Renée: End-to-end training of Extreme Classification Models

Vidit Jain [1] **Jatin Prakash** [1] **Deepak Saini** [2] **Jian Jiao** [2] **Ramachandran Ramjee** [1] **Manik Varma** [1]

Abstract

The goal of Extreme Multi-label Classification (XC) is to learn representations that enable mapping input texts to the most relevant subset of labels selected from an extremely large label set, potentially in hundreds of millions. Given the extreme scale, conventional wisdom believes it is infeasible to train an XC model in an end-to-end manner. Thus, for training efficiency, several modular and sampling-based approaches to XC training have been proposed in the literature. In this paper, we identify challenges in the end-to-end training of XC models and devise novel optimizations that improve training speed over an order of magnitude, making end-to-end XC model training practical. Furthermore, we show that our end-to-end trained model, *Renée* delivers state-of-the-art accuracy in a wide variety of XC benchmark datasets. Code for *Renée* is available at https://github.com/microsoft/renee.

1 Introduction

Extreme Multi-label Classification (XC) deals with the problem of mapping input data points to one or more labels, where the labels come from an extremely large set, potentially in the hundreds of millions. This problem arises in many practical settings; for example, a search engine needs to identify relevant ads to display corresponding to a user query, or an e-commerce website needs to display relevant products when a user searches for an item.

Given the importance of this problem, it has been studied extensively in the literature. Early approaches used sparse linear models (Babbar & Schölkopf, 2017; Prabhu et al., 2018) while recent approaches deliver superior performance using deep learning models (Liu et al., 2017; You et al., 2019; Chang et al., 2020; Zhang et al., 2021a; Dahiya et al., 2022). However, *it is widely acknowledged that end-to-end training of deep learning models is simply infeasible at XC scale* (Chang et al., 2020; Zhang et al., 2021a; Dahiya et al., 2022). By end-to-end training, we refer to training that uses all available training data and updates all model parameters based on a task-specific loss function.

For scalability, prior work decomposes the end-to-end problem into modular sub-problems with surrogate loss functions (Chang et al., 2020; Dahiya et al., 2021a;b) but at the cost of potential accuracy gains. Further, several negative mining techniques have been developed (Xiong et al., 2021; Dahiya et al., 2022) to improve training performance. These techniques select, for each data point, a subset of labels called "hard negatives", i.e., negative labels that are likely to be confused with positive labels. Then, each data point is trained using only the selected small subset of positive and hard negative labels. Using fewer labels improves training speed but hurts accuracy if incorrect negatives are chosen.

Deep learning-based XC models comprise an encoder (e.g., a transformer like BERT (Kenton & Toutanova, 2019)) and a classifier that maps the encoder embeddings to output classes. Chang et al. (2020) show that end-to-end training of a deep-learning-based XC model with 1 Million classes runs out of GPU memory. If the encoder embedding size is 1024 and there are 1 Million output classes, the classifier alone will have 1 Billion parameters and require 16 GB of memory (Rajbhandari et al., 2020). One could use a model-parallel architecture to distribute the model over multiple GPUs, but we do not see such a baseline utilized in the literature, presumably, due to its exorbitant training costs.

In this paper, we take an in-depth look at the cost of end-to-end training of XC models. We identify key memory and compute costs of the classifier and several optimizations for reducing these costs. First, we use a novel loss shortcut technique to optimize memory usage: instead of the usual approach of computing the loss during the forward pass and using automatic differentiation for the backward pass, which requires maintaining a large amount of intermediate state, we short circuit the computation by skipping the loss altogether, thereby, saving a large amount of memory and compute. Second, we adopt a hybrid data and model-parallel

[1]Microsoft Research, Bangalore, India [2]Microsoft. Correspondence to: Vidit Jain <jainvidit@microsoft.com>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

architecture, where encoders are trained in a data-parallel fashion, and the large classifier is trained in a model-parallel manner. Finally, we employ several optimizations such as a split backward pass for overlapping compute with communication, a bottleneck layer between encoder and classifier, and kernel fusion to improve performance further. We call our extReme ENd-to-End trainEd model, *Renée*.

We evaluate *Renée* on publicly available XC datasets (Bhatia et al., 2016) that have up to 3 million classes and a proprietary dataset that has 120 million classes. Apriori, it is unclear whether end-to-end training at XC scale can achieve high accuracy or if it will even converge.

On datasets without label features (e.g., opaque product id label), we show that *Renée* achieves significant gains over prior state-of-the-art results. For example, *on Amazon-670K, Renée achieves Precision@1 of 54.23, 5% higher than 49.11 achieved by previous state-of-the-art XR-Transformer model* (Zhang et al., 2021a). On datasets that have label features (i.e., label is also a text), we show that *Renée* can achieve state-of-the-art results when we either augment the input training data with additional label text-based data points or start with an encoder initialized via pre-training on the dataset.

Finally, we show that end-to-end training at an extreme scale is also practical. For smaller datasets with up to a few million classes, we find that *Renée trains* $3-14\times$ *faster than a standard hybrid data-model parallel implementation and has comparable or lower total training time than prior approaches*. We also show that an 8 billion parameter *Renée* model with 120 million classes can be trained on a single DGX-2 node with 16 V100 GPUs and is $15\times$ faster than the standard implementation.

In summary, we show that end-to-end training of extreme classification models is not only practical but that it can provide meaningful accuracy gains over prior approaches that viewed end-to-end training as simply infeasible. We believe an end-to-end trained model like *Renée* can serve as a strong baseline for further research in this important area. Code for *Renée* is available at https://github.com/microsoft/renee.

2 Related Work

Extreme Classification. Early works in XC primarily focused on training accurate and scalable classifiers for fixed features such as bag-of-words (Babbar & Schölkopf, 2017; Prabhu et al., 2018; Khandagale et al., 2020) or pre-trained features such as FastText (Joulin et al., 2016) or CDSSM (Jain et al., 2019; Huang et al., 2013).

Recent works offer much better accuracy by utilizing deep learning, learning task-dependent features and classifiers. This class of algorithms use a range of diverse feature extraction architectures, including bag-of-embeddings (Medini et al., 2019; Dahiya et al., 2021b;a), CNN (Liu et al., 2017), attention (You et al., 2019), and transformer-based architectures (Jiang et al., 2021; Chang et al., 2020; Zhang et al., 2021a; Dahiya et al., 2022). Transformer-based architectures are the current state-of-the-art models in XC.

Modular training and Negative Mining. The conventional wisdom in the XC community is that training with all labels becomes infeasible when the number of labels is large (Chang et al., 2020; Dahiya et al., 2021a;b). Therefore, there has been much focus on selecting a small set of irrelevant labels, or "negatives", per training point for scaling XC algorithms. Broadly, negative-mining techniques can be classified into three categories: random (Guo et al., 2019; Chen et al., 2020b), feature-aware (Lee et al., 2018; Hofstätter et al., 2021) and task-aware (Qu et al., 2020; Dahiya et al., 2021a; 2022), each class with its trade-offs in computational cost and quality of negatives.

Moreover, existing literature approaches the XC task in a modular fashion, generally training the feature extraction module on surrogate tasks in the earlier modules and then freezing it in later modules to generate negatives efficiently (Dahiya et al., 2021b; Yu et al., 2022). This, again, is done to enable scaling to the case where labels are in millions; however, it is well known that end-to-end pipelines often beat their modular counterparts (Mirowski et al., 2016).

One-vs-all methods (OvA). OvA methods such as ProXML (Babbar & Schölkopf, 2019) and DiSMEC (Babbar & Schölkopf, 2017) treat each label as a binary classification problem and classification tasks independent of each other. This class of models had mainly been ignored due to expensive computational costs. As a result, targeted optimizations have been made to improve OvA scalability (Yen et al., 2017; Prabhu et al., 2018; Chen et al., 2020a; Zhang et al., 2021a). However, the current state-of-the-art XC methods (Dahiya et al., 2022) based on negative mining are more accurate across many tasks.

Systems approaches. Several techniques (Chen et al., 2018; Li et al., 2020; Zheng et al., 2020; 2022) have helped improve the performance of deep learning training jobs. However, the skip loss optimization used in *Renée* is only possible by giving up the final loss calculation, a trade-off that cannot automatically be identified by these techniques that preserve fidelity. We evaluated various compilers to empirically validate this. Alpa (Zheng et al., 2022) is able to identify a parallelization plan on a simplified version of an end-to-end XC model but is unable to perform the skip loss optimization. Apache TVM Unity (Chen et al., 2018) uses graph optimization which again cannot do the skip loss optimization without explicitly removing the loss computation from the graph. Finally, TASO (Jia et al., 2019a) is targeted only towards model inference.

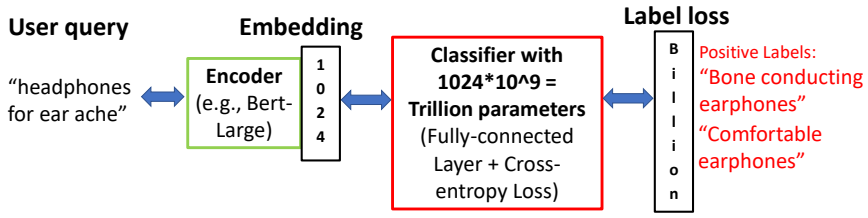

Figure 1. Renée model with 1 Billion classes

Several systems have focused on optimizing the performance of large models by distributing them efficiently across multiple GPUs (Shazeer et al., 2018; Huang et al., 2019; Rajbhandari et al., 2020; Rasley et al., 2020). Techniques for automating this distribution have also been proposed (Jia et al., 2019b; Athlur et al., 2022).

While *Renée* could benefit from such systems when used with a very large encoder, for the encoder sizes evaluated in this paper, a simple hybrid data-model parallel architecture (Krizhevsky, 2014) is most efficient. Finally, the compute-communication overlap optimization of *Renée* could be incorporated in some of these systems.

