Net2Tab: Neural Network to Tables for Data Prefetching

Pengmiao Zhang, Neelesh Gupta, Rajgopal Kannan, Viktor K. Prasanna *Fellow, IEEE*

***Abstract*—Neural Networks (NNs) have demonstrated exceptional performance in memory access prediction through their ability to extract latent features from complex patterns and generalize to unseen data. However, deploying NN-based models for hardware data prefetching poses two key challenges: 1) balancing the computational demands of NNs with hardware constraints, and 2) designing custom prefetchers that leverage diverse NN architectures. To address these challenges, we propose *Net2Tab*, a framework that transfers knowledge from a NN to a hierarchy of tables, enabling the practical implementation of NN-based hardware prefetchers. Net2Tab introduces: 1) a novel *CDTF (Configuration, Distillation, Tabularization, and Fine-tuning) methodology* for creating table-based NN approximations that meet the design constraints of a hardware prefetcher, and 2) a set of *tabularization kernels* that convert various NN architectures into table lookups, facilitating the design of custom prefetchers. We evaluate the tabularization kernels using NNs based on linear, LSTM, convolution, and attention architectures. Results show that Net2Tab reduces arithmetic operations by 98.98%-99.83%, achieves 46.81×-92.07× acceleration, and maintains 88.50%-89.31% of memory access prediction F1-score compared to the original NN-based models. We apply Net2Tab to convert state-of-the-art NN-based prefetchers into table-based approximations. These approximations achieve 18.24×–67.65× speedups and 99.79%–99.99% reduction in arithmetic operations compared to the original, computationally demanding NN models, resulting in 2.84%–13.46% higher IPC improvement over the best-performing rule-based prefetcher BO.**

***Index Terms*—memory access prediction, prefetching, neural network, knowledge distillation, product quantization**

I. Introduction

PREFETCHING is a key technique for mitigating the memory wall problem in modern computer systems by predicting and preloading data to reduce memory access latency and enhance system throughput [1], [2]. Research on hardware prefetcher design focuses on two main approaches: rule-based and machine learning (ML)-based prefetchers [3].

Rule-based prefetchers, such as the widely used Best-Offset prefetcher (BO) [4] and Irregular Stream Buffer (ISB) [5], rely on heuristic rules to identify memory access patterns. These prefetchers have low hardware overhead, using compact tables to record recent patterns and predict future accesses. While effective for regular access patterns with high spatial and temporal locality, their predefined rules limit effectiveness in complex workloads and unseen patterns.

P. Zhang, N. Gupta, and V. K. Prasanna are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, USA. E-mail: {pengmiao, neeleshg, prasanna}@usc.edu

R. Kannan is with the DEVCOM Army Research Office, Los Angeles, USA. E-mail: rajgopal.kannan.civ@army.mil

Distribution Statement A: Approved for public release. Distribution is unlimited.

In response, ML algorithms have been applied to address the limitations of traditional rule-based prefetchers [6], [7]. Particularly, Neural Networks (NNs) have demonstrated high accuracy in the task of memory access prediction due to their ability to extract latent features from complex patterns and generalize predictions to unseen data [3], [8], [9].

However, deploying NN-based memory access prediction models for hardware data prefetching presents practical challenges. *1) Balancing the computational demands of NNs with hardware constraints.* High-performing NNs often have a large number of parameters, placing significant demands on system resources. Their complexity can result in high inference latency, leading to untimely data prefetching. Furthermore, the intensity of BLAS operations involved in NN inference, mainly matrix multiplications, consume substantial resources and energy. Unfortunately, current ML approaches that use pruning [10], quantization [11], and parallel implementations [12] to accelerate NNs cannot fully address this issue, as they still require numerous matrix multiplications during model inference. *2) Designing custom prefetchers using diverse NN architectures.* Existing NN-based prefetchers have explored various architectures to extract latent patterns in memory access sequences, enhancing prediction and prefetching performance. These architectures include Multi-Layer Perceptron (MLP) [13], [14], Long Short-Term Memory (LSTM) [6], [7], [15]–[17], Convolution [18], [19], and Attention [3], [20], [21]. The rapid advancement of NN-based prefetcher research continually introduces new models that incorporate various combinations of these architectures, such as Voyager [9], which uses LSTM and Attention, and TransFetch [3], which uses MLP and Attention. Hence, hardware designs for these highly customized and specialized models require lengthy development cycles and are not easily scalable.

To address these challenges, we propose *Net2Tab*, a framework that transfers knowledge from a NN to table hierarchies, improving practicality of NN-based prefetchers for hardware implementation. Net2Tab consists of 1) a novel *CDTF (Configuration, Distillation, Tabularization, and Fine-tuning) methodology* that constructs table-based NN approximations given prefetcher design constraints, and 2) a set of *tabularization kernels* that convert various basic NN architectures into table lookups, facilitating custom prefetcher designs. In this paper, we use the term *TabNN* to indicate the Table-based approximation of a Neural Network.

The CDTF (Configuration, Distillation, Tabularization, and Fine-tuning) methodology aims to reduce NN complexity to meet hardware prefetcher design constraints while maintaining model performance. To achieve this, we introduce several optimizations. First, to meet design constraints on model storage cost and inference latency, we design a *NN-Table Co-Configuration* module that adjusts NN layers, hidden dimensions, and TabNN configurations to ensure that storage and latency requirements are satisfied. Second, we apply knowledge distillation to transfer knowledge from the trained memory access prediction model to the model configured by the structure tuner. Then, we propose approximating NN layers with simple and fast table lookups based on Product Quantization (PQ) [22], referred to as *tabularization*. This approximates the entire model as table hierarchies (i.e., TabNN), eliminating all matrix multiplications during inference. Finally, we introduce layer-wise fine-tuning to mitigate error accumulation. This approach learns the mapping from TabNN layer inputs to NN layer outputs, calibrating the data distribution of TabNN layers. Through these optimizations, we transfer knowledge from a large, high-complexity NN to a simple hierarchy of tables that meets hardware prefetcher design constraints.

We design tabularization kernels for common NN architectures in prefetching to develop a general TabNN generation framework. Based on product quantization, each kernel is optimized for the unique architectural characteristics. For the linear kernel, we extend PQ from vector dot products to higher-dimension inputs and integrate bias into the constructed table. For the LSTM kernel, we unroll the recurrent structure into a multi-layer NN and construct tables per time step for hidden states. For the convolution kernel, we use the im2col [23] approach to convert convolution operations into matrix multiplications. For the attention kernel, we use pairwise dot products and two-step quantization to construct the table approximation to replace two matrix multiplications.

We summarize our main contributions below:

- We propose Net2Tab, a framework that converts NN-based memory access prediction models into table-based approximations, towards practical hardware prefetchers.
- We outline a general CDTF methodology for Net2Tab that transfers knowledge from a NN to a hierarchy of tables. This process involves model configuration, knowledge distillation, tabularization, and fine-tuning, while meeting hardware implementation constraints and maintaining NN performance.
- We design kernels in Net2Tab to convert commonly-used NN architectures—linear, LSTM, convolution, and attention—into tables. These kernels enable the construction of table-based approximations (TabNNs) to generate custom NN-based prefetchers consisting of these layers.
- We evaluate the tabularization kernels of Net2Tab. Compared to the standard NNs, the tabularized models reduce arithmetic operations by 98.98%-99.83%, achieve 46.81×-92.07× acceleration, and maintain 88.50%-89.31% of memory access prediction F1-score.
- We apply Net2Tab to convert state-of-the-art NN-based prefetchers to TabNNs. TabNN-based prefetchers achieve 18.24×–67.65× acceleration and cut arithmetic operations by 99.79%–99.99% compared to the original, computationally demanding NN-based models, resulting in 2.84%–13.46% higher IPC improvement over the best rule-based prefetcher BO at similar latencies.

II. Background

A. Memory Access Prediction Using Neural Networks

Memory access prediction is crucial for identifying patterns in past behavior and accurately forecasting future addresses, enabling efficient data prefetching to reduce latency and boost performance. While traditional prefetchers leverage spatial or temporal locality, Neural Network (NN) models allow for predicting irregular access patterns. The problem definition of NN-based memory access prediction is outlined below.

Problem Definition. Let $X_t = \{x_1, x_2, \ldots, x_N\}$ denote the sequence of N historical memory accesses at time t, and $Y_t = \{y_1, y_2, \ldots, y_k\}$ represent the k future accesses. A neural network can approximate $P(Y_t \mid X_t)$, the probability of accessing the future accesses Y_t based on the history X_t.

By capturing complex dependencies, NNs improve prediction accuracy over traditional methods [3]. While existing models emphasize improving memory access prediction accuracy, this work focuses on making these models more hardware-friendly by reducing their computational complexity with only a small performance trade-off.

B. Product Quantization

Product Quantization (PQ) [22] is an algorithm that approximates computing vectors' inner product through quantization and precomputation. Given a vector $\boldsymbol{a} \in \mathbb{R}^D$ from a training set $\tilde{\boldsymbol{A}} \in \mathbb{R}^{N\times D}$ with N samples, and a fixed weight vector $\boldsymbol{b} \in \mathbb{R}^D$, PQ generates a quantized approximation $\hat{\boldsymbol{a}}$ such that $\hat{\boldsymbol{a}}^\top \boldsymbol{b} \approx \boldsymbol{a}^\top \boldsymbol{b}$. To quantize $\boldsymbol{a}$, the D-dimensional vector is divided into C subspaces, each of dimension V. Within each subspace, K prototypes are learned as quantized subvectors. Since $\hat{\boldsymbol{a}}$ is quantized and $\boldsymbol{b}$ is fixed, $\hat{\boldsymbol{a}}^\top \boldsymbol{b}$ can be precomputed and reused during queries. Figure 1 provides an overview of PQ, with the detailed process described below.

1) Training: PQ training consists of two main steps: prototype learning for each subspace and constructing the table to store precomputed results.

