Alternative implementations of the Auxiliary Duplicating Permutation Invariant Training

Jens Gulin[1,2], Karl Åström[1]

[1]Computer Vision and Machine Learning, Centre for Mathematical Sciences, Lund University, Lund, Sweden
[2]Sony Europe B.V., Lund, Sweden

This work was partially supported by the strategic research project ELLIIT, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Abstract

Simultaneous sound event localization and detection (SELD) for multi-source sound events is an open research field. The Multi-ACCDOA format is a popular way to handle activity-coupled sound events where the same class occurs at multiple locations at the same time. An important part is the Auxiliary Duplicating Permutation Invariant Training (ADPIT) paradigm that calculates the loss for order-agnostic output. The baseline system for the DCASE SELD challenge 2024 has an implementation of ADPIT. In this paper we discuss alternative ways to implement ADPIT with the goal to reduce multiplications, to make the equivalent calculations faster. ADPIT duplicates output when there are fewer events than tracks. A brief discussion how this differs from permutation invariant training without duplicated output is also included. The loss calculations are likely not the execution bottleneck in the current challenge setup, but ADPIT scales poorly for an increased number of tracks and improved efficiency is thus of general interest for audio localization.

1. Introduction

This work presents different ways to implement ADPIT [1] and a discussion on their computational requirements. For this paper, we consider the problem in abstraction and for no specific computational platform. Because of that, there is neither implementation nor formal performance analysis included. Although it may be considered a work-in-progress, we want to present the discussion and welcome input from the community.

Although zeroth-order optimization [2] is advancing, stochastic gradient descent (SGD) based methods like Adam [3] are the default choices. Either way, the loss function is iterated repeatedly during training. The discussion will refer to *branch free* loss calculation, meaning a vectorizable implementation not branching off depending on data content. This is not to be confused with [1] using "without branching" to describe a unified SELD network, with sound detection, classification and localization in the same flow.

Time complexity is commonly discussed in terms of asymptotic cost ($\mathcal{O}$) when the number of samples increases [4]. From that point of view, the number of tracks (or slots, as we will call them) is fixed, and efficiency in terms of tracks is just affecting the coefficient and not $\mathcal{O}$ complexity. On the other hand, having a large constant factor may be prohibiting to both large and small sample sizes, and calculations wasted are good for nothing. This is the motivation to discuss how to reduce the cost per track. With control of that cost, we argue that also higher number of tracks should be explored.

2. Background

This section provides terminology and an in-depth introduction to ADPIT, as well as a brief discussion of related work.

Proceedings of the Work-in-Progress Papers at the 14th International Conference on Indoor Positioning and Indoor Navigation (IPIN-WiP 2024), October 14–17, 2024, Kowloon, Hong Kong
jens.gulin@sony.com (J. Gulin); karl.astrom@math.lth.se (K. Åström)
0000-0002-3656-759X (J. Gulin); 0000-0002-8689-7810 (K. Åström)

2.1. Problem formulation

Consider the actual problem as the efficient execution of the loss function during permutation invariant training of a SELD network. To simplify discussion, we will describe a more abstract task and formulate the problem more generally.

Assume that we have recordings of an environment containing the sounds of *events* and non-events, together with a (fairly accurate) *ground truth (GT)* list of these events (as provided from oracle knowledge). Although training is meant to give the network the generalized ability to separate them, the environment-specific semantic difference of events and non-events is disregarded here: What makes it an event is simply that it is listed as GT. At any time, there may be no event active, one event, or multiple events overlapping fully or partially. The events have a start and end time, a source *position* in space, and an audio *class*. The classes are predefined for the environment, e.g. female speech, clapping and music. The time may be divided into discrete *frames*, so that the event duration is instead represented as a number of consecutive frame-wise events.

The *network* can be any entity that takes input, say raw audio, and produces a set of *tokens* as output. Each token represents an estimation of one event inferred from the input, we will however reserve the word event for GT and reserve token for the estimates. The *loss function* evaluates the output quality compared to the GT, and provides feedback (*gradients*) to guide the training. The internal representation of the token and the event may be different, but the loss function is able to compare them.

