TUTOR: **T**raining Ne**u**ral Ne**t**works Using Decisi**o**n **R**ules as Model Priors

Shayan Hassantabar, Prerit Terway, and Niraj K. Jha, *Fellow, IEEE*

Abstract—The human brain has the ability to carry out new tasks with limited experience. It utilizes prior learning experiences to adapt the solution strategy to new domains. On the other hand, deep neural networks (DNNs) generally need large amounts of data and computational resources for training. However, this requirement is not met in many settings. To address these challenges, we propose the TUTOR DNN synthesis framework. TUTOR targets non-image datasets. It synthesizes accurate DNN models with limited available data, and reduced memory and computational requirements. It consists of three sequential steps: (1) drawing synthetic data from the same probability distribution as the training data and labeling the synthetic data based on a set of rules extracted from the real dataset, (2) use of two training schemes that combine synthetic data and training data to learn DNN weights, and (3) employing a grow-and-prune synthesis paradigm to learn both the weights and the architecture of the DNN to reduce model size while ensuring its accuracy. We show that in comparison with fully-connected DNNs, on an average TUTOR reduces the need for data by $6.0\times$ (geometric mean), improves accuracy by 3.6%, and reduces the number of parameters (floating-point operations) by $4.7\times$ ($4.3\times$) (geometric mean). Thus, TUTOR is a less data-hungry, accurate, and efficient DNN synthesis framework.

Index Terms—Architecture synthesis; compact network; compression; deep neural network; synthetic data generation.

1 Introduction

An exceptional aspect of human intelligence is its ability to solve new problems, even with limited experience in the new domain, by leveraging prior experiences. A similar ability to learn efficiently under limited available data and computational resources is also a desirable attribute of artificial intelligent (AI) agents.

Over the last decade, deep neural networks (DNNs) have revolutionized a variety of application domains, such as computer vision, speech recognition, and robotic control. However, data scarcity is still a limiting factor in training DNNs for applications such as healthcare and cognitive modeling of human decisions [1], [2]. Many such applications offer just a few thousand data instances for training. Training of DNN models with such small datasets may lead to over-fitting, hence limit their applicability to real-world environments.

Some of the drawbacks of the current DNN training process are as follows:

- *Need to obtain and label large datasets*: Collecting a large number of data instances and manually labeling them is costly and time-consuming, even more so in domains where experts are needed to label the data instances.
- *Ignoring domain knowledge*: Training DNN architectures only on available training data does not take into account available expert domain knowledge or the set of rules that may have been derived for the domain. As a result, the process is not efficient and, in turn, exacerbates the need for large datasets.
- *Substantial redundancy*: Most DNN architectures need substantial storage, memory bandwidth, and computational resources for training. However, recent work shows that it is possible to significantly reduce the number of parameters and floating-point operations (FLOPs), with no loss in accuracy [3], [4].

There has been a recent spurt of interest in combining domain-specific information in the form of rule-based AI with modern statistical AI [5], [6]. In such works, rule-based AI can act as an inductive bias to enhance the learning efficacy for novel tasks. Imposing prior knowledge mitigates the need for a large amount of training data. However, extracting domain knowledge or creating a set of rules for a novel domain is often challenging as it requires specific expertise in the area. Nevertheless, domains such as natural language processing often have pre-defined rules that make such a combination attractive [7].

To address the above problems, we propose a new DNN training and synthesis framework called TUTOR, which is particularly suitable for tasks where limited data and computational resources are available for training. It relies on generation of synthetic data from the same probability distribution as the available training data, to enable the DNN to start its training from a better initialization point. TUTOR consists of three sequential stages: synthetic data generation and labeling, training schemes for incorporating synthetic and real data into the training process, and a grow-and-prune DNN synthesis to ensure model compactness as well as to enhance performance.

Data augmentation is an effective technique, often used for image classification to increase dataset size, help the model learn invariance, and regularize the model [8]. TUTOR targets non-image datasets. In TUTOR, the synthetic data are used to pre-train the DNN model. They are labeled by incorporating a set of rules extracted from real data. These rules are obtained from a random forest model trained on

This work was supported by NSF under Grant No. CNS-1907381. *Shayan Hassantabar, Prerit Terway, and Niraj K. Jha are with the Department of Electrical Engineering, Princeton University, Princeton, NJ, 08544 USA, e-mail:{seyedh, pterway,jha}@princeton.edu.*

real data. The intuition is that training the DNN with the synthetic data labeled in this fashion provides a suitable inductive bias to the DNN. It thus addresses the problem of limited availability of data.

Furthermore, to address the problem of network redundancy, we use the grow-and-prune synthesis paradigm [4] by leveraging three different operations: connection growth, neuron growth, and connection pruning. It allows the DNN architecture to grow neurons and connections to adapt to the prediction problem at hand. Subsequently, it prunes away the neurons and connections of little importance. In addition, it does not fix the number of layers in the architecture beforehand, enabling the architecture to be learned during the training process.

The major contributions of this article are as follows:

- TUTOR addresses two challenges in the design of predictive DNN models: lack of sufficent data and computational resources. It focuses on non-image datasets and addresses the lack of sufficient data by generating synthetic data from the same probability distribution as the training data using three different density estimation methods. By relying on decision rules generated from a random forest machine learning model, it obviates the need for domain experts to label the synthetic data.
- TUTOR uses a training flow to combine the synthetic and real data to train the DNN models and learn both the weights and the architecture by using a grow-and-prune synthesis paradigm. This addresses the problem of fixed network capacity, and also reduces memory and computational costs.
- Less data: TUTOR requires $6\times$ fewer data instances to match or exceed the classification accuracy of conventional fully-connected (FC) DNNs.
- Efficiency: TUTOR uses $4.7\times$ fewer parameters and $4.3\times$ fewer FLOPs relative to FC DNNs.
- Accuracy: TUTOR enhances classification accuracy by 3.6% over the conventional FC DNNs.
- Privacy enhancement: The synthetic data generated by TUTOR can be separately used to train a DNN, with little to no drop in accuracy, without the need for real data. This is useful in applications where privacy issues makes sharing of data infeasible.

The rest of the article is organized as follows. Section 2 covers related work. Section 3 provides some necessary background and the theoretical motivation behind this work. Then, in Section 4, we discuss the proposed TUTOR framework in detail. Section 5 presents evaluation results for TUTOR. In Section 6, we provide a short discussion on analogies between the human brain and the TUTOR training flow. Finally, Section 7 concludes the paper.

2 Related Work

In this section, we review several related works along four different dimensions. First, we discuss how logical AI can be combined with data-driven statistical AI. Second, we review work that combines decision trees with DNNs. Third, we discuss some of the recent work in efficient DNN synthesis. Finally, we discuss various data augmentation methods..

2.1 Combining Logical AI with Statistical AI

Recently, there have been several attempts to combine propositional and first-order logic (FOL) with the statistical techniques of machine learning. For instance, Markov logic network (MLN) [5] is based on probabilistic logic that combines Markov networks with FOL. The first-order knowledge base (KB) is viewed as imposing a set of hard constraints, where violation of a formula reduces the probability of the targeted application world to zero. An MLN softens these constraints by associating a weight with each FOL formula in the KB. These weights are proportional to the probability of the FOL formula being true. The incorporation of FOL enables reasoning about pre-existing domain knowledge. The Markov networks model uncertainty in reasoning. Nodes in the MLN define the FOL predicates. An arc exists between two nodes if the predicates appear in the formula. The probability of a world x being true is given by:

$$P(x) = \frac{1}{Z} \exp\left(\sum_{k \in \text{ground formulas}} w_k f_k(x)\right)$$

where Z is a normalization constant and w_k is the weight of the k^{th} formula given by $f_k(x)$. A formula in which the variables are replaced by constants is referred to as a ground formula. MLN learns the weights by maximizing the likelihood of a query given some evidence. Prior knowledge imposes an inductive bias that allows learning with small amounts of data. However, the main drawback of an MLN is the need for domain expertise to obtain the formulas, thereby limiting its usage.