Prototype Learning (p)**.** Let $\tilde{\boldsymbol{A}}_c \in \mathbb{R}^{N\times V}$ represent the subvectors in the c-th subspace from the training set $\tilde{\boldsymbol{A}}$. The goal is to learn K prototypes $\boldsymbol{P}_{c,k}$ by minimizing the distance between each subvector $\tilde{\boldsymbol{A}}_c$ and its nearest prototype $\boldsymbol{P}_{c,k}$, where k indexes the prototypes within subspace c. This process is formulated in Equation 1.

$$p(\tilde{\boldsymbol{A}})_c \triangleq \arg\min_{P} \sum_{c} \sum_{i} \left\| \tilde{\boldsymbol{A}}_{c,i} - \boldsymbol{P}_{c,k} \right\|^2 \quad (1)$$

Table Construction (h)**.** A table is constructed by computing and storing the dot products of prototypes $\boldsymbol{P}_{c,k}$ from each subspace with their corresponding weight vectors $\boldsymbol{b}_c$. As shown in Equation 2, the function $h(\boldsymbol{b})_{c,k}$ represents the entry at index (c, k) in the table.

$$h(\boldsymbol{b})_{c,k} \triangleq \boldsymbol{b}_c{}^\top \cdot \boldsymbol{P}_{c,k} \quad (2)$$

2) Query: The query process eliminates the need for direct multiplication when computing dot products by encoding the query vector to the nearest prototypes, retrieving precomputed results from the table, and performing aggregation.

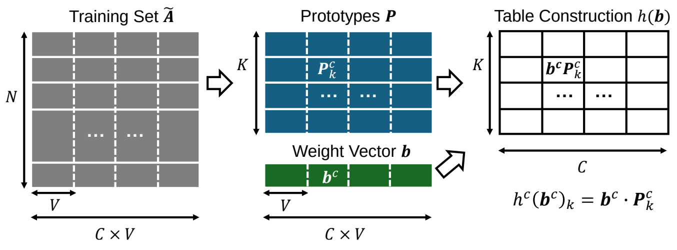
Vector Encoding (g). Given the query vector $\boldsymbol{a}$, $g(\boldsymbol{a})_c$ identifies the closest prototype $\boldsymbol{P}_{c,k}$ in each subspace c by finding the index k that minimizes the distance to $\boldsymbol{a}_c$, as shown in Equation 3. The result is a set of indices representing the encoding of $\boldsymbol{a}$ using the prototypes.

$$g\left(\boldsymbol{a}\right)_c \triangleq \arg\min_k \|\boldsymbol{a}_c - \boldsymbol{P}_{c,k}\|^2 \tag{3}$$

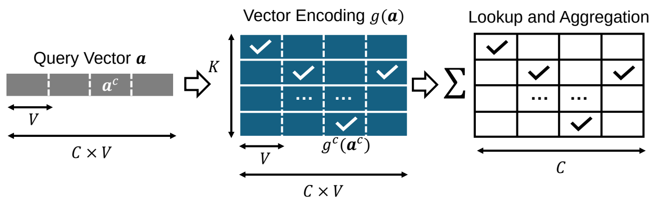
Table Lookup and Aggregation (f). The dot product $\boldsymbol{a}^\top\boldsymbol{b}$ is approximated by retrieving precomputed values via the encoded indices and aggregating the subspaces using an aggregation function f, as shown in Equation 4.

$$f(\boldsymbol{a},\boldsymbol{b}) \triangleq \sum_c h(\boldsymbol{b})_{c,k}, \quad k = g(\boldsymbol{a})_c \tag{4}$$

This query method bypasses direct dot product computation by using table lookups. By employing locality-sensitive hashing [24] for encoding and parallel summation for aggregation, the computational complexity is significantly reduced, particularly for high-dimensional vectors.


Training Set $\tilde{\boldsymbol{A}}$
N
V
$C \times V$
Prototypes $\boldsymbol{P}$
K
$\boldsymbol{P}_k^c$
Weight Vector $\boldsymbol{b}$
$\boldsymbol{b}^c$
V
$C \times V$
Table Construction $h(\boldsymbol{b})$
K
$\boldsymbol{b}^c\boldsymbol{P}_k^c$
C
$h^c(\boldsymbol{b}^c)_k = \boldsymbol{b}^c \cdot \boldsymbol{P}_k^c$

(a) Product quantization training. Prototypes are learned for each subspace, and dot products are precomputed and stored in a table.



(b) Product quantization query. The input vector is encoded to identify the nearest prototype index, which is then retrieved from the table for aggregation to produce the final result.

Fig. 1. Product Quantization (PQ) accelerates dot products by retrieving precomputed values through table lookups.

III. Related Work

A. Neural Networks for Data Prefetching

Neural networks (NNs) have been increasingly explored for memory access prediction due to their adaptability to complex patterns and generalizability to unseen data [25], [26]. Peled et al. [13] use a Multi-Layer Perceptron (MLP) to learn correlations between contextual information and memory access patterns. Hashemi et al. [15] extensively evaluate Long Short-Term Memory (LSTM) networks, highlighting their effectiveness in predicting complex patterns. Srivastava et al. [16] propose a compact LSTM-based model using memory access jumps (delta) as input and binary encoded deltas as output. Tan et al. [19] apply convolutional neural networks (CNNs) for prefetching due to their efficiency and stability. CNNs have also served as baselines in [18] and [3]. Recently, the attention mechanism [27] has significantly improved NN-based prefetching. Voyager [9] combines LSTM and attention to enhance predictions, while TMP [19] integrates convolution and attention to fully exploit memory access correlations. TransFetch [3] employs a fully attention-based network with context input, achieving state-of-the-art prefetching performance. Despite these successes, existing NN-based prefetchers often prioritize prediction accuracy over model complexity, which is critical for real-world deployment due to constraints like latency and storage [28]. This work focuses on reducing the complexity of memory access prediction models to facilitate practical implementation in hardware prefetching.

B. Neural Network Acceleration and Approximation

Various approaches have been explored to reduce Neural Network (NN) complexity and accelerate inference. Model compression techniques such as parameter pruning [10], [29], quantization [11], [30], low-rank factorization [31], and knowledge distillation [32]–[34] aim to minimize redundancy and model size to enhance performance. Pipeline and parallel designs have been explored to accelerate convolution and attention-based models [35], [36]. MEViT [12] and AttMemo [37] accelerate Transformers by optimizing memory usage. Although these methods effectively reduce model complexity or accelerate model inference, they still rely on matrix multiplications. Researchers have explored reducing matrix multiplications in neural networks. Lin et al. [38] use trained binary parameters to eliminate multiplication, while shift-and-add strategies [39], [40] replace multiplications with additions and bit-shifts for better power efficiency. Despite reducing computational complexity, these methods retain original NN structure and require all computation steps. Razlighi et al. [41] quantize NN inputs and weights and store pairwise multiplications in tables, which incurs additional storage costs. Blalock et al. [24] use product quantization (PQ) to turn multiplications between inputs and weights into table lookups, mainly applying this method to the last linear layer to avoid error accumulation across layers. Our previous work [42] extends PQ to attention-based models and mitigates error accumulation through fine-tuning, but it lacks scalability and is limited to a specific memory access prediction model. In contrast, this paper introduces a novel framework that transforms the entire neural network into a table hierarchy for hardware data prefetching, supports various neural network structures for customized models, and provides a comprehensive design methodology and training scheme.

IV. Net2Tab Framework: Overview

We introduce a novel generic framework Net2Tab (Network to Tables) that transfers knowledge from a Neural Network (NN) to table hierarchies, which meets the constraints of hardware prefetcher design in a memory system.

A. Problem Definition

Given a trained NN for memory access prediction $\mathcal{M}$ with parameters $\boldsymbol{\theta}$, training data $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ from memory accesses, and hardware prefetcher design constraints—including latency τ and storage s—our objective is to construct a table-based approximation $\mathcal{T}$ with parameters ϕ, referred to as TabNN, such that $\mathcal{T}(\mathbf{x};\phi)$ closely approximates the output of $\mathcal{M}$ while adhering to the hardware design constraints. We formalize this as follows:

$$\min_{\phi}\left[\frac{1}{N}\sum_{i=1}^{N}\|\mathcal{M}(\mathbf{x}_i;\boldsymbol{\theta})-\mathcal{T}(\mathbf{x}_i;\phi)\|^2\right] \qquad (5)$$
$$\text{s.t.} \quad \mathcal{L}(\mathcal{T})<\tau, \quad \mathcal{S}(\mathcal{T})<s$$

where $\mathcal{L}(\cdot)$ represents the model inference latency and $\mathcal{S}(\cdot)$ represents the model storage cost.

B. Workflow

Figure 2 illustrates the overall structure of Net2Tab and how it enables practical hardware implementation for NN-based last-level cache (LLC) prefetching. To construct a table-based approximation of a given neural network, Net2Tab employs a novel CDTF method, consisting of four key steps: Configuration, Distillation, Tabularization, and Fine-tuning.

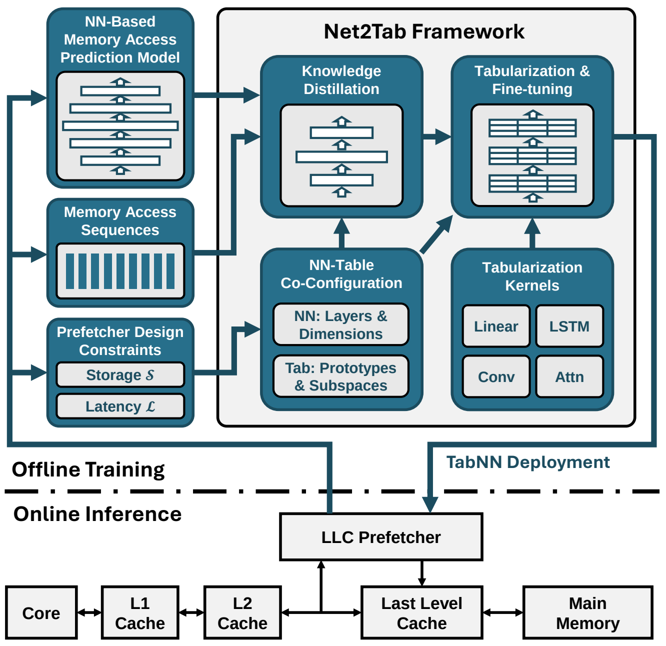


Fig. 2. Net2Tab constructs a table-based approximation (TabNN) of a neural network model for practical hardware prefetcher implementation.

