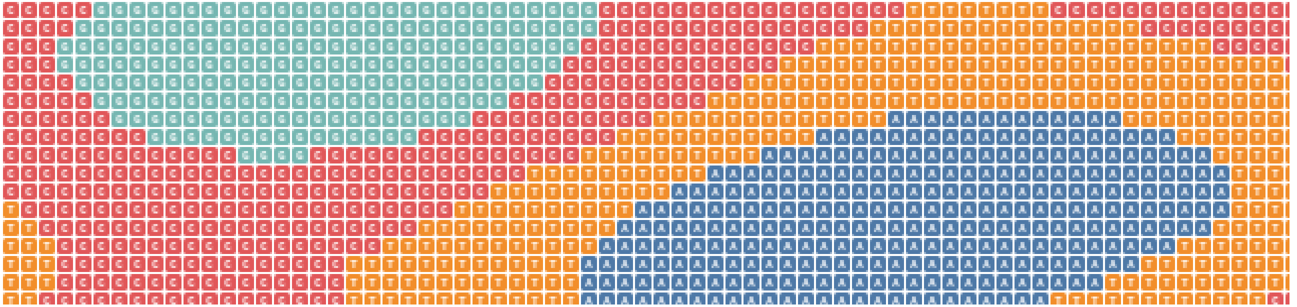
Tokenization with Character Resolution



Abstract visualization of the four base pairs A, C, G, and T that make up all of our DNA.

AUTHORS

Qiuyi Li[2], Leandro von Werra[1]

AFFILIATIONS

1. Hugging Face
2. GenerTeam

PUBLISHED

June 9, 2026

Table of Contents

To BPE or not to BPE
- Conceptual Trade-offs
- Experimental Data
- Biological Considerations

Factorized Nucleotide Supervision
- FNS Code Deep Dive
- Inference with FNS
- Training FNS Loss
- FNS for Sequence Scoring
- Try it out

Conclusion

Resources

DNA language models borrow many ideas from large language models. Both operate on sequences, both use tokens, and both can be trained by predicting what comes next. But DNA is **not** natural language, and there are some significant differences. Not all design choices that work well for English or code automatically translate to genomic sequences. On the other hand, the special structure of DNA sequences may give us some advantage over dealing with natural language text.

One of the most important choices is **tokenization**: how we turn a DNA sequence into tokens that a model can read, score, and generate. Coming from natural language, the naive way to go about tokenization is to apply BPE tokenization. If you also immediately thought about BPE, don't be offended by the use of the word naive! We've been there, we thought the same, and we tried to tame DNA sequences in various BPE experiments. Unfortunately, we failed.

In this blog post we want to share what we learned from these experiments and what we settled on in the end for the tokenizer of the Carbon DNA model: a custom tokenization scheme that chunks sequences with equal length (6 characters in our case or also called 6-mers in bio lingo) and a powerful method we discovered that enables us to maintain single character resolution in the loss during training and at inference time called **Factorized Nucleotide Supervision (FNS)**.

Some of the aspects we'll explore are specific to DNA sequences, others may apply to other settings as well. Let's dive into it by exploring the elephant in the room: why not just use BPE?

To BPE or not to BPE

If you are reading this blog, chances are you come from an LLM background. As such, you might think: "Why not just use BPE?". And you'd have good reason for that. BPE has been battle tested in many language models, and despite efforts to come up with smarter ways to tokenize sequences, we haven't made it far beyond the original algorithm. Instead of changing the core algorithm, we have optimized parts of it in specific settings:

- **Math:** There has been a long debate around how to best tokenize numbers to get the best math performance out of language models. Initially (GPT 1-3) didn't discriminate numbers from alphabetic characters and applied arbitrary splits based on frequency. Later, GPT-3.5/4 started to systematically split 1-,2-, and 3-digit numbers as tokens, left to right, while Llama and PaLM tokenized numbers always into single digits. Later work found that right-to-left chunking in triplets works better (e.g. `10000000` becomes `10|000|000` instead of `100|000|00`), as it naturally aligns with arithmetic operations.
- **Code:** Also, coding has been an area of interest for tokenization optimizations. Code has a lot of repeating patterns even across whitespaces (e.g. `import numpy as np` as well as many different patterns for whitespaces (especially in Python). While merging across whitespaces typically didn't help, making sure the whitespace patterns are well represented helps. Also, for code generation, people found that with a tokenization trick, you can also do fill-in-the-middle with a strictly left-to-right autoregressive language model.
- **Multilingual:** Balancing a tokenizer to represent common as well as rare languages fairly is a tricky undertaking. The statistical nature of BPE naturally adds more tokens for frequent languages and thus tokenizes them more efficiently, while rare language tokens are underrepresented and thus smaller fragments

lead to less efficient tokenization. One can balance the tokenizer vocabulary a bit by either sampling more evenly during tokenizer training.

So BPE has been battle tested in LLM training runs on a wide scale and generally works across different texts quite well, so naturally, we want to throw it at DNA sequences, too. Let's have a look at some conceptual considerations when choosing a tokenizer for DNA sequences.