Offloading-based approaches(e.g., (Ren et al., 2021)) have helped alleviate the memory bottleneck issues associated with large model training by leveraging compute and memory resources on the host CPU to execute the optimizer. Such approaches may enable training on a larger scale but lead to large overheads due to CPU-GPU memory migration. Along with the skip-loss optimization, which gives performance gains orthogonal to that of offloading-based approaches, *Renée* also uses a split-optimizer strategy to reduce the parameter memory by up to 50%.

3 *Renée* Bottleneck Analysis

Figure 1 illustrates end-to-end training of a *Renée* model for a potential search-based retrieval application. The input is a user search query, and the model maps it into one or more labels out of an extremely large set of one billion labels (e.g., advertiser keywords in case of sponsored search). To accomplish this mapping, the model consists of an encoder and a classifier. The training dataset consists of user queries and a few positive labels for each query, with the rest of the labels considered negative. The encoder computes an embedding for the user query and the classifier uses the embedding and the labels to compute the cross-entropy loss for each of the billion labels. Encoder and classifier gradients are then calculated by backpropagation, thereby achieving end-to-end model training.

Typical encoders used in the recent literature are all BERT-based models such as 6-layer distilBERT (Sanh et al., 2019), 12-layer BERT-base (Kenton & Toutanova, 2019), 24-layer RoBERTa-large (Liu et al., 2019), etc. These models produce an embedding vector of size 768 or 1024 (large). If the XC application has a billion labels, a fully-connected classifier will have a trillion parameters, which is cost-prohibitive to train. We now take a closer look at the compute and memory costs involved in training such a model, which will motivate our design choices in the next section.

3.1 Compute

The *Renée* model consists of an encoder and a classifier. The typical encoder used today is a BERT-based model. The compute requirements for training a BERT-large model (330 Million parameters) with a sequence length of 512 is 1 TFlop/example, accounting for all the matrix multiplication operations that comprise the bulk of the cost of training such a model. Now consider the classifier. In the forward pass, assuming a batch size of B, a matrix of $B \times 1024$ is multiplied with a matrix of 1024×10^9, which results in approximately $2B \times 10^{12}$ operations or 2 TFlops/example. Accounting for another two similar-cost matrix multiplications in the backward pass, the classifier-compute cost becomes 6 TFlops/example. The classifier compute requirement is significant but not so outrageous even at the scale of 1 Billion classes, especially when compared to training large encoders. Thus, training an XC model in an end-to-end manner is not infeasible from a compute point of view.

3.2 Memory

BERT-large encoders with 330 Million parameters can be fine-tuned on a single GPU with as little as 16GB memory. Let us now look at the memory requirements of the classifier. The number of classifier parameters is $embed\_size \times num\_classes$. For an embedding size of 1024 and 1 billion classes, the classifier has 1 Trillion parameters. This implies that just the model parameters (including gradients and optimizer state for ADAM, which holds first and second moments in fp32 precision) will occupy $16 * num\_parameters$ (Rasley et al., 2020) bytes = 16 TB of memory, or 1000x the memory requirements of the encoder!

Now consider the memory requirements for intermediate states, such as activation buffers. Figure 2 depicts a simplified version of the various steps involved in computing

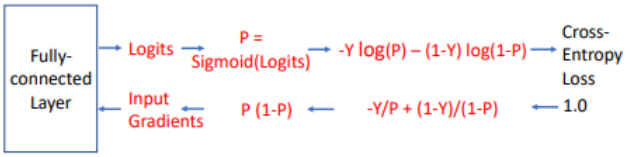

Figure 2. Renée classifier. Y = 1 (0) for positive (negative) labels.

Algorithm 1 Sparse BCE loss computation

```
//x is output logits of shape (batch_size, num_classes)
//pos_1,pos_2 are indices where target is positive
loss = x.clamp(min=0.0).sum()
loss -= x[pos_1,pos_2].sum()
loss += (1+(-torch.abs(x)).exp()).log().sum()
```

the cross-entropy loss from the logits that are output from the matrix multiplication. For each input we must store four vectors of size $num\_classes$ in full 32-bit precision: i) the logits, ii) the probability matrix, P, iii) the full target label matrix, Y, and iv) the gradients of logits with respect to the final loss. Therefore, the classifier requires $16{*}num\_classes$ bytes = 16 GB per input example for computing the loss and the gradients of the loss in the backward pass. For typical mini-batch sizes of 1K, the intermediate state for the classifier will match the parameter memory requirements of 16 TB!

Thus, the infeasibility of end-to-end training comes predominantly from the memory requirements for the classifier, and it is clear why (Chang et al., 2020) reported that end-to-end training went out-of-memory at a batch size of one even for XC models with 1 million classes.

4 Design

This section discusses various design choices that optimize the memory needed for the classifier and, as a side-effect, also optimizes the classifier's compute requirements. Note that this paper will not focus on optimizing the encoder as i) the encoder's requirements are not as challenging as the classifier's and ii) encoder architecture is an active area of research and a suite of encoders of varying compute and memory requirements such as TinyBERT (Jiao et al., 2020), DistilBERT (Sanh et al., 2019), etc. are available.

4.1 Optimizing Intermediate Memory

As we saw in Section 3, the intermediate memory required in the loss computation can be massive, as much as 16GB per example. We now look at various ways of reducing this cost. In this paper, we will discuss the optimization details using the specific example of binary cross entropy loss (which we use in our evaluation), but the techniques described are broadly applicable and would result in similar savings for other loss functions, such as Softmax as well.

The outputs of the linear layer of the classifier are known as logits. As shown in Figure 2, these logits are first converted into probabilities for each class by applying the sigmoid function on the logits. Using the probabilities, P, and the actual class labels, Y, where Y=1 is a positive label and 0 is negative, the cross-entropy loss is given by

$$BCE = -Ylog(P) - (1-Y)log(1-P) \quad (1)$$

However, computing the loss as described above can lead to numerical stability issues. This is because as the network drives P (or $1-P$) close to 0, log(P) tends to $-\infty$, and gradients tend to ∞, leading to overflows and NaNs.

Thus, the cross-entropy loss is typically computed in an equivalent but stable way by combining the sigmoid computation of probabilities and equation 1.

This loss function is called BCEWithLogitsLoss (PyTorch) or $sigmoid\_cross\_entropy\_with\_logits$ (TensorFlow). It takes the output logits and the full target label matrix of dimensions $(batch\_size, num\_classes)$. However, this is inefficient in XC applications since $num\_classes$ is large while most of the elements in the label matrix are zeros. Thus, our first optimization is to convert the standard BCE loss computation to use sparse target labels. If the output logits are x, a numerically stable BCEWithLogitsLoss is computed as:

$$\begin{aligned} BCE &= -Ylog(\frac{1}{1+e^{-x}}) - (1-Y)log(\frac{e^{-x}}{1+e^{-x}}) \\ &= Ylog(1+e^{-x}) + (1-Y)(x + log(1+e^{-x})) \\ &= x - xY + log(1+e^{-x}) \\ &= max(x,0) - xY + log(1+e^{-|x|}) \end{aligned} \quad (2)$$

Sparse loss computation. Equation 2 is numerically stable because the term $1+e^{-|x|}$ is bounded in range 1 to 2, making the log stable. This equation can be converted to a sparse computation as shown in Algorithm 1 with the three lines computing the three terms of the equation. Instead of passing the full label matrix, we pass in the index vectors where the label matrix elements are positive and compute xY in equation 2 by indexing into only those locations of x where label elements are positive and taking their sum.

While the above optimization saves the memory needed for the full label matrix, it still has many other intermediate states needed for the gradient computation. Can we optimize the memory usage further? The derivative of the sigmoid and loss functions are shown in the bottom row in Figure 2. If we fuse the derivatives of all of the terms in the bottom, the input gradient to the linear layer is given by

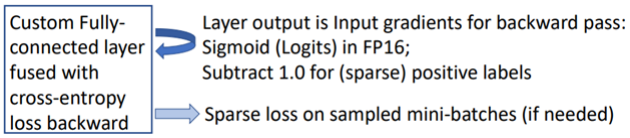

Figure 3. Optimized classifier with fused linear and cross entropy loss backward layer and optional sparse loss computation.

$$\begin{aligned} grad &= P(1-P)*(-Y/P+(1-Y)/(1-P))*1 \\ &= -(1-P)Y+P(1-Y) \\ &= P-Y \\ &= 1/(1+e^{-x})-Y \end{aligned} \tag{3}$$

Skipping loss computation. *Instead of computing the loss and employing automatic differentiation in the backward pass as is typically done, we can skip loss computation entirely and go directly from the output logits x of the linear layer to the input gradient as shown in Figure 3, avoiding all the intermediate computations of Figure 2!* In fact, we can fuse the input gradient computation shown in equation 3 into the matrix multiplication kernel of the linear layer. Since the output logits matrix with shape $(batch\_size, num\_classes)$ is large, reading and writing the matrix for computing equation 3 as a separate kernel will be memory bound and expensive. By fusing it along with the matrix computation in a single kernel, we read in the matrix output from fast local registers, compute the gradient and write out the gradients into memory. Also note that we don't need the full label matrix Y in equation 3 – since most of the values are zeros, we can perform a sparse computation by subtracting 1 from only those gradient indices where the labels are positive. Therefore, while skipping the loss computation, we must only store the logits and the logit gradients. We avoid another copy to store logit gradients by kernel fusion, consequently needing only one vector of size $num\_classes$ per input. Thus, skipping loss computation eliminates the need to store 3 out of 4 large matrices, leading to a $4\times$ memory reduction. Finally, equation 3 is numerically stable in 16-bit precision (unlike the loss computation, which requires 32 bits) and requires only 2 bytes for each of the output terms. Thus, *by skipping loss computation, we require only* $2*num\_classes$ *bytes of memory per example, an* $8\times$ *memory reduction compared to the standard XC implementation (Section 3)*

While computing the final loss is not essential for gradient-descent training, some users may still want to monitor loss to make sure the training is proceeding well. For this, one can use the sparse loss computation on sampled mini-batches (e.g., 1 in 100) which results in accurate loss estimates. This will negate the fused matrix multiply optimization for those samples (since output logits are necessary for computing loss), but it still avoids the full cost of the automatic differentiation approach for most of the mini-batches.