Step 1: Configuration. The complexity of the final TabNN model depends on both the neural network architecture and the dimensions of the constructed table. To ensure compliance with hardware implementation constraints, we design an NN-Table Co-Configuration module in Net2Tab, which determines the neural network structure (e.g., the number of layers and hidden dimensions) and table configurations (e.g., the number of prototypes and subspaces) that satisfy storage cost and latency requirements, referred to as ”valid” structures.

Step 2: Distillation. After configuring the NN structure that can lead to a valid TabNN model, we can apply Knowledge Distillation (KD) [32] to transfer knowledge from the trained large memory access prediction model (as teacher model) to the smaller valid neural network (as student model).

Step 3: Tabularization. We convert each layer of the distilled model to a table-based approximation using Product Quantization. We design tabularization kernels for commonly used neural network architectures, including linear, LSTM, convolutional, and attention. Whole model tabularization is streamlined through these readily available kernels.

Step 4: Fine-tuning. As the number of tabularized layers increases, error accumulation can degrade approximation performance. We address this with a layer-wise fine-tuning method that retrains NN layer weights based on preceding tabular layer outputs, calibrating distribution shifts.

The trained table-based model can be deployed to the last-level cache (LLC) for hardware prefetching. The prediction process primarily involves table lookups with minimal arithmetic operations. Net2Tab ensures fast model inference and straightforward hardware implementation.

V. Tabularization Kernels

We design tabularization kernels for commonly used neural network architectures, including linear, LSTM, convolutional, and attention. These kernels facilitate the tabularization of the entire model and provide complexity references for configuring neural networks and table approximations under design constraints.

A. Linear Kernel

A linear layer in a neural network is shown in Equation 6:

$$\text{Linear}\,(\boldsymbol{X})=\boldsymbol{W}\boldsymbol{X}+\boldsymbol{B} \qquad (6)$$

where $\boldsymbol{W}\in\mathbb{R}^{D_O\times D_I}$ is the layer weight, $\boldsymbol{B}\in\mathbb{R}^{D_O\times T}$ is the layer bias, and $\boldsymbol{X}\in\mathbb{R}^{D_I\times T}$ is the layer input. D_I and D_O are the input dimension and output dimension, respectively. T represents the input length. $T>1$ when a linear layer follows an LSTM, Attention or CNN layers where T represents the input sequence length or number of channels. $T=1$ when the input is a single-feature vector.

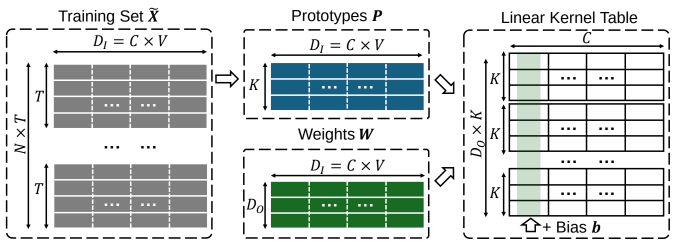


Fig. 3. Linear kernel that transfers knowledge from a linear layer to tables.

1) Training: Figure 3 shows the process of transferring knowledge from a linear layer to tables. Using the training set $\tilde{\boldsymbol{X}}\in\mathbb{R}^{N\times T\times D_I}$, we train prototypes using locality-sensitive hashing [24] within each subspace of size V across all $N\times T$ input vectors $\tilde{\boldsymbol{X}}_r\in\mathbb{R}^{NT\times D_I}$. Then, the prototype $\boldsymbol{P}_{c,k}$ in subspace index c and prototype index k performs a dot product with the linear layer weight $\boldsymbol{W}_{o,c}$ (output index o, subspace index c) across all output dimensions. This results in D_O

tables for all output dimensions, each storing the precomputed products using $K \times C$ entries. Additionally, we merge the bias into the tables. The bias $\boldsymbol{B}$ in Equation 6 is a T-repeat of the bias vector $\boldsymbol{b} \in \mathbb{R}^{D_O}$ added to all T outputs. We add the bias $\boldsymbol{b_O}$ to a single subspace of its corresponding output table. In this way, the bias can be aggregated into the final result during query. In practice, we reshape the bias to match the constructed table $\boldsymbol{b}_r \in \mathbb{R}^{D_O \times K \times C}$, repeating values in each Do dimension K times for the first C subspace column and filling the rest with zeroes. $\boldsymbol{b}_r$ is then added to the learned table. The formal expression for the table construction is:

$$h(\boldsymbol{W})_{o,c,k} = \boldsymbol{W}_{o,c}{}^\top \cdot p(\tilde{\boldsymbol{X}}_r)_{c,k} + \boldsymbol{b}_r \tag{7}$$

2) Query: For a query input $\boldsymbol{X} \in \mathbb{R}^{T \times D_I}$, the t-th row vector $\boldsymbol{X}^t$ is encoded to generate results for all D_O output dimensions following the PQ query process. The query of each dimensions are independent and can execute in parallel. Formally, the linear output $\hat{\boldsymbol{Y}} \in \mathbb{R}^{T \times D_O}$ is generated through:

$$\hat{\boldsymbol{Y}}_o^t = \sum_c h(\boldsymbol{W})_{o,c,k}, k = g(\boldsymbol{X}^t)_c \tag{8}$$

3) Complexity: Equation 9 shows the linear kernel latency. It comprises the encoding of subvectors g, the table lookups h, and subspace aggregation f. We use the locality-sensitive hashing (LSH) in [24] as the encoding function, the latency is $\log(K)$ for K prototypes. In this paper, we use "⇒" to show the complexity analysis assuming fully parallel implementation and LSH for encoding.

$$\mathcal{L} = \mathcal{L}_g + \mathcal{L}_f + \mathcal{L}_h \Rightarrow \log(K) + \log(C) + 1 \tag{9}$$

Equation 10 shows the linear kernel storage cost in bits. The encoding process g requires TC indices of prototypes, with each index requiring $\log(K)$ bits. Additionally, there are $D_O KC$ entries in the constructed tables, each taking d bits.

$$\mathcal{S} = \mathcal{S}_g + \mathcal{S}_h \Rightarrow TC\log(K) + D_O KCd \tag{10}$$

Equation 11 shows the arithmetic operations in the inference of table-based linear layer. It consists of two parts, the encoding g to get table indexes and the aggregation f after table look-up to get the output results.

$$\mathcal{A} = \mathcal{A}_g + \mathcal{A}_f \Rightarrow TC\log(K) + TD_O\log(C) \tag{11}$$

B. LSTM Kernel

Long Short-Term Memory (LSTM) [43] is a variant of Recurrent Neural Network (RNN) that overcomes the gradient vanishing problems of basic RNNs. At time step t, an LSTM unit processes the input gate $\boldsymbol{i}^t$, the forget gate $\boldsymbol{f}^t$, the output gate $\boldsymbol{o}^t$, and the cell input $\tilde{\boldsymbol{c}}^t$, generating the cell state $\boldsymbol{c}^t$ and hidden state $\boldsymbol{h}^t$ that can be used for the next time step. The formal expression of LSTM is:

$$\begin{aligned}
\boldsymbol{i}^t &= \sigma(\boldsymbol{W}^i\boldsymbol{x}^t + \boldsymbol{U}^i\boldsymbol{h}^{t-1} + \boldsymbol{b}^i)\\
\boldsymbol{f}^t &= \sigma(\boldsymbol{W}^f\boldsymbol{x}^t + \boldsymbol{U}^f\boldsymbol{h}^{t-1} + \boldsymbol{b}^f)\\
\boldsymbol{o}^t &= \sigma(\boldsymbol{W}^o\boldsymbol{x}^t + \boldsymbol{U}^o\boldsymbol{h}^{t-1} + \boldsymbol{b}^o)\\
\tilde{\boldsymbol{c}}^t &= \tanh(\boldsymbol{W}^c\boldsymbol{x}^t + \boldsymbol{U}^c\boldsymbol{h}^{t-1} + \boldsymbol{b}^c)\\
\boldsymbol{c}^t &= \boldsymbol{f}^t \odot \boldsymbol{c}^{t-1} + \boldsymbol{i}^t \odot \tilde{\boldsymbol{c}}^t\\
\boldsymbol{h}^t &= \boldsymbol{o}^t \odot \tanh(\boldsymbol{c}^t)
\end{aligned} \tag{12}$$

where $\boldsymbol{x}^t \in \mathbb{R}^{D_I}$ is the input at time step t. $\boldsymbol{W}^i, \boldsymbol{W}^f, \boldsymbol{W}^o, \boldsymbol{W}^c \in \mathbb{R}^{D_H \times D_I}$ are the weight matrices for the input gate, forget gate, output gate, and cell candidate, respectively, while $\boldsymbol{U}^i$, $\boldsymbol{U}^f$, $\boldsymbol{U}^o$, $\boldsymbol{U}^c \in \mathbb{R}^{D_H \times D_H}$ are the corresponding hidden state weight matrices. $\boldsymbol{b}^i, \boldsymbol{b}^f, \boldsymbol{b}^o, \boldsymbol{b}^c \in \mathbb{R}^{D_H}$ are the bias terms. σ denotes the sigmoid function, $\tanh$ denotes the hyperbolic tangent function, and $\odot$ denotes element-wise multiplication. $\boldsymbol{c}^{t-1}$ and $\boldsymbol{h}^{t-1}$ are the previous cell state and hidden state. D_I and D_H are the input and hidden dimensions.