The logic tensor network (LTN) [6] uses fuzzy logic to combine DNNs with FOL. Logical predicates represent concepts that are combined to form logical formulas. Fuzzy logic enables a formula to be partially true and take a value between 0 and 1. In LTN, a DNN learns the membership of an object in a concept. Each concept or formula is a point in a feature space and the DNN outputs a number between 0 and 1 to denote the degree of membership.

2.2 Combining Decision Trees with DNNs

Neural-backed decision trees (NBDTs) [9] combine decision trees with a DNN to enhance DNN explainability. Decision trees make higher-level decisions and the DNN makes low-level decisions. NBDTs also make intermediate decisions as opposed to the DNN that outputs decisions only at the last layer. NBDTs achieve performance close to that of the DNN while improving explainability. A neural decision tree [7] incorporates a nonlinear splitting criterion in its nodes by using a DNN to make the splitting decision. Deep neural decision forests [10] present an end-to-end learning framework based on convolutional neural networks (CNNs) and a stochastic differentiable decision tree for image classification applications. This approach uses back-propagation to learn the parameter to split at a node. Deep neural decision trees [11] use a soft binning function modeled by a DNN to split nodes into multiple leaves.

Initializing the weights of a DNN from a good starting point can also be used to improve its performance. Humbird et al. [12] use a decision tree model to construct the architecture and initialize the weights of the DNN. The decision

paths in a decision tree determines the DNN architecture. The tree acts as a warm-start to the training process.

2.3 Efficient DNN Synthesis

In this section, we discuss several different approaches for synthesizing efficient DNNs.

To reduce the need for data, a popular approach involves the use of transfer learning [13] to transfer weights learned for one task to initialize weights of a network for a similar task. This approach is based on the fact that some of the learned features are similar across related tasks.

A common approach for reducing computational cost is the use of efficient building blocks. For example, MobileNetV2 [14] uses inverted residual blocks to reduce model size and FLOPs. ShuffleNet-v2 [15] uses depth-wise separable convolutions and channel-shuffling operations to ensure model compactness. Spatial convolution is one of the most expensive operations in CNN architectures. To reduce its computational cost, Shift [16] uses shift-based modules that combine shifts and point-wise convolutions that significantly reduce computational cost and storage needs. FBNet-v2 uses differentiable neural architecture search to automatically generate compact architectures. Efficient performance predictors, e.g., for accuracy, latency, and energy, are also used to accelerate the DNN search process [17], [18].

DNN compression methods can be used to remove redundancy in DNN models. Han et al. [19] proposed a pruning methodology to remove redundancy from large CNN architectures, such as AlexNet and VGG. Pruning methods are also effective on recurrent neural networks [20]. Combining network growth with pruning enables a sparser, yet more accurate, architecture. Dai et al. [4], [21] use the grow-and-prune synthesis paradigm to generate efficient CNNs and long short-term memories. SCANN [22] uses feature dimensionality reduction alongside grow-and-prune synthesis to generate very compact models for deployment on edge devices and Internet-of-Things (IoT) sensors.

Orthogonal to the above works, low-bit quantization of DNN weights can also be used to reduce FLOPs. A ternary weight representation is used in [23] to significantly reduce computation and memory costs of ResNet-56, with a limited reduction in accuracy.

2.4 Data Augmentation for Enhancing Performance

Learning with sparse data is challenging as machine learning algorithms, especially DNNs, face the overfitting problem in a small-data regime. To overcome this problem, several techniques have been developed to augment the dataset. Data augmentation for images enhances the accuracy of image classification and reinforcement learning tasks. AutoAugment [24] uses reinforcement learning to obtain a policy for optimal image transformation to improve performance on various computer vision tasks. The optimal transformation is dependent on the dataset. UniformAugment [8] augments images by uniformly sampling from the continuous space of augmentation transformations, hence, avoiding the costly search process for finding augmentation policies. Laskin et al. [25] use two new augmentation schemes of random translation as well as random amplitude scaling alongside standard augmentation techniques (crop, cutout, flip, rotate, etc.). They use these augmentation schemes to achieve the state-of-the-art performance for reinforcement learning tasks. Antoniou et al. [26] use generative adversarial networks to augment the images for few-shot learning in the low-data regime tasks.

3 Background

In this section, we discuss background material that motivates the TUTOR framework. First, we discuss different probability density estimation methods. We use such methods in the TUTOR synthetic data generation module. Next, we discuss the Hierarchical Bayes and Empirical Bayes concepts. We also discuss linking of gradient-based learning with hierarchical Bayes. These methods motivate our training schemes (explained in Section 4.4) for the way we combine synthetic and real data to learn the parameters of the DNN models.

3.1 Probability Density Estimation

The probability density function (pdf) is a fundamental concept in statistics. It is used to compute various probabilities associated with a random variable. Probability density estimation deals with estimating the density of function f with $\hat{f}$ given random samples $x_1, \ldots, x_n \sim f$.

Predominantly, probability density estimation can be categorized into two groups: parametric and non-parametric. In parametric density estimation, pdf f is assumed to be a member of a parametric family. Hence, density estimation is transformed into finding estimates of the various parameters of the parametric family. For example, in the case of a normal distribution, density estimation aims to find the mean μ and standard deviation σ. In general, the classic parametric mixture of C Gaussian models has a pdf of the form:

$$p(x_i|\Theta) = \Sigma_{c=1}^{C} p(x_i|z_i = c, \Theta) p(z_i = c|\Theta)$$

where each component c is a d-dimensional multivariate Gaussian distribution with mean vector μ_c and covariance matrix Σ_c in the form of:

$$p(x_i|z_i = c, \Theta) = \left(\frac{1}{2\pi}\right)^{(d/2)} |\Sigma_c|^{-1/2} \exp\left(-\frac{1}{2}(x_i - \mu_c)^T \Sigma_c^{-1} (x_i - \mu_c)\right)$$

where $|\Sigma_c|$ is the determinant of the covariance matrix. In addition, the mixture probabilities are modeled as categorical:

$$p(z_i = c|\Theta) = \theta_c$$

A non-parametric density estimation scheme can generate any possible pdf. It does not make any distributional assumptions and covers a broad class of functions. Kernel density estimation is a non-parametric density estimation approach that approximates a probability distribution as a sum of many *kernel* functions. Each kernel function K satisfies the following property:

$$\int_{-\infty}^{\infty} K(y)dy = 1 \quad , \quad K(y) > 0 \quad \forall y$$

There may be various options for choosing function K, such as the pdf of the normal distribution. Another important parameter is the kernel *bandwidth*, h, that rescales the kernel function. Kernel density estimation can be formulated as:

$$\hat{f}(y) = \frac{1}{nh}\sum_{i=1}^{n} K(\frac{y - x_i}{h})$$

where n is the number of samples and x_i is the i^{th} sample.

3.2 Hierarchical Bayes and Empirical Bayes

Creating sub-models can be a way to model the distribution of data with a hierarchical structure. Each sub-model has its own parameters that are interlinked to model the complete distribution. Hierarchical Bayes models the data distribution at a prior level by using Bayes inference that links the parameters and propagates uncertainties among the sub-models. It uses the Bayesian network assumption: variables are conditionally independent of their non-descendants, given thair parents. The equation below shows hierarchical Bayes modeling of the distribution of data X [27].

$$X \longrightarrow \theta \longrightarrow \theta_1 \longrightarrow \theta_2 \longrightarrow \ldots \longrightarrow \theta_t$$

where θ is the parameter and $\theta_1, \theta_2, \ldots, \theta_t$ are the hyper-parameters. Based on the Bayes assumption, using only the following conditionals specifies the complete model: $[X|\theta]$, $[\theta|\theta_1]$, $[\theta_1|\theta_2]$, ..., $[\theta_{t-1}|\theta_t]$. By exploiting the inherent structure of the distribution, hierarchical modeling reduces the number of parameters needed in its modeling. Hence, it is less prone to overfitting. Empirical Bayes [27] is similar to Hierarchical Bayes. However, Empirical Bayes determines the hyperparameters by using the data instead of the hyper-prior distribution.