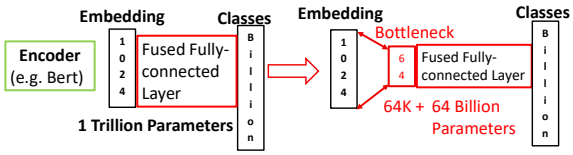

Figure 4. Use of a bottleneck layer to optimize classifier parameter memory and compute requirements.

4.2 Optimizing Parameter Memory

The second significant component of XC model memory usage is the memory required for the classifier parameters and the associated optimizer state. This depends on the number of classifier parameters, given by $embed\_size * num\_classes$. Since the number of classes is a function of the problem domain, the only way to reduce the number of classifier parameters is to reduce the embedding size.

Bottleneck layer. Thus, a simple way to reduce parameter memory is to introduce a *bottleneck layer* between the encoder and the classifier. The bottleneck layer is a linear layer that maps the encoder output from a high-dimension space to a low one. For example, as shown in Figure 4, a linear layer with 64K parameters can map a 1024-sized embedding to a 64-sized smaller embedding. This implies that the number of classifier parameters is now $64*num\_classes$ or a 16x reduction in parameter memory, which can more than makeup for the additional 64K parameters of the bottleneck layer for most XC applications.

However, using the bottleneck layer can reduce the model accuracy. In our experiments, we found that a 64-sized embedding can deliver accuracy within 1% of a full-sized embedding. Thus, the trade-off may be worthwhile, especially when the number of classes is in the hundreds of millions. Note that the $16\times$ reduction in parameter also results in a $16\times$ reduction in compute required for the classifier.

Split optimizer. A second reason for the high cost of parameter memory is the use of Adam optimizer (Kingma & Ba, 2015), which requires the storage of first and second moments in full 32-bit precision. While Adam is critical for training transformer-based encoder models like BERT, we found that the classifier can be trained equally well using the SGD optimizer, which does not require second moments. In fact, in some models, we found that SGD without momentum can match the accuracy of SGD with momentum. Thus, in *Renée* we use the Adam optimizer to train the encoder and SGD (with or without momentum) to train the classifier. This further reduces the memory required for the classifier parameters by 25% (or 50% without momentum). Thus, the 16 TB required for a 1 Billion class XC model (Section 3) is now reduced through the use of a bottleneck layer and SGD

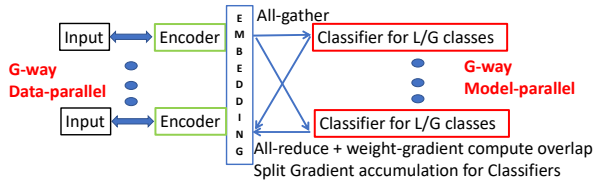

Figure 5. Hybrid data-model parallel architecture with all-reduce communication overlapped with weight gradient computation and split gradient accumulation for classifiers.

optimizer to about 768 GB (or 512 GB), i.e., up to a 32× memory reduction.

4.3 Hybrid Data-Model Parallel Training

While the optimizations described so far can reduce the memory needed for end-to-end training by an order of magnitude or more, the memory requirements may still be beyond the capabilities of a single GPU (e.g., 32GB V100) when the number of classes is in the hundreds of millions. In these cases or for speeding up training, even when a single GPU can fit the model, it makes sense to utilize multiple GPUs to train a *Renée* model.

While various data-parallel and model-parallel architectures have been explored in the literature (Rasley et al., 2020), a hybrid architecture (Krizhevsky, 2014; Narayanan et al., 2019) that uses a mix of data-parallel and model-parallel elements is the best fit for a *Renée* model.

Figure 5 depicts the hybrid data-model parallel architecture for training *Renée* on G GPUs. The encoders are trained in a data-parallel manner, while the classifier is split in a model-parallel manner among the GPUs. Thus, the encoders produce encoding of the input data in parallel. An all-gather is then used to distribute the embedding to all GPUs, which then train the classifier for a subset L/G of the classes. In the backward pass, the gradients are all-reduced (or reduce-scattered) and sent to the respective encoders.

Overlapping all-reduce with computation. While the above architecture allows efficient parallel training of *Renée* models, the all-gather/all-reduce communication is synchronous and can increase multi-GPU training overhead depending on the network connectivity. We now describe a novel optimization that *Renée* uses to overlap some of this communication with compute, thereby reducing training overhead. *The key observation behind the optimization is that the backward pass of the linear layer in the classifier entails the computation of two large matrix multiplications, one for computing the gradient that needs to be passed to the encoder and another for computing the gradient for updating the classifier weights*. Thus, in *Renée* we first compute the gradient for the encoder and overlap the communication of these gradients with the second matrix multiplication for computing the gradient for the classifier weights.

Finally, one can also overlap the forward pass all-gather with compute by using gradient accumulation and splitting a mini-batch into multiple micro-batches. This can also reduce the memory requirements of the intermediate state for both the encoder and classifier (by reducing the batch size). However, GPU compute performance typically increases with increased batch size (until it saturates). Thus, care must be taken to have a large enough micro-batch size so that sufficient parallelism is available for each matrix multiplication in the encoder. For typical encoder models, we found that reducing the batch size can significantly impact performance; hence, this optimization does not help.

Split gradient accumulation. Instead, *Renée uses an alternative optimization that we term split gradient accumulation that saves GPU memory without hurting performance: use gradient accumulation only for training the classifier*. This optimization is especially useful when training on multiple GPUs since the classifier gets $G * batch\_size$ embeddings from all the encoders, a much larger batch size than necessary from a GPU compute performance perspective. Thus, with this optimization, each encoder produces the embedding for their respective mini-batch of examples. However, then the combined embeddings of all encoders are divided into micro-batches (a configurable parameter), and the forward and backward passes with gradient accumulation are performed for each micro-batch for the classifier. Finally, the mini-batch backward pass is performed for the encoders. Thus, the classifier memory requirement is reduced from $2 * num\_classes * G * mini\_batch\_size$ to $2 * num\_classes * micro\_batch\_size$, enabling another order of magnitude reduction in GPU memory usage.

4.4 Inference

While we have focused so far on training performance, inference performance is even more critical from an application deployment perspective.

Approximate nearest neighbor search. Thus, for inference, we eschew the fully-connected classifier and instead rely on approximate nearest neighbors search algorithms such as DiskANN (Jayaram Subramanya et al., 2019). DiskANN performs *searches in logarithmic time in the number of labels and can search over a billion vectors on a single CPU in a few milliseconds*.

We use *Renée*'s classifier weights (L2 normalized to the unit norm) as vectors in the DiskANN database. When a user query arrives, a lightweight encoder such as DistilBERT is used to get an embedding vector. We then perform a nearest neighbor search of this vector against the classifier weights and return the top-k best matches. For the XC datasets that we have evaluated, DiskANN is efficient and able to

achieve 99+% of the exact search retrieval accuracy using the fully-connected layer.

4.5 Complex Classifiers

This work considers the classifier to be a single fully-connected layer like prior state-of-the-art models (Dahiya et al., 2021a; 2022), since we find that even a single FC layer in *Renée* gives state-of-the-art accuracy when trained end-to-end. However, one may use more complex classifiers like a multi-layer network and still observe *Renée*'s speedups since the last layer will fan out to $num\_classes$ dimensions, and therefore, its compute and memory costs will far outweigh the costs of the initial layers.

5 Implementation

Renée implementation uses the PyTorch framework (Paszke et al., 2019) and spans about 1500 lines of Python. As described in Section 4, a key optimization in *Renée* is skipping the loss computation and explicitly computing the input gradients for the classifier. This is implemented using a custom fused kernel based on NVIDIA's CUTLASS (NVIDIA, b) library such that the output of the forward pass matrix multiplication is the input gradient for the backward pass and consists of about a few hundred lines of C++.

CUTLASS consists of various CUDA C++ template abstractions for implementing high-performance matrix multiplication (GEMM), enabling this fused kernel implementation. Specifically, we leverage CUTLASS's epilogue model, where an epilogue function can be added to the output values of any GEMM kernel, thereby avoiding unnecessary reads/writes to GPU memory. Thus, we add a sigmoid epilogue function to the `volta_tensorop_gemm` kernel to create a fused sigmoid GEMM kernel. We then bind a python function to this kernel using pybind so that Python code in PyTorch can use this optimized kernel.

Once the classifier input gradients are available, we implement a custom backward pass for the classifier in PyTorch. This consists of a matrix multiplication for computing the input gradients to the encoder followed by an asynchronous all-reduce operation[1] to collect the respective gradients from all the classifiers to all the encoders. In parallel with this all-reduce operation, we schedule the second matrix multiplication of the classifier backward-pass to compute the gradients for the classifier weights, thereby, overlapping compute with communication.

Once the encoder input gradients are available, we switch to PyTorch's automatic differentiation to perform the remainder of the backward pass and compute gradients for the encoder. This is feasible since PyTorch supports calling backward on any tensor with the appropriate input gradients as parameters. Thus, instead of the standard loss.backward(), we call embed.backward(input_gradients), where embed is the encoder embedding tensor.

Finally, we found that FP16 training with *Renée* using the standard practice of scaling the output loss resulted in NANs. Instead, when we apply loss scaling only to the input gradients of the encoder and train the classifier without loss scaling, i.e. split loss scaling, we were able to train *Renée* using FP16, thereby further improving performance by using the tensor cores available in modern GPUs.

[1]Reduce-scatter suffices but PyTorch/NCCL does not have good support for reduce-scatter at this time.

6 Evaluation

We start by comparing the accuracy of *Renée* against prior approaches and then evaluate performance.

