete when the integrated model can take, say, a 1000-base sequence and predict some next bytes (randomly initialized prediction) without crashing or shape mismatches.

4. **Milestone 4: End-to-End Training on Small Scale** – *Timeline: Week 6-10.* Begin training the full model on a smaller dataset or shorter sequences to ensure everything trains properly. For example, use a single human chromosome or a microbiome genome (~10 million bases) to train on, with a reduced model size (maybe a few layers global, small embedding sizes) just to observe learning. Monitor training loss to see if the model is actually learning next-token prediction. During this phase, tune hyperparameters like the entropy threshold (maybe start with a fixed patch max length to simplify, then introduce entropy-based splits gradually). Evaluate on held-out sequences whether the model can continue sequences realistically. Achieving this milestone means we have a proof-of-concept model that learns something (e.g., it should produce outputs with realistic nucleotide distribution and maybe some simple repeats or motifs correctly).

5. **Milestone 5: Scaling Up Pretraining** – *Timeline: Week 10 onward.* Using the insights from the small-scale runs, scale up to the full dataset and model size. This involves:

6. Expanding the model to the target parameter count (e.g., 7B or 40B parameters as in Evo models, depending on our compute resources).
7. Increasing context length gradually up to the full 1M (possibly by curriculum: train at 100k, then 300k, etc., to stabilize optimization).
8. Running on a multi-GPU cluster with distributed training (DeepSpeed ZeRO, etc.). At this stage, we engage Savanna if available: adapt its training scripts (`train.py` or Slurm scripts) to our model configuration so we can leverage multi-node training easily [17].
9. Monitoring throughput and optimizing (e.g., adjusting batch sizes, sequence lengths for efficiency, ensuring memory fits, etc.).

10. This stage will be the most time-consuming, potentially running for multiple weeks to ingest trillions of bases. We will checkpoint the model periodically (possibly following Chinchilla scaling laws to determine training steps per parameter/token ratio). Milestone 5 is essentially the core pretraining completion: having a trained Hyena-BLT-Genome model that has seen a vast amount of genomic data.

11. **Milestone 6: Evaluation and Fine-tuning** – *Timeline: After pretraining.* Evaluate the pretrained model on various tasks to gauge its capabilities:

12. Zero-shot or prompt-based tasks: e.g., ask it to continue a sequence, or predict missing sequences between known flanking regions, etc.
13. Fine-tune on downstream tasks: For example, integrate a linear classifier on top of the model for enhancer prediction, or fine-tune it on masked language modeling if we want to test reconstruction abilities.
14. Compare performance with baseline models (like Nucleotide Transformer, Enformer, or Evo 2 on tasks if possible).

15. Fine-tune the model on specialized data (like epigenomic data if doing multimodal, see Future Directions). This milestone ensures the model is not just trained, but also useful and validated.

16. **Milestone 7: Repository Release and Documentation** – *Timeline: Ongoing, complete by project end.* Throughout development, maintain updated documentation (like this guide). Create example notebooks (`examples/`) demonstrating how to use the model: e.g., a notebook showing how to generate DNA sequences with prompts, how to extract embeddings for a sequence, or how to fine-

tune on a small task. Polish the README with quickstart instructions. Finally, release the code and (if possible) some pretrained weights on a platform like Hugging Face for community use.

Each milestone builds on the previous, and our development approach remains iterative. We anticipate revisiting earlier components as we progress (for instance, going back to tweak the patching once we see how it behaves in full training). By following this roadmap, new contributors or AI agents can easily see what has been done and what's next, and jump into the project at any stage.

Setup, Training, and Testing Examples

This section provides example commands and code snippets to help developers get started with the Hyena-BLT-Genome project. We assume a Linux environment with Python 3.11+ and at least one CUDA-compatible GPU (NVIDIA A100/H100 recommended given the model size). Before running any training, ensure you have the necessary data (e.g., a corpus of genomic sequences) available or specified in the config.

Installation and Environment Setup

First, clone the repository and install the required packages. We recommend using a conda environment:

```
# Clone the repository
git clone https://github.com/YourOrg/hyena-blt-genome.git
cd hyena-blt-genome

# (Optional) Create a conda environment
conda create -n hyenablt python=3.11 -y
conda activate hyenablt

# Install dependencies
pip install -r requirements.txt
```

The `requirements.txt` likely includes libraries such as PyTorch (with CUDA support), DeepSpeed (if using), and any other needed packages (transformer engine for FP8, etc.). If using **TransformerEngine** for faster matrix ops on Hopper GPUs, ensure the hardware is compatible (H100 or equivalent) [20].

If integrated with Savanna, one might need to install it or ensure the submodule is initialized:

```
# If Savanna is included as a submodule for training:
git submodule update --init --recursive
```

Preparing Data

Make sure you have access to a dataset of genomic sequences. For example, if using the **OpenGenome2** dataset [14], download it or have the path ready. Alternatively, use any large FASTA files or a collection of genomes.

