Can Semi-Supervised Learning Improve Prediction of Deep Learning Model Resource Consumption?

Karthick Panner Selvam, Mats Brorsson
Snt, University of Luxembourg, Luxembourg

***Abstract*—As computational demands for deep learning models escalate, accurately predicting training characteristics like training time and memory usage has become crucial. These predictions are essential for optimal hardware resource allocation. Traditional performance prediction methods primarily rely on supervised learning paradigms. Our novel approach, TraPPM (Training characteristics Performance Predictive Model), combines the strengths of unsupervised and supervised learning to enhance prediction accuracy. We use an unsupervised Graph Neural Network (GNN) to extract complex graph representations from unlabeled deep learning architectures. These representations are then integrated with a sophisticated, supervised GNN-based performance regressor. Our hybrid model excels in predicting training characteristics with greater precision. Through empirical evaluation using the Mean Absolute Percentage Error (MAPE) metric, TraPPM demonstrates notable efficacy. The model achieves a MAPE of 9.51% for predicting training step duration and 4.92% for memory usage estimation. These results affirm TraPPM's enhanced predictive accuracy, significantly surpassing traditional supervised prediction methods. Code and data are available at: https://github.com/karthickai/trappm**

***Keywords*—Performance model; deep learning; Graph neural network**

I. Introduction

Deep learning (DL) has significantly advanced various fields by analyzing complex patterns in extensive datasets. The escalating complexity of DL models, driven by advances in computational resources and data availability, necessitates increased memory and computational power for training. This heightened demand complicates the training process and increases costs. Accurately predicting both memory consumption and step time for DL models is challenging due to a variety of hidden factors, including the choice of convolutional algorithms, garbage collection mechanisms, memory pre-allocation strategies, and the specifics of implementations like cuDNN [1]. These factors complicate the task of making precise predictions, highlighting the need for sophisticated approaches to accurately estimate these critical training characteristics. Effective prediction of memory and step time is not only essential for preventing out-of-memory errors but also plays a crucial role in optimizing resource allocation and enhancing the effectiveness of neural architectural search (NAS), ultimately enhancing the efficiency of the model development process

In past studies, researchers primarily employed supervised Multi-Layer Perceptron (MLP) and GNNs to predict the training and inference attributes of DL models [1]–[10]. These methods, while effective, are confined to the limits of supervised learning and do not fully exploit the potential of unlabeled data, which can significantly enhance prediction performance.

In response to this gap, we introduce TraPPM, a novel approach that leverages semi-supervised learning. First, we utilize unsupervised GNN to learn graph representations from an unlabeled dataset. GNNs are adept at capturing patterns and relationships within graph-structured data. Next, we combine the learned graph representations with static features of a DL model. With this integrated vector, we train the supervised GNN-based performance regressor using a labeled dataset, allowing it to accurately estimate the training step time and memory usage of a given DL model. Utilizing an embedding generated from unsupervised learning in conjunction with supervised training boosts performance prediction accuracy compared to relying solely on supervised training. Our key contributions include the following:

- We have implemented TraPPM, a novel methodology that leverages the unsupervised GNN for learning embeddings from unlabelled datasets. And combine the embedding with DL static features to train the GNN-based regressor model using a labeled dataset to predict the training characteristics without running it on target hardware.
- We rigorously assessed the performance of TraPPM against state-of-the-art baselines, including supervised GNN, MLP, and GBoost, TraPPM exhibits superior performance, achieving a remarkable 910 MB RMSE and 4.92% MAPE for memory and 23 ms RMSE and 9.51% MAPE for step-time prediction. This superior performance underscores the efficacy of harnessing unlabeled data for performance prediction.
- Furthermore, our comprehensive dataset, encompassing 8,079 labeled graphs and 25,053 unlabelled graphs from various DL model families, presents a substantial contribution to the community, paving the way for future research in performance prediction and optimization.

II. Background

1) Computational graphs: Deep learning models are usually represented as directed computational graphs, where each node represents mathematical operations, such as matrix multiplication, and edges represent the data flow between these nodes. For example, a simple Convolutional Neural Network (CNN) model. The image data is fed into the network via the input node, and it just passes data to the next node. The Conv nodes perform convolution operations on the input image data. The Pooling node is responsible for reducing the spatial dimensions of the input data to reduce computational requirements. The Fully Connected (FC) node, where each neuron is interconnected with all neurons from the previous layer, applies the activation function to a weighted sum of their inputs. Finally, the output node takes the data from the FC node and provides prediction results.

2) Graph neural networks: GNNs constitute a specialized class of deep learning models that operate on graph-structured data, denoted as $\mathcal{G} = (V, E)$, where V represents the set of nodes and E represents the set of edges in the graph. Each node $v_i \in V$ is associated with a feature vector, which encodes information about that node. The fundamental principle underlying GNNs is the iterative process known as message passing, which facilitates the generation of embeddings for nodes or entire graphs.

In the message passing process, each node v_i updates its embedding by aggregating information from its neighboring nodes. This aggregation is achieved through functions such as summation, averaging, or more intricate operations like neural networks or attention mechanisms. Let $\mathbf{h}_i^{(l)}$ denote the embedding of node v_i after l message passing iterations, where l represents the layer in the GNN. Initially, $\mathbf{h}_i^{(0)}$ corresponds to the node's original feature vector.

The update equation for node v_i at layer l in a GNN can be expressed as follows:

$$\mathbf{h}_i^{(l)} = \mathrm{TRANSFORM}\left(\mathbf{h}_i^{(l-1)}, \left\{\mathbf{h}_j^{(l-1)} : v_j \in \mathcal{N}(v_i)\right\}\right)$$

Here, $\mathbf{h}_j^{(l-1)}$ represents the embeddings of the neighboring nodes of v_i at the $(l-1)$-th layer, and $\mathcal{N}(v_i)$ denotes the set of neighbors of node v_i. The `TRANSFORM` function combines the embeddings of the node's neighbors with its own embedding from the previous layer. Through multiple layers of message passing, each node gathers information from an increasingly wider neighborhood in the graph. Thus, the final embedding $\mathbf{h}_i^{(l)}$ for node v_i after l layers encapsulates information from both its immediate and more distant neighbors within the graph. GNNs have demonstrated remarkable success in various graph-related tasks, including node classification, link prediction, and graph-level classification. Prominent GNN variants such as GraphSAGE [11], Graph Attention Networks (GAT) [12], and Graph Convolutional Networks (GCN) [13] have gained widespread adoption in the research community and have yielded state-of-the-art results in these tasks.