6.1 Accuracy

Datasets. Multiple benchmark short-text and long-text datasets with and without label features are considered in this paper. These datasets cover a variety of applications, including product-to-product recommendation (AmazonTitles-670K, Amazon-670K, AmazonTitles-3M, Amazon-3M, LF-Amazon-131K, LF-AmazonTitles-131K, and LF-AmazonTitles-1.3M), predicting related Wikipedia pages (LF-WikiSeeAlso-320K) and predicting Wikipedia categories (Wiki-500K, LF-Wikipedia-500K). Wiki-500K dataset can be obtained from AttentionXML's (You et al., 2019) official GitHub repository. All others can be downloaded from the Extreme Classification Repository (Bhatia et al., 2016). Results are also reported on a proprietary dataset with up to 120 million labels for matching user queries to advertiser bid phrases (Q2BP-120M), created by mining search engine click logs, where a query was treated as a data point and clicked advertiser bid phrases became its labels. Dataset statistics are given in the supplementary.

Baselines. For datasets without label features, state-of-the-art transformer-based methods such as AttentionXML (You et al., 2019), XR-Transformer (Zhang et al., 2021a), X-Transformer (Chang et al., 2020), and LightXML (Jiang et al., 2021) are the primary baselines for *Renée*. For datasets with label features, we compare with state-of-the-art Siamese methods such as SiameseXML (Dahiya et al., 2021a) and NGAME (Dahiya et al., 2022), as well as transformer-based baselines. Results for other methods are reported in the supplementary material.

Prior work includes a combination of results from single models and ensembles. Ensembling is a well-known technique of using multiple models to boost accuracy (Dietterich, 2000) but is orthogonal to XC. Thus, all *Renée* results are reported based on a single model. However, for completeness,

we report prior results based on ensembles.

Hyperparameters. *Renée*'s hyper-parameters are: (i) batch size, (ii) dropout, (iii) linear layer weight decay, (iv) encoder learning rate and (v) classifier learning rate. As mentioned in Section 4.2, the Adam and SGD optimizers were used for the encoder and the classifier respectively. The baseline algorithms' hyper-parameters were set as their authors suggested wherever applicable and by fine-grained validation otherwise. Please refer to the supplementary for *Renée*'s hyper-parameter settings.

Evaluation Metrics. Algorithms were evaluated using Precision@k (P@k, k $\in \{1, 3, 5\}$) since this is the most widely used metric for evaluation in the XC literature (Prabhu et al., 2018; You et al., 2019; Zhang et al., 2021a). Results on other metrics, such as nDCG@k (N@k) and the Propensity-scored variants (PSP@k and PSN@k) are in the supplementary material. Definitions of all these metrics are available at (Bhatia et al., 2016).

6.1.1 Datasets without label features

Table 1 compares *Renée* against baselines on five widely-used XC datasets without label features: AmazonTitles-670K, Amazon-670K, AmazonTitles-3M, Amazon-3M and Wiki-500K. We see that ***Renée* achieves P@1 of up to 5% higher** than leading transformer-based models and *obtains state-of-the-art results on all five datasets*, even when compared to prior work that use ensembles of up to 9 models.

6.1.2 Datasets with label features

We now evaluate datasets where label features are available in the form of label text. Five widely-used XC benchmark datasets are considered: LF-AmazonTitles-131K, LF-Amazon-131K, LF-AmazonTitles-1.3M, LF-WikiSeeAlsoTitles-320K and Wikipedia-500K.

Augmentation with Label Text: In order to inform *Renée* training of label features, we augment the training data with label texts as training documents with the corresponding label id as a positive label.

Initialization with pre-trained encoder: For the short-text datasets (LF-AmazonTitles-131K, LF-AmazonTitles-1.3M), we initialize the weights of *Renée* encoder using NGAME's Module 1 encoder, which is trained on the dataset with a contrastive loss. This may be viewed as an XC-specific pre-training step. This initialization is followed by standard *Renée* end-to-end training.

Results from Table 2 show that ***Renée* achieves P@1 of up to 2% higher** than leading Siamese methods and achieves *state-of-the-art results in 4 out of 5 datasets*. An ablation of *Renée* results with and without label text augmentation and pre-trained initialization is included in the supplementary.

6.1.3 Proprietary dataset with 120M labels

Table 3 shows results on a proprietary dataset, Q2BP-120M, with 120 million labels and 370 million training points for matching user queries to advertiser bid phrases. ***Renée* is over 16% better** than leading XC methods like NGAME and SiameseXML in P@3 and P@5 metrics on Q2BP-120M. Instead of doing *Renée* inference over the full label space in linear complexity, we can perform inference using an approximate nearest neighbor search algorithm like DiskANN in logarithmic complexity. This results in an accuracy loss of less than 0.5% while limiting end-to-end inference latency for a user query to under 20 ms on a DGX-2 machine.

6.1.4 Impact of encoder size

We do an ablation to study the effect of encoder size on *Renée* accuracy, i.e., $P@k$ metric. *Renée* results in Table 2 are based on Distil-RoBERTa, a 6-layer encoder. We train *Renée* with varying encoder sizes on the LF-WikiSeeAlso-320K dataset for this ablation. Encoders used include a 3 and 6 layered MiniLM (Wang et al., 2020), 6-layer Distil-RoBERTa (Sanh et al., 2019) and 12-layer base and 24-layer large variants of RoBERTa (Liu et al., 2019). In Table 4, we see a trend that as we increase the encoder size, we see an improvement in the overall P@1 and P@5, as expected. With a large RoBERTa encoder, ***Renée* P@1 improves to 49.8, over 4% higher than previous state-of-the-art, NGAME**. However, we see that the increase in accuracy is not linear with encoder parameter size, and it starts saturating. This shows that we do not need a powerful encoder for obtaining the accuracy gains from *Renée*. A small encoder followed by the classifiers can provide a good trade-off between performance and accuracy.

6.1.5 Impact of hybrid optimizer

We do an ablation to study the effect of using the Adam optimizer instead of SGD for classifier training on *Renée* accuracy, i.e., $P@k$ metric. We consider 2 datasets, LF-AmazonTitles-131K and LF-WikiSeeAlso-320K for this ablation. In Table 5, we notice that using SGD for training the classifier is better than using Adam by 1 point. Therefore, it seems that using the split optimizer setting to train *Renée* helps with both accuracy and savings in memory and compute.

6.2 Performance: Training Time

Tables 6 & 7 show *Renée*'s total training time and prior methods for various datasets. For datasets without label features, *Renée* and XR-Transformer have the lowest total training time in Wiki-500K and Amazon-3M datasets, respectively, while AttentionXML and LightXML are an order of magnitude slower. While SiameseXML has the lowest training time for datasets with label features, its accu-

Table 1. Results on Non-Label features datasets comparing *Renée* to baselines. To obtain results on AmazonTitles-670K/3M for LightXML and XR-Transformer, official code of (Zhang et al., 2021b; Jiang et al., 2021) was used, other dataset results are from (Zhang et al., 2021b). Other prior results have been taken from (Bhatia et al., 2016). "-" means the result is not available. We include ensemble numbers reported in the literature for reference; the number denotes no. of models used for ensembles, e.g., XR-Transformer-3 uses 3 model ensemble. ***Renée* outperforms every method, including ensemble, using just a single model and achieves state-of-the-art results in all datasets.**

Methods	AmazonTitles-670K			Amazon-670K			AmazonTitles-3M			Amazon-3M			Wiki-500K		
	P@1	P@3	P@5	P@1	P@3	P@5	P@1	P@3	P@5	P@1	P@3	P@5	P@1	P@3	P@5
AttentionXML	37.92	33.73	30.57	47.14	42.7	38.99	46	42.81	40.59	50.86	48.04	45.83	76.95	58.42	46.14
LightXML	41.7	37.3	34.2	47.3	42.2	38.5	–Not-Scalable–			–Not-Scalable–			76.19	57.22	44.12
XR-Transformer	41.07	36.66	33.55	49.11	43.8	40	48.72	45.74	43.35	52.6	49.4	46.9	78.1	57.6	45
***Renée* (ours)**	**45.2**	**40.24**	**36.61**	**54.23**	**48.22**	**43.83**	**51.81**	**48.84**	**46.54**	**54.84**	**52.08**	**49.77**	**79.47**	**60.37**	**46.84**
Ensemble Models (Shown for Completeness)															
X-Transformer-9	-	-	-	48.07	42.96	39.12	-	-	-	51.2	47.81	45.07	77.09	57.51	45.28
LightXML-3	43.1	38.7	35.5	49.1	44.17	40.25	–Not-Scalable–			–Not-Scalable–			77.78	58.85	45.57
XR-Transformer-3	41.94	37.44	34.19	50.11	44.56	40.64	50.5	47.41	45	54.2	50.81	48.26	79.4	59.02	46.25

Table 2. Results on Label features datasets comparing *Renée* to baselines. Results for NGAME and other methods have been taken from (Bhatia et al., 2016). NGAME denotes classifier (M2 Module) performance in (Dahiya et al., 2022). We also report the results of the NGAME ensemble model as NGAME-2 (Fusion of NGAME M1 and M2). ***Renée* outperforms most prior methods, including ensembles, using just a single model and achieves state-of-the-art results in 14 out of 15 columns.**

Methods	LF-AmazonTitles-131K			LF-Wikipedia-500K			LF-AmazonTitles-1.3M			LF-WikiSeeAlso-320K			LF-Amazon-131K		
	P@1	P@3	P@5	P@1	P@3	P@5	P@1	P@3	P@5	P@1	P@3	P@5	P@1	P@3	P@5
XR-Transformer	38.1	25.57	18.32	81.62	61.38	47.85	50.14	44.07	39.98	42.57	28.24	21.3	45.61	30.85	22.32
LightXML	35.6	24.15	17.45	81.59	61.78	47.64	–Not-Scalable–			34.5	22.31	16.83	41.49	28.32	20.75
SiameseXML	41.42	27.92	21.21	67.26	44.82	33.73	49.02	42.72	38.52	42.16	28.14	21.39	44.81	30.19	21.94
NGAME	44.69	29.89	21.21	84.78	65.72	50.92	54.99	48.09	43.11	45.72	29.61	22.06	46.63	30.94	22.03
***Renée* (ours)**	**46.05**	**30.81**	**22.04**	**84.95**	**66.25**	**51.68**	56.04	**49.91**	**45.32**	**47.77**	**31.90**	**23.82**	**48.05**	**32.33**	**23.26**
Ensemble Models (Shown for Completeness)															
NGAME-2	46.01	30.28	21.47	84.01	64.69	49.97	**56.75**	49.19	44.09	47.65	31.56	23.68	46.53	30.89	22.02

Table 3. Results on the Q2BP-120M proprietary dataset having 120M labels comparing *Renée* with recent XC methods. Dataset matches user queries to advertiser bid phrases.