1) Training: The four gates follow the same structure so we use general notations $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{b}$ to show the design of one gate. Let the input training set be $\tilde{\boldsymbol{X}} \in \mathbb{R}^{N \times T \times D_I}$, where T is the number of time steps for an LSTM layer to generate the output result, N is the number of training samples. Let $\tilde{\boldsymbol{H}} \in \mathbb{R}^{N \times T \times D_H}$ be the hidden states generated for all time steps given the input $\tilde{\boldsymbol{X}}$. Figure 4 shows the process of constructing tables for an LSTM kernel. First, we train general input prototypes for $\boldsymbol{X}$ across all time steps using the reshaped input $\tilde{\boldsymbol{X}}_r \in \mathbb{R}^{NT \times D_I}$, and store the $\boldsymbol{WX}$ result in tables using Equation 7, referred to as WX tables. For the hidden states, we unroll the recurrent structure, train prototypes for each time step, and store the multiplications with weights $\boldsymbol{U}$ using tables for each time step, referred to as UH tables, shown in Equation 13. The reshaped bias $\boldsymbol{b}_r$ is added to the WX tables to reduce the query complexity.

$$h^t_{o,c,k}(\boldsymbol{U}) = \boldsymbol{U}_{o,c}{}^\top \cdot p_{c,k}(\tilde{\boldsymbol{H}}^{t-1}) \tag{13}$$

where o, c, k are the indexes of the table entries for the output dimension D_O, subspaces C, and prototypes K.

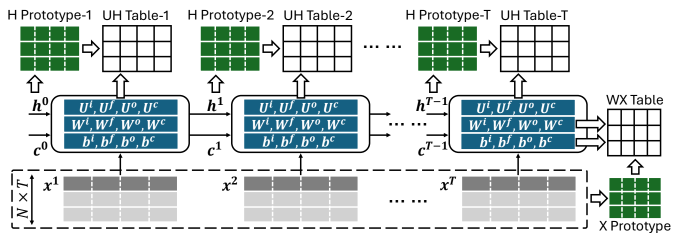


Fig. 4. LSTM kernel training. We construct a WX table using the input vectors across all time steps and samples. We construct UH tables for each time step by unrolling the recurrent structure.

2) Query: The query vector $\boldsymbol{x}^t$ at time steps t is encoded to generate results from the WX tables. The hidden state $\boldsymbol{h}^{t-1}$ is encoded to look up results from the UH table for step t. The gate result before the activation function is $\boldsymbol{y}^t$. The query process can be expressed as:

$$\begin{gathered}
\hat{\boldsymbol{y}}_o^t = \sum_c h(\boldsymbol{W})_{o,c,i} + \sum_c h(\boldsymbol{U})^t_{o,c,j}\\
i = g(\boldsymbol{x}^t)_c, j = g(\boldsymbol{h}^{t-1})_c
\end{gathered} \tag{14}$$

where o, c are the indexes of output dimension and subspaces. i, j are the indexes of the closest prototypes for the input vector and the hidden vector and subspace c.

3) Complexity: Equation 15 shows the latency of a table-based LSTM layer assuming a parallel process among the independent linear operations. $\mathcal{L}_\sigma$ is the activation function latency and $\mathcal{L}_{em}$ is the element-wise multiplication latency. $\mathcal{L}_\sigma = 1$ for lookup table implementation and $\mathcal{L}_{em} = 1$ for fully parallel processing.

$$\begin{aligned} \mathcal{L} &= T(\mathcal{L}_g + \mathcal{L}_f + \mathcal{L}_h + 2\mathcal{L}_\sigma + 2\mathcal{L}_{em}) \\ &\Rightarrow T(\log(K) + \log(C) + 1 + 2(\mathcal{L}_\sigma + \mathcal{L}_{em})) \end{aligned} \tag{15}$$

Equation 16 shows the storage cost of an LSTM kernel. The WX table, S_{h_w}, is independent of time steps, while the encodings S_{g_w} and S_{g_u} are local to each time step. The UH table, S_{h_u}, is stored for all time steps. S_{act} represents the storage cost for the activation functions.

$$\begin{aligned} \mathcal{S} &= 4(\mathcal{S}_{g_w} + \mathcal{S}^d_{h_w} + \mathcal{S}_{g_u}) + 4T\mathcal{S}^d_{h_u} + \mathcal{S}_{act} \\ &\Rightarrow 8C\log(K) + 4(T+1)D_H KCd + \mathcal{S}_\sigma + \mathcal{S}_{\text{tanh}} \end{aligned} \tag{16}$$

Equation 17 shows the arithmetic operations in an LSTM kernel. It consists of encoding and aggregation process for all gates across all time steps, as well as the element-wise multiplications.

$$\begin{aligned} \mathcal{A} &= 4T(\mathcal{A}_{g_x} + T\mathcal{A}_{f_x} + \mathcal{A}_{g_h} + T\mathcal{A}_{f_h}) + 3T\mathcal{A}_{em} \\ &\Rightarrow 8T(C\log(K) + D_H\log(C)) + 3TD_H \end{aligned} \tag{17}$$

C. Convolution Kernel

To convert a convolution layer to table lookups, we first employ im2col method [23] to map the convolution operation into a matrix multiplication, then apply product quantization to approximate the matrix multiplication. To avoid confusion, we use the term "filter" to represent the feature extraction matrix for convolution and use "kernel" to represent the tabularization module that transfers a convolution operation to table lookups.

1) Training: Figure 5 illustrates the table construction for the convolution operation. The im2col algorithm reshapes the convolutional layer inputs from $D_I \times H \times W$ to an input patch matrix of size $HW \times F_H F_W D_I$ (assuming padding), where D_I represents the number of channels, H and W are the input data dimensions, and F_H and F_W are the filter dimensions. For N input samples, we train K input prototypes for the NHW vectors within each subspace of size V. Concurrently, we reshape the D_O filters from dimensions $F_H \times F_W \times D_I$ to a filter path matrix of size $D_O \times F_H F_W D_I$, converting the convolution operation into a dot product between the input patch and the filter patch. The dot products between the learned prototypes and the filter patch vectors are precomputed and stored in the convolution kernel table. The product quantization step is the same as a linear kernel, following Equation 7.

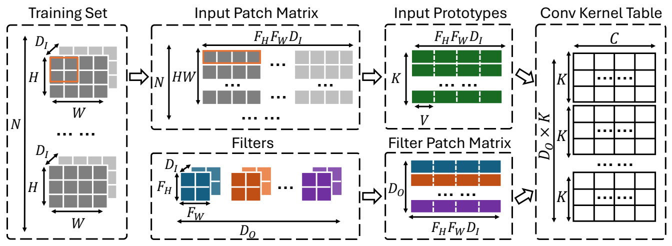


Fig. 5. Convolution kernel training. We employ im2col method to convert the input training set and the filters into matrixes, then apply product quantization to construct tables that stores the matrix multiplcation results.

2) Query: Incoming query data is reshaped into a 2D patch matrix using the im2col method. Each row is divided into C subspaces, and each subvector is matched with a prototype using a trained clustering or hashing function. The $HW \times C$ subvectors operate independently, enabling parallel prototype matching. The indexes of the matched prototypes are used to retrieve precomputed dot products from the D_O trained tables for each output channel. These lookups are also independent, allowing parallel output channel processing. The subspace results are then aggregated, avoiding matrix multiplications in convolution inference, following Equation 8.

3) Complexity: Given the convolution operation for an input with D_I channels of size $H \times W$, stride T, and padding P with D_O filters of size $F_W \times F_H$, we define $H' = \frac{H - F_H + 2P}{T} + 1$ and $W' = \frac{W - F_W + 2P}{T} + 1$. Equation 18 shows the latency of convolution kernel inference, consisting of the reshape of im2col $\mathcal{L}_{rs}$ and a linear operation latency $\mathcal{L}_l$. Equation 19 and Equation 20 show the storage cost and arithmetic operations of the table-based convolution, which are similar to the analysis of linear kernel.

$$\mathcal{L} = \mathcal{L}_{rs} + \mathcal{L}_l \Rightarrow \log(K) + \log(C) + 2 \tag{18}$$

$$\mathcal{S} = \mathcal{S}_g + \mathcal{S}_h \Rightarrow H'W'C\log(K) + D_O CKd \tag{19}$$

$$\mathcal{A} = \mathcal{A}_g + \mathcal{A}_f \Rightarrow H'W'(C\log(K) + D_O\log(C)) \tag{20}$$

D. Attention Kernel

The attention mechanism excels in memory access prediction due to its high accuracy, adaptability, and parallelizability [3], [27]. Given three distinct matrices: $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{T \times D_k}$ the scaled dot-product attention is defined as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{D_k}}\right)\boldsymbol{V} \tag{21}$$

where D_k is the dimension of $\boldsymbol{K}$. Tabularizing attention processes is challenging due to the absence of a fixed weight matrix, making precomputation in linear kernels unfeasible. To address this, we propose to tabularize the pairwise dot products of the quantized inputs. Additionally, attention requires two matrix multiplications, which can inflate table depth to K^3 for K prototypes, leading to high storage use. Our solution involves secondary quantization of the intermediate results to reduce table depth to $2K^2$. Lastly, operations like scaling and Softmax activation add complexity. We streamline this by integrating these operations into the prototypes during training, eliminating such operations during a query.