The output is arranged in a fixed number of N *slots*. All slots need to be filled, when the network infer fewer tokens, slots are marked empty by the *zero* ($\oslash$) token. As mentioned, the output is a set, where order is disregarded, but in practice the output slots are ordered. For permutation invariant training (PIT), the problem is to find the best way to map each output to a distinct event from GT, finding the *sequence* of N events that best matches the ordered output. With $M \leq N$ actual GT events in a scope, each M will introduce a permutation *category*, and we label the events accordingly; $\oslash$, {A0}, {B0, B1}, etc.

We can assume that the loss function directly compares one token to one event, even if multiple entries are input together. A further generalization is that the loss operates independently within a *scope*, where a scope may be e.g. frame-wise or frame-class-wise. That is, when scoped for both frame and class, there is no need to evaluate a token towards another frame or another class. Finally, *batches* are processed together, and the combined loss is the average over all samples. Batching allows for efficient vectorization with limited memory usage and it also provides a stochastic regularization smoothing gradients.

2.2. ADPIT

This section introduces the work of Shimada et al. presented in [1]. Everything here reference their paper to provide the background needed, only rephrased slightly to fit the discussion. Any commentary given is that of our own.

The ADPIT method is described as PIT with auxiliary duplication into empty slots. Our interpretation is that *duplication* introduces identical copies of enough events to fill all slots, while *auxiliary* means that copying is meant to support the result. Equivalently, the network is trained to produce close to identical copies, while free to chose which tokens to duplicate and how similar they would be in any way that could support training the most. Compared to PIT, the experimental results indicated that ADPIT made a large improvement. Our hypothesis is that the duplication gives the network better learning possibilities through redundancy. The Multi-ACCDOA format extends the Single-ACCDOA with multiple slots per class. The paper claims that this gave better results for cases where there were multiple active events from the same class, and did not worsen the normal case.

ADPIT scope is frame-based processed class-wise. There are a fixed number of N output slots[1] per class c for each frame t, and M_{ct} is the number of active events of that class in the current frame. Due to duplication, there should be no $\oslash$-tokens unless there are no estimates at all for the scope.

[1] Called *tracks* in reference, but there is no tracking over time so we call them slots.

Their eq. 6, here slightly reformulated, gives an upper limit of number of permutations possible for a scope:

$$\hat{K}(N, M) = \begin{cases} {}_N P_M \times M^{N-M} & \text{if } M > 0, \\ 1 & \text{if } M = 0, \end{cases} \tag{1}$$

where $\times$ is scalar multiplication, ${}_N P_M$ counts the number of (partial) permutations of M events onto N slots without repetition. The exponential M^{N-M} counts all combinations of the M events copied into the (possibly) remaining $N - M$ slots. We rename it $\hat{K}(N, M)$ to clarify that it is an upper limit and show that the function is the same regardless how the variables are scoped. They acknowledge that the unique number of sequences may be lower than $\hat{K}$, since (1) does not account for that a duplicate is indistinguishable from the original.

In the example of $N = 3$ slots and $M = 2$ events, calculation yields $\hat{K}(3, 2) = 12$, but the actual number of permutations is 6. Table 1 lists all twelve variants grouped into six cells to show this more clearly. We refer to indistinguishable sequences (variants within a cell) as *aliasing* and they should be processed as only one in an efficient implementation.

Table 1
Illustration of $\hat{K}(3, 2)$, where the GT events B0 and B1 are naively permuted on three slots using ADPIT rules. Duplicates (in italics) are aliases for the original. Thus, the twelve sequences are in fact only six.

B0 *B0* B1	B0 B1 *B0*	B1 B0 *B0*	B1 *B1* B0	B1 B0 *B1*	B0 B1 *B1*
B0 B0 B1	*B0* B1 B0	B1 *B0* B0	*B1* B1 B0	*B1* B0 B1	B0 *B1* B1

2.3. Baseline implementation

The baseline system for the DCASE SELD challenge 2024 has an implementation[2] of ADPIT. In overview it has several steps: **1)** preprocess GT and introduce a *dummy* dimension that separates the events based on M, **2a)** at training, load preprocessed scope and calculate loss for each possible sequence, for all categories at the same time, **2b)** pick the minimum loss and use that for gradient calculation and back-propagation, **3)** at inference, identify and discard duplicate output.

