A Efficiency of Partitioning

For generating the similarity graph for the nearest neighbor function, on CIFAR-100 data, the similarity graph is of size $400^2 \times 100$ (the 400 comes from 40000/100, 40000 is the size of the dataset, and 100 is the number of classes) and takes around 3 minute on a 16 core CPU machine. On the Imagenet dataset, the graph size is roughly $1300^2 \times 1000$, and it takes around 89 minutes to generate on the same machine. The memory costs in either case are small, as we compute the similarities class by class.

In Figure 5 we present the running time of Algorithm 1 with the lazy evaluation trick with various ground set sizes. Particularly, we use the same f_{NN} similarity graph as we use for the CIFAR-100 experiment, and we random sample a subset from the ground set as the input to the partitioning algorithm. For various $|V|$ values, we fix m, the number of blocks, to be 20, and let $k = |V|/m$. We run the algorithm on a single Intel Xeon CPU 2.60GHz CPU, and we record the min wall clock time over 5 runs to reduce the influence of other system processes over our timing experiment.

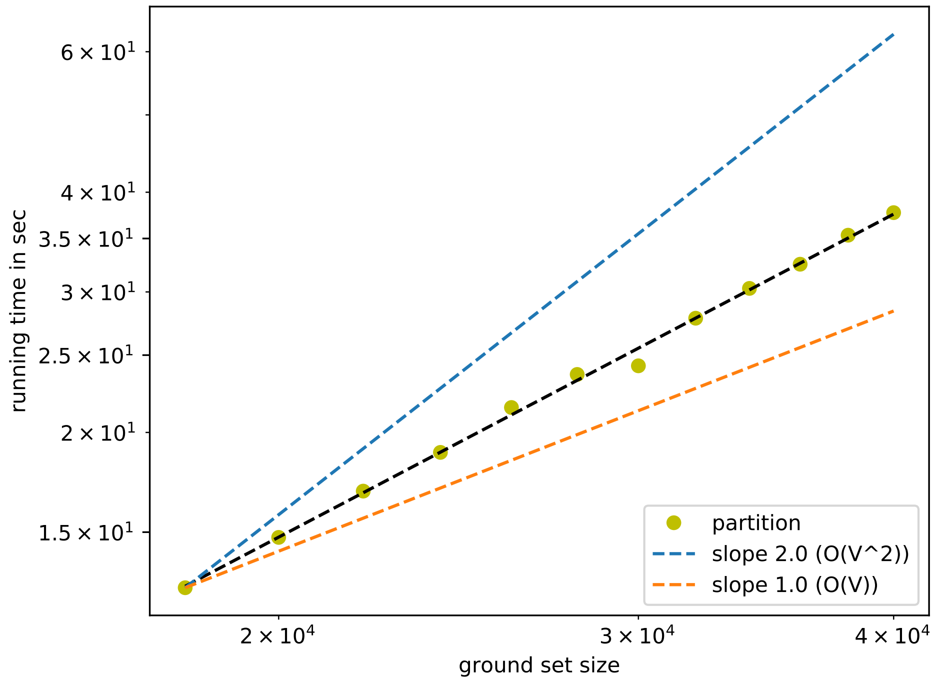

Figure 5: Log-log plot of the running time of the partition algorithm v.s. the ground set size. The slope of the regression line of the partition points is 1.345.

As shown in Figure 5, with the lazy evaluation trick, the running time of Algorithm 1 is significantly less than the theoretical worst case $\mathcal{O}(|V|^2)$ (the slope of a line in log-log scale reflects the complexity). We note that without the lazy evaluation trick, the running time would be $\mathcal{O}(|V|^2)$, but the lazy evaluation trick greatly increases the memory cost of the partitioning algorithm, for this experiment in particular, the memory cost is merely 20 times as the ordinary case since $m = 20$. Only with the hierarchical partitioning algorithm can we benefit from the speed-up achieved by the lazy evaluation trick, yet with little increase in memory cost. For example, if we set the block size constraints accordingly with $m_1 = 2$, $m_2 = 2$, and $m_3 = 5$, we get $20 = 2 \times 2 \times 5$ blocks in the end, and the memory cost with the lazy evaluation trick is only twice the ordinary case.