1) Training: Figure 6 illustrates the table construction process for attention operation. First, we train K prototypes for each of the input training set $\tilde{\boldsymbol{Q}} \in \mathbb{R}^{N \times T \times D_k}$, $\tilde{\boldsymbol{K}} \in \mathbb{R}^{N \times T \times D_k}$, and $\tilde{\boldsymbol{V}} \in \mathbb{R}^{N \times T \times D_k}$. We perform a pairwise product between the Q prototypes and the K prototypes within a subspace, generating a K^2 depth table with width C_k, referred to as QK table, as shown in Equation 22:

$$h(\tilde{\boldsymbol{Q}}, \tilde{\boldsymbol{K}})_{c,i,j} = p(\tilde{\boldsymbol{Q}}_r)_{c,i} \cdot p(\tilde{\boldsymbol{K}}_r)_{c,j} \tag{22}$$

where $\tilde{\boldsymbol{Q}}_r$ and $\tilde{\boldsymbol{K}}_r$ are reshaped input matrix with size $\mathbb{R}^{NT \times D_k}$, i and j are the index of prototypes. Using the trained

QK table, we can generate the set of approximated $\tilde{\boldsymbol{Q}}\tilde{\boldsymbol{K}}^{\top t}$, which are the output of the QK table after aggregation, shown as below:

$$\tilde{\boldsymbol{Q}}\tilde{\boldsymbol{K}}^{\top t} = \sum_c h(\tilde{\boldsymbol{Q}}^t, \tilde{\boldsymbol{K}}^t)_{c,i,j}, i = g_c(\tilde{\boldsymbol{Q}}^t), j = g_c(\tilde{\boldsymbol{K}}^t) \quad (23)$$

The generated results from QK table are processed by scaling and softmax activation following the attention mechanism. This intermediate results are quantized to prototypes, refered to as QK prototype. Eventually, the QK prototype along with the V prototypes perform pairwise dot product and construct the QKV table:

$$h(\boldsymbol{Q}\tilde{\boldsymbol{K}}^{\top}, \tilde{\boldsymbol{V}})_{c,i,j} = \text{Softmax}(p(\boldsymbol{Q}\tilde{\boldsymbol{K}}^{\top})_{c,i}/\sqrt{D_k}) \cdot p(\tilde{\boldsymbol{V}}_r)_{c,j} \quad (24)$$

The two constructed tables are stored for query while the learned prototypes are not.

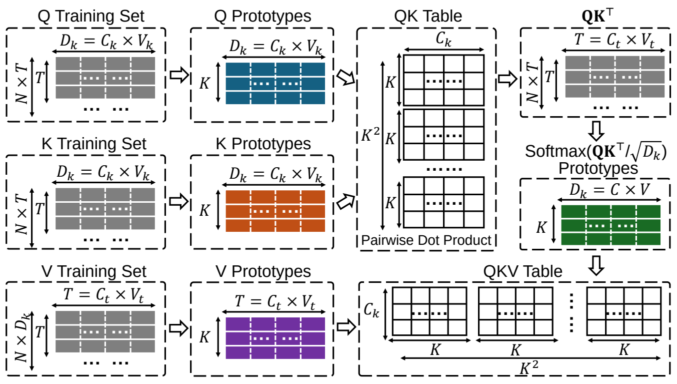


Fig. 6. Attention kernel training. We store pairwise products of learned prototypes for query reuse. To prevent table size expansion, we quantize the products of Q and K. We also merge Softmax and scaling within the learned prototypes and integrate them into the stored results in the QKV table.

2) Query: The query process involves two key steps of table lookups. First, the input matrices $\boldsymbol{Q}$ and $\boldsymbol{K}$ are encoded, and their dot product results are retrieved from the QK table as per Equation 22, resulting in the estimation $\boldsymbol{Q}\hat{\boldsymbol{K}}^{\top}$. Next, $\boldsymbol{Q}\hat{\boldsymbol{K}}^{\top}$ is encoded and, along with the encoded input $\boldsymbol{V}$, looked up in the QKV table. The results are then aggregated to provide the final estimation of the attention operation, as shown in Equation 25.

$$\hat{\boldsymbol{Y}}^t = \sum_c h(\boldsymbol{Q}\hat{\boldsymbol{K}}^{\top}, \boldsymbol{V})_{c,i,j}, i = g_c(\boldsymbol{Q}\hat{\boldsymbol{K}}^{\top t}), j = g(\boldsymbol{V}^t)_c \quad (25)$$

The attention kernel query eliminates matrix multiplications, scaling calculations, and activation operations.

3) Complexity: Equation 26 shows the attention kernel latency, consisting of the input encoding latency g for $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$, the aggregation for the QK Table, the encoding g_{qk} for the approximated $\boldsymbol{Q}\boldsymbol{K}^{\top}$, and the final aggregation f:

$$\begin{aligned} \mathcal{L} &= \mathcal{L}_{g_i, g_{qk}} + \mathcal{L}_{f_{qk}, f_{qkv}} + \mathcal{L}_{h_{qk}, h_{qkv}} \\ &\Rightarrow 2\log(K) + \log(C_k) + \log(C_t) + 2 \end{aligned} \quad (26)$$

Equation 27 shows the attention kernel storage cost, consisting of four encoding operations and two tables, where g_q, g_k, g_v, g_{qk} is for the encoding of attention input matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ and intermediate result $\boldsymbol{Q}\boldsymbol{K}^{\top}$.

$$\begin{aligned} \mathcal{S} &= \mathcal{S}_{g_q, g_k, g_{qk}, g_v} + \mathcal{S}^d_{h_{qk}, h_{qkv}} \\ &\Rightarrow (2TC_k + TC_t + D_k C_t)\log(K) + K^2(C_k + C_t)d \end{aligned} \quad (27)$$

Equation 28 shows the arithmetic operations. There are four encoding process g and two aggregation f processes.

$$\begin{aligned} \mathcal{A} &= \mathcal{A}(g_q, g_k, g_{qk}, g_v) + \mathcal{A}(f_{qk}, f_{qkv}) \\ &\Rightarrow (2TC_k + TC_t + D_k C_t)\log(K) \\ &\quad + T^2\log(C_k) + D_k^2\log(C_t) \end{aligned} \quad (28)$$

VI. Constructing Table-Based Approximations of a Neural Network

To construct a table-based approximation of a whole Neural Network (NN), i.e., TabNN, we propose a novel CDTF procedure, including steps of Configuration, Distillation, Tabularization, and Fine-tuning.

A. NN-Table Co-Configuration Module

As analyzed in Section V, the latency and storage cost of a tabularization kernel depend on both the original NN dimensions and the structure of the constructed table. Converting a multi-layer NN into a hierarchy of tables is affected by the number of layers L, as it determines the critical path and table lookup layers, influencing the tabularized model's complexity. The aim of designing a smaller, valid NN is to enable homogeneous knowledge distillation from the original model. This process retains the core structure while adjusting dimensions and depth, yielding a table-based model that satisfies hardware prefetcher constraints on latency and storage.

We design a NN-Table Co-Configuration module that searches for valid structures of a neural network and its table approximation, ensuring that storage cost and latency meet the implementation constraints. For a given model, we define candidate configuration lists for the number of neural network layers L, layer hyperparameters (e.g., hidden dimension D, filter sizes F_W, F_H, number of heads in multi-head attention H), and the table configuration (the number of prototypes K and subspaces C). Using these configurations, we generate a dictionary based on the complexity analysis in Section V that maps each specific configuration to its latency and storage cost. We then perform a latency-major greedy approach to search among both network and table configurations using the dictionary. The process starts by identifying configurations with the highest latency less than the latency constraint τ. Among these, it selects the configuration with the maximum storage capacity less than the storage constraint s. If no suitable configuration is found, it proceeds to configurations with lower latency, continuing until a configuration satisfying both the latency and storage constraints is identified.

B. Knowledge Distillation for Complexity Reduction

With the valid neural network model configuration, we apply Knowledge Distillation (KD) [32] to train the valid compact model (student model) using the original trained large

Algorithm 1 Layer-Wise Tabularization with Fine-Tuning

```
1:  Input: Trained $N$-layer model $\mathcal{M}$, Training input data $\mathcal{D}$
2:  Initialize: Model layer $i$ output $L[i] \leftarrow \mathcal{M}[0:i](\mathcal{D})$
3:  Initialize: Table hierarchy $\mathcal{T}$, fine-tune epoch $E$
4:  Initialize: Configuration lists prototypes $K$, subspace $C$
5:  for $i$ in $0$ to $N-1$ do
6:      if $i > 0$ and trainable parameters in $\mathcal{M}[i]$ then
7:          $\mathcal{M}[i] \leftarrow$ FINETUNE($\mathcal{M}[i], \mathcal{T}[0:i\text{-}1](\mathcal{D}), L[i], E$)
8:      if $\mathcal{M}[i]$ is a linear layer then
9:          $\mathcal{T}_\text{i} \leftarrow$ LINEARKERNEL($\mathcal{M}[i], K[i], C[i]$)
10:     else if $\mathcal{M}[i]$ is an LSTM layer then
11:         $\mathcal{T}_\text{i} \leftarrow$ LSTMKERNEL($\mathcal{M}[i], K[i], C[i]$)
12:     else if $\mathcal{M}[i]$ is a convolution layer then
13:         $\mathcal{T}_\text{i} \leftarrow$ CONVKERNEL($\mathcal{M}[i], K[i], C[i]$)
14:     else if $\mathcal{M}[i]$ is an attention layer then
15:         $\mathcal{T}_\text{i} \leftarrow$ ATTENTIONKERNEL($\mathcal{M}[i], K[i], C[i]$)
16:     else
17:         $\mathcal{T}_\text{i} \leftarrow \mathcal{M}[i]$
18:     Push $\mathcal{T}_\text{i}$ to $\mathcal{T}$
19: Output: $\mathcal{T}$
```

model (teacher model). Since popular prefetching models, such as TransFetch [3], use multi-label classification instead of the single-label classification of vanilla KD, we extend the vanilla KD T-Softmax [32] to a T-Sigmoid function for binary cross entropy (BCE) in multi-label classification training. The T-Sigmoid function is defined in Equation 29.

$$z_i = p(y_i)_{t=T} = \sigma\left(\frac{y_i}{T}\right) = \frac{1}{1+e^{-y_i/T}} \tag{29}$$

The complete loss function encompasses both the BCE loss ($Loss_{BCE}$) and the soft KD loss ($Loss_{KD}$), as defined in Equation 30:

$$\begin{aligned} Loss_{\text{KD}} &= \sum_{k=1}^{q} \text{KL}\left(\left[z_i^{tch}, 1-z_i^{tch}\right] \,\|\, \left[z_i^{stu}, 1-z_i^{stu}\right]\right) \\ Loss &= \lambda Loss_{\text{KD}} + (1-\lambda) Loss_{\text{BCE}} \end{aligned} \tag{30}$$

where $\text{KL}(\cdot\|\cdot)$ is the Kullback-Leibler divergence [44], λ is a hyper-parameter tuning the weights of the two losses, z_i^{tch} and z_i^{stu} are T-Sigmoid output of teacher and student models.