3) Graph auto encoders: GAEs [14], play a critical role in unsupervised learning with graphs. They are particularly useful when we have a lot of unlabeled data. A GAE comprises two essential parts: an encoder and a decoder. The encoder's role is to transform the input graph into lower-dimensional representations known as *embeddings* of nodes. This is often accomplished with a Graph Convolution Network (GCN), converting the input adjacency matrix A and feature matrix X into an embedding matrix Z. Where the adjacency matrix represents the connectivity between nodes in a graph. This can be written as $Z = encoder(X, A)$. The decoder takes the node embeddings produced by the encoder, the matrix Z, and tries to rebuild the original adjacency matrix. A common way of achieving this is using the node embeddings' inner product as the decoder function. The motivation here is that the inner product, as a similarity measure, can capture the likelihood of a link between two nodes. $A' = decoder(Z)$. The effectiveness of this transformation is evaluated using a loss function. This function measures the reconstruction error - the difference between the original adjacency matrix A and the reconstructed one A'. This discrepancy is usually calculated using a method like Binary Cross Entropy (BCE). The model is trained to minimize this loss, thus improving the GAE's precision. Having an established foundational understanding of DL as a computational graph and GAE, we can now delve into TraPPM's methodology. TraPPM leverages unsupervised GNN, particularly with a GAE, to learn the graph representation of unlabelled datasets. We utilize the computation graph as input to the TraPPM, with nodes representing operators and node features corresponding to operator attributes. The edges signify the connections between operators. We will explore it further in Section IV.

III. Related Work

The study of performance prediction of deep learning models is relatively recent, having only started to receive focus just a few years ago. Qi et al. [15] use a straightforward approach to estimate the training time of DL models, layer by layer, using an analytical model. They calculated each step duration and summed it to calculate the overall estimation. The model presumes no concurrent operations, which may only be accurate for some hardware types. Gao et al. [1] also used an analytical model to predict the memory consumption for the training DL model. Bouhali et al. [16] used an MLP-based regressor to predict the execution time of a DL model. They used input features such as trainable parameter count, memory size, and input size to predict the execution time. Nevertheless, the traditional MLP method could have been more effective due to its limited understanding of the DL layers.

Justus et al. [2] used the layer-by-layer technique proposed by Qi et al. [15] to improve the performance prediction accuracy. But use an MLP-based regression model instead of an analytic model. Gianti et al. [7] used a layer-by-layer technique as Justus et al. [2]. Instead of layer parameters, they used complex parameters such as FLOPS to predict the execution time and power of an individual layer of the DL model. Other researchers [17]–[20] also used the same layerwise approach to predict the execution time, memory allocation, and power consumption of the DL model. Yu et al. [3] employed a wave-scaling method for estimating the training step time of the deep learning model on a GPU. They also used the layerwise approach. However, this wave scaling technique necessitates the availability of a GPU to facilitate the prediction.

On the other hand, researchers used a graph neural network instead of MLP in a layerwise approach to predict the performance of the DL model [8], [9]. The layerwise approach did not capture the DL model network topology, and therefore prediction accuracy is sub-optimal [10]. To solve the above problem, Gao et al. [4] and other researchers, [5], [6], [10], used a graph learning to understand the model network topology by generating embeddings. Furthermore, they combine embeddings with overall DL features to predict the training and inference characteristics. The majority of prior studies utilized supervised techniques for DL model performance prediction, neglecting the vast pool of unlabelled DL model data. Our innovative approach, TraPPM, bridges this gap using a semi-supervised learning paradigm, enhancing prediction accuracy by harnessing unlabelled data.

In the first step, we employ an unsupervised graph neural network using unlabelled DL models. This network generates embeddings for input DL models, facilitating an in-depth understanding of the input DL model's network topology. In the subsequent step, we combine the embeddings with the static features extracted from a DL model. This fused data is utilized for training a GNN-based regressor using labeled data to predict the training characteristics. Our approach provides a more comprehensive and effective performance prediction mechanism than the previous works.

IV. Methodology

Our methodology consists of two phases. **Phase 1:** Unsupervised Learning, unlabelled DL graphs are trained using a GAE to generate embeddings, as explained in Section IV-B. However, we cannot directly feed the DL model in Open Neural Network Exchange (ONNX)[1] format into the input of GAE for training. Instead, we need to convert it to PyTorch Geometric (PyG) [21] data format before training, as explained in Section IV-A. **Phase 2:** Supervised Learning, the trained encoder from the GAE is utilized to generate embeddings for the labeled DL model. These embeddings and static features, along with the labeled DL models to, train GNN-based regressors to predict the training characteristics, as described in Section IV-C.

A. Graph Transformation

Given a DL model M with operations $O = \{o1, o2, ..., on\}$, we transform M (in ONNX format) into a graph G compatible with PyG. In G, nodes represent M's operations stored in the node feature matrix X, while A captures directed relationships. Specifically, $G = (X, A)$ where $X = O$ and $A[i][j] = 1$ if a directed edge exists from o_i to o_j, else $A[i][j] = 0$. If M is labeled, we incorporate a target vector Y into PyG data. For each node v in the DL model graph, we define an attribute vector A_v as: $[\mathbf{E}_O(v), I_v, \mathcal{O}_v, \text{Mac}_v, P_v, M_v]$. Here, $\mathbf{E}_O(v)$ is a one-hot encoded vector of length $|O|$, where $|O| = 98$, surpassing the previous work supported only 32 operators [10]. The vectors I_v and $\mathcal{O}_v$, each of length 6, encapsulate the input and output shape, respectively, with an extension to consider 3D convolution. The attributes Mac_v, P_v, and M_v symbolize the MAC, parameters, and memory of node v, respectively. Thus, our node feature vector n has a dimensionality of 113, offering a more exhaustive representation as shown in Fig. 1. To the best of our knowledge, this is the first work to incorporate 2D and 3D convolutions, alongside transformer-based architectures, into node features. This advancement distinguishes our approach from prior studies that were limited to 2D convolutions.

