Naive inference algorithm
Naively, we would attempt batch proximal gradient descent on this objective function, which would involve the following steps:
1. Given current iterate θ, calculate current λ for all bases \( i \in [0,N] \) by
   \[
   λ_i = \exp\left( \sum_{k \in [1,K]} \sum_{j \in [-M,M-1]} θ^k_{(i+j,j)} - θ_0 \right).
   \]
2. Given current λ calculate the per base gradient vector
   \[
   d \log(λ_i) = err_i = c_i - λ_i.
   \]
3. Propagate the errors back to the parameter θ. Let \( s \) be the integer index corresponding to a k-mer. Then the gradient of this kmer \( s \) with off set \( j \) is
   \[
   dθ^k_{s,j} = \sum_{\{ k = s \}} err_{i+j} \quad \text{and} \quad dθ^0_0 = \sum_{i=1}^N err_i.
   \]
4. Update the current parameter with stepsize alpha.
   \[
   θ^k = θ^k + α dθ^k
   \]
5. Update the constant offset
   \[
   θ_0 = θ_0 - α dθ_0
   \]
6. Apply the proximal operator for \( L_1 \) regularization
   \[
   θ^k_{(s,j)} = \begin{cases} 
   θ^k_{(s,j)} - αη & \text{if } |θ^k_{(s,j)}| > αη \\
   0 & \text{otherwise}
   \end{cases}
   \]

This algorithm is prohibitively slow, with an iteration runtime of \( O(NMK + 4^K M) \). In practice, contribution from NMK dwarfs that of \( 4^K M \) since the gradient computation is cache incoherent and \( N \approx 3 \times 10^9 \) which is much greater than \( 4^K M \approx 6 \times 10^4 \).

There are two free parameters (\( α \) and \( η \)). The value for \( η \) is set via grid-search over values of \( η \) using held-out sets starting with the maximal feasible \( η \). This maximum is calculated analytically as the maximal \( η \) for which all \( K \)mers are nonzero. We will discuss setting \( α \) below.

Accelerating the algorithm
Prefix compression
Note that due to our k-mer matching scheme, a 3-mer can be represented as the sum of the 4 possible 4-mers whose prefix matches the 3-mer. Utilizing this fact, we can obtain runtimes of \( O(NM + 4^K M + 4^K K) \) and also reduce cache incoherence substantially.

We maintain a matrix \( φ \) of size \( 4^k \times 2M \) which represents only the longest k-mers. We then modify the first through fourth steps to use \( φ \) instead of \( θ \). Since every k-mer has a unique prefix match, this reduction maintains correctness of the algorithm.

Finally before step 6 we apply a decoding step. Let \( g(s,k) \) be a set-valued function consisting of all k-mers whose first k-1 characters match \( s \).

\[
    dθ^k_s = \sum_{s' \in g(s,k)} dφ^k_{s'}
\]

The use of dynamic programming (generate \( θ^k_{s-1} \) followed by \( θ^k_{s-2} \)) gives a runtime of \( O(4^K) \) to decode the compressed representation.

After step 6 we re-encode the parameter matrix into the compressed representation. Given a k-mer \( s \), let \( f(s,k) \) be the set valued function returning the k character prefix of \( s \).
\[ \phi_s = \sum_{k=1}^{K} \theta_{f(s,k)}^k \]

This takes runtime \( O(4^K K) \).

Representing the k-mers as bitstrings where each two bits represents a base allows for the query operations to be done nearly entirely bitshifts and cache-coherent additions, which allows for unmeasurably fast encoding and decoding for typical values of \( K=8 \) and \( M=500 \).

**More efficient proximal operators**

We derive a provably correct and more efficient proximal operator for the gradient descent algorithm.

The basic algorithm uses the standard \( L_1 \) soft-threshold prox operator:

\[
\theta_{s,j}^k = \begin{cases} 
\theta_{s,j}^k - \alpha \eta & \text{if } |\theta_{s,j}^k| > \alpha \eta \\
0 & \text{otherwise}
\end{cases}
\]

However, we note that this solution can be strictly improved with little extra effort. Using the same insight as our prefix compression scheme, note that adding a constant \( c \) to a k-mer and adding the same constant to the 4 possible k-1 mer prefix matches returns the same predicted \( \lambda \) values but have different \( L_1 \) penalty values. Using this idea we can decrease the \( L_1 \) penalty without affecting the goodness of fit.

This algorithmically captures the intuition that if \{ATA,ATC,ATG,ATT\} all have similar and positive effects, we can better represent the effect using just AT.

Define the median of a k-mer as the median parameter value of the prefix matching k+1-mers and negative of itself. Define \( g(s,k) \) as before as the set-valued function returning the four possible one character continuations of \( s \) (for example, given AT, \( g(s,3) = \{ATA,ATC,ATG,ATT\} \)) and the function \( m(s,j) \) as:

\[
m(s,j) = \text{median}(\theta_{g(s,k),j}^{k+1} - \theta_{s,j}^k).
\]

Then the parameters for any kmers \( s' \in h(s) \) can be updated as

\[
\theta_{s',j}^k = \theta_{s,j}^k - m(s,j)
\]

and

\[
\theta_{s,j}^k = \theta_{s,j}^k + m(s,j).
\]

This is a dynamic programming algorithm starting at K-1 and stopping at k=1. This procedure is guaranteed to not change the likelihood term depending on \( \lambda \) while strictly shrinking the \( L_1 \) norm of \( \theta \).

**Stochastic gradient descent**

We find that gradient descent is still far too slow to run on a single 32-core machine in less than a week. We achieve nearly ten-fold speedup by utilizing stochastic rather than batch gradient descent.

The variant of gradient descent we use is a minibatch-gradient, where we calculate the gradient and error over a smaller subregion of the genome. We use twenty million bases as our minibatch size (which we will refer to as \( B \)).

To control the step-size more intelligently, we also use a variant of stochastic gradient descent known as Adagrad [1]. We maintain a separate history \( \delta_s \) for every k-mer which we increment with the norm of the gradient.

In our variant, we cut the genome into twenty-million base chunks called minibatches. Let \( l \in [1,2,...,\lceil \frac{N}{B} \rceil] \) and \( \sigma(l) \) be a permutation of \( l \).

The steps one to three in the previous algorithm becomes:
1. At the beginning of every pass over the full genome, we generate a new random permutation $\sigma(l)$
2. Pick a global step size $\alpha$ by doing a line-search along the region of size $B$ with largest number of reads.
3. For $i \in [1 \ldots \lfloor \frac{N}{B} \rfloor]$ do a full update (naïve algorithm) on the subset of bases $(\sigma(l))_{i-1}B + [0,B].$
4. For each k-mer $s$, update its value with
   \[ \theta^k_s = \theta^k_s + \alpha \frac{d\theta^k_s}{\sqrt{\delta_s}} \]
5. If the average function value of all minibatches is more than 10% greater than the previous iteration, set $\alpha = \alpha/2$, reset parameters and redo the loop
6. Else return the averaged iterates over the whole pass.

While this algorithm gives no asymptotic performance improvement over the batch gradient, in practice it returns a solution equivalent to the batch gradient in time that is 10-20 times faster than the batch method. This is a well-documented effect in the literature.

Reference