C. Layer-Wise Tabularization

With the configured table structure and the trained compact neural network, we perform layer-wise tabularization, converting the network into table hierarchies layer by layer. Lines 8-18 in Algorithm 1 illustrate this process. For each neural network layer, we detect the operation and apply the corresponding tabularization kernel introduced in Section V, covering linear, LSTM, convolution, and attention operations. We initialize the output table hierarchy as an empty queue $\mathcal{T}$ and push each converted layer to $\mathcal{T}$.

D. Fine-Tuning

While performing layer-wise tabularization for the neural network model $\mathcal{M}$, the input of layer i for the table-based approximation $\mathcal{T}$ is $\hat{\boldsymbol{X}} = \boldsymbol{X} + \epsilon_x$, where $\boldsymbol{X}$ is the original NN layer input, and ϵ_x is the error introduced by tabularization. This leads to an error in the tabularized output for the matrix multiplication between weights and inputs, $\boldsymbol{W}\hat{\boldsymbol{X}} + \boldsymbol{b} = \boldsymbol{Y} + \epsilon_y$. As the number of tabularized layers increases, errors accumulate, reducing approximation performance.

To mitigate this issue, we fine-tune the layer weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$. For layer i, we initialize $\boldsymbol{W}$ and $\boldsymbol{b}$ using the trained model, taking the output of the $i-1$ layer of TabNN as input $\hat{\boldsymbol{X}}$ and the original NN layer output $\boldsymbol{Y}$ as the target, performing E epochs of NN layer weights retraining. We use the Mean Squared Error (MSE) loss function to learn the updated layer $\mathcal{M}'[i]$ with the new weights $\boldsymbol{W}'$ and biases $\boldsymbol{b}'$, as shown in Equation 31. Fine-tuning is applied to layers $i > 0$ when there are trainable parameters, as shown in Algorithm lines 6-7. Then, the i-th layer tabularization is based on the fine-tuned NN weights. The fine-tuning calibrates the table approximation using the NN layer outputs, effectively mitigating error accumulation.

$$(\boldsymbol{W}', \boldsymbol{b}') = \arg\min_{\boldsymbol{W},\boldsymbol{b}} \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{d} \Big(\boldsymbol{Y}_{ij} - (\boldsymbol{W}\hat{\boldsymbol{X}}_{ij} + \boldsymbol{b})\Big)^2 \tag{31}$$

VII. Evaluation

A. Experimental Setup

1) Dataset: We evaluate the prediction and prefetching performance of Net2Tab using traces generated from benchmarks *SPEC CPU 2006* [45], *SPEC CPU 2017* [46], and *GAP* [47]. We use the first 50M instructions for model training, and the following 50M instructions for model testing and prefetching simulation. Table I shows the statistics of the benchmarks, including the number of unique program counters (PCs), addresses, page addresses, and deltas.

TABLE I
BENCHMARK STATISTICS

Benchmark	# PCs	# Addresses	# Pages	# Deltas
SPEC 06	23∼893	60.0K∼2.21M	2.51K∼88.9K	23.6K∼2.01M
SPEC 17	26∼1126	62.1K∼1.78M	7.99K∼ 0.26M	3.18K∼0.72M
GAP	63∼118	0.56M∼1.25M	8.27K∼ 27.2K	0.30M∼1.20M

2) Simulator: Following existing literature, we use ChampSim [48] to generate traces and evaluate our approach. ChampSim simulates a modern multi-core system with a configurable memory hierarchy. Table III details the simulation parameters, with prefetchers simulated at the last-level cache (LLC).

B. Evaluation of Prediction Performance

Existing NN-based prefetchers use various input formats and output targets, making direct performance comparisons infeasible. To evaluate Net2Tab for transferring knowledge from neural networks to tables, we fix the model input and output and apply Net2Tab to various NN architectures, the experimental results are shown in Table II.

TABLE II
F1-Score of Models Applying Net2Tab for Memory Access Prediction

							F1-Score												
Model†				Complexity			SPEC 2006				SPEC 2017				GAP				
Backbone	KD	TA	FT	$\mathcal{L}$(Cycle)	$\mathcal{S}$(KB)	$\mathcal{A}$(K)	bwaves	milc	lesl	libq	gcc	mcf	lbm	wrf	bc	bfs	cc	pr	Mean
MLP (Tch)	✗	✗	✗	3867	2157.2	549.9	0.935	0.813	0.590	0.995	0.940	0.776	0.664	0.657	0.331	0.557	0.401	0.644	0.692
MLP (Stu)	✗	✗	✗	663	139.3	35.1	0.902	0.745	0.570	0.995	0.933	0.741	0.629	0.648	0.321	0.557	0.367	0.535	0.662
MLP (Stu)	✓	✗	✗	663	139.3	35.1	0.939	0.770	0.577	0.995	0.937	0.743	0.641	0.649	0.314	0.557	0.399	0.570	0.674
MLP (Tab)	✓	✓	✗	42	896.2	5.6	0.679	0.527	0.497	0.995	0.936	0.591	0.611	0.646	0.303	0.557	0.317	0.527	0.599
MLP (Tab)	✓	✓	✓	42	896.2	5.6	0.784	0.578	0.514	0.995	**0.937**	0.651	0.612	0.646	**0.303**	**0.557**	0.323	0.522	0.618
LSTM (Tch)	✗	✗	✗	5693	1329.0	2797.1	0.949	0.844	0.634	0.995	0.935	0.817	0.673	0.681	0.363	0.565	0.396	0.644	0.708
LSTM (Stu)	✗	✗	✗	989	55.0	62.9	0.878	0.787	0.599	0.995	0.932	0.741	0.632	0.649	0.301	0.557	0.353	0.604	0.669
LSTM (Stu)	✓	✗	✗	989	55.0	62.9	0.895	0.833	0.608	0.995	0.933	0.746	0.647	0.656	0.316	0.558	0.384	0.607	0.681
LSTM (Tab)	✓	✓	✗	99	968.1	11.9	0.800	0.569	0.525	0.995	0.935	0.654	0.608	0.634	0.299	0.557	0.250	0.524	0.612
LSTM (Tab)	✓	✓	✓	99	968.1	11.9	**0.809***	0.596	0.537	0.995	0.935	0.684	**0.621**	0.638	0.290	**0.557**	0.249	**0.530**	0.620
CNN (Tch)	✗	✗	✗	5305	2862.9	5082.3	0.938	0.822	0.601	0.995	0.940	0.790	0.674	0.658	0.349	0.557	0.401	0.678	0.700
CNN (Stu)	✗	✗	✗	1179	36.4	93.4	0.848	0.789	0.511	0.995	0.934	0.769	0.649	0.648	0.335	0.557	0.359	0.556	0.663
CNN (Stu)	✓	✗	✗	1179	36.4	93.4	0.896	0.802	0.522	0.995	0.936	0.770	0.655	0.648	0.338	0.557	0.360	0.575	0.671
CNN (Tab)	✓	✓	✗	80	682.3	16.5	0.638	0.544	0.487	0.994	0.821	0.696	0.541	0.636	0.304	0.557	0.256	0.500	0.581
CNN (Tab)	✓	✓	✓	80	682.3	16.5	0.795	0.595	0.493	0.995	0.936	0.677	0.524	0.635	0.293	**0.557**	0.262	0.498	0.605
Attention (Tch)	✗	✗	✗	4541	2452.5	6540.4	0.969	0.863	0.628	0.995	0.952	0.791	0.686	0.677	0.350	0.558	0.403	0.689	0.713
Attention (Stu)	✗	✗	✗	905	77.0	125.0	0.923	0.725	0.589	0.995	0.946	0.772	0.631	0.660	0.338	0.557	0.372	0.622	0.678
Attention (Stu)	✓	✗	✗	905	77.0	125.0	0.936	0.849	0.601	0.995	0.947	0.779	0.654	0.660	0.343	0.557	0.388	0.659	0.697
Attention (Tab)	✓	✓	✗	97	864.4	10.8	0.783	0.643	0.416	0.995	0.903	0.687	0.535	0.626	0.268	0.539	0.286	0.487	0.597
Attention (Tab)	✓	✓	✓	97	864.4	10.8	0.796	**0.666**	**0.542**	**0.995**	0.933	**0.689**	0.544	**0.651**	0.285	**0.557**	**0.391**	0.520	**0.631**

† Tch: teacher model; Stu: student model; Tab: TabNN model; KD: knowledge distillation; TA: tabularization; FT: fine-tuning.
∗ Bold values are the highest among all fine-tuned TabNN models.