[1]https://github.com/onnx/onnx

$OneHot(Op_v)$	I_v	O_v	Mac_v	P_v	M_v
98	6	6	1	1	1

Fig. 1. The graph's nodes are augmented with node features, each consisting of 113 elements. To accommodate 3D convolution, padding was appended at the end of both the input and output shapes.

B. Phase 1: Unsupervised Learning

In order to leverage the potential of unlabelled data, we train the GAE model in unsupervised manner. The fundamentals of GAE are explained in Section II-3. However, it is not possible to directly use the DL model in ONNX format as input to the GAE. Therefore, we first transform the ONNX format to G as described in Section IV-A. The overview of our GAE is illustrated in Fig. 2.

The GAE's encoder is composed of four GraphSAGE convolution layers, which process node features of dimension $[\#nodes, 113]$. These layers aggregate neighborhood features, followed by batch normalization and ReLU activation to introduce non-linearity and enhance training stability. A dropout layer with a rate of 0.5 prevents overfitting. The encoder outputs embeddings Z in a latent space of dimension $[\#nodes, 512]$, as depicted in Fig. 2. The decoder reconstructs the adjacency matrix $\hat{A}$ using the embeddings Z through the operation $\sigma(ZZ^T)$, where σ is the sigmoid function. The BCE loss for the GAE is defined as:

$$L_{\text{BCE}} = -\log(\hat{A}(z, i_{\text{pos}}, j_{\text{pos}}) + \epsilon) - \log(1 - \hat{A}(z, i_{\text{neg}}, j_{\text{neg}}) + \epsilon)$$

In this equation, $\hat{A}$ denotes the predicted adjacency matrix. The terms $i_{\text{pos}}, j_{\text{pos}}$ signify the indices of positive edges, while $i_{\text{neg}}, j_{\text{neg}}$ correspond to negative edges, obtained through negative sampling. To ensure stability during the computation of logarithms, we used a small constant $\epsilon = 1 \times 10^{-15}$. The essence of this loss metric lies in its ability to guide the GAE towards accurately reflecting the original graph structure. The model optimizes this loss, aiming to accurately reconstruct the graph's adjacency matrix. Upon minimizing this loss, the weights of the GAE's encoder are frozen, setting the stage for Phase 2's supervised training.

C. Phase 2: Supervised Learning

The primary objective of TraPPM is to predict training characteristics such as memory usage (MB) and training step time (ms). To achieve this, we employ a GNN-based regressor for prediction. The overview of supervised learning is shown in Fig. 3. The input G includes both actual values, represented by [mb, W], and static characteristics. The static features encompass the batch size B, the total number of nodes N_t, the total number of edges E_t, total MAC operations (MAC_t), total parameters (P_t), and total memory (M_t). The values N_t and E_t are directly extracted from G, while the values MAC_t, P_t, and M_t are obtained using the ONNX tool. Consequently, the static feature vector F_s has a length of 6. Supervised learning consists of three components.

GNN Component: It consists of two layers of the SAGEConv layer, and each SAGEConv layer is succeeded by a ReLU activation function and a dropout mechanism with a rate of 0.05.

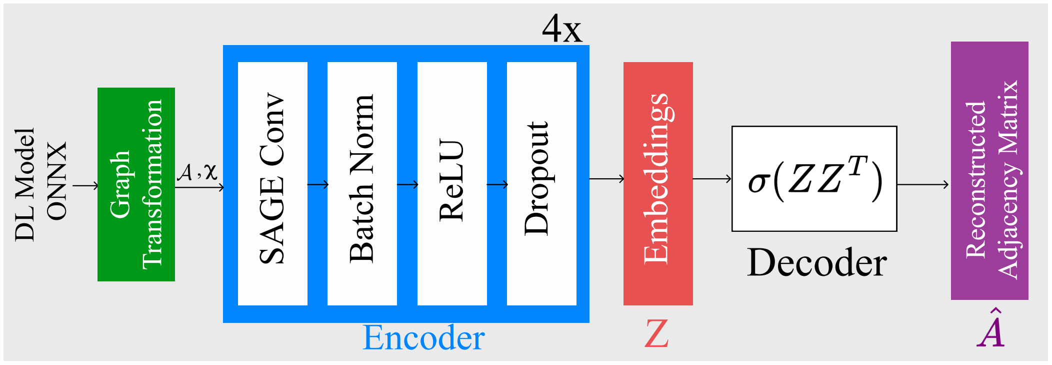


Fig. 2. Unsupervised Learning - Training Graph Auto Encoder to minimize reconstruction loss of unlabelled DL model graphs.

Feature Aggregation Component: The node features produced by the SAGEConv layer were aggregated using the sum reduce function, resulting in a [1, 512] dimension. Similarly, embeddings generated from the GAE were reduced to [1, 512] using another sum aggregator function. These two embeddings were then combined with a static feature F_s, forming a vector of dimensions [1, 1030], which was subsequently fed into the MLP component.

MLP Component: The concatenated feature vector is passed through two Fully connected (FC) layers. Both FC layers are succeeded by ReLU activations and dropout layers with a rate of 0.05. The processed features are passed through a final layer that produces a single output value.

In the forward pass, the model processes the input G, performs graph convolutions, aggregates node features, integrates it with static features and aggregated embeddings generated from GAE, and passes it through the FC layers to produce the final prediction. We employed the Mean Squared Error (MSE) as our loss function and utilized the Adam optimizer for the training phase. In the backward pass, the model updates the parameters θ, in both the GNN and MLP components. To individually predict memory usage (MB) and step time (ms), we have trained two distinct models: M1 for memory and M2 for step time and frozen their weights. A given input G is simultaneously processed by all two models (M1 and M2). Alongside G, each model also receives the static feature vector F_s and the aggregated embeddings generated by the GAE as we discussed earlier. The combined input helps these models produce accurate predictions on the training characteristics of a given DL model.