The two training schemes of TUTOR (explained in Section 4.4) model the learning of the parameters of the DNN with Hierarchical Bayes. In addition, similar to Empirical Bayes, TUTOR estimates the hyperparameters by using the available data. In this process, it first uses probability density estimation to estimate the distribution of the training dataset. It uses the sum of the log-probabilities of the data instances in the validation set to determine the hyperparameter λ of the probability density estimation function. TUTOR generates synthetic data by sampling from this function. In the training schemes, it learns parameters θ of the DNN model by using the synthetic data to pre-train the weights of the model. Therefore, we can obtain a Hierarchical Bayes model of the parameters as follows:

$$X \longrightarrow \lambda \longrightarrow \theta$$

The posterior distribution can be written as:

$$p(\lambda, \theta \mid X) \propto p(X \mid \theta, \lambda) p(\theta \mid \lambda) p(\lambda)$$

where X denotes the combined training and validation set, θ the parameter of the model after training with synthetic data, and λ the hyperparameter of the probability density estimation function.

3.3 Linking Gradient-based Learning with Synthetic Data and Hierarchical Bayes

Gradient based learning combined with a Bayesian network enables transfer of knowledge across various domains in a sample-efficient manner. Model-agnostic meta-learning (MAML) [28] trains a model across a variety of tasks and fine-tunes it using gradient-descent on a new learning task. As a result, MAML uses the other related tasks to initialize the weights of the DNN instead of using random initialization. Grant et. al. [29] cast MAML as Empirical Bayes using a hierarchical probabilistic model. It learns a prior that is adapted to new tasks to augment gradient-based meta-learning. Learning on one task influences the parameters of another task through a meta-level parameter that determines the task-specific parameters.

In a similar fashion to MAML, TUTOR generates synthetic data and uses them to learn parameters θ of the model. Then, it learns parameters ϕ of the DNN from the initialization with θ followed by gradient descent using the real training data.

4 Synthesis Methodology

In this section, we discuss the TUTOR DNN synthesis framework in detail. First, we give a high-level overview of the TUTOR DNN training methodology. We then zoom into each part of the methodology, including the three synthetic data generation methods, labeling of the synthetic dataset, and the two training schemes that combine the real and synthetic data. We then explain our grow-and-prune synthesis algorithm. Finally, we discuss the use of synthetic data alone as a proxy for real data to train DNNs, without any loss in accuracy, for the sake of privacy enhancement.

4.1 Framework Overview

We illustrate the proposed TUTOR framework in Fig. 1. It consists of three main parts: (1) synthetic data generation and labeling with decision rules, (2) two pre-training schemes that combine synthetic data with real data, and (3) grow-and-prune synthesis to decrease network redundancy and computational cost for inference while improving performance.

The synthetic data generation module uses the training and validation sets as inputs to generate synthetic data instances. We formulate this step as an optimization problem. The objective is to find the best set of hyperparameters (λ) of the probability density estimation function computed on the training data ($\hat{f}(X_{train}|\lambda)$). To this end, we use the sum of the log probabilities of instances in the validation set ($\texttt{score}(X_{validation})$) as a measure to compare different functions. We sample a pre-defined number of instances ($count$) from the density estimation function $\hat{f}$ with an optimal set of hyperparameters λ^* to generate the synthetic data:

$$\lambda^* = \arg\max_{\lambda} \left(\hat{f}(X_{train}|\lambda).\texttt{score}(X_{validation}) \right)$$

$$X_{syn} = \hat{f}(X_{train}|\lambda^*).\texttt{sample}(count)$$

Section 4.2 explains the process of synthetic data generation.

To label the synthetic dataset, we use decision rules to obviate the need for the expensive and laborious process of labeling by an expert. To do so, we use a random forest model trained on the data used as a KB. Section 4.3 explains this process.

In the two training Schemes A and B, TUTOR uses the synthetic data alongside the original training and validation data. As explained in Section 3.3, these schemes learn the DNN weights by combining gradient based learning with Hierarchical Bayes modeling. TUTOR uses synthetic data to learn parameters θ of the DNN model to act as a better initialization point for the next training step. Then, it learns parameters ϕ of the DNN by performing gradient-descent training with real training data starting from initialization point θ. Section 4.4 explains these two training schemes.

Finally, we perform grow-and-prune synthesis [4], [22] on the FC DNN models that are the outputs of Schemes A and B. This step uses gradient-based learning to obtain both the final architecture and the final DNN parameters. Grow-and-prune synthesis uses three different architecture-changing operations that are explained in Section 4.5.

4.2 Synthetic Data Generation

In this section, we discuss how TUTOR uses three different probability density estimation models to generate synthetic data. We use both parametric and non-parametric density estimation approaches (explained in Section 3.1). The first approach involves the use of multivariate normal distribution to estimate the probability distribution of training data. This is an example of parametric density estimation. We also use an approach based on kernel density estimation that resides on the other side of the spectrum, being a non-parametric density estimation approach. Furthermore, we use Gaussian mixture models (GMMs) to estimate the training data distribution. This approach can be considered a trade-off between the other two approaches. It is based on the sum of a limited number of multivariate normal distributions.

4.2.1 Multivariate normal distribution (MND)

Next, we discuss modeling data with a multivariate normal distribution. The parameters of this distribution are computed based on the training data. The sample mean vector $\hat{\mu}$ can be estimated as:

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

where N is the total number of training data instances and x_i is the d-dimensional vector that depicts the i^{th} training instance.

To estimate the $d \times d$ covariance matrix of the multivariate normal distribution, the maximum likelihood estimator of the covariance matrix is given by:

$$\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{\mu})(x_i - \hat{\mu})^T$$

As a result, a multivariate normal distribution with the d-dimensional mean vector $\hat{\mu}$ and $d \times d$ covariance matrix $\hat{\Sigma}$ estimates the density. Therefore, the joint density function of a random vector X based on this density function is given as:

$$p_X(x|\hat{\mu},\hat{\Sigma}) = \left(\tfrac{1}{2\pi}\right)^{d/2} |\hat{\Sigma}|^{-1/2} \exp\left(-\tfrac{1}{2}(x-\hat{\mu})^T\hat{\Sigma}^{-1}(x-\hat{\mu})\right)$$

where $|\hat{\Sigma}|$ is the determinant of matrix $\hat{\Sigma}$. The synthetic data are generated by sampling the distribution:

$$X \sim \mathcal{N}(\hat{\mu},\hat{\Sigma})$$

4.2.2 Gaussian mixture model (GMM)

The second approach involves the use of a multi-dimensional GMM to model the training data distribution. The data are modeled as a mixture of C Gaussian models that lead to a pdf of the form:

$$p_X(x|\Theta) = \Sigma_{c=1}^{C} p_{X|Z}(x|z=c,\Theta)p_Z(z=c|\Theta)$$

where Θ depicts the parameters of the Gaussian model, X the observed variables, and Z the hidden state variables that indicate Gaussian model assignment, with the prior probability of each model c being:

$$p_Z(z=c) = \theta_c$$

As a result, the GMM can simply be written as:

$$p_{X|C}(x) = \sum_{c=1}^{C} \mathcal{N}(x|\mu_c,\Sigma_c)\,\pi_c$$

Here, μ_c represents the mean vector, Σ_c the covariance matrix, and π_c the weight of GMM component c, with a total of C components.

The drawback of this approach lies in its learning complexity. To address this issue, we use the iterative Expectation-Maximization (EM) algorithm to determine the GMM parameters. The EM algorithm solves this optimization problem in two steps:

- E-step: given the current parameters θ_i and observations in data $\mathcal{D}$, estimate the values of z_i.
- M-step: with the new estimate of z_i, find the new parameters, θ_{i+1}.

As a result, in the E-$step$, for each training data instance x_i, the most likely cluster z_i is obtained. After cluster assignment is done in the E-$step$, in the M-$step$ the cluster parameters, i.e., cluster mean μ_c and covariance matrix Σ_c, are obtained based on current cluster assignments. This procedure is repeated until convergence. There are several choices for the covariance matrix *(diagonal, spherical, full)* that can be used to control the degree of cluster freedom. In our setup, we specify covariance to be of the *full* type to enable modeling of clusters with arbitrary orientation.