TABLE III
Simulation Parameters

Parameter	Value
CPU	4 GHz, 4 cores, 4-wide OoO, 256-entry ROB, 64-entry LSQ
L1 I-cache	64 KB, 8-way, 8-entry MSHR, 4-cycle
L1 D-cache	64 KB, 12-way, 16-entry MSHR, 5-cycle
L2 Cache	1 MB, 8-way, 32-entry MSHR, 10-cycle
LL Cache	8 MB, 16-way, 64-entry MSHR, 20-cycle
DRAM	$t_{RP} = t_{RCD} = t_{CAS} =$ 12.5 ns, 2 channels 8 ranks, 8 banks, 32K rows, 8GB/s bandwidth

TABLE IV
Configurations of the Implemented Models

Backbone	Version†	Configuration∗	Backbone	Version	Configuration
MLP	Tch	L=8, D=256	CNN	Tch	L=18, D=16, F=3
	Stu	L=4, D=64		Stu	L=9, D=4, F=3
	Tab	C=2, K=256		Tab	C=2, K=256
LSTM	Tch	L=2, D=128	Attention	Tch	L=2, D=128, H=4
	Stu	L=1, D=32		Stu	L=1, D=32, H=2
	Tab	C=1, K=128		Tab	C=2, K=128

† Tch: teacher model; Stu: student model; Tab: TabNN model.
∗ L: layer; D: dimension; F: filter; H: head; C: subspace; K: prototype.

1) Input and Target: Following TransFetch, we use segmented memory access addresses as input and block address delta bitmaps as the target [3].

2) Metrics: The models perform multi-label binary classification, so we use F1-Score to evaluate the performance [49].

3) Model Configurations: We implement and train teacher models using architectures aligned with our designed kernels, including MLP (based on linear operations), LSTM, CNN (convolution with residual connections [50]), and Attention (using ViT architecture [3], [51]). Under latency $\mathcal{L} < 100$ cycles and storage $\mathcal{S} < 1$ MB constraints, the NN-Table Co-Configuration module provides valid student models and TabNN structures, as shown in Table IV.

4) Results: Table II presents the results of models with configurations detailed in Table IV. For models based on MLP, LSTM, CNN, and Attention, knowledge distillation enhances the average F1-Score of student models by 1.87%, 1.87%, 1.30%, and 2.94%, respectively; fine-tuning improves the average F1-Score of the TabNN models by 3.17%, 1.30%, 4.13%, and 5.70%.

Compared to teacher models, TabNN accelerates model inference by 92.07×, 57.50×, 66.31×, and 46.81× for MLP, LSTM, CNN, and Attention, respectively, while reducing arithmetic operations by 98.98%, 99.57%, 99.67%, and 99.83%. Despite these optimizations, TabNN maintains F1-Score performance at 89.31%, 87.57%, 86.43%, and 88.50% for the respective models. In comparison to student models with an equal number of layers as the table-based models, TabNN provides a speedup in inference by 15.78×, 9.98×, 14.73×, and 9.33× for MLP, LSTM, CNN, and Attention. It also decreases arithmetic operations by 84.04%, 81.08%, 82.33%, and 91.36%, respectively, while preserving F1-Score performance at 93.35%, 92.67%, 91.25%, and 93.07%.

Among all model backbones, the Attention model achieves the highest mean F1-Score of 0.631, followed by the LSTM model at 0.620. In this memory access prediction task, MLP outperforms CNN due to latency constraints limiting the depth of the CNN model, which typically relies on deeper architectures for feature extraction. The LSTM model demonstrates

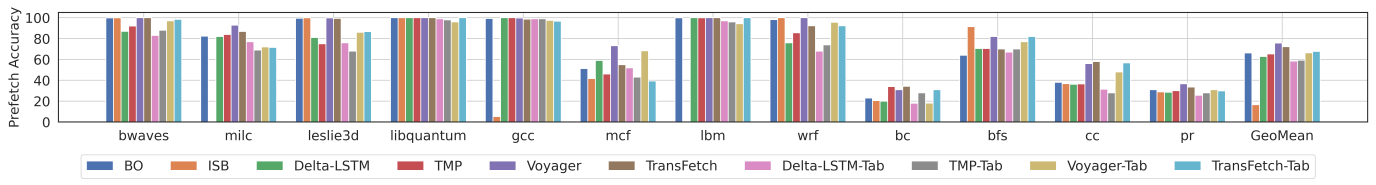


Fig. 7. Prefetch accuracy of baseline rule-based prefetchers, baseline NN-based prefetchers, and TabNN-based approximations of the NN-based prefetchers.

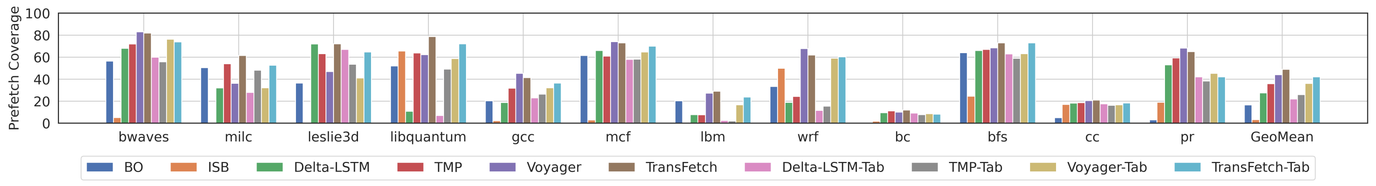


Fig. 8. Prefetch coverage of baseline rule-based prefetchers, baseline NN-based prefetchers, and TabNN-based approximations of the NN-based prefetchers.

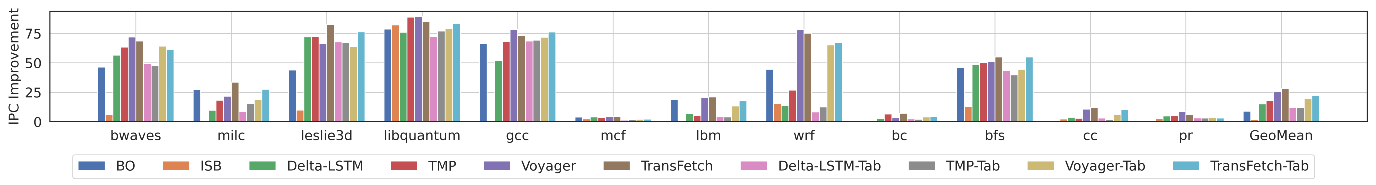


Fig. 9. IPC Improvement of baseline rule-based prefetchers, baseline NN-based prefetchers, and TabNN-based approximations of the NN-based prefetchers.

strong performance within these latency constraints, as the input history window is small (10 time steps), making the recurrent steps similar in depth to the student Attention model (8 layers) and the student CNN model (9 layers).

C. Evaluation of Prefetching Performance

We evaluate the overall effectiveness of Net2Tab framework for data prefetching by converting existing NN-based prefetchers to TabNN models and simulating on ChampSim.

1) Implemented Prefetchers: We implement state-of-the-art rule-based and NN-based prefetchers as baselines, and construct table-based approximations of the NN-based prefetchers (TabNN-based prefetchers) generated by Net2Tab. Table V details the implemented prefetchers. For the rule-based prefetchers, we implement the spatial Best-Offset Prefetcher (BO) and the temporal Irregular Stream Buffer (ISB). Rule-based prefetchers have a very low number of arithmetic operations, which can be ignored. For NN-based prefetchers, we implement state-of-the-art models that use various neural network structures, including the LSTM-based Delta-LSTM and Voyager, the convolution-based TMP (TCN-based Memory Prefetcher), and the attention-based TransFetch. By applying the proposed Net2Tab framework, we convert these NN models into tables under the constraints of 200 cycles latency and 4MB storage cost.

2) Metrics: We evaluate the performance of prefetchers using prefetch accuracy, prefetch coverage, and IPC improvement [52] compared to a baseline system with no prefetcher.

3) Results: The TabNN-based models generated from our Net2Tab framework result in accelerations of 67.65×, 29.90×, 18.24×, and 23.59× for Delta-LSTM, TMP, Voyager, and TransFetch, respectively, resulting in comparable latencies to rule-based prefetchers. Net2Tab reduces the number of arithmetic operations by 99.80%, 99.97%, 99.99%, and 99.95% for Delta-LSTM, TMP, Voyager, and TransFetch, respectively.

TABLE V
IMPLEMENTED PREFETCHERS AND THEIR COMPLEXITY

Type	Prefetcher	$\mathcal{L}$(Cycle)	$\mathcal{S}$(B)	$\mathcal{A}$(OPs)	Mechanism
Rule	BO [4]	60	4K	-	Spatial locality
	ISB [5]	30	8K	-	Temporal locality
NN	Delta-LSTM [16]	3247	1.96M	2.50M	LSTM, Linear
	TMP [19]	5652	6.54M	21.7M	Conv, Attn, Linear
	Voyager [9]	2773	14.9M	154M	LSTM, Attn, Linear
	TransFetch [3]	4506	13.8M	37.6M	Attn, Linear
TabNN	Delta-LSTM-Tab	48	3.77M	5.10K	LSTM, Linear
	TMP-Tab	189	1.19M	6.13K	Conv, Attn, Linear
	Voyager-Tab	152	3.88M	5.56K	LSTM, Attn, Linear
	TransFetch-Tab	191	3.75M	17.5K	Attn, Linear

Figures 7, 8, and 9 illustrate prefetch accuracy, coverage, and IPC improvement using the baseline rule-based and NN-based prefetchers, as well as the TabNN-based prefetchers generated by Net2Tab, denoted by ”-Tab”.

For prefetch accuracy, Delta-LSTM-Tab, TMP-Tab, Voyager-Tab, and TransFetch-Tab achieve accuracies of 58.44%, 59.27%, 66.27%, and 67.67%. These represent drops in accuracy of 4.36%, 6.02%, 9.41%, and 4.53% compared to NN-based implementations. When compared to BO (66.19%) and ISB (16.6%), TransFetch-Tab outperforms both, while the other models show slightly lower accuracy than BO.

Regarding prefetch coverage, Delta-LSTM-Tab, TMP-Tab, Voyager-Tab, and TransFetch-Tab achieve coverage of 22.03%, 25.99%, 36.16%, and 42.07% respectively. These figures show reductions of 5.47%, 9.96%, 7.89%, and 6.81% compared to the NN-based implementations. In comparison to BO (16.61%) and ISB (3.27%), the TabNN-based prefetchers outperform the rule-based prefetchers in terms of coverage.