D. Evaluation Metrics

To assess the performance of our TraPPM model compared to the baseline models, we employ two widely used evaluation metrics: MAPE and Root Mean Square Error (RMSE). We chose MAPE because it measures the average percentage difference between the predicted and actual values. It allows us to assess the relative accuracy of the predictions as shown in Eq. (1). On the other hand, RMSE is used to measure the overall magnitude of prediction errors on the same scale as the predicted variable, providing a standardized and interpretable metric for assessing the performance of prediction models as shown in Eq. (2). By utilizing both MAPE and RMSE in our experiment, we thoroughly evaluate TraPPM's performance compared to the baseline models.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100 \tag{1}$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}, \tag{2}$$

V. Experiments and Results

A. Enviromental Setup

We used two hardware configurations for data collection: the first comprised an AMD EPYC 7402 processor with two sockets (24 cores per socket), 512 GB DDR4-3200 RAM, and a NVIDIA A100 GPU with 40 GB HBM; while the second utilized 2× Intel Xeon Gold 6148 CPUs (2× 20 cores at 2.4 GHz) and a NVIDIA V100 GPU with 16 GB HBM. However, we exclusively used the hardware equipped with the NVIDIA A100 GPU for the experiments. The experimental environment for developing TraPPM involved the utilization of several essential Python libraries. The important libraries used were PyTorch 2.0.0, torch-geometric 2.3.0, torch-cluster 1.6.1, ONNX 1.13.1, and torch-sparse 0.6.17. These libraries played a crucial role in implementing and training the TraPPM model. The experiments for training TraPPM and generating the dataset were conducted on the abovementioned system using CUDA 11.7.

B. Datasets

For our TraPPM experiment, we employed a dual-method approach, harnessing both unsupervised and supervised datasets. As we already discussed, unsupervised datasets are used for training GAE, and the supervised dataset is used to train the GNN-based regressor.

1) Unsupervised dataset: For the TraPPM experiment, we harnessed the Timm library [22] to generate a diverse unsupervised dataset comprising various CNN and transformer-based architectures. These models were exported in ONNX format and subsequently converted to PyG data, a process detailed in Section IV-A. The dataset includes 25,053 unlabeled DL models, spanning eleven distinct model families as outlined

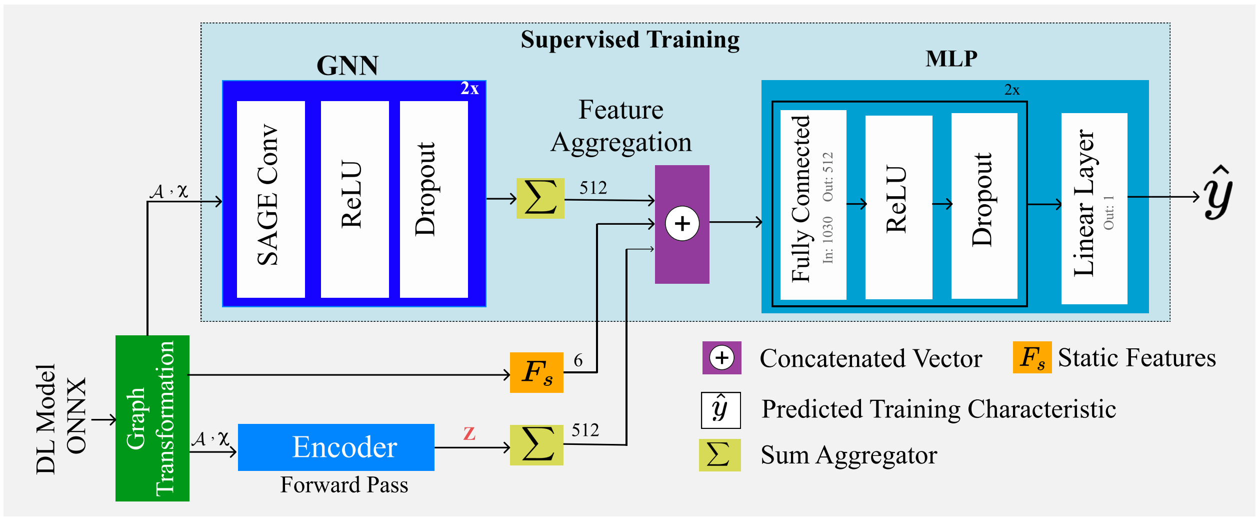


Fig. 3. Supervised Learning - Training a GNN regressor using MSE loss to minimize the actual y vs. predicted $\hat{y}$. We train three different models separately for predicting step time (ms), memory usage (MB), and power consumption (W).

in Table I. This extensive collection of models underpins our unsupervised learning approach, as elaborated in Section IV-B.

The dataset features a range of model variants within each family. For instance, the ConvNext family [23] encompasses variants like Base, Large, Small, and Tiny. The DenseNet family [24] includes DenseNet121, 161, 169, and 201. Other represented families are EfficientNet [25], MnasNet [26], MobileNet [27], PoolFormer [28], ResNet [29], Swin [30], VGG [31], along with additional models such as Visformer [32] and ViT [33]. This breadth ensures a comprehensive representation of current DL model architectures, facilitating robust unsupervised learning.

2) Supervised dataset: Our supervised dataset is subset of unsupervised dataset. The data collection was conducted using two GPUs: the NVIDIA A100 and the NVIDIA V100, as described in Section V-A. Specifically, using the A100 GPU, we collected a total of 7536 labeled DL models. Conversely, with the V100 GPU, we gathered 543 labeled DL models. For baseline model comparisons, we primarily utilized the labeled DL models from the A100 GPU. Meanwhile, the dataset collected from the V100 GPU was exclusively reserved for evaluating TraPPM's transfer learning capabilities. We again utilized the Timm library to generate DL models. However, instead of saving them to the ONNX format, we trained each model for 55 iterations, with the initial five iterations serving as a warm-up phase. We calculated the CUDA time during each iteration, representing the time taken to process a single iteration or step time in the training process. Our focus was primarily on step time, as it remains consistent during the training of the DL model, except for the initial few iterations that may exhibit variations due to warm-up effects. Therefore, we excluded the first five iterations when calculating the metrics. Additionally, we collected memory usage and power consumption data using the NVML[2] Python library. For each of the eleven different model families, we repeated this process, averaging the step time (ms), memory usage (MB), and power consumption (W). The results, along

TABLE I. TraPPM: Dataset Distribution

Family	Unsupervised	Supervised	
		A100	V100
DenseNet	838	466	27
EfficientNet	1370	566	44
MnasNet	7208	795	64
MobileNet	2449	1613	123
PoolFormer	601	377	36
ResNet	1805	821	56
Swin	787	421	36
VGG	6171	937	61
VisFormer	237	235	17
ConvNext	1530	439	27
ViT	2057	866	52
Total	**25053**	**7536**	**543**

with the corresponding ONNX model files, were saved. During converting these models to PyG data format, we appended the measured values into the graph data Y.