Specifically, for the CIFAR100 experiment with mini-batch size 128, the ordinary partitioning (Algorithm 1) has peak memory of 234MBs, while the hierarchical method with $k_1 = 1024$, $k_2 = 512$, $k_3 = 128$, has peak memory of only 30MBs. For the ImageNet experiment with a mini-batch size of 128, the ordinary partitioning (Algorithm 1) has peak memory of 151GBs, while the hierarchical method with $k_1 = 32768$, $k_2 = 16384$, $k_3 = 8192$, $k_4 = 4096$, $k_5 = 128$, has peak memory of only 4.6 GBs.

B On the fly Priority Queue for Robust Partitioning with Lazy Evaluation

In Algorithm 3, we show details about building the priority queues on demand rather than initializing m priority queues with all elements in V for robust partitioning with lazy evaluation. The general idea is simple: we keep a global list of singleton values $f(v)\forall v \in V$ shared by all partitions, initialize the priority queue for every block to be empty, and gradually push the gain of an element with respect to certain block into the priority queue for such block. Under the ordinary approach, the peak memory happens near the initialization phase, as every priority queue is initialized with all the elements in the ground set, which is a big waste, since the singleton gains are the same. We address this issue by keeping a shared global list of singleton values.

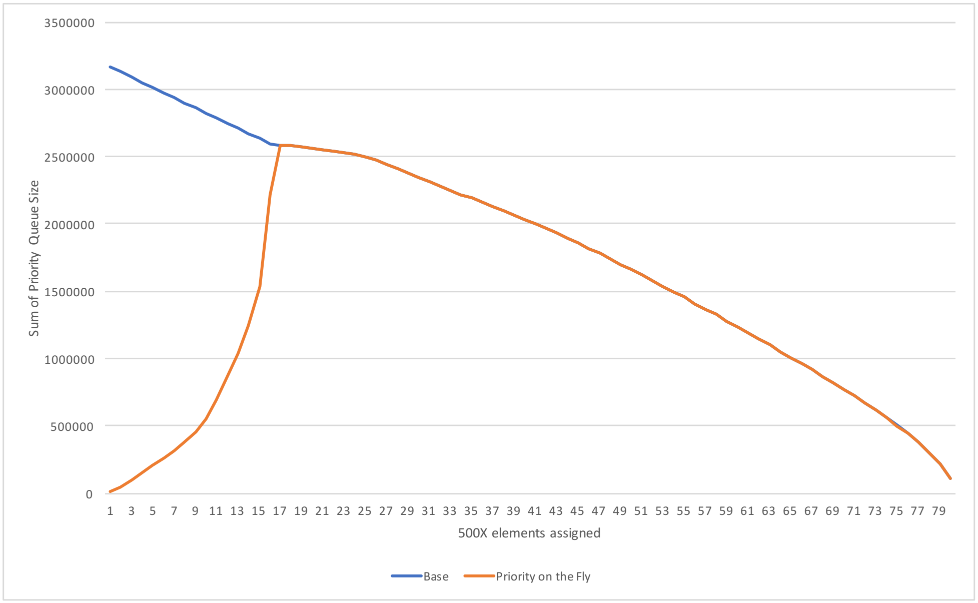

Figure 6: Comparison of the memory (as the sum of priority queue sizes) of the ordinary lazy evaluation to the on-the-fly priority queue lazy evaluation for robust submodular partitioning. The peak memory cost for the ordinary case starts at the beginning when we initialize m priority queues with all singleton values, while for the on-the-fly case, we gradually push elements into each priority queue, which reduces the peak memory cost by around 25%.

In the worst case, Algorithm 3 does not save memory, unfortunately. In practice, however, we regularly observe a constant reduction of memory cost (around 25%) as shown in Figure 6.