The implementation is scoped for $N = 3$ slots for each class and time frame. GT is preprocessed to do the proper scoping and at the same time M is calculated. Step 1 applies $d = 6$ dummy events. The idea is to give the entire set {A0, B0, B1, C0, C1, C2} in parallel. For a specific scope, only one M can be the actual target and in result only one of the subsets A, B or C are active, while the other dummy entries are all zeros. In step 2, the events of A, B and C are permuted separately. The number of permutations of B is correctly set to 6. With 1 for A and 6 for C, the total is 13 sequences per scope, again processed together. Now, if not cared for, the inactive dummy entries could severely disrupt the assumption that the lowest loss is the best matching sequence. To regulate the numerous all-zero entries, an overlay strategy ensures that these zero-entries are converted into existing sequences that do not interfere. All sequences and the overlays are shown in Table 2. Note that when $M = 0$, we can see that every row will turn into all zeros, which is an appropriate target for $\oslash$-events, and this category requires no additional logic.

The purpose for processing also "useless" dummy losses is to make a branch free implementation, which allows full vectorization and automatic gradient calculations: Each scope is processed exactly the same, regardless of the data. The overlay strategy is however not optimal, and Section 3.2 proposes another way. The step 3 has a fixed angular threshold and does not rely on event-pair permutations (since no GT knowledge is available). It might be improved still, but this step is not part of the learning loop, thus less of a priority here.

In this context, the exact implementation of the loss function is not important. For the sake of completeness, the loss function is the *mean squared error (MSE)* of the estimated position and the event

[2] https://github.com/partha2409/DCASE2024_seld_baseline/tree/0b07887

Table 2
Baseline implementation overlay strategy. Depending on M, only A, B or C is active. Inactive entries are zero and the three columns are added to form the target sequence. Thus, when A is active, all 13 sequences are A0A0A0. When C is active, 8 of the sequences are C0C1C2.

Permutations	*Overlay 1*	*Overlay 2*
A0 A0 A0	B0 B0 B1	C0 C1 C2
B0 B0 B1	A0 A0 A0	C0 C1 C2
B0 B1 B0	—‖—	—‖—
B0 B1 B1	—‖—	—‖—
B1 B0 B0	—‖—	—‖—
B1 B0 B1	—‖—	—‖—
B1 B1 B0	—‖—	—‖—
C0 C1 C2	A0 A0 A0	B0 B0 B1
C0 C2 C1	—‖—	—‖—
C1 C0 C2	—‖—	—‖—
C1 C2 C0	—‖—	—‖—
C2 C0 C1	—‖—	—‖—
C2 C1 C0	—‖—	—‖—

position. Since the scope is class-based, the loss function can disregard class. In the end, the loss will still penalize a token with no matching event of that class or an event with no matching token.

Note: The baseline implementation also has a function to calculate various metrics against GT, for logging during training or standalone evaluation. The `linear_sum_assignment` function from SciPy [5], is used to find the best token-event pairing. The implementation is not optimized for the specific problem and reusing the token pairing (algorithm) from the loss function might be beneficial, but this is left for future study.

2.4. PIT and Related Work

Permutation invariant training has been used for different tasks, such as speaker separation [6] and SELD with track-wise loss [7]. ADPIT is in [8] both used for the usual frame-class-scope SELD and, in a novel setting, for training a multi-source time-of-arrival feature extractor.

Relying on exhaustive permutation and taking the sequence with minimum loss, [6] mentions $M!$ sequences. A naive permutation strategy is however $N!$ and to avoid that for $M < N$, some slots need to be discarded. The auxiliary autoencoding PIT (A2PIT) [9], providing fault tolerance for speech separation, replaces empty targets with the original input. From a permutation perspective, any unique encoding for empty targets is however equivalent. It is clarified that A2PIT first identify the assumed valid (non-empty) outputs and then add (empty) or remove (valid) slots randomly to get exactly as many as the M real targets. This is simple, but not a perfect strategy, as random selection does not guarantee best fit. A possibility is that the loss function still does $N!$ permutations, and that M is used only for the balance factor between valid and empty pairs.