C. Training - Graph Auto Encoder

The first phase of the TraPPM experiment involves training the GAE, a key component of our TraPPM. Initially, we considered the Masked Graph Autoencoder technique, as presented in Hou's study [34]. This method masks random node features and attempts to reconstruct them, facilitating graph representation learning. However, our node features, largely sparse due to one-hot encoding as explained in Section IV-A, did not align well with this strategy. As a result, we turned to the classical GAE, which proved to be a better fit for our needs. The GAE model was developed using the PyG Library, the detailed model architecture explained in Section IV-B. For training, we employed the BCE loss function and utilized the Adam optimizer with a lr=5×10^{-4}, betas=(0.9, 0.999), eps=1×10^{-8}. To train the GAE, we utilized an unsupervised dataset, as described in Section V-B1, for a total of 400 epochs. Finally, we have achieved a BCE loss of 0.9291. The entire training process for the GAE took approximately 25.6 hours on a single A100 GPU. We employed t-SNE [35] to visualize the

[2]https://pypi.org/project/pynvml/

sum aggregated embeddings generated by the GAE, as shown in Fig. 4. The widespread distribution of the ResNet models is due to its numerous variants, distinguishing it from other models.

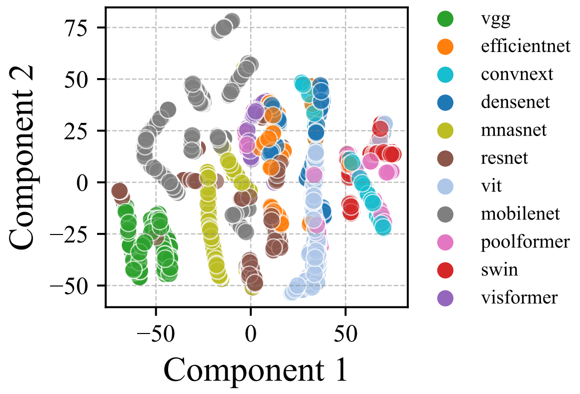


Fig. 4. t-SNE visualization of sum aggregated embeddings generated by the GAE.

D. Training - TraPPM

The core part of the experiment involves training the TraPPM model, with the detailed architecture explained in Section IV-C. We used the PyTorch library to create the TraPPM model. We used a labeled dataset collected from A100 GPU to train the model, as mentioned in Section V-B2. We partitioned our dataset according to a 70:30 ratio for each model family. Specifically, 70% of the data was used for training and 30% for testing. This split aligns with the standards established in previous research [10]. However, instead of a conventional split, we adopted a Monte Carlo validation approach. To ensure robustness and reliability in our results, we employed five distinct seeds: 1337, 1338, 1339, 1340, and 1341. By utilizing these seeds, we generated five different dataset splits and subsequently averaged the results to derive a more comprehensive performance evaluation. During the training process, we utilized the Adam optimizer with a lr=1×10^{-3}, betas=(0.9, 0.999), eps=1×10^{-8}. The training was performed over 100 epochs to fair comparison with baseline models. Training the TraPPM model for a single fold takes about 1 hour and 17 minutes for 100 epochs.

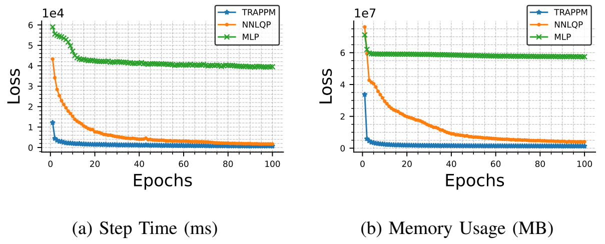
E. Baseline Models

In our evaluation, we compared TraPPM with three baseline models: Gboost, MLP, and the supervised GNN. Gboost served as a strong foundation for further development, while MLP was chosen for its wide usage in performance prediction [2]. Finally, we included the supervised GNN model introduced by Lu et al. [10], referred to as NNLQP. This model served as a reference for evaluating the performance of TraPPM in relation to a well-established supervised GNN approach.

1) Gradient boosting: To develop the GBoost model, we conducted training using the XGBoost [36] python library. The training process involved utilizing a supervised dataset that solely consisted of DL static features as input. To optimize its hyperparameters, we conducted a grid search. The hyperparameters explored during the grid search were estimators with values [500, 1000, 2000], lr with values [1×10^{-3}, 1×10^{-4}], max depth with values [10, 30, 50], subsample with values [0.5, 0.75, 1], and colsample bytree with values [0.5, 0.75, 1]. After performing the grid search, we identified the best hyperparameters as follows: colsample bytree: 1, lr: 1×10^{-3}, max depth: 50, estimators: 2000, subsample: 1.

2) MLP: We created a baseline MLP model that is similar to the TraPPM MLP component, with the only difference being that it accepts only static features as input during training. We trained the baseline MLP model using 100 epochs, utilizing the MSE loss function and the Adam optimizer with lr=1×10^{-3}, which is the same setting as the TraPPM supervised training.

3) NNLQP: It is important to note that a key distinction between the NNLQP model and the TraPPM model is that the NNLQP model is unable to utilize unsupervised datasets. It can only operate with supervised datasets. To ensure a fair comparison, we kept the model architecture unchanged, only adapting the node features to accommodate the TraPPM dataset as discussed in Section IV-A. We trained the model for 100 epochs using the Adam optimizer with lr=1×10^{-3}, following the same settings as the TraPPM model. The NNLQP model takes the graph representation G as input, generates embeddings, concatenates them with static features, and employs an MLP to predict performance.



(a) Step Time (ms)

(b) Memory Usage (MB)

Fig. 5. Epoch vs Loss plot comparing the convergence rates of TraPPM, NNLQP, and MLP. TraPPM showcases rapid convergence due to its ability to leverage unsupervised learning from unlabeled data, as trained over 100 epochs.

F. Baseline - Comparison

We assessed the performance of the TraPPM model by comparing it with baseline models. Both the TraPPM model and the baselines were trained using a supervised dataset of A100 GPU, with a specific focus on predicting step time (ms) and memory usage (MB). We trained the TraPPM, NNLQP, and MLP models for 100 epochs, repeating the process five times using different seeds as outlined in Section V-D. When we assessed the models for their capability to predict memory usage and step time, the epoch-versus-loss plot as shown in Fig. 5, revealed that the TraPPM model converges more rapidly compared to both NNLQP and MLP. This faster convergence can be attributed to TraPPM's ability to leverage unsupervised learning from unlabeled data.