C Proof of Theorem 1

Theorem 1. *For submodular function f on ground set V and block size (mini-batch size) constraint k, suppose $m = |V|/k$, Algorithm 1 gives an approximation ratio of* $\frac{e-1}{(e-1)m+1}$.

Proof. At each iteration of Algorithm 1, it finds the smallest partially filled block (i.e., block size is less than k) and assigns the element with the largest gain to that block. The objective value we care about is the minimum block value. Therefore, at the end of some iteration (line 6 of Algorithm 1), if we find j s.t. $|A_j| = k$ where $j \in \text{argmin}_j f(A_j)$, we know the final minimum block value is $f(A_j)$, even though there are still elements unassigned at that moment (i.e. $|R| > 0$). The reason is quite simple, block A_j is full so its value will not change afterwards and all other block values will not decrease since f is monotone non-decreasing. A_j will be minimum ever after. Therefore, in this proof and also for the proof of Theorem 2, we assume that we are at line 6 of the earliest iteration when there exists j s.t. $|A_j| = k$ where $j \in \text{argmin}_j f(A_j)$. We pause the algorithm and prove $\min_j f(A_j)$ has a lower bound at that time. This lower bound will also apply for the final $\{A_j\}s$.

This assumption is useful since at the beginning of the current and all previous iterations (line 3), we know that for all $j \in \text{argmin}_j f(A_j)$, $|A_j| < k$. Then in line 3, $j^* \in \text{argmin}_{j,|A_j|<k} f(A_j)$ is equivalent with $j^* \in \text{argmin}_j f(A_j)$, which will make the analysis simpler. Unfortunately, this is not the end of Algorithm 1, so that $A_1, A_2, \ldots, A_m$

may not partition the entire ground set V, or more precisely, $\cup_{i=1}^m A_i \subseteq V$. We will be mindful of this in the below.

We begin the proof with several definitions and lemmas.

Definition 2. *Let OPT_f be $\max_{\pi\in\Pi(V,k)} \min_i f(\pi_i(V))$ for submodular function f defined on ground set V. Let $OPT = OPT_f$ for simplicity.*

Definition 3. *Let $f_A(X) : 2^V \to \mathbb{R}$ be a set function with $f_A(X) = f(X \cap A)$, and $X \subseteq V$.*

Immediately, we notice that if f is monotone non-decreasing submodular, so is f_A. Those two definitions are only valid in the scope of this section.

Lemma 1. *For monotone non-decreasing submodular function f, for any $A \subseteq V$, $OPT_f \le OPT_{f_{V\setminus A}} + f(A)$.*

Proof. First, we have $f(X) \le f(X \setminus A) + f(X \cap A) \le f_{V\setminus A}(X) + f(A)$ for all $X \subseteq V$ according to submodularity and monotonicity.

Let $\pi^* \in \operatorname{argmax}_{\pi\in\Pi(V,k)} \min_i f(\pi_i(V))$. Then we have $\mathrm{OPT}_{f_{V\setminus A}} \ge \min_i f_{V\setminus A}(\pi^*_i(V)) \ge \min_i[f(\pi^*_i(V)) - f(A)] = \mathrm{OPT}_f - f(A)$.

□

Lemma 2. *For monotone non-decreasing submodular function f, $OPT_f \le \min_{X\subseteq V, |X|=m-1} \max_{Y\subseteq V\setminus X, |Y|\le k} f(Y)$ (Recall the m is the number of blocks).*

Proof. Let $\pi^* \in \operatorname{argmax}_{\pi\in\Pi(V,k)} \min_i f(\pi_i(V))$. Note that $\mathrm{OPT}_f \le f(\pi^*_i(V))$ for $i = 1, \dots, m$.