You might need to preprocess the data into a binary format or shard it for performance (Savanna supports splitting data across nodes). For initial tests, you can use a smaller sample:

```
# Example: download a reference genome (e.g., human chromosome 1) for quick test
wget http://hgdownload.soe.ucsc.edu/goldenPath/hg38/chromosomes/chr1.fa.gz -O
data/chr1.fa.gz
```

Our training script will likely accept an argument or config entry for the data path (for instance, a text file listing all sequences or a folder). Ensure this is pointed correctly.

Running Training

To kick off training, use the `train.py` script. You can specify configurations either via a config file or command-line arguments. For example:

```
# Example training command (single-node, single-GPU for demonstration)
python train.py --config configs/pretrain.yaml --data data/chr1.fa.gz --output
runs/test-run/ --context_length 100000 --batch_size 1
```

In this example, `pretrain.yaml` might define the model architecture (layer sizes, etc.) and training hyperparameters. We override some parameters just for demonstration (like context_length). The script will initialize the model and begin training on the provided data.

For multi-GPU training (distributed data parallel or model parallel), you might use DeepSpeed or PyTorch's launcher. For instance, with DeepSpeed:

```
deepspeed --num_gpus 8 train.py --config configs/pretrain.yaml --deepspeed
configs/deepspeed_config.json
```

(Savanna would handle launching across many nodes with its own scripts, which we would document separately in `scripts/`.)

During training, checkpoints will be periodically saved (e.g., in `runs/test-run/checkpoints/`). Monitor the console or log files for training loss and, if implemented, validation loss on a held-out set of sequences. We expect to see the training bits-per-base (or perplexity) decrease over time, indicating the model is learning to predict genomic sequences better.

Resuming Training

If training is interrupted or when moving from stage to stage, you can resume from a checkpoint:

```
python train.py --config configs/pretrain.yaml --resume runs/test-run/
checkpoints/epoch_10.ckpt
```

This will load the model and optimizer state from the checkpoint and continue training.

Testing and Generation

After training (or during, if you want to test intermediate models), you can use the model for inference. There might be a separate script or interactive prompt for generation. For example, suppose we have a script `generate.py` for autoregressive generation:

```
# Generate a continuation for a given input sequence
python generate.py --model runs/test-run/checkpoints/final_model.pt --prompt
"ACGTACGT..." --max_length 1000
```

This would load the trained model and print out a generated sequence of up to 1000 bases that follows the given prompt. Under the hood, the generation process uses the entropy-based patching to determine patch boundaries as it generates new bases, ensuring consistency with how the model was trained (so it will merge low-entropy outputs into longer patches on the fly).

For a programmatic interface, the core model might be used as follows in Python:

```
from hyena_blt.model import HyenaBLTModel

# Load the trained model (this could also be via torch.load if it's a state
dict)
model = HyenaBLTModel.load_from_checkpoint("runs/test-run/checkpoints/
final_model.pt")
model.eval().to("cuda")

# Provide an input DNA sequence
prompt = "ATGCGTACGTACG"  # example prefix
generated = model.generate(prompt, max_length=500)
print("Generated continuation:", generated)
```

The `model.generate` method would handle the iterative patch generation (likely using greedy or sampling strategies for picking the next base). We will include options for nucleus sampling or temperature scaling when generating sequences, as these can be useful to introduce diversity if treating the model as a generative tool for DNA design.

Evaluation on a Task

If we fine-tuned the model for a task (say, predicting regulatory region activity), we would provide an example of running evaluation. For instance:

```
python eval_task.py --model path/to/finetuned_model.pt --task chromatin_profile
--data data/task_dataset.csv
```

This would output metrics like ROC AUC, etc., depending on the task. (The specifics would be developed when we approach those tasks.)

For now, as a language model for DNA, a basic evaluation is to compute the bits-per-byte (BPB) or perplexity on a held-out sequence. We can do this with a script:

```
python evaluate_bpb.py --model runs/test-run/checkpoints/final_model.pt --data
data/validation_sequence.fa
```

This would use the model to calculate the cross-entropy loss on the validation sequences and report BPB (bits per nucleotide), which is analogous to perplexity for byte-level models [13].

All these commands and code snippets are designed to make it easy for developers to train and use the Hyena-BLT-Genome model. More detailed documentation will be provided in the `README.md` or `docs/` for each script and module.

Future Directions

The Hyena-BLT-Genome project opens up many exciting avenues for extension and improvement. Here we outline a few **future directions** and ideas that could be explored:

- **Multimodal Genomics (Sequence + Epigenetics):** Genomic sequence is just one aspect of DNA. Important biological insights come from combining sequence with epigenetic signals (such as DNA methylation, histone modifications, chromatin accessibility) or gene expression data. A future model could extend the architecture to accept not just the nucleotide sequence but also parallel tracks of signals. For example, alongside the sequence bytes, patches could carry associated vectors representing epigenetic features in that region. The model (perhaps with minor modifications to the encoder/decoder to handle multiple modalities) could then learn from sequence + signal jointly. This multimodal approach could improve tasks like regulatory element prediction or variant effect prediction, where sequence context *and* epigenetic context matter.