TABLE II. Comparative Performance Analysis of Memory Usage (MB) Prediction: Averaged Results Over Five Distinct Splits. Results Highlight TraPPM's Enhanced Accuracy Compared with Baseline Models. The Lower the Values, the Higher the Accuracy

Family	**TraPPM**		**NNLQP**		**MLP**		**GBoost**	
	MAPE	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**
convnext	**4.95%**	**1005.71**	7.01%	1490.67	54.74%	8906.85	14.62%	3230.67
densenet	**3.52%**	**730.47**	8.29%	1675.17	66.44%	8317.23	14.64%	3113.40
efficientnet	**3.55%**	**537.84**	7.67%	1687.05	51.70%	11952.91	16.70%	3458.01
mnasnet	**4.53%**	**585.65**	6.75%	1804.18	94.42%	4908.33	14.72%	2756.27
mobilenet	**5.28%**	**633.74**	6.74%	1587.84	108.22%	5051.35	24.65%	2879.35
poolformer	**4.41%**	**1441.70**	6.96%	2027.24	76.10%	8951.67	15.04%	3332.57
resnet	**4.65%**	**658.40**	8.09%	1229.99	124.26%	7393.93	16.78%	2479.84
swin	**5.09%**	**774.91**	10.37%	1853.26	53.68%	8294.61	15.12%	2909.59
vgg	**10.48%**	2341.17	10.76%	**2271.89**	42.29%	7145.38	15.87%	3911.28
visformer	**3.92%**	**318.97**	9.49%	722.83	191.19%	8671.54	13.97%	1170.94
vit	**3.78%**	**985.22**	9.07%	2219.88	72.05%	8908.69	14.99%	3444.81

TABLE III. Comparative Performance Analysis of Step Time (ms) Prediction: Averaged Results Over Five Distinct Splits. Results Highlight TraPPM's Enhanced Accuracy Compared with Baseline Models

Family	**TraPPM**		**NNLQP**		**MLP**		**GBoost**	
	MAPE	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**
convnext	**8.09%**	**46.00**	9.50%	61.06	64.92%	354.59	16.15%	102.72
densenet	**6.69%**	**15.46**	18.41%	36.50	104.45%	155.09	14.08%	33.61
efficientnet	**6.81%**	**13.46**	9.45%	22.51	49.18%	118.56	15.36%	34.94
mnasnet	**7.98%**	**12.72**	18.59%	40.05	106.86%	80.87	14.49%	41.18
mobilenet	**9.20%**	**8.95**	14.66%	21.69	116.48%	52.97	25.79%	31.44
poolformer	**13.02%**	27.05	13.23%	**26.79**	166.75%	119.56	14.59%	32.51
resnet	**11.26%**	**16.20**	25.45%	36.02	192.95%	122.75	24.13%	46.33
swin	9.01%	35.18	**8.89%**	**33.66**	60.08%	263.86	15.68%	72.44
vgg	**10.74%**	**22.51**	13.20%	30.29	69.91%	83.70	15.89%	43.59
visformer	14.79%	17.88	18.61%	15.29	437.33%	287.67	**13.99%**	**14.85**
vit	**7.06%**	**40.13**	9.15%	83.41	105.84%	432.32	16.60%	146.39

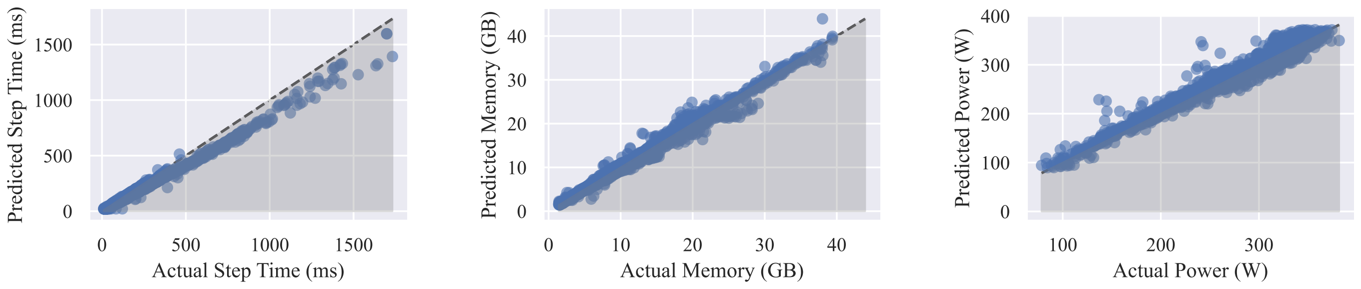


Fig. 6. Comparison of actual values with predictions from TraPPM on the test set. The model was trained for 100 epochs using a supervised dataset, with a split seed of 1337.

TABLE IV. Average Performance Comparison of TraPPM with Baseline Models

	Memory Usage (MB)		**Step Time (ms)**	
Model	**MAPE ↓**	**RMSE ↓**	**MAPE ↓**	**RMSE ↓**
TraPPM	**4.92%**	**910.34**	**9.51%**	**23.23**
NNLQP	8.29%	1688.18	14.47%	37.02
MLP	85.01%	8045.68	134.07%	188.36
GBoost	16.10%	2971.52	16.98%	54.54

G. Model Evaluation and Comparison

The performance evaluation of the TraPPM model against baseline models was conducted using a test dataset, with MAPE and RMSE as the key metrics, as detailed in Section IV-D. These metrics were computed for each model family individually to provide a comprehensive performance assessment. Lower MAPE and RMSE values indicate closer alignment of predictions with actual values.

Table II details the predictive accuracy for memory consumption, while Table III focuses on training step latency. The TraPPM model notably outperforms the baselines in both aspects. In memory consumption prediction, TraPPM achieves a significant relative improvement in MAPE of 40.6% over the NNLQP model, demonstrating its robustness. Similarly, for training step time predictions, TraPPM exhibits superior accuracy, with a relative MAPE improvement of 34.2% compared to NNLQP. Additionally, the aggregate performance of the TraPPM model, encompassing all tested model families, is summarized in Table IV. This table provides a holistic view of the TraPPM model's performance across various prediction tasks, reinforcing its overall efficacy in comparison to the baseline models.