Conceptual Trade-offs

As an alternative to BPE, we could also create a simpler tokenizer that just uses fixed size chunks (in biology chunks of k base pairs are called a k-mer). Let's have a look at how each of those tokenizers split a DNA sequence. In the widget below, click on the Tokenize button to see how each tokenizer differs::

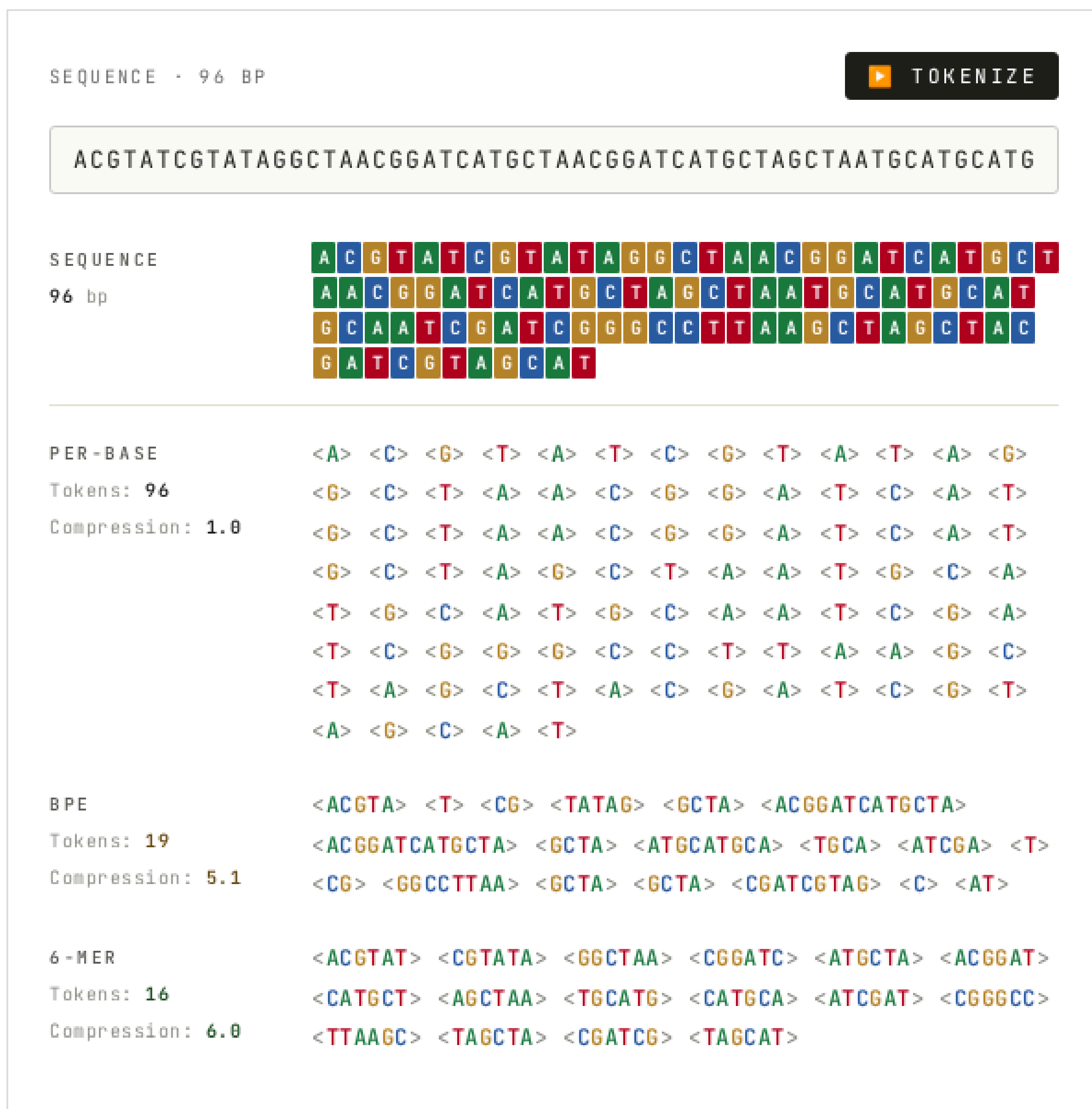


Figure 1 · Tokenization Strategies A demonstration of single-nucleotide tokenization, BPE, and 6-mer tokenization on the same DNA sequence.

Clearly, the 1-mer tokenizer is the least efficient, and while the BPE tokenizer can be more dynamic by having variable length tokens, in this case, the static 6-mer tokenizer wins. But does that also translate to performance gains when training a model with such a tokenizer?

Experimental Data

In ML, the only way to really know what works and what doesn't on real data is to actually train a model. So we trained a bunch of small models and evaluated their performance. The following plot shows the performance on a popular DNA benchmark with different kinds of tokenizers:

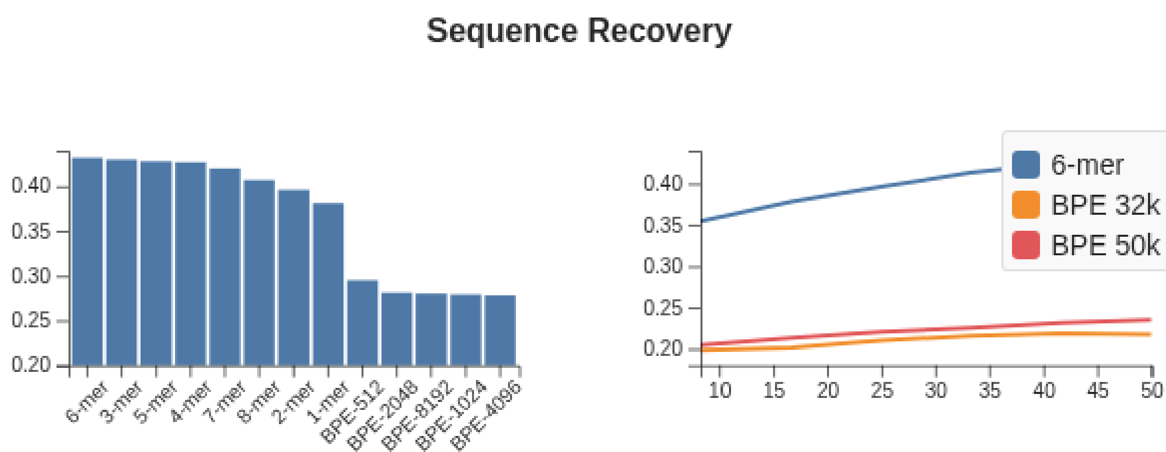


Figure 2 · Comparing Different Tokenizers on the Sequence Recovery Benchmark. Under the same training budget, fixed-length k-mer tokenizers generally outperform BPE variants, with 6-mer achieving the best sequence recovery. Among k-mer tokenizers, 1-mer lags behind longer k-mers in this comparison. BPE tokenizers of varying vocabulary sizes consistently show lower recovery accuracy.

This data comes from experiments conducted as part of the GENERator (left) and Carbon (right) models. Notably, BPE suffers from noticeable performance degradation compared to the simple fixed-length chunking approaches. And under the same training budget, 6-mers appear to perform better than 1-mers among k-mer tokenizers.

There is a reason why BPE might be performing so badly on DNA sequences. DNA sequences are very long without natural chunks (as whitespaces), so there is inherently a **prefix ambiguity.** When BPE segments a sequence, multiple vocabulary entries may share the same nucleotide prefix at any given position. All of them extend the DNA correctly, but teacher forcing only accepts the exact to-

ken from the original segmentation. This is the prefix ambiguity: a prediction can be locally correct in DNA sequence space while being wrong in token space.

Here's an example:

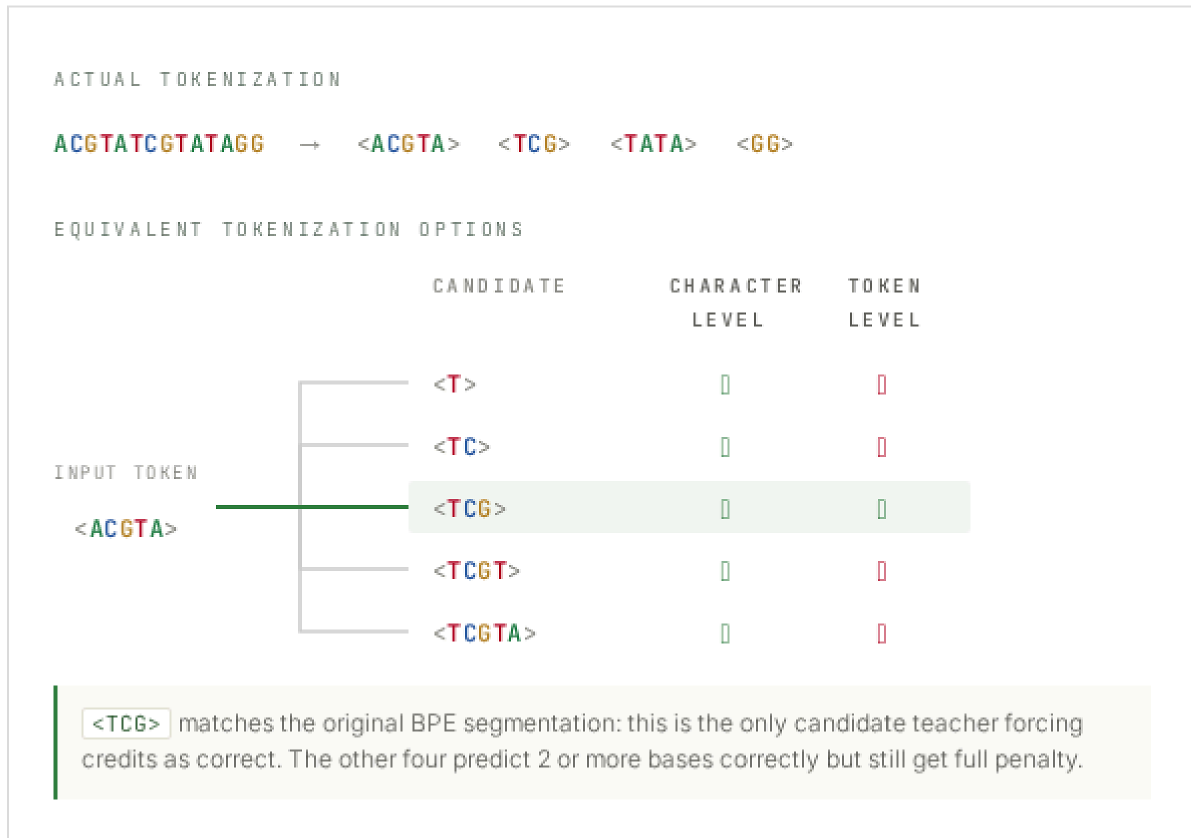


Figure 3 · Prefix Ambiguity of BPE Tokenization Line 1 shows how BPE segments the full sequence. Line 2 zooms into the step right after <ACGTA>: the same DNA prefix can be extended by five different vocab entries, but only one matches the original segmentation.

So clearly every candidate got at least one or more bases right, and in the first three cases, all bases in the predicted token are correct. But the issue is that the model predicted a different continuation after the correctly predicted bases, thus changing the tokenization entirely. When training a model with this tokenizer, it would get any learning signal for all but one choice. So based on these experiments, should we just train with a 6-mer tokenizer?