To avoid overfitting on the training data, the likelihood of validation data is obtained by varying the number of mixtures, C. The sum of log probabilities of instances in the validation set is used to evaluate the quality of the model. The number of components that maximizes this criterion is chosen as the optimal number of components required to model the distribution:

$$C^* = \underset{C}{arg\,max}\left(p_{X|C}(x).\texttt{score}(X_{validation})\right)$$

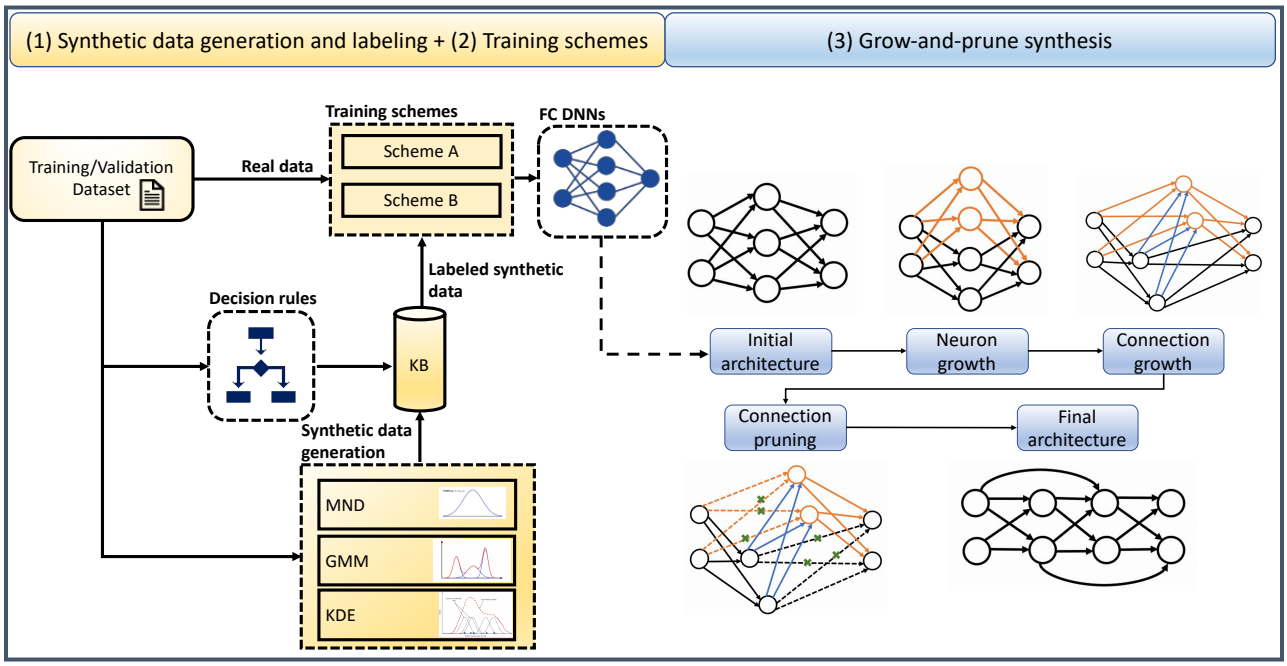

Fig. 1. Block diagram of the TUTOR DNN synthesis framework: (1) synthetic data generation and labeling, (2) two training schemes, and (3) grow-and-prune synthesis.

The GMM parameters thus obtained are then used to generate synthetic data by sampling a pre-defined number of instances ($count$) from this distribution.

$$X_{syn} = p_{X|C^*}(x).\texttt{sample}(count)$$

4.2.3 *Kernel density estimation (KDE)*

Compared to the parametric mixture models, non-parametric mixture models assign one component to every training example. In general, this can be formulated as:

$$p(x|\Theta) = \sum_{i=1}^{N} p_Z(z_i = c|\Theta) p_{X|Z}(x|z = c, \Theta)$$

In our setup, we use the pdf of the normal distribution as the kernel function. An important hyperparameter of the KDE model is kernel bandwidth h that controls the smoothness of the estimated function. The bandwidth has a large impact on the bias-variance trade-off of the estimator. Essentially, high-variance models estimate the training data well, but suffer from overfitting on the noisy training data. On the other hand, high-bias estimators are simpler models, but suffer from underfitting on the training data and fail to capture important regularities. In the KDE models, small bias is achieved when $h \to 0$ and small variance when $h \to \infty$. Therefore, the choice of the h value directly impacts function performance on unseen data. Hence, as in the case of finding the optimal number of components in a GMM model, we use the validation set to find the optimal bandwidth value h^*. We use the sum of log probabilities of the instances in the validation set to evaluate the KDE probability estimator:

$$h^* = \underset{h}{arg\,max} \left(p_{X|h}(x).\texttt{score}(X_{validation}) \right)$$

After obtaining all the parameters of the KDE estimator, we sample a pre-defined number of instances ($count$) from this distribution to generate the synthetic data.

$$X_{syn} = p_{X|h^*}(x).\texttt{sample}(count)$$

4.3 Labeling the Synthetic Data

In the case of supervised machine learning, domain experts are generally asked to label the data. However, this process can be very expensive and time-consuming. Furthermore, in many settings, researchers may not have access to domain experts. To obviate the need for domain experts, we instead use decision rules extracted from a random forest model trained on the real training data.

In a decision tree, each path starting from the root and ending at the leaves can be considered a *rule*. Hence, as shown in Fig. 1, we consider labeling the synthetic data with rules derived from the random forest model that performs the best on the validation set. We evaluate various random forest models using different splitting criteria (such as Gini index or entropy) and different depth limitations on the decision trees.

4.4 TUTOR Training Schemes

In this section, we explain how TUTOR uses the labeled synthetic data to obtain a prior on the weights of the DNN architecture. We introduce two schemes, termed Schemes A and B, to combine the synthetic and real data to train two FC DNN models.

4.4.1 *Scheme A*

Scheme A combines gradient-based learning with the Hierarchical Bayes approach to estimate the parameters of the model in two steps (as explained in Section 3.3). Algorithm 1 summarizes the process of training DNN weights using synthetic and real data. Scheme A uses the data generated by the three different synthetic data generation methods to

pre-train the network architecture. Pre-training is followed by use of the real training dataset to fine-tune and train the weights of the network. Scheme A outputs the model that performs the best on the validation set. Then, we use the test set for evaluation.

Algorithm 1 Scheme A

Input: (X,y): Training data; *Methods*: the three methods for synthetic data generation; *Arch*: DNN architecture; KB: knowledge-base (random forest model)
  for *method* in *Methods* **do**
    X_{Syn}: synthetic data based on *method*
    y_{syn} = KB (x_{syn})
    Pre-train *Arch* with (X_{Syn}, y_{Syn})
    Train *Arch* with (X, y)
    $Model_{method}$ = *Arch* with the learned weights
  end for
  $Model_A$: best model as evaluated on the validation set
  $testAcc$: Evaluate $Model_A$ on test data
Output: $Model_A$ and $testAcc$

4.4.2 Scheme B

Algorithm 2 summarizes Scheme B. First, we train two instances of the same DNN architecture. We train the first DNN on the training dataset. This network is responsible for learning the data. On the other hand, we train the second DNN on the synthetic data. Hence, this network is responsible for learning the KB, i.e., the random forest model. Next, we obtain another architecture by concatenating the outputs of these two networks and adding two FC layers followed by an output layer. We then train this architecture on the training dataset and evaluate it on the validation set.

The two added FC layers are responsible for learning the importance of the output from each of the two DNN components in making the final prediction. In this aspect, Scheme B takes inspiration from an MLN [5]. An MLN associates a weight with each FOL formula in a KB to depict their relative importance. Furthermore, an MLN uses Markov networks to model the uncertainty in reasoning based on the KB. Similarly, Scheme B uses the added two FC layers to learn the relative importance of prediction based on training data and based on synthetic data that represent the KB, in the final prediction. Finally, we record the best model and evaluate it on the test set.

4.5 Grow-and-prune Synthesis

In this section, we discuss the grow-and-prune synthesis step. This approach, first proposed in [4], [22], allows both the architecture and weights to be learned during the training process, enhancing both model accuracy and compactness. We apply the grow-and-prune synthesis to models A and B that are the outputs of Algorithms 1 and 2, respectively.