For all $X \subseteq V$ with $|X| = m-1$, there is at least one $\pi^*_i(V)$ that does not contain any element from X since $\{\pi^*_i(A)\}_{i=1}^m$ are m disjoint sets. Assume $\pi^*_j(V) \cap X = \emptyset$, then $\pi^*_j(V) \subseteq V \setminus X$. Since $|\pi^*_j(V)| \le k$, we have $\mathrm{OPT}_f \le f(\pi^*_j(V)) \le \max_{Y\subseteq V\setminus X, |Y|\le k} f(Y)$. Note that this holds for all $X \subseteq V$ with $|X| = m-1$. Therefore, $\mathrm{OPT}_f \le \min_{X\subseteq V, |X|=m-1} \max_{Y\subseteq V\setminus X, |Y|\le k} f(Y)$

□

Now, we move to the main part of the proof for Theorem 1. We assume $m \in \operatorname{argmin}_{i=1,2,\dots,m} f(A_i)$ w.l.o.g. (assume the last block has the min evaluation). So $\min_{i=1,2,\dots,m} f(A_i) = f(A_m)$.

For $i = 1, 2, \dots, m-1$, assume A_i is not empty; otherwise $|V| < m$ and $\mathrm{OPT}_f = 0$ which immediately proves the theorem. Let a_i be the last added element in block A_i. We claim $f(A_i \setminus \{a_i\}) \le f(A_m)$, as the greedy process of the algorithm always puts an element into the block with minimal evaluation.

Let $f'(X) = f_{V\setminus[\cup_{i=1:m-1}(A_i\setminus\{a_i\})]}(X)$. We note that

$$\mathrm{OPT}_{f'} \le \max_{Y\subseteq V\setminus\{a_1,a_2,\dots,a_{m-1}\}, |Y|\le k} f'(Y) \tag{5}$$

$$= \max_{Y\subseteq V', |Y|\le k} f(Y) \tag{6}$$

according to lemma 2, where $V' = V \setminus (\cup_{i=1:m-1} A_i)$. Next, we want to prove $\mathrm{OPT}_{f'} \le \frac{1}{\gamma} f(A_m)$.

If $|V'| \le k$, then $\mathrm{OPT}_{f'} \le f(A_m)$.

If $|V'| > k$, then $|A_m| = k$. Recall that in the greedy process, at each step, we always choose the block with minimal evaluation and add the element with the largest gain from the remaining set R. Let us focus only on the m-th block. Let $A_m = \{v_1, v_2, \dots, v_k\}$ where each element is labeled in the greedy order. Assume that just before v_i is picked, the current remaining set of elements is R_i. Given the greedy process, we have $v_i \in \operatorname{argmax}_{v\in R_i} f(v|\{v_1, v_2, \dots, v_{i-1}\})$ for all $i = 1, 2, \dots, k$. We notice that $v_i \in B_i \cap V'$ for all i and $B_i \cap V' = V' \setminus \{v_1, v_2, \dots, v_{i-1}\}$. Therefore,

$$v_i \in \operatorname*{argmax}_{v\in V'\setminus\{v_1,v_2,\dots,v_{i-1}\}} f(v|\{v_1, v_2, \dots, v_{i-1}\}) \tag{7}$$

for all $i = 1, 2, \dots, k$.

Interestingly, this is just the greedy algorithm that chooses k elements from V'. Immediately, we have

$$f(A_m) \geq \gamma \max_{Y \subseteq V', |Y| \leq k} f(Y) \tag{8}$$

where $\gamma = 1 - \frac{1}{e}$ (Nemhauser et al., 1978). Therefore, $\text{OPT}_{f'} \leq \frac{1}{\gamma} f(A_m)$.

Then we use lemma 1,

$$\begin{aligned} \text{OPT}_f &\leq \text{OPT}_{f'} + f(\cup_{i=1,2,\ldots,m-1}(A_i \setminus \{a_i\})) && (9)\\ &\leq \text{OPT}_{f'} + \sum_{i=1,2,\ldots,m-1} f(A_i \setminus \{a_i\}) && (10)\\ &\leq \frac{1}{\gamma} f(A_m) + (m-1) f(A_m) && (11)\\ & && (12) \end{aligned}$$

since $f(A_i \setminus \{a_i\}) \leq f(A_m)$.