The advantages over BPE are convincing: Every six consecutive bases become one token. This is not a biological claim about DNA organization, but it's a **deterministic compression unit**. Each autoregressive step predicts exactly one six-base block.

Compared with single-nucleotide tokenization, 6-mer tokenization delivers compound speedups:

- **6× reduction in sequence length** — a 6,000 bp sequence becomes 1,000 tokens instead of 6,000
- **36× reduction in attention cost** — Transformer attention scales quadratically, so 6× shorter sequences yield 36× faster computation
- **6× faster generation** — each autoregressive step generates 6 bases instead of 1, requiring 6× fewer decoding steps

In addition, when compared with BPE, it:

- Keeps the next-token target stable and unambiguous
- Eliminates prefix ambiguity
- Makes training more stable

So again, should we just train with a 6-mer tokenizer? The answer is a "yes, but...". Let's see why a simple 6-mer tokenizer doesn't fully reflect the biological reality.

Biological Considerations

So the conceptual analysis and experiments indicate that training with a 6-mer tokenizer would be the best. But first, let's step back and ask a deeper question: **why doesn't BPE work for DNA from a biological perspective?**

Natural language, code, and math are designed to communicate information. Fixed letter combinations carry stable, unambiguous meanings. BPE thrives there because it discovers these meaningful units.

DNA is different. It wasn't designed, but it evolved through billions of years of trial and error. The result is a sequence full of redundancy, neutral drift, and noise. BPE's attempt to force DNA into "meaningful units" is a strong prior and, unfortunately, likely a wrong one.

Consider an alternative philosophy: **neutral tokenization**. Don't impose biological meaning at the tokenizer level. Let the model figure out what matters. This is exactly what protein language models do: they tokenize at the amino acid level (1-mer), and there's virtually no debate. It's the consensus.

For DNA, the natural extension would be 1-mer tokenization (single nucleotides). It gives us **single-nucleotide resolution**. This is essential for tasks like modeling SNPs (single base changes), splice-site mutations (disrupting critical dinucleotides), codon-level alterations (changing an amino acid or introducing a premature stop codon), or regulatory variants (altering a transcription factor binding site). The model can reason at the level of individual bases without baked-in assumptions.

But there's a problem: DNA is long. A typical human gene (e.g. *CFTR*) can span over 190,000 base pairs. That's 190,000 tokens for a 1-mer tokenizer, just for a single gene. Compare that to proteins, where a typical sequence is just a few hundred amino acids. 1-mer DNA tokenization is computationally brutal: it kills **long-context modeling** and **computational efficiency**. And we need that long context: distal enhancers regulate genes across large distances, gene neighborhoods carry functional organization, chromatin structure connects distant loci, and evolutionary constraints only become visible when enough surrounding sequence is available.

So we face a tension:

- **Single-nucleotide resolution** → 1-mer
- **Long-context + efficiency** → 6-mer

We want both. A pure 6-mer tokenizer gives us speed and length, but loses single-nucleotide resolution. We need a way to have our cake and eat it too.

Let's see how we can marry the two requirements by creating a 6-mer tokenizer with single base resolution!

Factorized Nucleotide Supervision

So our 6-mer tokenizer predicts DNA in 6-mer tokens. At each auto-regressive step, the model outputs logits over all 4,096 possible 6-mers. This is efficient, but many biological tasks need **base-level probabilities**. For example, we may want to know:

> *What is the probability that the first base of the next 6-mer is A?*

FNS answers this by **marginalizing the 6-mer distribution**. Instead of asking for the probability of one complete 6-mer, we sum the probabilities of all 6-mers that share the same base at the position we care about.

GROUND TRUTH: A C G T C G

$p^{(1)}(A) = \Sigma_* \, p($ A G C A A G $)$

$p^{(2)}(C) = \Sigma_* \, p($ G C G A T C $)$

$p^{(3)}(G) = \Sigma_* \, p($ G C G T A G $)$

$p^{(4)}(T) = \Sigma_* \, p($ T T G T C A $)$

$p^{(5)}(C) = \Sigma_* \, p($ G T A C C A $)$

$p^{(6)}(G) = \Sigma_* \, p($ A C T T G G $)$

$$\text{loss} = -{}^{1}/_{6}$$
$$\log\left(p^{(1)}(A)\, p^{(2)}(C)\, p^{(3)}(G)\, p^{(4)}(T)\, p^{(5)}(C)\, p^{(6)}(G) \right)$$

Reading the notation. The model only ever predicts whole 6-mers, so it never directly reports a probability for a single base. $p^{(1)}(A)$ is what we want: the probability that position 1 of the next 6-mer is **A**. We recover it by *marginalizing*, summing the probability of every 6-mer that carries **A** in slot 1 while the other five positions range over all bases (the shuffling boxes, written Σ_*).

Computing the loss. Doing this at each slot gives six per-position probabilities, one for each base of the ground-truth 6-mer. Their product is the model's probability of getting *every* position right; taking `-log` turns it into a loss, and dividing by 6 averages it per base. That is the `loss = -¹/₆log(…)` expression above. Because the credit is split across positions, a near-miss like TATATT still scores well for the five bases it got right, unlike plain 6-mer cross-entropy, which treats any imperfect token as equally wrong.

Figure 4 · Factorized Nucleotide Supervision (FNS). The model scores the whole 6-mer at once, but FNS reads it one position at a time: for each slot, sum every 6-mer that carries the observed base there. Edit the ground-truth 6-mer to watch the factorization and the loss update.

So we can, on the one hand, assign a probability to each base rather than just tokens, and, on the other hand, smooth the loss by assigning partial credit for partial overlap between tokens and

the ground truth.