Even $N!$ sequences of PIT are often fewer than exhaustive permutations for ADPIT. With ADPIT, the slots are potentially encoded in many different representations, adding permutation complexity. It is also more difficult to filter out the "empty" ADPIT slots, for the same reason. In result, PIT handles any M within the same permutations and no extra care is needed for a branch free implementation. It would be interesting to see more papers compare if the ADPIT complexity is worthwhile.

Exhaustive permutation is a naive way to solve the *linear assignment problem*, or equivalently to find the *minimum weight matching in bipartite graphs*. The weights are pre-calculated as a cost matrix with $\mathcal{N}_r$ rows and $\mathcal{N}_c$ columns. The Hungarian algorithm [10] (also known as Kuhn–Munkres) originally solved this for an $\mathcal{N}$ square matrix in $\mathcal{O}(\mathcal{N}^4)$. Variants, such as the modified Jonker-Volgenant algorithm [11] implemented in SciPy [5], reach $\mathcal{O}(\mathcal{N}^3)$ and handle $\mathcal{N}_r \neq \mathcal{N}_c$. The implementation is not branch free, and since $\mathcal{N}! < \mathcal{N}^3$ until $\mathcal{N} = 6$ we rely on exhaustive summation for now. The auctioning algorithm is according to [11] rarely faster and the worst case sometimes very slow, but it may be better

suited for parallelization and well-structured problems. In addition, for branching algorithms, the best or average case is often faster than the worst case.

3. Proposals: K(N,M), ARMPIT and BRADPIT

This section presents a number of simple and elegant implementation strategies for ADPIT. We collectively name the branch free ones, especially the combination of them, *ADPIT with Reduced Multiplications (ARMPIT)*. In addition, we believe that a branching approach may be a suitable implementation, and present also *Branch Remedied ADPIT (BRADPIT)*. First step is to clarify the permutation requirements.

3.1. Number of sequences

The area of combinatorics is a well developed study of permutations and combinations within mathematics. The interpretation provided for (1) uses some commonly understood rules, but a broader search also provides the exact count we need. A table of thirty conceptual counts [12, page 26] identifies our problem as *Count B3*, providing the exact number of sequences,

$$\underset{\text{for } 1 \le M \le N}{K(N,M)} = M!\, S(N,M) \tag{2}$$

$$= \sum_{i=0}^{M} (-1)^{M-i} \binom{M}{i} i^N = M! \sum_{i=0}^{M} \frac{(-1)^{M-i}\, i^N}{(M-i)!\, i!}, \tag{3}$$

where $S(N,M)$ counts the number of ways to arrange all N outputs into M unlabeled groups, with no group empty. The factorial $M!$, alternatively ${}_M P_M$, then counts the ways to label the M groups (as M events). $S(N,M)$ are known as the *Stirling numbers of the second kind* [13], defined as the number of ways to partition a set of N elements into M non-empty subsets. The On-Line Encyclopedia of Integer Sequences (OEIS) [14] provides values for the Stirling numbers in entry A008277, as well as the pre-calculated (2) and its maximum value for each N in entries A019538 and A002869 respectively. To provide an understanding of the equation beyond Stirling numbers, (3) is an adaptation from [15, eq. 6.19] and the last step follows from the definition of the binomial coefficients. Note that the factorial is absorbed by the binomial in (2) and from symmetry some references substitute $j = M - i$.

The case with $N = 0$ slots has no practical application, but $M = 0$ events occur whenever there is "silence". For these cases, $S(N,0) = 0$ and there are indeed no ways to label the output if there are no events. However, following (1), originally from [1], and defining

$$\underset{\text{for } M = 0,\, M < N}{K(N,M)} = 1 \tag{4}$$

makes sense, considering this as a special case where each output slot is expected to approach the zero-event $\oslash$. The number of slots should be chosen to exceed any expected number of events. Whenever $M > N$, there are too few slots to output all events, and thus no slots for duplicates. PIT can still score a (best subset) loss for the outputs given. The number of sequences are then *Count B0* [12],

$$\underset{\text{for } 1 \le N < M}{K(N,M)} = {}_M P_N, \tag{5}$$

which concludes all cases.