Therefore, we have

$$\min_{i=1,\ldots,m} f(A_i) = f(A_m) \geq \frac{1}{\frac{1}{\gamma} - 1 + m} \text{OPT}_f = \frac{e-1}{(e-1)m+1} \text{OPT}_f \tag{13}$$

□

D Proof of Theorem 2

Theorem 2. *If we have $\tau \geq 2$ as defined in Def. 1 for every call to Algorithm 1 from Algorithm 2, then we achieve an approximation ratio of $(\frac{\tau-1}{2\tau-1})^r \frac{k_r}{|V|}$.*

Proof. First, we show a simple counter example which can make the bound arbitrarily bad for Algorithm 1 without any partitions. Let $|V| = 8$, $m = 4$, $k_1 = 4$, $k_2 = 2$ and f is modular. The eight elements have weights $1-2\epsilon, 0.25, 0.25, 0.25, 0.25, \epsilon, \epsilon, \epsilon$ respectively, with $\epsilon > 0$ as a small value. Clearly, OPT=$0.25+\epsilon$, yet Algorithm 2 will give blocks with weights $(0.25, 0.25), (0.25, 0.25), (1-2\epsilon, \epsilon)$ and (ϵ, ϵ), and the min block evaluation is 2ϵ.

Next, we prove Theorem 2 under certain mild assumption.

Definition 4. *For any monotone non-decreasing submodular f and number of partitions M, Let OPT_M be $\max_{\pi \in \Pi(V, |V|/M)} \min_i f(\pi_i(V))$.*

The previous definition is valid only within the scope of this section.

Lemma 3. *For any monotone non-decreasing submodular f, and $M_1, M_2 \in \mathbb{Z}^+$, if $\frac{M_2}{M_1} \in \mathbb{Z}^+$, we have $OPT_{M_1} \geq OPT_{M_2}$.*

Proof. For any optimal partition of M_2, we can group them into M_1 partitions and each partition's function value is greater or equal to OPT_{M_2}. □

In Algorithm 2, there are r iterations and the partition will form a tree structure (Figure 2). The root is V and $Q_1 = \{V\}$. In the i-th iteration, it will partition every block A in $Q_i = \{A_{i,j}\}_{j=1}^{|Q_i|}$ into m_i smaller sets and those $|Q_i| m_i$ sets form Q_{i+1}. Immediately, we have that $|Q_i| = \Pi_{j=1}^{i} m_{j-1} = \frac{|V|}{k_{i-1}}$, where m_0 is set to 1. Let $M_i = |Q_i|$ and Q_i is a M_i-partition.

Definition 5. *For any $i = 1, \ldots, r$, $A \in Q_i$ is partitioned into $A_1, A_2, \ldots, A_{m_i}$, we call $child(A) = \{A_1, A_2, \ldots, A_{m_i}\}$ and $parent(A_j) = A$.*

Algorithm 2 is repeatedly calling Algorithm 1, and as we have discussed in Appendix C, we can end Algorithm 1 early to achieve the desired bound. The end condition is that there exists j s.t. $A_j \in \text{argmin}_i A_i$ and $|A_j| = k$. At the time we early stop Algorithm 1, we name the resulting blocks, whose sizes are possibly less than the cardinality constraint, as $\bar{A}_1, \bar{A}_2, \ldots, \bar{A}_{|V|/k}$. We notice $\bar{A}_j \subseteq A_j$ and $\min_i A_i = \min_i \bar{A}_i$. We assume that Algorithm 1 also outputs $\bar{A}_1, \bar{A}_2, \ldots, \bar{A}_{|V|/k}$.

Definition 6. *For any $i = 1, \dots, r$, $A \in Q_i$, after running Algorithm 1 on A, let $t(A) = \min_{i=1}^{m_i} |\bar{A}_i|$.*

Definition 7. *For $i = 1, \dots, r$ and $\tau \geq 2$ is an integer, we say Property $P(i, \tau)$ is true if and only if $t(A) \geq \tau$ for all $A \in Q_i$.*

Definition 8. *For $i = 1, \dots, r+1$ and $\tau \geq 2$ is an integer, we say Property $G(i, \tau)$ is true if and only $f(A) \geq \frac{1}{(\frac{2\tau-1}{\tau-1})^{i-1} M_i} OPT_{M_i}$ for all $A \in Q_i$.*

$G(1, \tau)$ is true immediately, since $M_1 = 1$. The ultimate goal is to prove $G(r+1, \tau)$ by induction.