H. Theoretical Insights into TraPPM's Semi-Supervised Learning Approach

The TraPPM model leverages a semi-supervised framework, integrating unsupervised learning for generating embeddings, which significantly enhances its predictive capabilities for step time and memory usage. This methodology stands in contrast to the purely supervised models like NNLQP, which rely exclusively on labeled data. Theoretically, the effectiveness of TraPPM is attributed to its ability to access a richer representation space, capturing latent structural features within the data through these unsupervised embeddings, features that remain elusive in a solely supervised paradigm.

From a mathematical perspective, the TraPPM model can be seen as operating within an expanded function space, F', compared to the more limited function space, F, accessible by conventional supervised learning. This expanded space F', achieved through the integration of unsupervised embeddings, encapsulates the original space F but extends further to incorporate additional dimensions reflecting data variance and underlying structure. The empirical benefits of this expansion are evidenced by the improved MAPE and RMSE metrics detailed in Tables II and III, with aggregate performance enhancements further demonstrated in Table IV.

To provide empirical validation of these theoretical and mathematical concepts, we include actual versus predicted plots in Fig. 6. These plots vividly illustrate the alignment between TraPPM's predictions and actual outcomes, thereby substantiating the model's proficiency in accurately forecasting training characteristics. They visually reinforce the theoretical and mathematical merits of the semi-supervised learning approach employed by TraPPM, highlighting its superiority in a variety of prediction tasks across multiple model families.

I. Ablation Study: Impact of Weight Initialization

In this ablation study, we examine the influence of weight initialization on the TraPPM model's performance, focusing on two distinct GAE configurations: one using pre-trained weights and another with randomly initialized weights. Both configurations are integrated with a GNN for regression tasks, as detailed in Section IV-C.

Empirical results indicate a marked difference in performance based on the initialization approach. The model with pre-trained weights demonstrates a notable decrease in MSE loss for training step time, starting from 1.26×10^4 and reaching 6.82×10^2 by the 100th epoch, signifying effective and efficient learning. In contrast, the randomly initialized model begins with a substantially higher initial MSE loss of approximately 5.43×10^{12}, which only marginally improves to 1.05×10^5 by the 2nd epoch and then stagnates, showing no further significant decrease in subsequent epochs. This pattern is consistently observed for both training step time and memory consumption prediction tasks.

Theoretically, this disparity can be attributed to the different starting points in the parameter space optimization

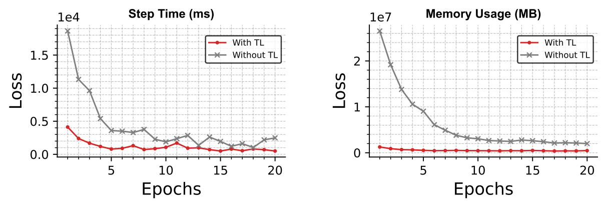


Fig. 7. Epoch vs. Loss plot demonstrating TraPPM's enhanced convergence through transfer learning.

landscape. Pre-trained weights provide a beneficial starting position, facilitating a more focused and stable gradient descent path ($\nabla_\theta L$). Conversely, random initialization tends to place the model in a less favorable starting point, often characterized by steeper initial gradients and a higher likelihood of getting trapped in local minima.

These observations underscore the critical role of initial weight settings in the performance of GAEs, especially in the context of the TraPPM model. The study highlights the substantial advantage of employing pre-trained weights for complex structured data tasks, as they significantly enhance the model's ability to learn efficiently and effectively.

VI. Discussion

A. TraPPM's Adaptability to Predicting Diverse Metrics

The TraPPM model demonstrates its versatility in metric prediction, such as power consumption, by leveraging the GAE's embeddings Z. These embeddings, derived from DL models, are crucial for extending prediction capabilities beyond standard metrics like memory usage and step time.

The GAE transforms high-dimensional inputs G into a comprehensive latent space $Z = f_{\text{GAE}}(G)$, forming the foundation for a GNN-based regression model as explained in the Section IV-B. For power consumption prediction, this GNN model, trained on the supervised dataset for 100 epochs (as outlined in Section V-D), aims to minimize the MSE between the predicted $\hat{y}_{\text{power}}$ and actual power consumption values y_{power}:

$$L_{\text{power}} = \frac{1}{n} \sum_{i=1}^{n} (y_{i,\text{power}} - \hat{y}_{i,\text{power}})^2$$

This method highlights TraPPM's adaptability in using the same set of GAE embeddings for diverse predictions. The effectiveness of this approach is validated by TraPPM's performance in power consumption prediction, achieving a MAPE of 5.01% and an RMSE of 17 W, thereby demonstrating the robustness and versatility of GAE embeddings in various predictive scenarios. Fig. 6, clearly depicts TraPPM's predictive accuracy, illustrating the close alignment between predicted and actual power consumption values.

B. Transfer Learning Capability of TraPPM

Transfer learning, crucial in DL when labeled data is scarce, was employed in TraPPM to address the limited labeled data for the V100 GPU (see Section V-B2). By initializing

```python
import trappm

config = { 'model': 'resnet101.onnx', 'batch_size': 32,
           'device': 'GPU:A100-SXM4-40GB:1' }
out = trappm.predict(config)
print("ms: {0}, MB: {1}, W: {2}".format(*out))
```

Fig. 8. A sample Python code using TraPPM to predict.

the V100 GPU training with weights W_{A100} from the A100 GPU-trained model, depicted in Fig. 7, we aimed to expedite convergence compared to starting from scratch.

In TraPPM, transfer learning theoretically embodies domain adaptation, transitioning the function $f_{\text{source}}(X; W_{A100})$ to $f_{\text{target}}(X; W)$. This strategy circumvents the initial generic feature learning phase, directly fine-tuning the model to the target dataset's specificities.

The effectiveness of this approach in TraPPM led to substantial relative improvements in prediction accuracy: approximately 55.03% in RMSE for Step Time and 48.76% for Memory usage. Table V details these enhancements, underscoring the robustness of transfer learning in optimizing TraPPM's predictive performance for different hardware contexts, especially where labeled data is limited.