Methods	P@1	P@3	P@5
SiameseXML	83.46	36.90	23.64
NGAME	**87.82**	38.39	24.35
***Renée* (DiskANN)**	83.30	55.01	41.06
***Renée* (Exact Search)**	83.78	**55.30**	**41.27**

racy is 5-17% worse than *Renée*. Compared to the previous state-of-the-art approach, NGAME, we see that *Renée* is $4.2 - 5.6\times$ faster.

6.3 Performance: Optimizations

In this section, we compare the performance of *Renée* with standard end-to-end training using a hybrid data-model parallel implementation in PyTorch to evaluate the various optimizations in *Renée*. We use the latest version of PyTorch (2.0) with CUDA version 11.7 for this evaluation. We make use of the compile feature for the encoder layer. For the standard implementation, we use the standard approach of defining the model and the loss function, which in our case is binary cross entropy with logits, performing the forward pass and using Pytorch's automatic differentiation for the backward pass. For the hybrid data-parallel architecture, we use an all-gather to send the embeddings from the encoders to the model-parallel classifiers, and in the backward pass, we use an all-reduce to distribute the gradients from the classifier to the encoders. We use *Renée*'s split-optimizer optimization for all experiments, including baseline, for a fair comparison. Specifically, we use Apex's (NVIDIA, a) Fused SGD optimizer for the classifier and Fused Adam optimizer for the encoder. We use 32GB V100 GPUs for single-node experiments and a DGX-2 with 16 V100s connected with NVLINK for multi-GPU experiments.

Table 4. Renée trained on LF-WikiSeeAlso-320K using different encoders and their respective $P@1$ and $P@5$ performance.

Encoder	Parameters	P@1	P@5
MiniLM-L3	15M	41.5	20.9
MiniLM-L6	20M	45.0	22.6
Distil-RoBERTa	67M	47.8	23.8
RoBERTa-base	125M	48.3	24.2
RoBERTa-large	330M	49.8	24.9

Table 5. Renée $P@1$ and $P@5$ performance when the classifier is trained with different optimizers.

Optimizer	P@1	P@5
LF-AmazonTitles-131K		
SGD	46.05	22.04
Adam	45.31	21.44
LF-WikiSeeAlso-320K		
SGD	47.77	23.82
Adam	46.29	22.63

Table 6. Similar to (Zhang et al., 2021b), we compare total training time (TT) in hours to get to state-of-the-art precision for different XC methods on 2 Non-Label features datasets using 8 Nvidia V100 GPUs. Baseline results are from (Zhang et al., 2021b). LightXML is not scalable for Amazon-3M.

Amazon-3M		Wiki-500K	
Method	**TT**	**Method**	**TT**
AttentionXML	54.8	AttentionXML	37.6
X-Transformer-9	542	X-Transformer-9	557
LightXML-3	-	LightXML-3	271
XR-Transformer-3	29.3	XR-Transformer-3	38
Renée	43.4	*Renée*	8.6

Since *Renée*'s optimizations are targeted at optimizing the compute and memory costs of the classifier, the performance gains of *Renée* would depend on the relative costs of the classifier versus the encoder. Thus, we evaluate a range of encoders from small to large to illustrate the range of performance gains achievable. Encoders considered are MiniLM (Wang et al., 2020) with 3 and 6 layers (15M and 20M parameters, respectively), DistilRoBERTa (Sanh et al., 2019) with 6 layers (67M parameters) and RoBERTa-large (Liu et al., 2019) with 24 layers (330M parameters).

6.3.1 *Single GPU performance*

Table 8 depicts the end-to-end training time for one epoch on various datasets for the baseline implementation that uses standard PyTorch optimizations and *Renée* performance with different approaches to computing loss as detailed in Section 4. Consider the MiniLM-L6 encoder on AmazonTitles-3M dataset. The baseline approach can only fit a batch size of 128 before it runs out of memory and achieves an epoch training time of 36:27 (minutes:seconds). We can increase the batch size used by the baseline with gradient accumulation, but even increasing the batch size to 1024 and using Pytorch 2.0's compile feature, the epoch time reduces to only 33:43. This is because the standard gradient accumulation does not increase the parallelism of the encoder and merely saves on the number of parameters updates. Using the Sparse loss computation and other optimizations, *Renée* can support a maximum batch size of 512 without gradient accumulation. The increased batch size and optimizations reduce the epoch time to 11:10. By using sampled sparse loss computation (sample of 1 in 100), the epoch time is further reduced to 6:43. Finally, by skipping loss computation altogether, *Renée* can accommodate a maximum batch size of 1024. Along with Pytorch compile and using the custom cuda kernel, the epoch time reduces to 4:54 or *almost* $7\times$ *reduction compared to baseline*.

Table 7. Comparing total training time (TT) in hours on a single Nvidia V100 GPU for various XC methods. We use 2 Label features datasets for this comparison. Results for methods other than *Renée* are taken from (Bhatia et al., 2016).

LF-WikiSeeAlso-320K		LF-Amazon-131K	
Method	**TT**	**Method**	**TT**
XR-Transformer	119.47	XR-Transformer	38.4
AttentionXML	90.37	AttentionXML	50.17
LightXML	249	LightXML	56.03
SiameseXML	2.33	SiameseXML	1.18
NGAME	75.39	NGAME	39.99
Renée	17.8	*Renée*	7.15

Table 8. Performance on 1 V100 GPU with 32GB memory

Approach	Batch Size	Epoch Time (mm:ss)
AmazonTitles-3M, MiniLM-L6, Seqlen 32		
Baseline	128	36:27
Baseline (grad accum)	1024	34:10
Baseline (grad accum+compile)	1024	33:43
Sparse loss	512	11:10
Sparse loss(sample 1%)	512	6:43
Skip loss	1024	5:25
Skip loss (compile)	1024	4:56
Skip loss (compile+custom-cuda)	1024	4:54
Amazon-670K, Distil-RoBERTa, Seqlen 256		
Baseline (grad accum+compile)	256	48:23
Skip loss	256	13:20
AmazonTitles-3M, MiniLM-L3, Seqlen 16		
Baseline (grad accum+compile)	8192	17:00
Skip loss	8192	1:10

Next, consider the case of a bigger encoder, such as Distil-Roberta, on the Amazon-670K dataset. We also use a larger sequence length of 256 in this experiment, which further increases the compute requirements of the encoder. In this case, baseline takes about 48:23 for one epoch while *Renée* without loss can reduce it to 13:20 or *a* $3.6\times$ *reduction compared to baseline*.

Finally, we consider a smaller encoder, MiniLM, with 3 layers and use a smaller sequence length of 16. We further use

Table 9. Performance on 1 A6000 GPU

A6000			V100
Epoch Time (mm:ss)		Speedup	Speedup
Baseline	Skip loss		
AmazonTitles-3M, MiniLM-L6, Seqlen 32			
40:00	5:22	7.44	6.88
Amazon-670K, Distil-RoBERTa, Seqlen 256			
39:25	14:48	2.66	3.60
AmazonTitles-3M, MiniLM-L3, Seqlen 16			
24:00	1:43	13.87	14.57

Table 10. Performance on DGX-2 (16 V100 GPUs). Encoder is RoBERTa-Large.

Approach	Batch Size	Epoch Time (mm:ss)
Amazon-3M, Seqlen 256		
Baseline	32	74:23
Skip loss	48	17:11
AmazonTitles-3M, Seqlen 32		
Baseline	512	10:34
Skip loss	512	1:58

a bottleneck layer of 64 to accommodate larger batch sizes. In this case, the baseline's epoch time is 17:00 while *Renée* without loss can reduce it to 1:10 or *over* $14\times$ *reduction compared to baseline*.

We also compare *Renée* speedups on A6000, a newer version of GPU released in 2020. As seen in Table 9, the speedups obtained on an A6000 are comparable to the speedups observed on a V100.

6.3.2 *Multi GPU performance*

We now consider a large encoder, RoBERTa-large, with 330M parameters. Further, it produces a 1024-dimension embedding, which implies that the classifier parameters would be 3 Billion for a dataset with 3M labels. Without a bottleneck layer, such a model does not fit in a 32GB V100. Thus, we evaluate large encoders on a multi-GPU setting, namely, a 16 V100 DGX2 machine.

Table 10 depicts the end-to-end training time for one epoch on two datasets for the baseline hybrid data-model parallel PyTorch implementation and *Renée* with all its optimizations, including loss skipping. For all approaches, the maximum batch size per GPU is shown (and for the baseline, gradient accumulation is used to match the batch size of *Renée*). On the Amazon-3M dataset, baseline takes 74:23 for one epoch while *Renée* completes an epoch in 17:11 for $4.3\times$ *reduction*. For the AmazonTitles-3M dataset, we use a smaller sequence length of 32 since the data points are titles, which are shorter in length. This reduces the encoder computation time compared to Amazon-3M, which used a sequence length of 256. Thus, *Renée* provides a greater benefit achieving a per epoch time of 1:58 compared to the baseline of 10:34, resulting in *a* $5.3\times$ *reduction*.