The approach used here allows the depth of the DNN to change during the training process. This is due to allowing the neurons to feed their output to any neuron activated after them. As a result, DNN depth depends on how the neurons are connected in the architecture and can be changed during the grow-and-prune synthesis process. Our approach applies

Algorithm 2 Scheme B

Input: (X,y): Training data; *Methods*: the three methods for synthetic data generation; *Arch*: DNN architecture; KB: knowledge-base (random forest model)
  for *method* in *Methods* **do**
    X_{Syn}: synthetic data based on *method*
    y_{syn} = KB (x_{syn})
    $Model_1$: Train *Arch* on (X_{Syn}, y_{Syn})
    $Model_2$: Train *Arch* using (X,y)
    CombinedNet: `Concat` (Output of *Models* 1 and 2) + 2 FC layers + an output layer
    fine-tune *CombinedNet* on the training set
    Evaluate on the validation set
  end for
  $Model_B$: best *CombinedNet* on the validation set
  $testAcc$: Evaluate $Model_B$ on test data
Output: $Model_B$ and $testAcc$

three different architecture-changing operations for a predefined number of iterations (set to five in our experiments) . After each change, the DNN architecture is trained for a few epochs (set to 20 in our implementation) and its performance evaluated on the validation set. Finally, we choose the highest performing architecture on the validation set.

The three operations that are used in our grow-and-prune synthesis step are as follows:

Connection growth: Connection growth activates the dormant connections in the network, with their weights set to 0 initially. The two main approaches we use for connection growth are:

- **Gradient-based growth**: In this approach, we choose the added connections based on their effect on the loss function $\mathcal{L}$. Each weight matrix W has a corresponding binary mask $Mask$ that has the same size. The binary mask is used to disregard the inactive connections with zero-valued weights. Algorithm 3 shows the process of gradient-based growth. The goal of this process is to identify the dormant connections that are most effective in reducing the loss function value. For each mini batch, we extract the gradients for all the weight matrices $W.grad$ and accumulate them over a training epoch. An inactive connection is activated if its gradient is above a certain percentile threshold of the gradient magnitude of its associated layer matrix.
- **Full growth**: This approach restores all the dormant connections in the network to make the DNN fully connected.

Connection pruning: The connection pruning operation is based on the magnitude of the connection. We remove a connection w if and only if its magnitude is smaller than a certain percentile threshold of the weight magnitude of its associated layer matrix. Algorithm 4 shows this process.

Neuron growth: This step duplicates the existing neurons in the architecture, to add neurons to the network, and increases network size. Algorithm 5 explains the neuron growth process. In this process, we select the neuron with the highest activation function (neuron i) for duplication. After adding the new neuron (neuron j) to the network, we

Algorithm 3 Connection growth algorithm

Input: $W \in R^{M \times N}$: weight matrix of dimension $M \times N$; $Mask \in R^{M \times N}$: weight mask of the same dimension as the weight matrix; Network P; $W.grad$: gradient of the weight matrix (of dimension $M \times N$); data D; α: growth ratio
  if full growth **then**
    $Mask_{[1:M,1:N]} = 1$
  else if gradient-based growth **then**
    Forward propagation of data D through network P and then back propagation
    Accumulation of $W.grad$ for one training epoch
    $t = (\alpha \times MN)^{th}$ largest element in the $|W.grad|$ matrix
    for all $w.grad_{ij}$ **do**
      if $|w.grad_{ij}| > t$ **then**
        $Mask_{ij} = 1$
      end if
    end for
  end if
  $W = W \otimes Mask$
Output: Modified weight matrix W and mask matrix $Mask$

Algorithm 4 Connection pruning algorithm

Input: Weight matrix $W \in R^{M \times N}$; mask matrix $Mask$ of the same dimension as the weight matrix; α: pruning ratio
  $t = (\alpha \times MN)^{th}$ largest element in $|W|$
  for all w_{ij} **do**
    if $|w_{ij}| < t$ **then**
      $Mask_{ij} = 0$
    end if
  end for
  $W = W \otimes Mask$
Output: Modified weight matrix W and mask matrix $Mask$

use the original neuron i to set the new values for mask and weight matrices. In addition, to break the symmetry, random noise is added to the weights of all the connections related to the newly added neuron.

Algorithm 5 Neuron growth algorithm

Input: Network P; weight matrix $W \in R^{M \times N}$; mask matrix $Mask$ of the same dimension as the weight matrix; data D; candidate neuron n_j to be added; array A of activation values for all hidden neurons
  forward propagation through P using data D
  $i = argmax\ (A)$
  $Mask_{[j,1:N]} = Mask_{[i,1:N]}$
  $Mask_{[1:M,j]} = Mask_{[1:M,i]}$
  $W_{[j,1:N]} = W_{[i,1:N]} + noise$
  $W_{[1:M,j]} = W_{[1:M,i]} + noise$
Output: Modified weight matrix W and mask matrix $Mask$

We apply connection pruning after neuron growth and connection growth in each iteration. We apply these three operations for a pre-defined number of iterations. Finally, we select the best performing architecture on the validation set.

4.6 TUTOR Application: Using Synthetic Data Alone

It is common to assume a trusted curator gathers a dataset with sensitive information from a large number of individuals. Such a dataset can be used to train predictive machine learning models for specific tasks in a related domain. A dataset can contain *micro-data*, revealing information about individuals from whom the data are collected. For example, many healthcare datasets contain legally protected information, such as health histories of patients. Hence, there is a privacy risk in publishing or sharing such datasets. Narayanan et al. [30] have shown that an adversary with a small amount of background knowledge about an individual can use it to identify the individual's record in an anonymized dataset, with a high probability. This means that the adversary can learn private information about the individual.

In addition, a study [31] shows that using 1990 U.S. Census summary data, with a high probability, 87% of the U.S. population can be identified with only three features: ZIP Code, gender, and date of birth.

As a result, free flow of data between various organizations risks privacy loss. However, data sharing is necessary for building predictive machine learning models. To address this issue, we could share synthetic data, drawn from the same distribution as the real data, with the other parties [32]. The synthetic data generation module in TUTOR can be used to generate such a dataset. Furthermore, the dataset can be labeled using the most accurate DNN model (on the validation set) synthesized by the TUTOR framework. In Section 5.3, we show that DNN models trained on the synthetic dataset incur little to no drop in accuracy compared to DNNs trained on real data.

5 Experimental Results

In this section, we evaluate the performance of the TUTOR synthesis methodology on eight datasets. Table 1 shows their characteristics. These datasets are publicly available from the UCI machine learning repository [33], Kaggle classification datasets [34], [35], and Statlog collection [36]. Since TUTOR focuses on non-image datasets, none of these datasets are image-based. In addition, TUTOR focuses on settings where limited data are available. Hence, the chosen datasets are of small to medium size. First, we present results of applying TUTOR DNN synthesis to these datasets. In this case, we use the available data to synthesize the most accurate DNN models. We also evaluate the contribution of each step of the TUTOR framework. Next, we evaluate the ability of TUTOR to reduce the need for large amounts of data. In this context, we evaluate the performance of TUTOR-generated models when only a part of the training and validation data are available. Subsequently, we discuss an application of TUTOR in which only the synthetic data are shared with other parties, not the real data. We compare DNN models synthesized for each case.

5.1 TUTOR Performance Evaluation

In this section, we evaluate the performance of the TUTOR framework on eight datasets. We present results for various synthetic data generation methods and training schemes. To

TABLE 1
Characteristics of the datasets.

Dataset	Training Set	Validation Set	Test Set	Features	Classes
Sensorless Drive Diagnosis (SenDrive) [33]	40509	9000	9000	48	11
MIT-BIH Arrhythmia Dataset (BIH-arrhythmia) [35]	22628	7543	7543	187	2
PTB Diagnostic ECG Database (PTB-ECG) [34]	8730	2910	2910	187	2
Smartphone-based Human Activity Recognition (SHAR) [33]	6213	1554	3162	561	12
Epileptic Seizure Dataset (EpiSeizure) [33]	6560	1620	3320	178	2
Human Activity Recognition Dataset (HAR) [33]	5881	1471	2947	561	6
Statlog DNA (DNA) [36]	1400	600	1186	180	3
Breast Cancer Wisconsin Diagnostic Dataset (Breast Cancer) [33]	404	150	160	30	2

compare the different synthetic data generation methods, we use t-distributed Stochastic Neighbor Embedding (t-SNE) [37] to visualize high-dimensional data in a two-dimensional plane. Fig. 2 compares the distribution of the three synthetic data generation methods with that of the original training data for the HAR dataset.