Lemma 4. *For $i = 1, 2, \dots, r$ and $\tau \geq 2$ is an integer, $[G(i, \tau) \cap P(i, \tau)] \to G(i+1, \tau)$*

Proof. Given the assumption, we have both $G(i, \tau)$ and $P(i, \tau)$ to be true. So for all $A \in Q_i$, $f(A) \geq \frac{1}{(\frac{2\tau-1}{\tau-1})^{i-1} M_i} \mathrm{OPT}_{M_i} \geq \frac{1}{(\frac{2\tau-1}{\tau-1})^{i-1} M_i} \mathrm{OPT}_{M_{i+1}}$ according to Lemma 3. For the next step, we need to prove $f(A_j) \geq \frac{\tau-1}{(2\tau-1)m_i} f(A)$ for all $A_j \in child(A) = \{A_1, A_2, \dots, A_{m_i}\}$.

W.l.o.g, we assume $m_i \in \mathrm{argmin}_{j=1,2,\dots,m_i} f(A_j)$ and $|A_{m_i}| = |\bar{A}_{m_i}| = k_{i+1}$. For $j = 1, 2, \dots, m_i - 1$, we first look at $\bar{A}_j$. Since $P(i, \tau)$ is true, we know that $|\bar{A}_j| \geq \tau \geq 2$. Let a_j be the last added element in $\bar{A}_j$. We claim $f(\bar{A}_j \setminus a_j) \leq f(A_{m_i})$ given the greedy process and note that block A_{m_i} is not full (size smaller than k_{i+1}) at the time of adding a_j. Also, all elements in $\bar{A}_j$ are added by greedy order, so $f(\bar{A}_j) \leq \frac{|\bar{A}_j|}{|\bar{A}_j|-1} f(\bar{A}_j \setminus a_j) \leq \frac{\tau}{\tau-1} f(A_{m_i})$ for $j = 1, 2, \dots, m_i - 1$.

Next, there are still some elements in $A_j \setminus \bar{A}_j$ we have not analyzed. We notice $|\cup_{j=1,2,\dots,m_i-1} [A_j \setminus \bar{A}_j]| \leq |A| - (m_i - 1)\tau - k_i$, and the elements in $\cup_{j=1,2,\dots,m_i-1} [A_j \setminus \bar{A}_j]$ are not added to m_i block because they have less or equal marginal gains than any element in A_{m_i}. Therefore $f(A_{m_i} \cup [\cup_{j=1,2,\dots,m_i-1} (A_j \setminus \bar{A}_j)]) \leq \frac{|A| - (m_i-1)\tau}{k_i} f(A_{m_i})$.

$$
\begin{aligned}
f(A) &\leq f(A_{m_i} \cup [\cup_{j=1,2,\dots,m_i-1} (A_j \setminus \bar{A}_j)]) \\
&\quad + f(\cup_{j=1,2,\dots,m_i-1} f(\bar{A}_j)) && (14) \\
&\leq \left[\frac{|A| - (m_i - 1)\tau}{k_i} + \frac{\tau(m_i - 1)}{\tau - 1} \right] f(A_{m_i}) && (15) \\
&= \left[\frac{m_i k_i - (m_i - 1)\tau}{k_i} + \frac{\tau(m_i - 1)}{\tau - 1} \right] f(A_{m_i}) && (16) \\
&\leq \left[1 + \frac{\tau}{\tau - 1} \right] m_i f(A_{m_i}) && (17) \\
& && (18)
\end{aligned}
$$

Therefore, we have $f(A_{m_i}) \geq \frac{1}{\frac{2\tau-1}{\tau-1} m_i} f(A)$. And since, $m_i \in \mathrm{argmin}_{j=1,2,\dots,m_i} f(A_j)$, we have $\min_{B \in child(A)} f(B) \geq \frac{1}{\frac{2\tau-1}{\tau-1} m_i} f(A)$ for all $A \in Q_i$. Recall that, we have, for all $A \in Q_i$, $f(A) \geq \frac{1}{(\frac{2\tau-1}{\tau-1})^{i-1} M_i} \mathrm{OPT}_{M_{i+1}}$.