Table 11. 120 Million label Dataset Performance on DGX-2

Approach	Batch Size	Epoch Time (hours)
Baseline	4	214
Baseline (grad accum)	512	204
Skip loss without fusion	512	15
Skip loss with fusion	512	13.5

Finally, Table 11 depicts the end-to-end training time for one epoch on the proprietary 120 Million label dataset. The encoder is a small three-layer transformer similar to MiniLM. We employ a bottleneck layer that reduces the classifier dimension to 64. With 120 Million classes, the *Renée* model has about 8 Billion parameters. We train this model using the baseline approach and *Renée* on a DGX-2 node. For the baseline, we can only fit a maximum per-GPU batch size of 4, resulting in 214 hours of training time per epoch (this reduces to 204 with a batch size of 512 using 128-way end-to-end gradient accumulation). For *Renée* we can fit a per-GPU batch size of 512 (with 8-way split gradient accumulation), and the training time reduces to 15 hours/epoch. Finally, using kernel fusion while skipping loss computation produces a further 10% reduction for this large model. Thus, *Renée* can reduce training time by over $15\times$.

In summary, we see that *Renée*'s optimizations deliver significant performance savings, ranging from $3-15\times$ reduction in training time depending on the compute cost of the encoder and the number of labels in the classifier. The highest gains are achieved when the encoder is small and/or the number of labels is large.

7 Conclusion

In this paper, contrary to conventional wisdom, we show that end-to-end training at extreme scale is practical. Further, we show that our end-to-end trained model, *Renée*, is able to achieve state-of-the-art results on a number of publicly available data sets. As part of future work, we are investigating further improvements to the performance and accuracy of end-to-end trained models using novel data augmentation and loss functions.

8 Acknowledgements

We thank our anonymous reviewers & Shepherd, Dr. Hao Zhang, for their helpful comments in improving the paper.

REFERENCES

Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 472–487, 2022.

Babbar, R. and Schölkopf, B. Dismec: Distributed sparse machines for extreme multi-label classification. In *Proceedings of the tenth ACM international conference on web search and data mining*, pp. 721–729, 2017.

Babbar, R. and Schölkopf, B. Data scarcity, robustness and extreme multi-label classification. *Machine Learning*, 108(8):1329–1351, 2019.

Bhatia, K., Dahiya, K., Jain, H., Kar, P., Mittal, A., Prabhu, Y., and Varma, M. The extreme classification repository: Multi-label datasets and code, 2016. URL http://manikvarma.org/downloads/XC/XMLRepository.html.

Chang, W.-C., Yu, H.-F., Zhong, K., Yang, Y., and Dhillon, I. S. Taming pretrained transformers for extreme multi-label text classification. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 3163–3171, 2020.

Chen, B., Medini, T., Farwell, J., Tai, C., Shrivastava, A., et al. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Proceedings of Machine Learning and Systems*, 2:291–306, 2020a.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR, 2020b.

Dahiya, K., Agarwal, A., Saini, D., Gururaj, K., Jiao, J., Singh, A., Agarwal, S., Kar, P., and Varma, M. Siamesexml: Siamese networks meet extreme classifiers with 100m labels. In *International Conference on Machine Learning*, pp. 2330–2340. PMLR, 2021a.

Dahiya, K., Saini, D., Mittal, A., Shaw, A., Dave, K., Soni, A., Jain, H., Agarwal, S., and Varma, M. Deepxml: A deep extreme multi-label learning framework applied to short text documents. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining*, pp. 31–39, 2021b.

Dahiya, K., Gupta, N., Saini, D., Soni, A., Wang, Y., Dave, K., Jiao, J., Dey, P., Singh, A., Hada, D., et al. Ngame: Negative mining-aware mini-batching for extreme classification. *arXiv preprint arXiv:2207.04452*, 2022.

Dietterich, T. G. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pp. 1–15. Springer, 2000.

Guo, C., Mousavi, A., Wu, X., Holtmann-Rice, D. N., Kale, S., Reddi, S., and Kumar, S. Breaking the glass ceiling for embedding-based classifiers for large output spaces. *Advances in Neural Information Processing Systems*, 32, 2019.

Hofstätter, S., Lin, S.-C., Yang, J.-H., Lin, J., and Hanbury, A. Efficiently teaching an effective dense retriever with balanced topic aware sampling. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 113–122, 2021.

Huang, P.-S., He, X., Gao, J., Deng, L., Acero, A., and Heck, L. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pp. 2333–2338, 2013.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jain, H., Balasubramanian, V., Chunduri, B., and Varma, M. Slice: Scalable linear extreme classifiers trained on 100 million labels for related searches. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pp. 528–536, 2019.

Jayaram Subramanya, S., Devvrit, F., Simhadri, H. V., Krishnawamy, R., and Kadekodi, R. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 47–62, New York, NY, USA, 2019a. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019b.

Jiang, T., Wang, D., Sun, L., Yang, H., Zhao, Z., and Zhuang, F. Lightxml: Transformer with dynamic negative sampling for high-performance extreme multi-label text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 7987–7994, 2021.

Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 4163–4174, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.372. URL https://aclanthology.org/2020.findings-emnlp.372.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

Kenton, J. D. M.-W. C. and Toutanova, L. K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

Khandagale, S., Xiao, H., and Babbar, R. Bonsai: diverse and shallow trees for extreme multi-label classification. *Machine Learning*, 109(11):2099–2119, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. iclr. 2015. *arXiv preprint arXiv:1412.6980*, 9, 2015.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

Lee, M.-C., Gao, B., and Zhang, R. Rare query expansion through generative adversarial networks in search advertising. In *Proceedings of the 24th acm sigkdd international conference on knowledge discovery & data mining*, pp. 500–508, 2018.

Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., Gan, L., Yang, G., and Qian, D. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

Liu, J., Chang, W.-C., Wu, Y., and Yang, Y. Deep learning for extreme multi-label text classification. In *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*, pp. 115–124, 2017.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Medini, T. K. R., Huang, Q., Wang, Y., Mohan, V., and Shrivastava, A. Extreme classification in log memory using count-min sketch: A case study of amazon search with 50m products. *Advances in Neural Information Processing Systems*, 32, 2019.

Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

NVIDIA. APEX library, a. URL https://github.com/nvidia/apex.

NVIDIA. Cutlass library, b. URL https://github.com/NVIDIA/cutlass.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-\high-performance-deep-learning-library.pdf.

Prabhu, Y., Kag, A., Harsola, S., Agrawal, R., and Varma, M. Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising. In *Proceedings of the 2018 World Wide Web Conference*, pp. 993–1002, 2018.

PyTorch. BCEWithLogitsLoss. URL https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html.

Qu, Y., Ding, Y., Liu, J., Liu, K., Ren, R., Zhao, W. X., Dong, D., Wu, H., and Wang, H. Rocketqa: An optimized training approach to dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2010.08191*, 2020.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL https://arxiv.org/abs/1908.10084.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training, 2021.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

TensorFlow. Sigmoid Cross Entropy With Logits. URL https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits.

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., and Zhou, M. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers, 2020.

Xiong, L., Xiong, C., Li, Y., Tang, K.-F., Liu, J., Bennett, P., Ahmed, J., and Overwijk, A. Approximate nearest neighbor negative contrastive learning for dense text retrieval. In *International Conference on Learning Representations, ICLR*, 2021.

Yen, I. E., Huang, X., Dai, W., Ravikumar, P., Dhillon, I., and Xing, E. Ppdsparse: A parallel primal-dual sparse method for extreme classification. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 545–553, 2017.

You, R., Zhang, Z., Wang, Z., Dai, S., Mamitsuka, H., and Zhu, S. Attentionxml: Label tree-based attention-aware deep model for high-performance extreme multi-label text classification. *Advances in Neural Information Processing Systems*, 32, 2019.

Yu, H.-F., Zhong, K., Zhang, J., Chang, W.-C., and Dhillon, I. S. Pecos: Prediction for enormous and correlated output spaces. *Journal of Machine Learning Research*, 23(98): 1–32, 2022.

Zhang, J., Chang, W.-C., Yu, H.-F., and Dhillon, I. Fast multi-resolution transformer fine-tuning for extreme multi-label text classification. *Advances in Neural Information Processing Systems*, 34:7267–7280, 2021a.

Zhang, J., Chang, W.-C., Yu, H.-F., and Dhillon, I. Fast multi-resolution transformer fine-tuning for extreme multi-label text classification. *Advances in Neural Information Processing Systems*, 34:7267–7280, 2021b.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 863–879, 2020.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Gonzalez, J. E., et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2022.

A Dataset Statistics

Multiple benchmark short-text and long-text datasets with and without label features are considered in this paper. These datasets cover a variety of applications including product-to-product recommendation (AmazonTitles-670K, Amazon-670K, AmazonTitles-3M, Amazon-3M, LF-Amazon-131K, LF-AmazonTitles-131K, and LF-AmazonTitles-1.3M), predicting related Wikipedia pages (LF-WikiSeeAlso-320K) and predicting Wikipedia categories (Wiki-500K, LF-Wikipedia-500K). Wiki-500K dataset can be obtained from AttentionXML's (You et al., 2019) official GitHub repository. All others can be downloaded from the Extreme Classification Repository (Bhatia et al., 2016). Results are also reported on a proprietary dataset with up to 120 million labels for matching user queries to advertiser bid phrases (Q2BP-120M). These were created by mining search engine click logs, where a query was treated as a data point and clicked advertiser bid phrases became its labels. Dataset statistics are given in Table 13.

B Full Results

Tables 16 and 17 present detailed results on all baseline methods for non-LF and LF datasets respectively.

C Ablations

NGAME M1 init. and label text augmentation: We study the effect of **I.** NGAME (Dahiya et al., 2022) M1 initialization of the *Renée* encoder, and, **II.** Augmenting the training data with label text. Table 12 demonstrates that initialization with M1 always helps versus not doing as long as label text augmentation is not used. This is possibly due to XC specific pre-training of the M1 encoder interacting with label text augmentation. However, when using label text augmentation, we do not always require M1 initialization to achieve state-of-the-art results (in case of LF-Amazon-131K, LF-Wikipedia-500K). Specifically, we find that M1 init. is required for best results only for the *Titles* datasets where the encoder has to rely on a short input text.