3.2. Optimal overlay

In the baseline implementation, the overlays are not used to their full potential. Instead of just safeguarding against empty entries, each overlay can permute one M category, see example in Table 3. The order of sequences within each overlay does not matter here, but it may be noted that the Overlay B arrangement is chosen for the purpose of upcoming Section 3.3.

The optimal overlay strategy has a certain beauty for the $N = 3$ case, since two of the three cases are exactly six entries. For other cases, one permutation category will dominate, and as for Overlay A there will be redundancy. Yet, this strategy is a simple drop in replacement that reduces loss iterations from the sum of all categories to the count of the largest category. For the implemented case, this halves the work instantly. We would also suggest moving all of the overlay calculations out of the training loop. Instead of passing six dummy events from preprocessing, provide the six sequences for the relevant category. Make sure to store the data in an efficient format, so that managing the increased data footprint is not slower than constructing the sequences on the fly.

Table 3
Optimal overlay strategy. Depending on M, only A, B or C is active. Inactive entries are zero and the overlays are added entry-wise to form the target sequences. Thus, when A is active, all six sequences are A0 A0 A0. When B or C is active, each of the sequences are unique.

Overlay C	*Overlay B*	*Overlay A*
C0 C1 C2	B0 B0 B1	A0 A0 A0
C0 C2 C1	B0 B1 B0	—"—
C1 C0 C2	B1 B0 B1	—"—
C1 C2 C0	B1 B1 B0	—"—
C2 C0 C1	B0 B1 B1	—"—
C2 C1 C0	B1 B0 B0	—"—

An additional insight is that the loss evaluation returns the gradients and the loss value, but not the event permutation order. Doing so would be slightly more complicated with this more compact overlay format. It is crucial to keep track of the calculated gradients, but the *min* function is a well established pooling function that should not pose any problem.

As for the baseline implementation, the loss function would evaluate each slot against the given event sequence and sum the components. The time complexity of this strategy is mainly determined by the distance calculations and matrix multiplications involved in evaluating the loss for a sequence. This approach evaluates $6 \times 3 = 18$ pairs instead of 13×3. The reduced number of sequences is a great improvement, but even for the Overlay C category, each token-event pair is evaluated twice.

3.3. Distance first

Although the permutations help make sure that all token-event pairs are evaluated, separating the enumeration of pairs from the permutation of events is our next approach. There are $N \times M = 9$ token-event pairs for the worst category. Lining up these pairs for vectorized calculation then turns the loss of each permutation into simple sums.

To make this a branch free algorithm, we break down the sums formed by the overlays in Table 3 and calculate each component separately. The sequences are reformulated in Table 4 using dummy event sums. There are six unique sums Xj, and when paired with the matching slots of the sequence S0S1S2, it turns out to be twelve unique pairs. This is slightly higher than the "worst category" nine, and a trade-off for being branch free. Due to an intentional symmetry-breaking ordering chosen, one sequence (in bold, X4X5X0) is entirely made up from pairs that are not reused. Thus the row can be handled as a sequence loss rather than the separate pairs, but from our complexity perspective that still calculates 12 loss-pairs.

Remember that sequence X4X5X0, and thus ⑫ ⑪ ⑩, is an overlay of three sequences where only one is active. We can replace it with two sequences that focus on each category instead. The C2C0C1 sequence is also X3X1X2, which is available for reuse in pairs ⑨, ⑦ and ⑤. When category C is active that concludes the needed permutations, for category B it produces B1B0B0 which is a permutation already covered (by the last sequence). Calculating B1B0B0 twice is wasteful, but harmless since we reuse existing pairs. In the same way, A0A0A0 is an obvious repetition, and we will ignore category A henceforth. The B0B1B1 sequence is still needed to close the category B permutations. The X1X3X3 is not possible as it would produce the C0C2C2 sequence, which is not a category C permutation since