TABLE V. COMPARISON OF METRICS WITH/WITHOUT TL

Label	Metric	With TL	Without TL
Step Time	MAPE (%)	**19.13**	28.24
	RMSE (ms)	**20.05**	44.59
Memory	MAPE (%)	**11.22**	28.49
	RMSE (MB)	**603.03**	1176.90

C. Ease of Use with TraPPM

We have developed a TraPPM as a Python library for predicting the step time, memory usage, and power consumption of DL models in the ONNX format. Users can effortlessly leverage TraPPM's performance prediction capabilities with just a few lines of code, as shown in Fig. 8.

D. Optimizing Cloud Costs and Resources with TraPPM

TraPPM is instrumental not only in Neural Architectural Search but also in datacenter job scheduling and cloud cost estimation. Its predictive capability enables efficient resource planning in datacenters and accurate estimation of cloud computing expenses. For example, using TraPPM to predict the training duration of an EfficientNet_b6 model, with an predicted step time of 350 ms over 2×10^5 iterations, yields a training time of around 19.44 hours. On a cloud platform with an A100 GPU costing 2.934 USD per hour, the total cost is approximately 57.04 USD. This application of TraPPM for cost prediction showcases its utility in optimizing computational resources and budgeting for cloud-based DL tasks.

VII. CONCLUSIONS

We present TraPPM, a novel framework that combines unsupervised GAE with a supervised GNN regressor to precisely predict DL model training characteristics without necessitating execution on target hardware, a significant departure from traditional approaches reliant solely on labeled datasets. TraPPM demonstrates exceptional predictive accuracy, achieving MAPEs of 4.92% for memory usage, 9.51% for step time, and 5.01% for power consumption, along with robust RMSE values. The release of our comprehensive dataset comprising 25,053 unlabelled DL graphs and 8,079 labeled DL graphs further enriches the field, providing a valuable resource for future research. TraPPM's innovative use of unlabeled data in a semi-supervised learning context marks a significant advancement in the DL performance prediction community.

ACKNOWLEDGMENT

This work has been done in the context of the MAELSTROM project, which has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955513. The JU receives support from the European Union's Horizon 2020 research and innovation program and the United Kingdom, Germany, Italy, Switzerland, Norway, and in Luxembourg by the Luxembourg National Research Fund (FNR) under contract number 15092355.

REFERENCES

[1] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating gpu memory consumption of deep learning models," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1342–1352.

[2] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 3873–3882.

[3] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021.

[4] Y. Gao, X. Gu, H. Zhang, H. Lin, and M. Yang, "Runtime performance prediction for deep learning models with graph neural network," in *ICSE '23*. IEEE/ACM, May 2023, the 45th International Conference on Software Engineering, Software Engineering in Practice (SEIP) Track.

[5] K. P. Selvam and M. Brorsson, "Dippm: a deep learning inference performance predictive model using graph neural networks," 2023.

[6] L. Bai, W. Ji, Q. Li, X. Yao, W. Xin, and W. Zhu, "Dnnabacus: Toward accurate computational cost prediction for deep neural networks," 2022.

[7] E. Gianniti, L. Zhang, and D. Ardagna, "Performance prediction of gpu-based deep learning applications," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 167–170.

[8] S. Kaufman, P. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, "A learned performance model for tensor processing units," in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 387–400.

[9] L. Dudziak, T. Chau, M. S. Abdelfattah, R. Lee, H. Kim, and N. D. Lane, "Brp-nas: Prediction-based nas using gcns," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[10] L. Liu, M. Shen, R. Gong, F. Yu, and H. Yang, "Nnlqp: A multi-platform neural network latency query and prediction system with an evolving database," in *51 International Conference on Parallel Processing - ICPP*, ser. ICPP '22. Association for Computing Machinery, 2022.

[11] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[12] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[13] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017.

[14] T. Kipf and M. Welling, "Variational graph auto-encoders," *NIPS Workshop on Bayesian Deep Learning*, 2016.

[15] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *International Conference on Learning Representations*, 2017.

[16] N. Bouhali, H. Ouarnoughi, S. Niar, and A. A. El Cadi, "Execution time modeling for cnn inference on embedded gpus," in *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, ser. DroneSE and RAPIDO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 59–65.

[17] M. Sponner, B. Waschneck, and A. Kumar, "Ai-driven performance modeling for ai inference workloads," *Electronics*, vol. 11, no. 15, 2022.

[18] Z. Lu, S. Rallapalli, K. Chan, S. Pu, and T. L. Porta, "Augur: Modeling the resource requirements of convnets on mobile devices," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 352–365, 2021.

[19] D. Velasco-Montero, J. Fernandez-Berni, R. Carmona-Galan, and A. Rodriguez-Vazquez, "Previous: A methodology for prediction of visual inference performance on iot devices," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9227–9240, 2020.

[20] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu, "*NeuralPower*: Predict and deploy energy-efficient convolutional neural networks," in *Proceedings of the Ninth Asian Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M.-L. Zhang and Y.-K. Noh, Eds., vol. 77. Yonsei University, Seoul, Republic of Korea: PMLR, 15–17 Nov 2017, pp. 622–637.

[21] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[22] R. Wightman, "Pytorch image models," https://github.com/rwightman/pytorch-image-models, 2019.

[23] Z. Liu, H. Mao, C. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2022, pp. 11 966–11 976. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR52688.2022.01167

[24] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2017, pp. 2261–2269. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.243

[25] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 6105–6114. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html

[26] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2019, pp. 2815–2823. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2019.00293

[27] A. Howard, M. Sandler, B. Chen, W. Wang, L. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for mobilenetv3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2019, pp. 1314–1324. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV.2019.00140

[28] W. Yu, C. Si, P. Zhou, M. Luo, Y. Zhou, J. Feng, S. Yan, and X. Wang, "Metaformer baselines for vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[30] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2021, pp. 9992–10 002. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.00986

[31] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[32] Z. Chen, L. Xie, J. Niu, X. Liu, L. Wei, and Q. Tian, "Visformer: The vision-friendly transformer," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2021, pp. 569–578. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.00063

[33] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," *ICLR*, 2021.

[34] Z. Hou, X. Liu, Y. Cen, Y. Dong, H. Yang, C. Wang, and J. Tang, "Graphmae: Self-supervised masked graph autoencoders," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 594–604.

[35] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html

[36] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785