D Hyper-parameters

The list below enumerates the various hyperparameters with a brief description and the values chosen.

1. **Encoder:** For each dataset, we choose pre-trained Roberta or Bert-based encoder variants from the sentence-transformers library (Reimers & Gurevych, 2019). A larger encoder leads to a better accuracy at the cost of increased training time. As the Encoder ablation indicates, the accuracy gains saturate while the training costs do not.
2. **Batch Size:** The number of datapoints chosen in a single mini-batch. The value for batch-size is heuristically chosen to reduce the convergence time and maximise accuracy.
3. **LR Encoder:** The learning rate used to update the learnable parameters of the encoder module. Generally, we try to keep it as high as possible for faster convergence while ensuring that the loss is not diverging.
4. **LR Classifier:** The learning rate used to update the learnable parameters of the linear layer. Various learning rates are tried in the range of 2e-3 and 2e-1, and the one that leads to the maximum accuracy is chosen.
5. **Weight Decay Classifier:** The value of weight decay used to regularize the parameters of the linear layer. Various values are tried in the range of 1e-2 and 1e-3, and the one that leads to the maximum accuracy is chosen.
6. **Weight Decay Encoder:** The value of weight decay used to regularize the parameters of the transformer-based encoder. Similar to the Classifier Weight Decay, various values are tried in the range of 1e-2 and 1e-3, and the one that leads to the maximum accuracy is chosen.
7. **Dropout:** The probability of randomly dropping the encoder outputs in order to regularise the network. High values of dropouts are required since the architectures being used are quite large compared to the amount of the data available.
8. **Warmup:** Warmup steps is the number of training iterations over which both the encoder and the classifier learning rates are linearly increased from 0 to the maximum value. Once the maximum value is achieved, it is then linearly reduced to 0 over the remaining training steps. This is done in order to reduce the impact of deviating the pre-trained model from learning on sudden new data set exposure. Generally, the warmup phase consists of around 5% of total training iterations.
9. **Epochs:** The number of training passes done over the all the datapoints present in the dataset. A large enough value is chosen to ensure convergence.
10. **Sequence Length:** The number of positional embeddings trained for the encoder module. All datapoint texts longer than the sequence length are truncated. Care is taken to ensure minimal data loss occurs due to truncation.

The various hyperparameter values chosen for each dataset are listed in Tables 14 and 15 for non-LF and LF datasets respectively. All other hyperparameters are constant across datasets and can be found in the *Renée* codebase.

E Artifact Appendix

E.1 Abstract

The artifact consists of a codebase to train *Renée* models for various extreme classification (XC) datasets. The codebase reproduces state-of-the-art numbers reported in the main publication. We include pre-processing scripts and training scripts for each dataset. *Renée* is built on top of Hugging Face's Transformers library and PyTorch. We hope that *Renée* will be useful for researchers working on XC classification tasks.

E.2 Artifact check-list (meta-information)

- **Algorithm:** We make end-to-end training feasible for Extreme Classification tasks via our end-to-end model, *Renée*.
- **Model:** We use HuggingFace models, which are automatically downloaded in the training scripts.
- **Data set:** We consider publicly available datasets which consider various XC problems. They are a few GBs in size. They can be downloaded from Extreme Classification Repository.
- **Run-time environment:** Conda-based virtual environment using Python 3.8.
- **Hardware:** We use DGX-2 node with 16 V100 GPUs for our experiments but any commodity GPU cluster can be used.
- **Execution:** Execution time depends on the dataset, model configuration and the hardware used.
- **Metrics:** Precision@k
- **Output:** TSV files with metrics on datasets.
- **Experiments:** Use provided scripts in the artifact to build the environment and run experiments. There can be a variation of upto 0.2% from reported results.
- **How much disk space required (approximately)?:** The artifact itself requires a few KBs of space. Some initialization checkpoints are provided, of size 500MB. Experimentation requires upto 50GB of space.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 1 week on specified hardware
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Provided here

E.3 Description

E.3.1 How delivered

The software can be obtained from Github: https://github.com/microsoft/renee.

E.3.2 Hardware dependencies

To train Renee, we require a Linux system with at least one GPU. For larger datasets, up to 16 GPUs with 32GB memory each may be required to fit in the model.

E.3.3 Software dependencies

A Python 3.8 and conda-based virtual environment installed with Pytorch and Hugging Face's Transformers library is required to train Renee.

E.3.4 Data sets

The datasets can be downloaded from Extreme Classification Repository.

E.4 Installation

Instructions to set up a conda virutal environment with the required packages is provided in the GitHub repository.

E.5 Experiment workflow

To run experiments, users must

1. Set up the virtual environment with the required packages using the install scripts. Refer to the Requirements Section.
2. Prepare the dataset by creating tokenisations for the transfomer-based encoder. The scripts required for tokenisation and optional label-text augmentation are provided in Data Preparation.
3. Run the training command corresponding to the dataset given in the scripts folder. Refer to Training.

E.6 Evaluation and expected result

The training scripts also contain code to evaluate the model at intermediate stages for the relevant metrics and output the results to the log. The expected result will match the numbers reported in Tables 1 & 2 of the manuscript. There may be a small difference of upto 0.2% in a some metrics due to the use of stochastic libraries.

E.7 Experiment customization

Renee can be trained for any Extreme Classification dataset. The hyperparamters can be tuned based on the hints given in the training script.

Table 12. Ablation study on the effect of NGAME M1 initialization and augmenting training data with label text on *Renée* accuracy. "M1 init." stands for M1 initialisation, "LT Aug." means label text augmentation.

Dataset	M1 init.	LT Aug.	Metrics	
			P@1	P@5
LF-AmazonTitles-131K	-	-	38.89	18.72
	Yes	-	45.70	21.58
	-	*Yes*	41.77	20.02
	Yes	*Yes*	46.05	22.04
LF-Amazon-131K	-	-	45.89	22.65
	Yes	-	47.35	23.00
	-	*Yes*	47.82	23.13
	Yes	*Yes*	48.05	23.26
LF-Wikipedia-500K	-	-	83.08	49.82
	Yes	-	83.27	50.06
	-	*Yes*	84.95	51.68
	Yes	*Yes*	84.5	51.31

Table 13. Dataset statistics. All the public datasets can be downloaded from (Bhatia et al., 2016), except for Wiki-500K, which can be obtained from the official implementation of (You et al., 2019). Datapoints typically have $\mathcal{O}(\log L)$ positive labels that are relevant to them. Other statistics of the Q2BP-120M proprietary dataset have been redacted using the ϕ symbol.

Dataset	Number of Labels L	Number of Train Points N	Number of Test Points N'	Avg. train points per label	Avg. labels per train point
Dataset with Label Features					
LF-AmazonTitles-131K	131,073	294,805	134,835	5.15	2.29
LF-Amazon-131K	131,073	294,805	134,835	5.15	2.29
LF-WikiSeeAlso-320K	312,330	693,082	177,515	4.67	2.11
LF-Wikipedia-500K	501,070	1,813,391	783,743	24.75	4.77
LF-AmazonTitles-1.3M	1,305,265	2,248,619	970,237	38.24	22.2
Dataset without Label Features					
AmazonTitles-670K	670,091	485,176	150,875	5.11	5.39
Amazon-670K	670,091	490,449	153,025	3.99	5.45
AmazonTitles-3M	2,812,281	1,712,536	739,665	31.55	36.18
Amazon-3M	2,812,281	1,717,899	742,507	31.64	36.17
Wiki-500K	501,070	1,779,881	769,421	16.86	4.75
Proprietary Dataset					
Q2BP-120M	120,293,341	370,080,440	92,532,582	ϕ	ϕ

E.8 Notes

The Github README of the project contains further information for training Renee for various XC datasets.

Table 14. Hyperparameters of *Renée* on non-label features (non-LF) datasets to facilitate reproducibility. Other parameters not mentioned below use the default values from the released code. LR stands for learning rate.

Dataset	Encoder	Batch Size	LR Encoder	LR Classifier	Weight Decay Classifier	Dropout	Warmup	Epochs	Sequence Length
AmazonTitles-670K	RoBERTa-Large	256	0.00005	0.01	0.001	0.8	10000	60	32
Amazon-670K	RoBERTa-Large	256	0.00004	0.01	0.001	0.8	10000	70	512
AmazonTitles-3M	RoBERTa-Large	256	0.00005	0.01	0.001	0.75	10000	60	32
Amazon-3M	RoBERTa-Large	256	0.00004	0.01	0.001	0.75	10000	60	256
Wiki-500K	RoBERTa-Base	2048	0.0001	0.002	0.0001	0.75	5000	100	256

Table 15. Hyperparameters of *Renée* on datasets having label features (LF datasets) to facilitate reproducibility. Other parameters not mentioned below use the default values from the released code. LR stands for learning rate. We augment the training set with label text across all LF datasets to expose *Renée* directly to label features during training (refer to subsection 6.1.2 of the main paper). Please refer to suppl. for more ablations on this.

Dataset	Encoder	Batch Size	LR Encoder	LR Classifier	Weight Decay Classifier	Dropout	Warmup	Epochs	Sequence Length
LF-AmazonTitles-131K	Distil-BERT	512	0.00001	0.05	0.0001	0.85	5000	100	32
LF-Wikipedia-500K	Distil-RoBERTa	2048	0.0001	0.002	0.001	0.7	5000	100	256
LF-AmazonTitles-1.3M	Distil-BERT	1024	0.000001	0.01	0.0001	0.7	15000	100	32
LF-WikiSeeAlso-320K	Distil-RoBERTa	2048	0.0002	0.2	0.0001	0.75	5000	100	128
LF-Amazon-131K	Distil-BERT	512	0.00001	0.05	0.0001	0.85	5000	100	128

Table 16. Results on other metrics, including $P@k$ for Non-Label features(non-LF) datasets. "-" means the result on that dataset and method pair are unavailable. Results for other methods are as reported in (Bhatia et al., 2016). We report ensemble performance for XR-Transformer, LightXML i.e. XR-Transformer-3, LightXML-3 in the table respectively. To get results on propensity-based metrics, we use the official code provided by (Zhang et al., 2021b; Jiang et al., 2021) to train models from scratch. * indicates the method trained by us to get performance on these propensity-based metrics.