Combining them together gives $G(i+1, \tau)$. ◻

Finally, we finish the proof of Theorem 2 by induction, $[\cap_{i=1,2,\dots,r} P(i, \tau)] \to G(r+1, \tau)$. ◻

E Binary Programming Formulation for Robust Partition Problem with a Cardinality Constraint

Consider X as a binary matrix of size $(n \times m)$, where every column of X (a binary vector of length n) corresponds to a subset of a ground set of size n. As we represent a set as a binary vector, a submodular function f takes in a binary vector and outputs a real value.

$$\max_X \min_{i=1:m} f(X[:,i]) \tag{19}$$

$$\text{s.t. } \forall j, \sum_{i=1:m} X[j,i] = 1 \text{ (partition constraint)} \tag{20}$$

$$\forall i, \sum_{j=1:n} X[j,i] = k \text{ (mini-batch size constraint)} \tag{21}$$

$$X[j,i] \in \{0,1\} \text{ for every } j,i \tag{22}$$

F Discussion about τ Values

For the CIFAR-100 experiment, we do a three-level hierarchical partitioning using the f_{NN} function defined in Section 5. The final mini-batch/block size is 128, and the τ value we get is 103 (τ is always less than the final block size), so the extra factor we get in Theorem 2 is 0.123. We also show the plot of the extra factor $(\frac{\tau-1}{2\tau-1})^r$ in Figure 7.

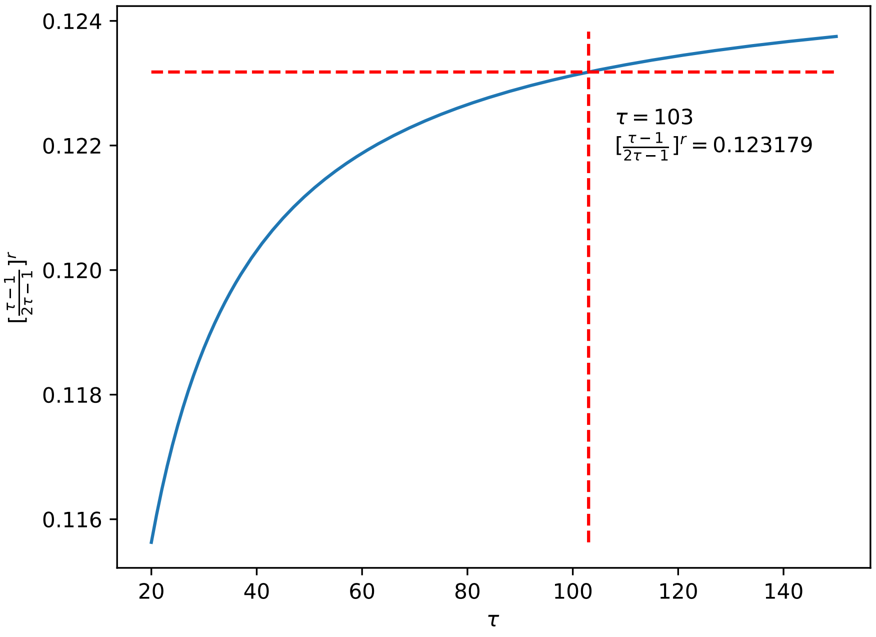

Figure 7: The extra factor in Theorem 2 v.s. τ on CIFAR-10 with f_{NN} function.

For the ImageNet experiment, we do a five-level partition. Like the CIFAR-100 case, we use the f_{NN} function and the final mini-batch size is 128. The τ value we get is 106 and the extra factor is 0.0305. For both cases, the τ values are close to the final block size, and far greater than the requirement ($\tau \geq 2$) given in Theorem 2. Therefore, in practice, the partitioning algorithm fills in blocks in a quite balanced manner.