One of the methods used in statistics to compare various models and measure how well they fit the data is the Akaike information criterion (AIC) [38]. When a model is used to represent the process that generated the data, some of the information is lost due to the model not being exact. AIC measures the relative information lost in the process of modeling that data. A lower AIC value shows a better fit. Fig. 3 shows the AIC values as a function of number of components in the GMM for modeling the training data of the HAR dataset. As can be seen, for this metric the best fit happens when the GMM has 22 components.

Next, we analyze the impact of different parts of TUTOR on DNN model performance. We compare three classes of DNN models. The first class consists of conventional FC DNNs trained on the real training dataset. We refer to such models as DNN 1. The second class of models, referred to as DNN 2, consists of the better model between those obtained by Schemes A and B. The third class of models, referred to as DNN 3, is based on grow-and-prune synthesis. The starting point for DNN 3 is the better one from those synthesized by Schemes A and B, as evaluated on the validation set.

Table 2 shows the comparisons. We report test accuracy, FLOPs, and number of parameters (#Param) for each model. For each dataset, we train various FC DNNs (with different number of layers and neurons per layer) and verify their performance on the validation set. The FC baseline (DNN 1) for each dataset is the one that performs the best on the validation set. Since Scheme A only changes the weights of the network using synthetic data, the network architecture remains the same. Therefore, the FLOPs and number of parameters in the network are the same as those of DNN 1 models. Compared to DNN 1 models, Schemes A and B that take help of synthetic data can be seen to perform better. On an average, DNN 2 models have 2.8% higher test accuracy relative to DNN 1 models. This shows the importance of making use of synthetic data when the available dataset size is not large. We summarize below the relative performance of the DNN models obtained through grow-and-prune synthesis using both real and synthetic data (DNN 3).

- **Better test accuracy:** Compared to DNN 1 (DNN 2), DNN 3 improves test accuracy on an average by 3.6% (2.5%).
- **Less computation:** There is a $4.3\times$ $(6.1\times)$ reduction in FLOPs per inference on an average (geometric mean) relative to DNN 1 (DNN 2).
- **Smaller model size:** There is a $4.7\times$ $(6.6\times)$ reduction in the number of parameters on an average (geometric mean) relative to DNN 1 (DNN 2). Hence, the memory requirements are also significantly reduced.

5.2 Reducing the Need for Data

In this section, we explore the impact of TUTOR in reducing the number of data instances needed. We apply TUTOR to only a part of the data selected using random sampling from the original dataset. We define the *data compression ratio* as the ratio of the original training (validation) dataset size over the sub-sampled training (validation) dataset size. For a fair comparison, the test set remains unchanged across various data compression ratios. Fig. 4 shows the results for all the datasets. We compare the three DNN models as the dataset size decreases. The dashed horizontal and vertical lines in Fig. 4 show the needed data for DNN 3 to achieve the same test accuracy as DNN 1 that relies on the whole training and validation sets. We summarize this information in Table 3. On an average (geometric mean), TUTOR-synthesized DNN 3 (DNN 2) reduces the need for data by $6.0\times$ $(3.9\times)$ while maintaining the same test accuracy.

5.3 TUTOR Application: Performance of Synthetic Data Alone

In this section, we evaluate the application of TUTOR in generating synthetic data that can be used alone to train DNN predictive models. In this application, to avoid exchanging personal information between various parties, we share instead the synthetic data generated by TUTOR. We use the synthetic data generation scheme that leads to the most accurate DNN model on the validation set. Furthermore, we use the most accurate DNN (on the validation set) synthesized by TUTOR to label the synthetic dataset. We generate 100,000 synthetic data instances.

We train a generic three-layer architecture, with two hidden layers with 100 neurons and an output layer, with only the synthetic data and evaluate its performance on the separate real test set. We compare the results with DNN 1 that is trained using the real training and validation sets. Table 4 shows the results. As one can see, using synthetic data alone to train the DNN model leads to similar performance as DNN 1 on the unseen test data. Since we can generate large synthetic datasets, in six of the eight cases, the test accuracy improves by 0.2% to 7.2%.

6 Discussion and Limitations

In this section, we discuss the inspiration we took from the human brain in designing the TUTOR framework. We also discuss TUTOR's limitations that can be addressed in the future.

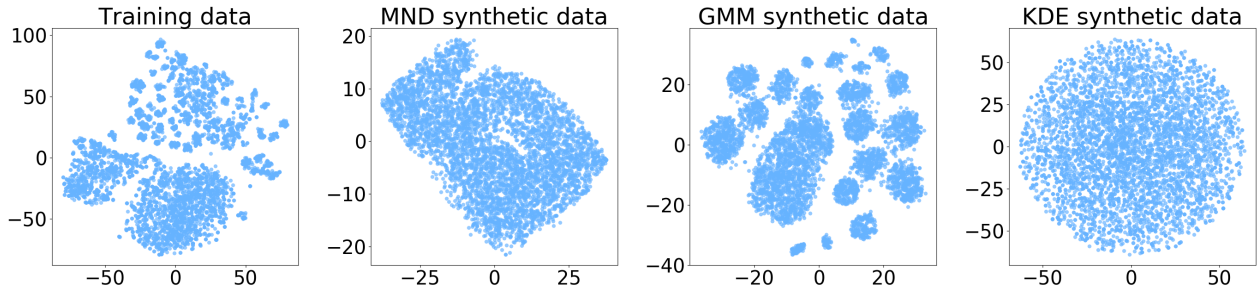

Fig. 2. Visualizing data distribution using t-SNE for 5000 data points obtained from various data generation methods applied to the HAR dataset, from left to right: (1) original training data, (2) multi-variate normal distribution, (3) GMM, and (4) kernel density estimation.

TABLE 2
Comparison results for various parts of TUTOR. The best test accuracies between Schemes A and B (DNN 2) and with grow-and-prune (GP) synthesis are shown in bold.

	FC Baseline			Scheme A (ACC. %)			Scheme B (Acc. %)					+ GP synthesis		
Dataset	Acc. (%)	FLOPs	#Param.	MND	GMM	KDE	MND	GMM	KDE	FLOPs	#Param.	Acc. (%)	FLOPs	#Param.
SenDrive	95.1	19.3k	9.0k	96.5	97.3	95.9	97.4	**97.5**	97.1	44.0k	20.8k	**99.0**	9.8k	5.0k
BIH-Arrhythmia	94.5	107.4k	54.1k	96.0	**96.5**	96.1	94.9	95.5	94.9	215.1k	108.4k	**96.6**	9.7k	5.0k
PTB-ECG	95.0	21.2k	10.7k	95.2	**96.2**	95.6	95.8	95.1	95.8	42.7k	21.6k	**97.3**	19.9k	5.0k
SHAR	91.6	707.6k	354.9k	93.4	93.0	93.3	93.2	**93.7**	92.7	1.4M	710.6k	**94.3**	19.2k	10.0k
EpiSeizure	89.4	95.5k	48.1k	93.0	92.4	93.0	**95.6**	94.6	95.0	191.9k	96.7k	**97.2**	69.7k	35.0k
HAR	93.2	703.1k	352.7k	**94.2**	93.8	94.1	93.3	93.6	93.6	1.4M	706.1k	**95.1**	49.2k	25.0k
DNA	92.3	130.3k	65.7k	93.9	94.8	93.5	93.5	**94.9**	94.0	260.9k	131.6k	**95.8**	49.6k	25.0k
Breast Cancer	93.7	7.2k	3.7k	94.4	**96.9**	94.4	94.4	96.6	95.0	14.7k	7.6k	**98.7**	2.9k	1.5k

TABLE 3
Comparison of the data needed for TUTOR-synthesized DNNs to achieve the DNN 1 accuracy.