This seems really cool, so you may wonder: why haven't we been doing this for natural language all along? There's a simple combinatorial reason. For DNA, we have only 4 atomic units, whereas for a byte-level BPE tokenizer, you have 256. So for a 6-character chunk, the vocab sizes are:

- DNA vocab:
- Byte-level BPE vocab:
- And even with just 26 characters:

So the vocab of a 6-character tokenizer for natural language would just be prohibitively large for practical applications. Glad we didn't tokenize natural language wrong all along!

But let's turn back to FNS now. To understand how FNS works in detail, lets work through some actual code.

FNS Code Deep Dive

Let's walk through how FNS works by walking through the code step-by-step. Since the Carbon models are natively integrated in transformers, we can easily load them. If you use `revision="fns"` you will also get all the helper functions built to make FNS usage seamless. The following code simply loads the model, feeds a sequence, and then gets all logits for the next k-mer prediction:

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer


model_id = "HuggingFaceBio/Carbon-3B"
revision = "fns"
device = "cuda"

tokenizer = AutoTokenizer.from_pretrained(
    model_id,
    revision=revision,
    trust_remote_code=True
)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    revision=revision,
    trust_remote_code=True,
    dtype=torch.bfloat16,
).to(device).eval()

dna_prompt = "TCGGATTAGCTGGAAAAATTGCTCCGGCAGCTCGCGGAGCTGGTTGGC"
prompt = f"<dna>{dna_prompt}"

inputs = tokenizer(
    prompt,
    return_tensors="pt",
    add_special_tokens=False,
).to(device)

with torch.no_grad():
    out = model(**inputs)
    next_logits = out.logits[:, -1, :]  # logits for the next 6-mer token

kmer_logits = next_logits[:, model._kmer_ids]           # [1, 4096]
kmer_probs = torch.softmax(kmer_logits.float(), dim=-1) # [1, 4096]
```

At this point, `kmer_probs[0]` is the model's probability distribution over all 4,096 possible 6-mer tokens. So, how do we get to the single base probability now? This is where the magic happens. We can simply sum the probabilities at a given position and give a

base over all tokens. Suppose we want `P(first base of next 6-mer = A)`. We recover it by summing over every 6-mer whose first position is A:

```
pos = 0 # First base inside the 6-mer
base_to_idx = {"A": 0, "T": 1, "C": 2, "G": 3}

idx = model._bp_base_index[pos] # [4096], base id for each 6-mer at position 0
mask = idx == base_to_idx["A"]  # True for 6-mers whose first base is A

p_first_base_A = kmer_probs[0, mask].sum()

print("P(first base = A):", p_first_base_A.item())
```

That is the core operation behind FNS: 6-mer distribution → six base-level distributions. The manual code above is useful for understanding the idea. In practice, Carbon wraps this operation in `compute_bp_probs()`:

```
with torch.no_grad():
    out = model(**inputs)
    next_logits = out.logits[:, -1, :]      # [1, vocab_size]
    bp_probs = model.compute_bp_probs(next_logits)

print(bp_probs.shape)  # [1, 6, 4]