With respect to IPC improvement compared to the system without a prefetcher, Delta-LSTM-Tab, TMP-Tab, Voyager-Tab, and TransFetch-Tab demonstrate improvements of 11.80%, 12.16%, 19.73%, and 22.41%, which indicate decreases of 3.37%, 5.84%, 6.05%, and 5.57% compared to the NN-based implementations. Notably, compared to BO (8.95%) and ISB (1.97%), the TabNN-based prefetchers show superior IPC improvement. Specifically, the highest-performing TabNN-based model, TransFetch-Tab, outperforms BO by 13.45%.

VIII. Conclusion

In this paper, we proposed Net2Tab, a framework that transfers knowledge from a neural network-based prefetching model to table hierarchies under hardware constraints for practical implementation. The key to our approach is a novel CDTF methodology for constructing table-based approximations of neural networks, along with a set of kernels that convert popular neural network layers into table lookups. We applied Net2Tab to construct table-based approximations of state-of-the-art NN-based prefetchers. Experimental results demonstrate that the table-based models accelerate the NN-based baseline prefetchers by 18.24×–67.65× and achieve 2.85%–13.45% higher IPC improvement than the best-performing rule-based baseline prefetcher.

In future work, we plan to optimize the table structure configuration and explore more efficient algorithms to identify valid NN and table structures under given accuracy and hardware constraints. Additionally, we aim to develop more efficient tabularization methods, such as converting multiple layers into a single table and introducing quantization to table entries, to further reduce model complexity.

Acknowledgment

This work has been supported by the U.S. National Science Foundation (NSF) under grant CNS-2009057 and SaTC-2104264, as well as the DEVCOM Army Research Lab (ARL) under grant W911NF2220159.

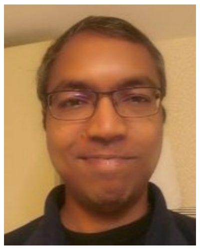
References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.
[2] M. Dubois, M. Annavaram, and P. Stenström, *Parallel computer organization and design*. cambridge university press, 2012.
[3] P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation for attention-based variable-degree prefetching," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 103–112.
[4] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 469–480.
[5] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 247–259.
[6] P. Zhang, A. Srivastava, T.-Y. Wang, C. A. De Rose, R. Kannan, and V. K. Prasanna, "C-memmap: clustering-driven compact, adaptable, and generalizable meta-lstm models for memory access prediction," *International Journal of Data Science and Analytics*, pp. 1–14, 2021.
[7] P. Zhang, A. Srivastava, B. Brooks, R. Kannan, and V. K. Prasanna, "Raop: Recurrent neural network augmented offset prefetcher," in *The International Symposium on Memory Systems*, 2020, pp. 352–362.
[8] P. Zhang, A. Srivastava, R. Kannan, A. V. Nori, and V. K. Prasanna, "Transformap: Transformer for memory access prediction," in *The International Symposium on Computer Architecture (ISCA), ML for Computer Architecture and Systems Workshop, 2021*, 2021.
[9] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 861–873.
[10] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021.
[11] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[12] K. Marino, P. Zhang, and V. K. Prasanna, "Me-vit: A single-load memory-efficient fpga accelerator for vision transformers," 2023.
[13] L. Peled, U. Weiser, and Y. Etsion, "A neural network prefetcher for arbitrary memory access patterns," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–27, 2019.
[14] H. Wang and Z. Luo, "Data cache prefetching with perceptron learning," *arXiv preprint arXiv:1712.00905*, 2017.
[15] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," *arXiv preprint arXiv:1803.02329*, 2018.
[16] A. Srivastava, A. Lazaris, B. Brooks, R. Kannan, and V. K. Prasanna, "Predicting memory accesses: the road to compact ml-driven prefetcher," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 461–470.
[17] A. Srivastava, T.-Y. Wang, P. Zhang, C. A. F. De Rose, R. Kannan, and V. K. Prasanna, "Memmap: Compact and generalizable meta-lstm models for memory access prediction," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2020, pp. 57–68.
[18] Y. Chen, Y. Zhang, J. Wu, J. Wang, and C. Xing, "Revisiting data prefetching for database systems with machine learning techniques," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 2165–2170.
[19] Y. Tan, Z. Zhang, Z. Ma, Y. Zhou, and D. Liu, "Towards the design of efficient tcn-basd prefetcher for hybrid nvm-dram memory," in *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2022, pp. 1–9.
[20] P. Zhang, R. Kannan, and V. K. Prasanna, "Phases, modalities, spatial and temporal locality: Domain specific ml prefetcher for accelerating graph analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–15.
[21] P. Zhang, R. Kannan, X. Tong, A. V. Nori, and V. K. Prasanna, "Sharp: Software hint-assisted memory access prediction for graph analytics," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–8.
[22] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.
[23] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.
[24] D. Blalock and J. Guttag, "Multiplying matrices without multiplying," in *International Conference on Machine Learning*. PMLR, 2021, pp. 992–1004.
[25] M. Islam, S. Banerjee, M. Meswani, and K. Kavi, "Prefetching as a potentially effective technique for hybrid memory optimization," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 220–231. [Online]. Available: https://doi.org/10.1145/2989081.2989129
[26] H. Choi and S. Park, "A survey of machine learning-based system performance optimization techniques," *Applied Sciences*, vol. 11, no. 7, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/7/3235
[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[28] S. Mohapatra and B. Panda, "Drishyam: An image is worth a data prefetcher," in *32nd ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '23)*, ACM. ACM New York, NY, USA, October 2023. [Online]. Available: https://doi.org/10.1145/3528416.3530224
[29] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," 2019.
[30] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.
[31] T. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," 10 2013, pp. 6655–6659.
[32] G. Hinton, O. Vinyals, J. Dean *et al.*, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, vol. 2, no. 7, 2015.
[33] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.
[34] N. Gupta, P. Zhang, R. Kannan, and V. Prasanna, "Packd: Pattern-clustered knowledge distillation for compressing memory access prediction models," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1–7.
[35] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," 2018.
[36] A. Mazaheri, T. Beringer, M. Moskewicz, F. Wolf, and A. Jannesari, "Accelerating winograd convolutions using symbolic computation and meta-programming," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.
[37] Y. Feng, H. Jeon, F. Blagojevic, C. Guyot, Q. Li, and D. Li, "Attmemo: Accelerating transformers with memoization on big memory systems," *arXiv preprint arXiv:2301.09262*, 2023.
[38] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.
[39] H. You, X. Chen, Y. Zhang, C. Li, S. Li, Z. Liu, Z. Wang, and Y. Lin, "Shiftaddnet: A hardware-inspired deep network," 2020.
[40] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, and J. Y. Li, "Deepshift: Towards multiplication-less neural networks," 2021.
[41] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "Looknn: Neural network with no multiplication," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1775–1780.
[42] P. Zhang, N. Gupta, R. Kannan, and V. K. Prasanna, "Attention, distillation, and tabularization: Towards practical neural network-based prefetching," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 876–888.
[43] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.
[44] J. M. Joyce, "Kullback-leibler divergence," in *International encyclopedia of statistical science*. Springer, 2011, pp. 720–722.
[45] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.
[46] S. CPU2017", "The standard performance evaluation corporation," https://www.spec.org/cpu2017/, 2017.
[47] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
[48] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," *arXiv preprint arXiv:2210.14324*, 2022.
[49] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *European conference on information retrieval*. Springer, 2005, pp. 345–359.
[50] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*. Springer, 2016, pp. 630–645.
[51] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.
[52] V. Srinivasan, E. S. Davidson, and G. S. Tyson, "A prefetch taxonomy," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 126–140, 2004.

Pengmiao Zhang received the BS degree in Electrical Engineering and Automation from Northeastern University (China) in 2013 and the MEng degree in Electrical Engineering from Harbin Institute of Technology in 2015. Currently, he is pursuing a Ph.D. degree in Computer Engineering at the University of Southern California. His research interests include time series data modeling, efficient machine learning, machine learning for computer systems, and memory system optimizations.

Neelesh Gupta is a fourth-year Bachelor's and joint Master's student majoring in Computer Science at the University of Southern California. He is interested in machine learning acceleration, machine learning-driven architectural support, and low-cost neural network inference.



Rajgopal Kannan received the Ph.D. degree in computer science from the University of Denver, in 1996. He is currently a Computer Scientist at the Army Research Lab in the Computing Architectures Branch and a research adjunct professor in electrical engineering at the University of Southern California. He was formerly a professor with the Department of Computer Science, Louisiana State University (2000–2015). His academic research was funded by DARPA, NSF and DOE and he has published more than 150 research papers in international journals and conferences with two patents awarded in the area of network optimization. His research interests are at the intersection of graph analytics, machine learning and edge computing — enabling application acceleration at the edge on low power devices, for example using Software-Defined Memory for memory bound applications. He is also interested in cyber–physical systems, especially data-driven models and analytics driving Smartgrid optimization and control.

Viktor K. Prasanna received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the Ph.D. degree in computer science from Pennsylvania State University. He is Charles Lee Powell chair in engineering in the Ming Hsieh Department of Electrical and Computer Engineering and professor of computer science at the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He serves as the director of the Center for Energy Informatics at USC. He served as the editor-in-chief of the IEEE Transactions on Computers during 2003–06. Currently, he is the editor-in-chief of the Journal of Parallel and Distributed Computing. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University, the 2019 Distinguished Alumnus Award from the Indian Institute of Science (IISc) and the 2016 Distinguished Alumnus Award from the University Visvesvaraya College of Engineering (UVCE), Bangalore University. He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. He is a fellow of the IEEE, the ACM, and the American Association for Advancement of Science (AAAS). He is an elected member of Academia Europaea.