	DNN 1			DNN 2			DNN 3		
Dataset	Acc. (%)	#Train	#Val	Acc. (%)	#Train	#Val	Acc. (%)	#Train	#Val
SenDrive	95.1	40509 (1×)	9000 (1×)	94.6	5063 (8×)	1125 (8×)	95.1	4050 (10×)	900 (10×)
BIH-Arrhythmia	94.5	22628 (1×)	7543 (1×)	95.6	11314 (2×)	3771 (2×)	94.5	5657 (4×)	1885 (4×)
PTB-ECG	95.0	8730 (1×)	2910 (1×)	96.2	8730 (1×)	2910 (1×)	95.1	4365 (2×)	1455 (2×)
SHAR	91.6	6213 (1×)	1554 (1×)	91.5	776 (8×)	194 (8×)	91.6	690 (9×)	172 (9×)
EpiSeizure	89.4	6560 (1×)	1620 (1×)	89.9	410 (16×)	101 (16×)	91.3	410 (16×)	101 (16×)
HAR	93.2	5881 (1×)	1471 (1×)	93.5	1960 (3×)	490 (3×)	93.3	840 (7×)	210 (7×)
DNA	92.3	1400 (1×)	600 (1×)	92.2	466 (3×)	200 (3×)	92.3	280 (5×)	120 (5×)
Breast Cancer	93.7	404 (1×)	150 (1×)	95.0	134 (3×)	50 (3×)	93.7	101 (4×)	38 (4×)

An interesting aspect of the human brain is its ability to quickly solve a problem in a new domain, despite limited experience. It intelligently utilizes prior learning experiences to adapt to the new domain. Inspired by this process, TUTOR addresses the need for a large amount of data by generating synthetic data from the same probability distribution as the limited available data. The synthetic data are used to warm-start the DNN training process, providing it with an appropriate inductive bias.

The human brain also changes its synaptic connections dynamically, to adapt to the task at hand. In fact, this is one way for it to acquire knowledge. TUTOR takes inspiration from this aspect of human intelligence in the grow-and-prune synthesis step. Allowing architectures to adapt during the training process enables TUTOR to generate accurate and efficient (both in terms of computation and memory requirements) architectures for the task at hand.

Although TUTOR addresses the small data problem of non-image datasets, the same problem may exist for image-based datasets. For example, image-based datasets in healthcare tend to be small, due to the difficulty of data collection and labeling. Although techniques like DeepInversion [39] have begun to address realistic but synthetic images from the same probability distribution, extending TUTOR to image-based applications can also be part of future work.

7 Conclusion

In this work, we proposed the TUTOR framework to address the need for large datasets in training DNN predictive models. TUTOR uses the synthetic dataset generated from the same distribution as the real training data, labeled through a random forest model trained on the real training data, to pre-train the DNN model, thus starting the training process with a better initialization point. We discussed two schemes for combining synthetic and real data for training. In addition, TUTOR utilizes the grow-and-prune synthesis paradigm to ensure model compactness while boosting accuracy. TUTOR can be particularly useful in settings where the available data are limited, such as in healthcare applications. Since the generated models are compact, they can be deployed on edge devices, such as smartphones and Internet-of-Things sensors.

TABLE 4
Comparison of the FC baseline trained on real data with an FC DNN trained on synthetic data.

	FC DNN trained on real data			FC DNN trained on syn. data			
Dataset	Acc. (%)	FLOPs	#Param.	Method	Acc. (%)	FLOPs	#Param.
SenDrive	95.1	19.3k	9.0k	GMM	92.5	31.6k	16.1k
BIH-Arrhythmia	94.5	107.4k	54.1k	KDE	95.1	57.6k	29.1k
PTB-ECG	95.0	21.2k	10.7k	GMM	92.6	57.6k	29.1k
SHAR	91.6	707.6k	354.9k	KDE	92.8	134.4k	67.5k
EpiSeizure	89.4	95.5k	48.1k	GMM	96.6	55.8k	28.2k
HAR	93.2	703.1k	352.7k	GMM	93.4	133.2k	66.9k
DNA	92.3	130.3k	65.7k	GMM	94.2	56.4k	28.5k
Breast Cancer	93.7	7.2k	3.7k	KDE	95.0	26.2k	13.4k

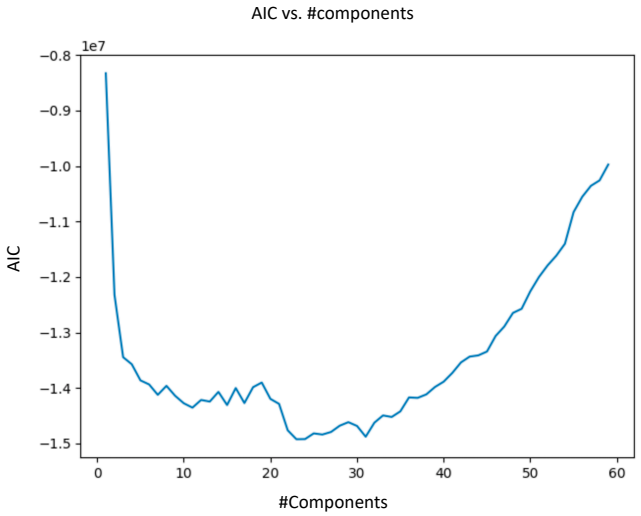

Fig. 3. AIC vs. number of components in GMM for the HAR dataset.

REFERENCES

[1] S. Hassantabar, N. Stefano, V. Ghanakota, A. Ferrari, G. N. Nicola, R. Bruno, I. R. Marino, and N. K. Jha, "CovidDeep: SARS-CoV-2/COVID-19 test based on wearable medical sensors and efficient neural networks," *arXiv preprint arXiv:2007.10497*, 2020.

[2] D. D. Bourgin, J. C. Peterson, D. Reichman, S. J. Russell, and T. L. Griffiths, "Cognitive model priors for predicting human decisions," in *Proc. Int. conf. Machine Learning*, 2019, pp. 5133–5141.

[3] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[4] X. Dai, H. Yin, and N. K. Jha, "NeST: A neural network synthesis tool based on a grow-and-prune paradigm," *IEEE Trans. Computers*, vol. 68, no. 10, pp. 1487–1497, 2019.

[5] M. Richardson and P. Domingos, "Markov logic networks," *Machine Learning*, vol. 62, no. 1-2, pp. 107–136, 2006.

[6] L. Serafini and A. D. Garcez, "Logic tensor networks: Deep learning and logical reasoning from data and knowledge," *arXiv preprint arXiv:1606.04422*, 2016.

[7] R. Balestriero, "Neural decision trees," *arXiv preprint arXiv:1702.07360*, 2017.

[8] T. C. LingChen, A. Khonsari, A. Lashkari, M. R. Nazari, J. S. Sambee, and M. A. Nascimento, "UniformAugment: A search-free probabilistic data augmentation approach," *arXiv preprint arXiv:2003.14348*, 2020.

[9] A. Wan, L. Dunlap, D. Ho, J. Yin, S. Lee, H. Jin, S. Petryk, S. A. Bargal, and J. E. Gonzalez, "NBDT: Neural-backed decision trees," *arXiv preprint arXiv:2004.00221*, 2020.

[10] P. Kontschieder, M. Fiterau, A. Criminisi, and S. Rota Bulo, "Deep neural decision forests," in *Proc. IEEE Conf. Computer Vision*, 2015, pp. 1467–1475.

[11] Y. Yang, I. G. Morillo, and T. M. Hospedales, "Deep neural decision trees," *arXiv preprint arXiv:1806.06988*, 2018.

[12] K. D. Humbird, J. L. Peterson, and R. G. McClarren, "Deep neural network initialization with decision trees," *IEEE Trans. Neural Networks and Learning Systems*, vol. 30, no. 5, pp. 1286–1295, 2018.

[13] M. Oquab, L. Bottou, I. Laptev, and J. Sivic, "Learning and transferring mid-level image representations using convolutional neural networks," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2014, pp. 1717–1724.