Methods	P@1	P@3	P@5	N@1	N@3	N@5	PSP@1	PSP@3	PSP@5	PSN@1	PSN@3	PSN@5
AmazonTitles-670K												
Bonsai	38.46	33.91	30.53	38.46	36.05	34.48	23.62	26.19	28.41	23.62	25.16	26.21
Parabel	38	33.54	30.1	38	35.62	33.98	23.1	25.57	27.61	23.1	24.55	25.48
Astec	40.63	36.22	33.00	40.63	38.45	37.09	28.07	30.17	32.07	28.07	29.20	29.98
AttentionXML	37.92	33.73	30.57	37.92	35.78	34.35	24.24	26.43	28.39	24.24	25.48	26.33
LightXML-3	43.1	38.7	35.5	-	-	-	-	-	-	-	-	-
XR-Transformer-3*	41.94	37.44	34.19	41.89	39.67	38.32	25.34	28.86	32.14	25.34	27.58	29.3
Renée	45.2	40.24	36.61	45.2	42.77	41.27	28.98	32.66	35.83	28.98	31.38	33.07
Amazon-670K												
Bonsai	45.58	40.39	36.6	45.58	42.79	41.05	27.08	30.79	34.11	-	-	-
Parabel	44.89	39.8	36	44.89	42.14	40.36	25.43	29.43	32.85	25.43	28.38	30.71
Astec	47.77	42.79	39.10	47.77	45.28	43.74	32.13	35.14	37.82	32.13	33.80	35.01
AttentionXML	47.58	42.61	38.92	47.58	45.07	43.5	30.29	33.85	37.13	-	-	-
LightXML-3	49.1	43.83	39.85	-	-	-	-	-	-	-	-	-
XR-Transformer-3*	50.13	44.6	40.69	50.13	47.28	45.6	29.9	34.35	38.63	29.9	32.75	35.03
Renée	54.23	48.22	43.83	54.23	51.23	49.41	34.16	39.14	43.39	34.16	37.48	39.83
AmazonTitles-3M												
Bonsai	46.89	44.38	42.3	46.89	45.46	44.35	13.78	16.66	18.75	13.78	15.75	17.1
Parabel	46.42	43.81	41.71	46.42	44.86	43.7	12.94	15.58	17.55	12.94	14.7	15.94
Astec	48.74	45.70	43.31	48.74	46.96	45.67	16.10	18.89	20.94	16.10	18.00	19.33
AttentionXML	46	42.81	40.59	46	43.94	42.61	12.81	15.03	16.71	12.8	14.23	15.25
LightXML-3	-	-	-	-	-	-	-	-	-	-	-	-
XR-Transformer-3*	50.5	47.41	45	50.5	48.79	47.57	15.81	19.03	21.34	15.81	18.14	19.75
Renée	51.81	48.84	46.54	51.81	50.08	48.86	14.49	17.43	19.66	14.49	16.5	17.95
Amazon-3M												
Bonsai	48.45	45.65	43.49	48.45	46.78	45.59	13.79	16.71	18.87	-	-	-
Parabel	47.48	44.65	42.53	47.48	45.73	44.53	12.82	15.61	17.73	12.82	14.89	16.38
AttentionXML	50.86	48.04	45.83	50.86	49.16	47.94	15.52	18.45	20.6	-	-	-
LightXML-3	-	-	-	-	-	-	-	-	-	-	-	-
XR-Transformer-3*	53.67	50.29	47.74	53.67	51.74	50.42	16.54	19.94	22.39	16.54	18.99	20.71
Renée	54.84	52.08	49.77	54.84	53.31	52.13	15.74	19.06	21.54	15.74	18.02	19.64
Wiki-500K												
Bonsai	68.7	49.57	38.64	-	-	-	-	-	-	-	-	-
Parabel	69.26	49.8	38.83	-	-	-	-	-	-	-	-	-
AttentionXML	76.95	58.42	46.14	-	-	-	30.69	38.92	44	-	-	-
LightXML-3	77.78	58.85	45.57	-	-	-	-	-	-	-	-	-
XR-Transformer-3	79.4	59.02	46.25	-	-	-	35.76	42.22	46.36	-	-	-
Renée	79.47	60.37	46.84	79.47	72.73	70.49	33.56	42.73	47.12	33.56	41.89	46

Table 17. Results on other metrics, including $P@k$ for Label features(LF) datasets. "-" means the result on that dataset and method pair are unavailable. NGAME-2 means the fusion (ensemble between encoder and classifier) performance as reported in (Dahiya et al., 2022). Results for other methods are as reported in (Bhatia et al., 2016).

Methods	P@1	P@3	P@5	N@1	N@3	N@5	PSP@1	PSP@3	PSP@5	PSN@1	PSN@3	PSN@5
LF-AmazonTitles-131K												
XR-Transformer	38.1	25.57	18.32	38.1	38.89	40.71	28.86	34.85	39.59	28.86	32.92	35.21
LightXML	35.6	24.15	17.45	35.6	36.33	38.17	25.67	31.66	36.44	25.67	29.43	31.68
DECAF	38.4	25.84	18.65	38.4	39.43	41.46	30.85	36.44	41.42	30.85	34.69	37.13
ECLARE	40.74	27.54	19.88	40.74	42.01	44.16	33.51	39.55	44.7	33.51	37.7	40.21
SiameseXML	41.42	27.92	21.21	41.42	42.65	44.95	35.8	40.96	46.19	35.8	39.36	41.95
NGAME-2	46.01	30.28	21.47	46.01	46.69	48.67	38.81	44.4	49.43	38.81	42.79	45.31
Renée	**46.05**	**30.81**	**22.04**	**46.05**	**47.46**	**49.68**	**39.08**	**45.12**	**50.48**	**39.08**	**43.56**	**46.24**
LF-Wikipedia-500K												
XR-Transformer	81.62	61.38	47.85	81.62	74.46	72.43	33.58	42.97	47.81	33.58	42.21	46.61
LightXML	81.59	61.78	47.64	81.59	74.73	72.23	31.99	42	46.53	31.99	40.99	45.18
ECLARE	68.04	46.44	35.74	68.04	58.15	56.37	31.02	35.39	38.29	31.02	35.66	34.5
SiameseXML	67.26	44.82	33.73	67.26	56.64	54.29	33.95	35.46	37.07	33.95	36.58	38.93
NGAME-2	84.01	64.69	49.97	84.01	78.25	75.97	**41.25**	52.57	**57.04**	**41.25**	**51.58**	**56.11**
Renée	**84.95**	**66.25**	**51.68**	**84.95**	**79.79**	**77.83**	39.89	**51.77**	56.7	39.89	50.73	55.57
LF-AmazonTitles-1.3M												
XR-Transformer	50.14	44.07	39.98	50.14	47.71	46.59	20.06	24.85	27.79	20.06	23.44	25.41
LightXML	-	-	-	-	-	-	-	-	-	-	-	-
DECAF	50.67	44.49	40.35	50.67	48.05	46.85	22.07	26.54	29.3	22.07	25.06	26.85
ECLARE	50.14	44.09	40	50.14	47.75	46.68	23.43	27.9	30.56	23.43	26.67	28.61
SiameseXML	49.02	42.72	38.52	49.02	46.38	45.15	27.12	30.43	32.52	27.12	29.41	30.9
NGAME-2	**56.75**	49.19	44.09	**56.75**	53.84	52.41	**29.18**	33.01	35.36	**29.18**	32.07	33.91
Renée	56.04	**49.91**	**45.32**	56.04	**54.21**	**53.15**	28.54	**33.38**	**36.14**	28.54	**32.15**	**34.18**
LF-WikiSeeAlso-320K												
XR-Transformer	42.57	28.24	21.3	42.57	41.99	43.44	25.18	30.13	33.79	25.18	29.84	32.59
LightXML	34.5	22.31	16.83	34.5	33.21	34.24	17.85	21.26	24.16	17.85	20.81	22.8
DECAF	41.36	28.04	21.38	41.36	41.55	43.32	25.72	30.93	34.89	25.72	30.69	33.69
ECLARE	40.58	26.86	20.14	40.48	40.05	41.23	26.04	30.09	33.01	26.04	30.06	32.32
SiameseXML	42.16	28.14	21.39	42.16	41.79	43.36	29.02	32.68	36.03	29.02	32.64	35.17
NGAME-2	47.65	31.56	23.68	47.65	47.5	48.99	**33.83**	**37.79**	**41.03**	**33.83**	**38.36**	**41.01**
Renée	**47.86**	**31.91**	**24.05**	**47.86**	**47.93**	**49.63**	32.02	37.07	40.9	32.02	37.52	40.6
LF-Amazon-131K												
XR-Transformer	45.61	30.85	22.32	45.61	47.1	49.65	34.93	42.83	49.24	34.93	40.67	43.91
LightXML	41.49	28.32	20.75	41.49	42.7	45.23	30.27	37.71	44.1	30.27	35.2	38.28
DECAF	42.94	28.79	21	42.94	44.25	46.84	34.52	41.14	47.33	34.52	39.35	42.48
ECLARE	43.56	29.65	21.57	43.56	45.24	47.82	34.98	42.38	48.53	34.98	40.3	43.37
SiameseXML	44.81	30.19	21.94	44.81	46.15	48.76	37.56	43.69	49.75	37.56	41.91	44.97
NGAME-2	46.53	30.89	22.02	46.53	47.44	49.58	38.53	44.95	50.45	38.53	43.07	45.81
Renée	**48.05**	**32.33**	**23.26**	**48.05**	**49.56**	**52.04**	**40.11**	**47.39**	**53.67**	**40.11**	**45.37**	**48.52**