# bp_probs[0, j, b] = P(base b at position j of the next 6-mer)
p_first_base_A = bp_probs[0, 0, 0]
print("P(first base = A):", p_first_base_A.item())
```

Let's have a look at what this means at inference time by exploring an example visually.

Inference with FNS

Let's explore this concrete example of 8 tokens as input with one ground truth output shown in section 1 of the visualization. If we pass this through the model the traditional way, we'll get a probability for each token and can, for example, see in section 2. We

can see that the ground truth in this case is the 5th most likely token to predict, and the distribution tail is so long that there is a 50% chance of predicting a token beyond the first 128 tokens. Then in section 3, we can see the marginalized probabilities of each base at each position. We can take it a step further and also look at the ranking of every possible k-mer token in Section 4. To make it simpler, we only color highlight the base at one position. An interesting example is to look at "A" at position 5. Essentially, all tokens ranked on top predict the same base "A" there.

CARBON-3B · 48 BP

● REAL INFERENCE

1 · INPUTS & GROUND TRUTH

INPUT SEQUENCE

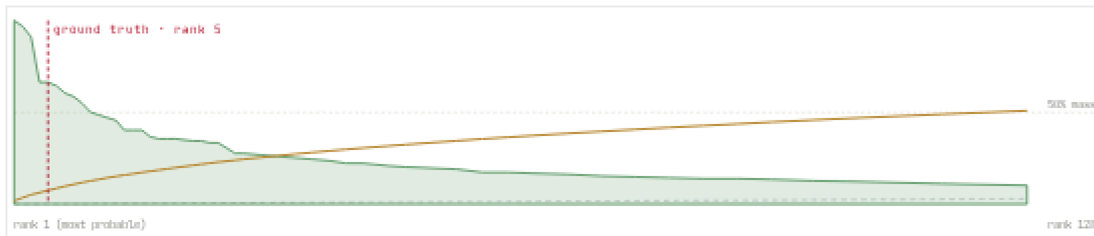
TCGGAT TAGCTG GAAAAA TTGCTC CGGCAG CTCGCG GAGCTG GTTGGC

TARGET TOKEN

→ GTCCAG

2 · TOKEN-LEVEL PREDICTION OVER ALL 4,096 6-MERS

Top prediction GGAGAG at **1.78%**. Half the probability mass falls in the top **120** of 4,096 tokens. Target token GTCCAG ranks **5** (1.19%).



PER-TOKEN PROBABILITY — CUMULATIVE MASS — UNIFORM REFERENCE

GROUND TRUTH RANK

MOST PROBABLE TOKENS

GGAGAG	GCTGAG	GCTCAG	GCCCAG	GTCCAG	CGAGAG	GGACAG	GCCGAG
1.78%	1.73%	1.62%	1.19%	1.19%	1.15%	1.08%	1.05%

3 · MARGINALIZED TO BASE LEVEL (THE 4,096-WAY DISTRIBUTION SUMMED PER POSITION · CLICK A POSITION)

POSITION 1	POSITION 2	POSITION 3	POSITION 4	POSITION 5	POSITION 6
G 45%	T 33%	C 34%	C 40%	A 88%	G 48%
C 25%	G 33%	T 25%	G 37%	G 6%	T 19%
A 19%	C 29%	A 23%	A 14%	C 3%	A 17%
T 11%	A 6%	G 18%	T 9%	T 3%	C 17%

4 · ALL 4,096 6-MERS, RANKED BY PROBABILITY · COLORED BY BASE AT POSITION 1

ALL A T C G

MOST PROBABLE → LEAST LIKELY

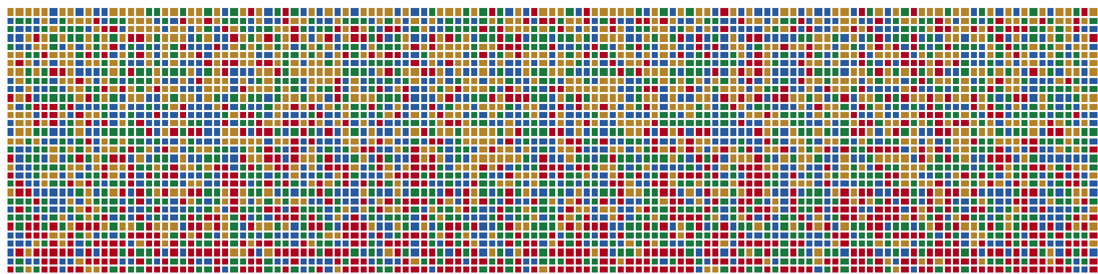


Figure 5 · FNS for Base-Level Generation. Real output from Carbon-3B. We feed an input sequence and ask the model to predict the next token. While the most probable 6-mer may not match the target token, the marginalized base distributions often align with the target at individual positions—demonstrating the benefit of FNS.

If you want to generate with this approach, call `model.generate_bp(...)` from the `fns` branch (a thin wrapper around `model.generate(...)` that installs the FNS logits processor). It accepts the same arguments as `generate`: `max_new_tokens`, `do_sample`, `temperature`, `top_p`, etc.

We can take this even a step further. Instead of independently marginalizing and selecting bases, we could also use conditional marginalization. For example, if we select "A" as the first base, we only need to marginalize over tokens starting with "A" for the remaining tokens. And we also don't need to do this left to right, we could start at any position the model is most confident about. Play with it yourself:

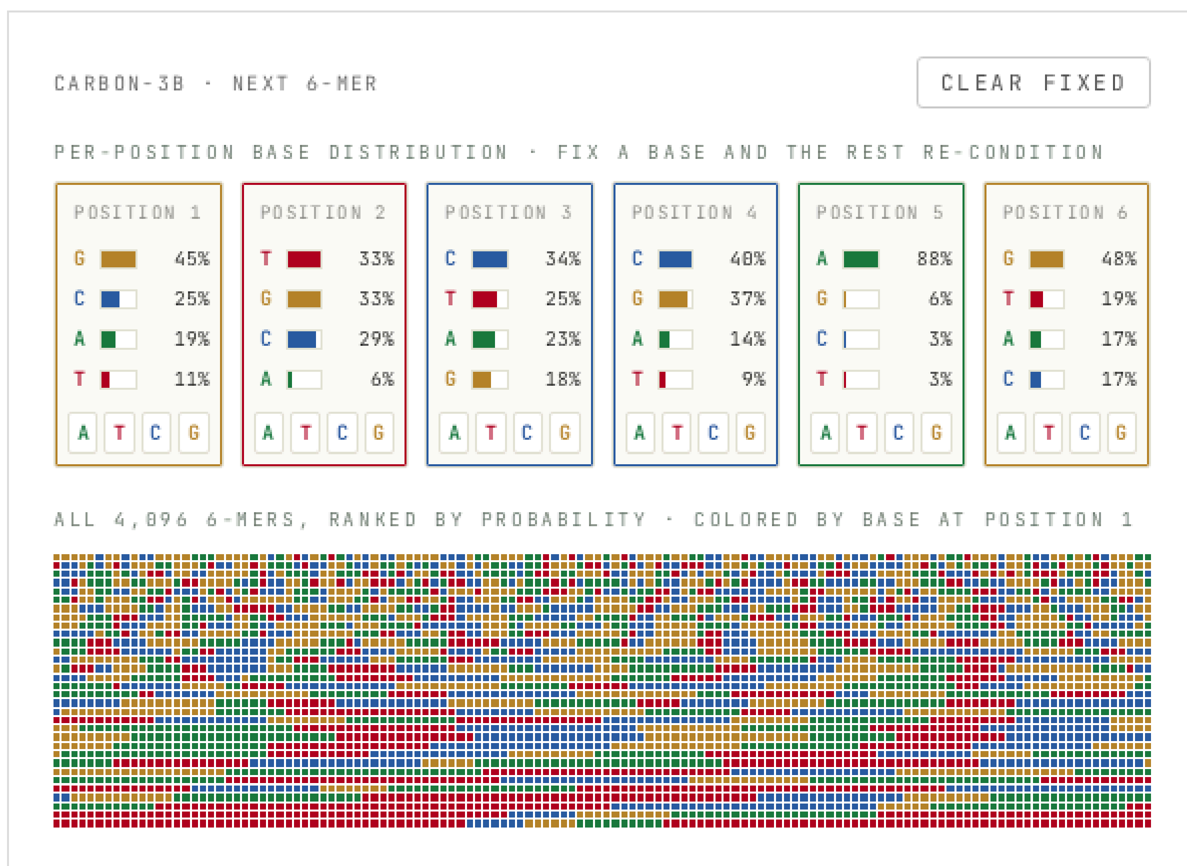


Figure 6 · FNS for Conditional Adaptation. Same Carbon-3B next-token distribution as Figure 4. Fix a base under any position and the open positions re-marginalize over only the 6-mers consistent with your choices; a fixed position freezes at its current distribution, and the grid below greys out the 6-mers that are no longer reachable.

We used single base resolution so far only for inference, but could we use that information also at training time? Yes!

Training FNS Loss

Standard 6-mer cross-entropy treats each 6-mer as an atomic class. If the target is `GTCCAG`, then `GGCCAG` gets the same loss as `AAAAAA`, even though `GGCCAG` matches five out of six bases and likely remains functional. **FNS changes the supervision signal.** Instead of asking only whether the exact 6-mer is correct, it asks whether each base position is correct.

For target `GTCCAG`, the standard cross-entropy loss is:

```
target = "GTCCAG"
target_id = tokenizer.encode(target)[0]  # full-vocab token id of "GTCCAG"
kmer_idx = (model._kmer_ids == target_id).nonzero(as_tuple=True)[0]  # 0..4095 position in the k-mer slice

ce_loss = -torch.log(kmer_probs[0, kmer_idx])
print("CE loss:", ce_loss.item())
```

But we can play the same game as for inference and compute the loss per position rather than over the whole token, and normalize it at the end:

```
base_to_idx = {"A": 0, "T": 1, "C": 2, "G": 3}

# bp_probs is reused from the inference example above; the loss value is illustrative,
# not the loss for a real "predict-GTCCAG-from-its-context" step.
fns_loss = torch.zeros((), device=bp_probs.device, dtype=bp_probs.dtype)

for pos, base in enumerate(target):
    base_id = base_to_idx[base]
    fns_loss = fns_loss - torch.log(bp_probs[0, pos, base_id])

fns_loss = fns_loss / 6
print("FNS loss:", fns_loss.item())
```

This gives **structured partial credit**. A near-miss 6-mer contributes probability mass to the positions it gets right. A completely mismatched 6-mer contributes little. We've seen the FNS loss greatly stabilize long training runs, leading to better performance.

Finally, let's look at a method that's especially important in genetics: scoring whole sequences.

FNS for Sequence Scoring

An important application of DNA models is to analyze the probability of each element in a sequence. This is especially important for variant effect prediction, which aims to determine if a mutation is benign or pathological. To do so, the probability difference between a reference and a mutated sequence is studied. FNS also provides a natural way to score DNA sequences at base resolution.

For a sequence, we want two things:

1. **P(A), P(T), P(C), P(G)** at each base position
2. **P(actual observed base)** at each position

The following example shows the base level scores of an original and a mutated sequence:

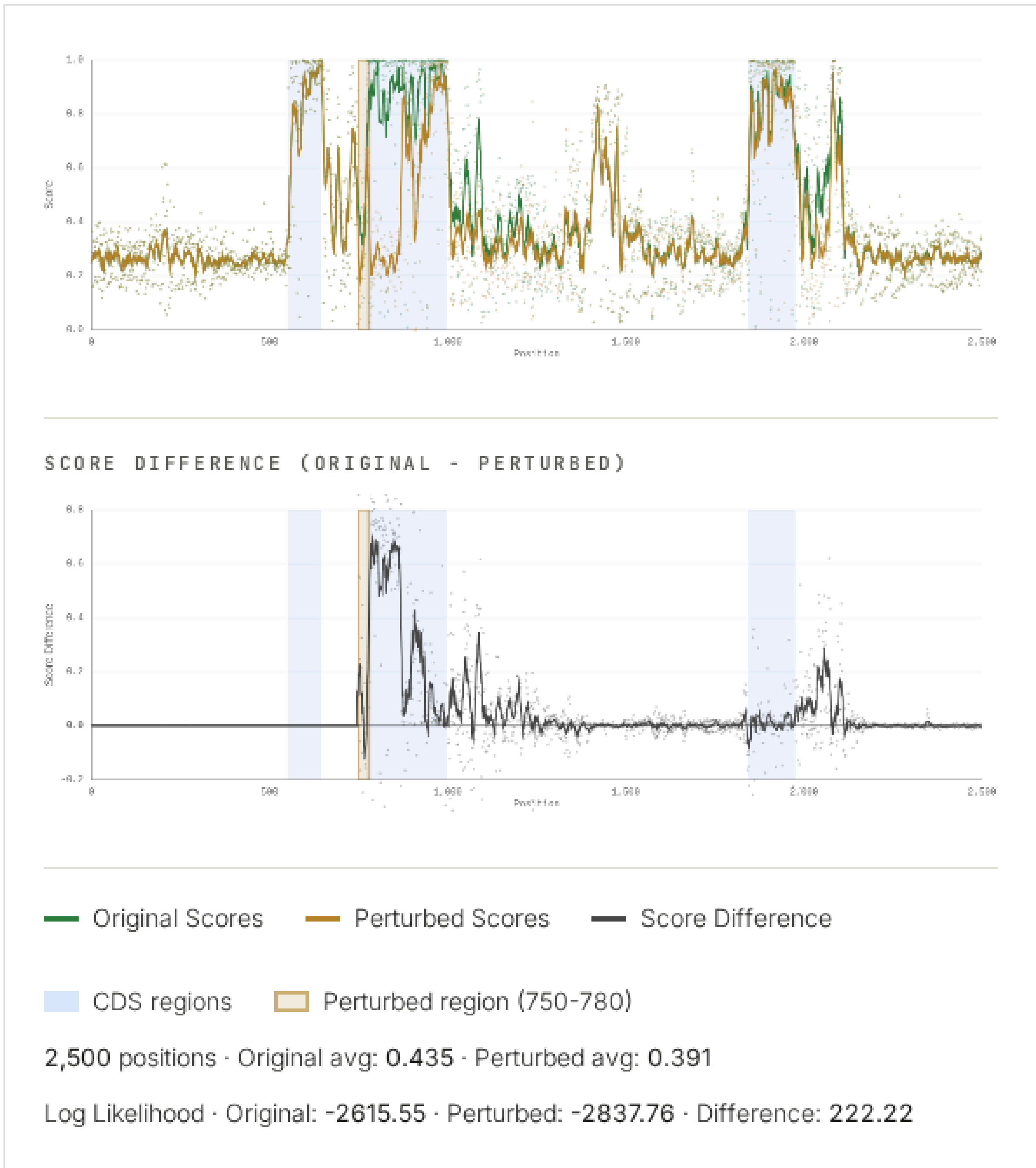


Figure 7 · FNS for Sequence Scoring. Comparing per-position scores before and after introducing a 30bp triplet expansion perturbation (positions 750–780) in the human hemoglobin beta (HBB) gene sequence. The original sequence shows baseline model predictions, while the perturbed version reveals how local sequence changes propagate through the model's scoring. Blue-highlighted regions mark the three coding sequence (CDS) exons. The visualization demonstrates both local effects within the perturbed region and potential downstream impacts on score distributions across the gene.

Carbon again wraps this whole process in a simple `score_sequence()` method:

```python
seq = "ATGCGCTAGCTACGATCGATCGTAGCTAGCTAGCTAGCTACG"

bp_probs, actual_probs = model.score_sequence(seq)

print(bp_probs.shape)      # [sequence_length, 4]
print(actual_probs.shape)  # [sequence_length]

# bp_probs[i] = [P(A), P(T), P(C), P(G)] at position i (i ranges 0 ... len(seq)-1)
# actual_probs[i] = probability assigned to the actual base in the input sequence
```

If a variant changes one base, FNS lets us directly compare the base-level probability of the reference allele and the alternate allele:

```python
position = 10

ref_base = "A"
alt_base = "G"

p_ref = bp_probs[position, base_to_idx[ref_base]]
p_alt = bp_probs[position, base_to_idx[alt_base]]

log_ratio = torch.log(p_ref) - torch.log(p_alt)

print("log P(ref) - log P(alt):", log_ratio.item())
# Higher positive log_ratio indicates higher risk that the
# alt_base is pathological.
```

The model is still a 6-mer Transformer internally, but FNS exposes a base-pair-level scoring interface. That concludes our tour of this unique tokenization strategy. Let's summarize what we learned.

Try it out

The reference implementation, the same `generate_bp`, `compute_bp_probs`, and `score_sequence` helpers used throughout this post:

```python
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch


tok = AutoTokenizer.from_pretrained(
    "HuggingFaceBio/Carbon-8B",
    revision="fns",
    trust_remote_code=True
)
model = AutoModelForCausalLM.from_pretrained(
    "HuggingFaceBio/Carbon-8B",
    revision="fns",
    trust_remote_code=True,
    dtype=torch.bfloat16,
).cuda().eval()

# Base-pair generation (greedy)
ids = tok(
    "<dna>ATGCGCTAGCTACG",
    return_tensors="pt",
    add_special_tokens=False
).input_ids.cuda()
out = model.generate_bp(ids, max_new_tokens=64, do_sample=False)
print(tok.decode(out[0], skip_special_tokens=True))

# Per-base scoring (useful for variant effect prediction)
bp_probs, actual_probs = model.score_sequence("ATGCGCTAGCTACGATCG...")
```

Conclusion

Carbon's tokenization strategy balances three competing demands:

1. **Efficiency**: 6-mer tokens reduce sequence length by 6×, making long genomic context affordable
2. **Resolution**: FNS recovers single-nucleotide probabilities from 6-mer distributions
3. **Flexibility**: The hybrid tokenizer supports both DNA and natural language

The key insight is that **you don't need to choose between token-level efficiency and base-level resolution.** FNS lets you have both: train and generate with 6-mer tokens for efficiency, then marginalize to base-level probabilities when you need single-nucleotide precision. This is especially useful for tasks such as:

- Sequence recovery
- Variant scoring
- Perturbation analysis
- Regulatory element prediction

This design makes FNS practical for real genomic tasks that require both long-range context and single-nucleotide accuracy.

Resources

- Carbon Collection
- Carbon demo Space
- Technical report (PDF)

Citation

For attribution in academic contexts, please cite this work as

```
Qiuyi Li, Leandro von Werra (2026). "Tokenization with Character Resolution".
```

BibTeX citation

```
@misc{li2026_tokenization_with_character_resolution,
  title={Tokenization with Character Resolution},
  author={Qiuyi Li and Leandro von Werra},
  year={2026}
}
```

Made with ❤️ with research article template editor