[14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNet v2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[15] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient CNN architecture design," in *Proc. European Conf. Computer Vision*, 2018, pp. 116–131.

[16] B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer, "Shift: A zero flop, zero parameter alternative to spatial convolutions," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2018, pp. 9127–9135.

[17] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, P. Vajda, M. Uyttendaele, and N. K. Jha, "ChamNet: Towards efficient network design through platform-aware model adaptation," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 2019.

[18] S. Hassantabar, X. Dai, and N. K. Jha, "STEERAGE: Synthesis of neural networks using architecture search and grow-and-prune methods," *arXiv preprint arXiv:1912.05831*, 2019.

[19] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *in Proc. Int. Conf. Learning Representations*, 2016.

[20] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[21] X. Dai, H. Yin, and N. K. Jha, "Grow and prune compact, fast, and accurate LSTMs," *IEEE Trans. Computers*, vol. 69, no. 3, pp. 441–452, 2019.

[22] S. Hassantabar, Z. Wang, and N. K. Jha, "SCANN: Synthesis of compact and accurate neural networks," *arXiv preprint arXiv:1904.09090*, 2019.

[23] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[24] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation policies from data," *arXiv preprint arXiv:1805.09501*, 2018.

[25] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas, "Reinforcement learning with augmented data," *arXiv preprint arXiv:2004.14990*, 2020.

[26] A. Antoniou, A. Storkey, and H. Edwards, "Data augmentation generative adversarial networks," *arXiv preprint arXiv:1711.04340*, 2017.

[27] A. Malinverno and V. A. Briggs, "Expanded uncertainty quantification in inverse problems: Hierarchical Bayes and Empirical Bayes," *Geophysics*, vol. 69, no. 4, pp. 1005–1016, 2004.

[28] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proc. Machine Learning Research*, vol. 70, 2017, pp. 1126–1135.

[29] E. Grant, C. Finn, S. Levine, T. Darrell, and T. Griffiths, "Recasting gradient-based meta-learning as Hierarchical Bayes," in *Proc. Int. Conf. Learning Representations*, 2018.

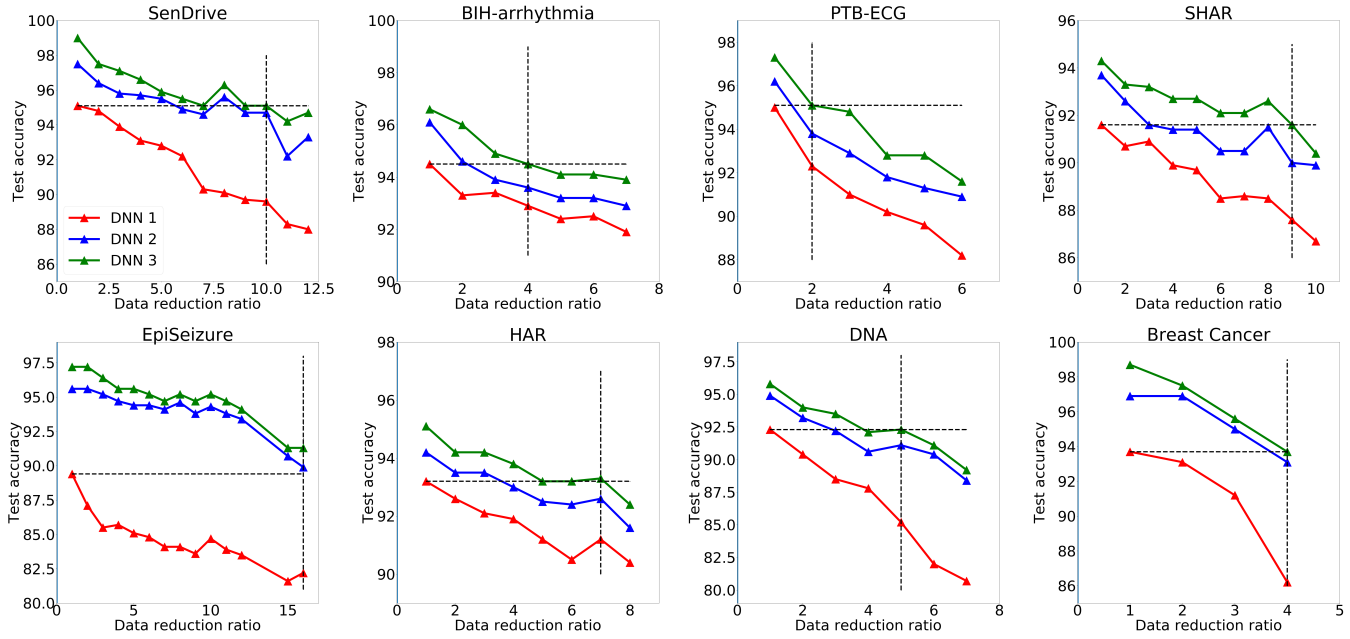

Fig. 4. Test accuracy vs. data compression ratio.

[30] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *Proc. IEEE Symp. Security and Privacy*, 2008, pp. 111–125.

[31] L. Sweeney, "Simple demographics often identify people uniquely," *Health (San Francisco)*, vol. 671, no. 2000, pp. 1–34, 2000.

[32] H. Surendra and H. Mohan, "A review of synthetic data generation methods for privacy preserving data publishing," *Int. Journal of Scientific and Technology Research*, vol. 6, no. 3, pp. 95–101, 2017.

[33] "UCI machine learning repository." [Online]. Available: https://archive.ics.uci.edu/ml/datasets.php

[34] "PTB diagnostic ECG database." [Online]. Available: https://www.kaggle.com/openmark/ptb-diagnostic-ecg-database

[35] "MIT-BIH arrhythmia database." [Online]. Available: https://www.kaggle.com/taejoongyoon/mitbit-arrhythmia-database

[36] C. W. Hsu and C. J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE Trans. Neural Networks*, vol. 13, no. 2, pp. 415–425, 2002.

[37] L. Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[38] S. I. Vrieze, "Model selection and psychological theory: A discussion of the differences between the Akaike information criterion (AIC) and the Bayesian information criterion (BIC)." *Psychological Methods*, vol. 17, no. 2, p. 228, 2012.

[39] H. Yin, P. Molchanov, J. M. Alvarez, Z. Li, A. Mallya, D. Hoiem, N. K. Jha, and J. Kautz, "Dreaming to distill: Data-free knowledge transfer via deepinversion," in *Proc. IEEE/CVF Conf. Computer Vision and Pattern Recognition*, 2020, pp. 8715–8724.

Shayan Hassantabar received his B.S. degree in Electrical Engineering, Digital Systems focus, from Sharif University of Technology, Iran. He also received his M.Math. degree in Computer Science from University of Waterloo, Canada, and his M.A. degree in Electrical Engineering from Princeton University. He is pursuing the Ph.D. degree in Electrical Engineering at Princeton University. His research interests include automated neural network architecture synthesis, neural network compression, and smart healthcare.

Prerit Terway received his M.S. degree from University of Michigan, Ann Arbor, and B.Tech. degree from Indian Institute of Technology, Gandhinagar, India, both in Electrical Engineering. He is currently a Ph.D. candidate in Electrical Engineering at Princeton University. His research interests include machine learning and cyber-physical systems.

Niraj K. Jha received his B.Tech. degree in Electronics and Electrical Communication Engineering from Indian Institute of Technology, Kharagpur, India in 1981 and Ph.D. degree in Electrical Engineering from University of Illinois at Urbana-Champaign, IL in 1985. He has been a faculty member of the Department of Electrical Engineering, Princeton University, since 1987. He is a Fellow of IEEE and ACM, and was given the Distinguished Alumnus Award by I.I.T., Kharagpur. He has also received the Princeton Graduate Mentoring Award.

He has served as the Editor-in-Chief of IEEE Transactions on VLSI Systems and as an Associate Editor of several other journals. He has co-authored five books that are widely used. His research has won 20 best paper awards or nominations and 21 patents. His research interests include smart healthcare, cybersecurity, machine learning, and monolithic 3D IC design. He has given several keynote speeches in the area of nanoelectronic design/test and smart healthcare.