Table 4
The sequences of Table 3 already overlayed, using dummy event sums named Xj. a) The dummy sequences with the events in slot order, and one permutation (based on category) per row. Then unique token-event pairs enumerated ① to ⑫ for clarity. The bold sequence is of special interest for the discussion. b) The unique dummy event sums. As before, only A, B or C is non-zero, depending on M.

a) Dummy sequences	
X1 X2 X3	① ② ③
X1 X3 X2	① ④ ⑤
X0 X1 X3	⑥ ⑦ ③
X0 X3 X1	⑥ ④ ⑧
X4 X5 X0	**⑫ ⑪ ⑩**
X3 X2 X1	⑨ ② ⑧

b) Dummy event sums
X1 = C0+B0+A0
X2 = C1+B0+A0
X3 = C2+B1+A0
X4 = C2+B0+A0
X5 = C0+B1+A0
X0 = C1+B1+A0

it allows duplication instead of the true C1 event. Instead we propose X1X3X0 (giving B0B1B1 and redundant C0C2C1), i.e. pairs ① and ④ reused and only ⑩ used only once. In total this makes ten loss iterations and seven sums to find the min loss. Instead of six dummy events sums, this strategy only uses four (discarding X4 and X5), which can be pre-processed. We have not found a way to reach nine loss iterations in a completely branch free manner.

3.4. Branching benefits (BRADPIT)

So far, we focused on the loss calculation for a single scope. Since Multi-ACCDOA has a fixed N for each scope, loss is calculated for many slots even if it is unlikely to have overlapping events. With many classes, the most likely GT is zero events for most of them, possibly one event for a few and in rare cases overlapping events of one class. The statistics of course depends on the dataset, but the general distribution is likely biased towards sparsity. An approach based on branch free processing needs to cover the worst case, even with an efficient implementation. There should be an advantage for an implementation that can recognize the majority of scopes as category A or ⊘ and avoid any permutation cost for them.

If we know the ground truth event number M, can we branch out the different categories without breaking the gradients? Will it lead to faster learning? Theoretically, a dataset biased towards the best case should benefit from an implementation with ability to adapt to M. Analysis for a specific use-case is needed in order to know if there is a significant improvement to learning time. Any improvements are still to the coefficient, not the $\mathcal{O}$ time complexity. The benefit however grows when N grows, as long as the dataset is biased towards lower M.

In pre-processing, pass on the specific value of M, it may replace the "active" field currently in the baseline implementation ($M > 0$ is active). A first approach simply finds the maximum M in the current batch and branches to an efficient implementation for that maximum. If a batch has only scopes of $M < 2$, the entire batch can have a permutation free loss calculation. From a gradient perspective, processing the entire branch the same way is essentially branch free. Note that the variation of training data within a batch should reflect the stochastic nature of the dataset. Batching samples strategically to achieve higher M consistency would impact learning negatively. Sorting within a batch just before loss calculation is however not changing the results, which leads to a second approach.

A batch is a number of samples grouped together to produce a common loss and a joint gradient. The loss is however calculated for several similar parts (e.g. individual scopes) and then joined (the average). Grouping the parts into sub-batches based on M can improve speed without affecting the total batch average. Since each time frame can have multiple scopes with different M, this strategy opts to separate them even if that breaks the benefit of continuous memory layout. Most importantly, the $M < 2$ categories are handled together, in the same way as above. The remaining few scopes can then be grouped together, also based on the maximum as above. The difference to the first approach may seem small, but it guarantees that the majority of the scopes gets the optimal implementation. If N is high or the batch size is large, more than two groupings may be beneficial. Groups that are "downwards compatible" to any lower M can accept spill-over from lower categories, which may help

balance vectorization. When a group is instead completely homogeneous, the implementation may be faster than one that caters to more categories. The earlier example for $N = 3$ has six pairs for $M = 2$, nine pairs for $M = 3$, but ten pairs for the combined implementation. A difference this small is however not worth pursuing, unless there are plenty of iterations in each group.