G Architecture of DNNs

See Table 1 for the structure of autoencoder for extracting features for CIFAR-100 experiment.

Algorithm 3: Priority Queue on the Fly for Robust Submodular Partition

```
Input: $n = |V|$, $k$, $f$, $m = n/k$ ;                    // $n$ ground set size, $m$ partition
Initialize $q_1, .., q_m$ ;                                 // $m$ priority queues
$q' = []$ ;                                                 // global sorted list
$a = [0, ..., 0]$ of length $m$ ;                           // pointer to $q'$ of every block
for $i = 1 : n$ do
    $q'$.push($(i, f(i))$);
end
Sort $q'$ descending by value (second element in the pair);
$\pi_j := \{\}$ for $j = 1 : m$ ;                           // blocks
$j' := 0$ ;                                                 // min partition index
$u := [False, False, ..., False]$ of length $n$ ;           // element $i$ assigned or not
while $\sum_j |\pi_j| < n$ do
    while $a[j'] < n$ do
        ;                                                   // find the next unselected element from the global list
        if $u[q'[a[j']].index()]$ then
            $a[j'] := a[j'] + 1$ ;                          // increment if element $q'[a[j']].index()$ already assigned
        end
    end
    if $|q_{j'}| > 0$ then
        ;                                                   // find the next unselected element from the priority queue of $j'$
        $i := q_{j'}$.pop().index();
        while $u[i]$ do
            if $|q_{j'}| > 0$ then
                $i := q_{j'}$.pop().index() ;               // skip if $i$ already assigned
            else
                $i := -1$ ;                                 // $i = -1$ indicating queue is empty
                break;
            end
        end
    else
        $i := -1$ ;                                         // $i = -1$ indicating queue is empty
    end
    if $i == -1$ then
        ;                                                   // choose the larger one from the two candidate elements
        $i := a[j'].index()$ ;                              // queue empty, get from the global sorted list
    else
        if $a[j'] < n$ then
            $i' := q'[a[j']].index()$ ;                     // have not traversed the entire global sorted list
            if $f(i'|\pi_{j'}) > f(i|\pi_{j'})$ then
                $i := i'$ ;                                 // Whether use the one from $q'$ or $q_{j'}$
            end
        end
    end
    Increment $a[j']$ until $q'[a[j']]$ is not assigned ;   // Get the next largest value from global queue
    Pop $q_{j'}$ until the top one is not assigned ;        // Get the next largest value from $q_{j'}$
    $w := max(q_{j'}.top().value(), q'[a[j']].value())$ ;   // The criteria is the larger one from the two queues
    if $f(i|\pi_{j'}) > w$ then
        ;                                                   // greedy step
        $\pi_{j'} := \pi_{j'} \cup \{i\}$;
        $u[i] := True$;
        $j' := argmin_j f(\pi_j)$ ;                         // choose new min block
    else
        $q_{j'}$.push( $(i, f(i|\pi_{j'}))$);
    end
end
```

Layer Group	Block Type (kernel size, stride, channels)	# Blocks
conv1	$[3 \times 3]$, 2, 64	1
conv2 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 64	3
conv3	$[3 \times 3]$, 2, 64	1
conv4 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 64	3
conv5	$[3 \times 3]$, 2, 64	1
conv6 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 64	3
conv7	$[3 \times 3]$, 2, 8	1
conv8 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 8	1
deconv9 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 8	1
deconv10	$[3 \times 3]$, 1, 64	1
deconv11 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 64	3
deconv12	$[3 \times 3]$, 2, 64	1
deconv13 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 64	3
deconv14	$[3 \times 3]$, 2, 64	1
deconv15 (residual)	$\begin{bmatrix} 3 \times 3 \\ 3 \times 3 \end{bmatrix}$, 1, 64	3
deconv16	$[3 \times 3]$, 2, 3	1

Table 1: Neural network structure of autoencoder for extracting features for CIFAR-100 experiment.