- **Graph-based Genome Representations:** As genomic research moves beyond a single reference genome, there is interest in **pangenome graphs** – where multiple genomes are represented in a graph structure with variant paths. Integrating the Hyena-BLT model with graph representations could be a compelling direction. One idea is to use the model's long-range capability to handle sequences extracted from different graph paths (representing different individuals or species) and possibly incorporate graph traversal logic into patching. Alternatively, the model's output could be conditioned on positions in a genome graph. This would enable **variant-aware sequence modeling**, where the model can follow different genomic paths and still maintain context. It might require adding gating to the model to handle branches, or an encoding of node connections in the patches.

- **3D Genome (Spatial Genomics Integration):** The linear sequence is only part of the story; DNA in the cell has a 3D organization (chromosome conformation). Future extensions might feed the model information about spatial proximity of regions (for instance, via an adjacency matrix of contacts from Hi-C data). A creative use of Hyena's convolutional nature could allow certain filters to operate over not just linear distance but over a graph of 3D contacts. Such a model could potentially read an entire genome and reason about which parts physically interact.

- **Variant Calling and Genetic Analysis:** The current model is generative (predict the next base), but with fine-tuning it could be applied to variant calling – identifying differences from a reference given sequence evidence. For example, by inputting sequencing reads (with IUPAC ambiguity codes for variants) and having the model output the true genotype sequence, possibly treating it as a sequence-to-sequence problem. The Hyena-BLT-Genome architecture might be well-suited to handle the long-range correlations in reads (or across phased haplotypes) needed for structural variant calling. Additionally, the model's understanding of genomic language might help prioritize which variants are benign or functional by modeling them in context (this drifts into the territory of predictive genomics).

- **In-context Learning for Genomics:** Similar to how large language models can perform tasks with prompts, a future direction is to utilize the long context to do **in-context learning** on genomic tasks. For example, we could feed the model a long prompt that includes a few examples of sequence annotations (like "Sequence -> property") and then have it predict the property for a new sequence. HyenaDNA already hinted at in-context learning with tuneable soft prompts. We could explore prompt-based queries like *"Given this whole genome and an annotation question, highlight the regions of interest"* to see if the model can zero-shot answer questions about the genome.

- **Adaptive Patching via Reinforcement Learning:** Our current entropy-based patching uses a fixed model to decide merges. One advanced direction is to make the patching decision part of the model's learned parameters – for instance, using a reinforcement learning approach where the model can decide to merge or split patches to improve some end-task reward (like minimizing perplexity or maximizing downstream task accuracy). This could involve training an agent that adjusts the entropy threshold or directly chooses patch breakpoints, possibly using policy gradients. If successful, the model might learn an even more optimal compression strategy tailored to genomic structure (e.g., maybe it learns to always isolate splice junctions as separate patches because they are important).

- **Scaling and Model Variants:** We plan to scale to large models (tens of billions of parameters) as resources allow. Future developers might attempt a **40B parameter Hyena-BLT model** or beyond, which, combined with patching, could effectively model entire chromosomes or even genomes in one forward pass. Also, exploring **smaller efficient models** for edge cases (like a 500M parameter model that can run on a single GPU for quick analyses) could broaden the utility. Quantization (e.g., 4-bit or 8-bit weights) and distillation of the big model to a smaller one are practical avenues for deployment.

- **Cross-Species and Evolutionary Modeling:** Since Evo was trained on genomes across all domains of life [14], our model could also ingest diverse species. A fascinating direction is to use the model to study evolutionary relationships: e.g., prompting the model with a sequence from an unknown organism and seeing which known sequences it predicts next (implicitly clustering it with its closest

relatives). The model might also be used generatively to **design novel DNA sequences** (within the bounds of biological realism) – for instance, proposing a DNA sequence with a desired protein-coding function or with regulatory properties, guided by the patterns it has learned from nature. This could have applications in synthetic biology and bioengineering.

We believe Hyena-BLT-Genome represents a significant step towards **scalable, efficient genomic foundation models**. By carefully documenting the design and providing a clear modular structure, we hope this guide enables developers and AI collaborators (like future instances of ChatGPT or other code-generation agents) to easily pick up the project, reproduce our results, and contribute new ideas. With its combination of dynamic token merging and a powerful long-range backbone, Hyena-BLT-Genome is poised to tackle the vast sequences of the genome, and we are excited to see how it evolves and what discoveries it may unlock in the world of computational genomics.

---

1 2 HyenaDNA: learning from DNA with 1 Million token context · Hazy Research
https://hazyresearch.stanford.edu/blog/2023-06-29-hyena-dna

3 14 20 GitHub - ArcInstitute/evo2: Genome modeling and design across all domains of life
https://github.com/ArcInstitute/evo2

4 5 6 7 8 9 13 15 16 18 19 Byte Latent Transformer: Patches Scale Better Than Tokens
https://arxiv.org/html/2412.09871v1

10 Paving the way to efficient architectures: StripedHyena-7B, open source models offering a glimpse into a world beyond Transformers
https://www.together.ai/blog/stripedhyena-7b

11 12 17 GitHub - Zymrael/savanna: Pretraining infrastructure for multi-hybrid AI model architectures
https://github.com/Zymrael/savanna