So far, the sub-batches are processed separately, only joined for the averaging step. The third approach instead follows a map-reduce strategy to join the steps that can be vectorized regardless of category: Once the token-event pairs have been formed (based on M), the MSE calculation is generic; with a list of templates referring to the MSE, the sums are generic. In a balanced manner, some steps may combine all categories, others split categories. This approach is probably not useful unless a well balanced implementation can be found. It is not obvious that a high N would make that more likely, the imbalance is rather increasing from more diversity.

4. Results

The loss execution cost is approximately $\mathcal{N}_{iter}$, the total number of token-event pairs evaluated in an epoch, as in (6). $\mathcal{N}_{pairs}$ may be different in each scope. If the pairs are not evaluated individually, the number depends on the number of permutation sequences, (8). With a frame-class scope, the number of scopes is (7).

$$\mathcal{N}_{iter} = \mathcal{N}_{scopes} \times \mathcal{N}_{pairs} \tag{6}$$

$$\mathcal{N}_{scopes} = \mathcal{N}_{frames} \times \mathcal{N}_{classes} \tag{7}$$

$$\mathcal{N}_{pairs} = \mathcal{N}_{seqs} \times N \tag{8}$$

A theoretic execution cost of these loss implementations on a fictive dataset similar to STARSS23 [16] is shown in Table 5. There are 13 classes, 200k frames and no frames have $M > 5$. Only 30k of the frames have $M > 1$, and even there it is unlikely that the overlapping events are of the same class. BRADPIT results depend on how many scopes that have $M > 1$ and we present two approximations. The optimistic value assumes that each scope can be calculated as a single sequence. This is not true, but even if we assume that each frame with overlap affects one scope, it only adds the cost for 30k scopes to the pessimistic $\mathcal{N}_{iter}$. The pessimistic $\mathcal{N}_{scopes}$ is unrealistic for frame-class scope, since not all overlapping events are of the same class.

Table 5
Approximative execution cost improvement gain factor relative to baseline. The pessimistic $\mathcal{N}_{scopes}$ value for BRADPIT adds the $M > 1$ category miscounted by the optimistic case. $\mathcal{N}_{pairs}$ for these scopes is the "dist. first" minus the optimistic number. For $N = 3$ the difference in $\mathcal{N}_{iter}$ is insignificant. For $N = 5$ the total cost is likely closer to the pessimistic result. k is thousands and M is millions.

	$N = 3$ *(with frame-class scope)*				$N = 5$ *(with frame scope)*			
Permutation Strategy	$\mathcal{N}_{scopes}$	$\mathcal{N}_{pairs}$	$\mathcal{N}_{iter}$	*gain* ↑	$\mathcal{N}_{scopes}$	$\mathcal{N}_{pairs}$	$\mathcal{N}_{iter}$	*gain* ↑
Baseline	$200\text{k} \times 13$	13×3	101M	1	200k	540×5	540M	1
ARMPIT opt. overlay	$200\text{k} \times 13$	6×3	46M	2.16	200k	240×5	240M	2.25
ARMPIT dist. first	$200\text{k} \times 13$	10	26M	3.9	200k	< 100	< 20M	> 27
BRADPIT (optimistic)	$200\text{k} \times 13$	1×3	7.8M	13	200k	1×5	1M	540
BRADPIT (pessimistic)	+30k	$10 - 3$	8.0M	12.7	+30k	$100 - 5$	3.85M	140

The Multi-ACCDOA format has multiple slots per class. Since overlapping events of the same class is rare, nearly all slots are empty or duplicates. If the scope is instead frame-wise, and the output format and loss function can handle tokens that combine class and location, selecting $N = 5$ is an option. With the baseline implementation, the cost for that may seem prohibitively large. With ARMPIT or BRADPIT however, Table 5 indicates that $N = 5$, from the perspective of loss iterations, may be better than $N = 3$. We have not done a full analysis of the $\mathcal{N}_{pairs}$ for $N = 5$, and these results should be considered speculative. In particular, ARMPIT "dist. first" has 5×5 worst category token-event pairs, but due to insights from Section 3.3, that number is quadrupled to give a conservative estimate that allows for a branch free loss.

The optimistic BRADPIT value for $N = 5$ again assumes that each scope can be calculated as a single sequence. This is not realistic, in fact the actual $M > 1$ count is 30k and the pessimistic number takes that into account. We consider it pessimistic as the ARMPIT pairs are likely higher than needed, but this time the reality is likely closer to pessimism. The reason is that the frame-scope already removed the empty scopes, and the penalty for higher N grows fast. Also note that the summing of sequences is disregarded in the metric, but that is increasingly costly when N grows.

5. Conclusions

How permutations are evaluated for an ADPIT loss implementation can potentially improve the evaluation time significantly. With these improvements in place, implementing frame-wise ARMPIT with $N = 5$ does not seem infeasible for STARSS23, perhaps even a better option than Multi-ACCDOA with $N = 3$. A natural continuation of the work is to verify the results with an implementation. We here present $K(N, M)$, the accurate number of ADPIT permutations for each category.

References

[1] K. Shimada, Y. Koyama, S. Takahashi, N. Takahashi, et al., Multi-ACCDOA: Localizing and detecting overlapping sounds from the same class with auxiliary duplicating permutation invariant training, in: Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2022, pp. 316–320.

[2] A. Chen, Y. Zhang, J. Jia, J. Diffenderfer, et al., DeepZero: Scaling up zeroth-order optimization for deep model training, in: International Conference on Learning Representations (ICLR), 2024.

[3] D. Kingma, J. Ba, Adam: A method for stochastic optimization, in: International Conference on Learning Representations (ICLR), 2015.

[4] M. Sipser, Introduction to the Theory of Computation, 2nd ed., Thomson Course Technology, 2006.

[5] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, 2024. URL: http://www.scipy.org/, version 1.14.0.

[6] D. Yu, M. Kolbæk, Z.-H. Tan, J. Jensen, Permutation invariant training of deep models for speaker-independent multi-talker speech separation, in: International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2017, pp. 241–245. doi:10.1109/ICASSP.2017.7952154.

[7] Y. Cao, T. Iqbal, Q. Kong, F. An, et al., An improved event-independent network for polyphonic sound event localization and detection, in: International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2021, pp. 885–889. doi:10.1109/ICASSP39728.2021.9413473.

[8] A. Berg, et al., Learning multi-target TDOA features for sound event localization and detection, in: Detection and Classification of Acoustic Scenes and Events (DCASE), 2024, pp. 16–20.

[9] Y. Luo, N. Mesgarani, Separating Varying Numbers of Sources with Auxiliary Autoencoding Loss, in: Proc. Interspeech 2020, 2020, pp. 2622–2626. doi:10.21437/Interspeech.2020-34.

[10] H. W. Kuhn, The hungarian method for the assignment problem, Naval Research Logistics (NRL) 52 (1955). URL: https://api.semanticscholar.org/CorpusID:9426884.

[11] D. F. Crouse, On implementing 2D rectangular assignment algorithms, IEEE Transactions on Aerospace and Electronic Systems 52 (2016) 1679–1696. doi:10.1109/TAES.2016.140952.

[12] R. A. Proctor, Let's Expand Rota's Twelvefold Way For Counting Partitions!, 2007. URL: https://arxiv.org/abs/math/0606404.

[13] O.-Y. Chan, D. V. Manna, Congruences for Stirling numbers of the second kind, in: Gems in Experimental Mathematics, 2010, pp. 97–111. doi:10.1090/conm/517.

[14] OEIS Foundation Inc., The On-Line Encyclopedia of Integer Sequences, 2024. At http://oeis.org.

[15] R. L. Graham, D. E. Knuth, O. Patashnik, Concrete Mathematics: A Foundation for Computer Science, second ed., Addison-Wesley, Reading, MA, 1994.

[16] K. Shimada, A. Politis, P. Sudarsanam, D. A. Krause, et al., STARSS23: An audio-visual dataset of spatial recordings of real scenes with spatiotemporal annotations of sound events, in: Advances in Neural Information Processing Systems, volume 36, 2023, pp. 72931–72957.