Machine Vision and Applications (2022) 33:59
https://doi.org/10.1007/s00138-022-01311-z

ORIGINAL PAPER

Dynamically throttleable neural networks

Hengyue Liu[1] · Samyak Parajuli[2] · Jesse Hostetler[3] · Sek Chai[4] · Bir Bhanu[1]

Received: 22 November 2021 / Accepted: 15 June 2022 / Published online: 7 July 2022

Abstract
Conditional computation for deep neural networks reduces overall computational load and improves model accuracy by running a subset of the network. In this work, we present a runtime dynamically throttleable neural network (DTNN) that can self-regulate its own performance target and computing resources by dynamically activating neurons in response to a single control signal, called *utilization*. We describe a generic formulation of throttleable neural networks (TNNs) by grouping and gating partial neural modules with various gating strategies. To directly optimize arbitrary application-level performance metrics and model complexity, a controller network is trained separately to predict a context-aware utilization via deep contextual bandits. Extensive experiments and comparisons on image classification and object detection tasks show that TNNs can be effectively throttled across a wide range of utilization settings, while having peak accuracy and lower cost that are comparable to corresponding vanilla architectures such as VGG, ResNet, ResNeXt, and DenseNet. We further demonstrate the effectiveness of the controller network on throttleable 3D convolutional networks (C3D) for video-based hand gesture recognition, which outperforms the vanilla C3D and all fixed utilization settings.

Keywords Throttleable neural network (TNN) · Conditional computation · Dynamic neural network · Contextual controller

1 Introduction

Recently, deep neural networks (DNNs) are prevailing in computer vision applications on edge devices and autonomous vehicles where real-time response is needed. The computation power and memory budget are drastically different across devices. Even on the same device, the runtime performance varies given different battery conditions, operation temperatures, etc. Researchers try to design lightweight neural networks [1–6] or perform neural architecture search (NAS) [7–11] with computation complexity constraints. Others seek ways of network pruning [12–15], model distillation and compression [16–19] or quantization [20–24] to reduce the memory footprint. These approaches typically provide offline-trained static models with a constant allocation of computation and memory resource. After training and deploying, the model is fixed such that the whole network has to be executed. However, the conditions in real-world setting are often different, whereby the runtime inference is neither optimal from an accuracy or efficiency perspective. The problem lies in that the naive training approaches can only produce static models with a specialized trade-off between performance and resource utilization. By leveraging runtime filter selection inspired from Dropout [25,26], this paper presents an adaptive system named dynamically throttleable neural network (DTNN) to tackle challenges in highly dynamic deployment environments. DTNN consists of a context-aware controller and a throttleable neural network. We use the term "throttleable" to suggest the ability of the model to adaptively balance performance and resource use in response to a control signal named *utilization*. TNNs flexibly support diverse and dynamic configurations under

✉ Hengyue Liu
hliu087@ucr.edu

Samyak Parajuli
samyak.parajuli@berkeley.edu

Jesse Hostetler
JesseHostetler@gmail.com

Sek Chai
sek@latentai.com

Bir Bhanu
bhanu@ee.ucr.edu

1 University of California, Riverside, Riverside 92521, CA, USA

2 University of California, Berkeley, Berkeley 94720, CA, USA

3 SRI International, Princeton, NJ 08540, USA

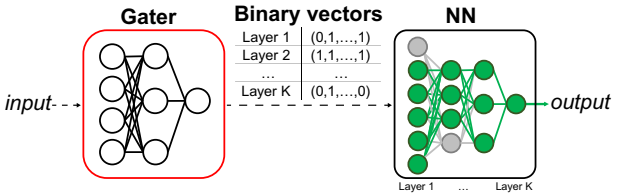
4 Latent AI, Skillman 08558, NJ, USA

different resource constraints without retraining or redeploying. The contextual controller learns the policy to generate the input-dependent utilization as the control signal for TNNs. For example, the controller predicts lower utilization for "easy" input data (e.g., video sequences of static objects), and higher utilization for "challenging" one (e.g., video sequences of moving pedestrians). Optimizing the best trade-off between performance metrics and utilization is carried out via deep contextual reinforcement learning [27–29]. In short, the controller determines how much to throttle, and the TNN determines how to throttle.

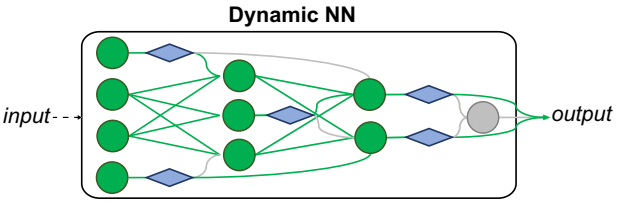
A major benefit of DTNN is the separation of the controller and TNN. Any controller that provides a single scalar can be used, such as heuristic and learnable policies. That is, the controller can be as simple as a fixed utilization parameter, or a state machine with trainable policies. Moreover, by using a *single* utilization parameter u to conditionally gate part of the network, the complexity of the controller is much reduced while retaining highly flexible inference paths. In addition to the input data, TNN only takes an extra input, the utilization, for task-specific predictions. In comparison, standard gating networks (Fig. 1a) produce a large number of vectors as control signals that make it hard to train and fine-tune the gating network. Another advantage of DTNN is its high flexibility and modularity. TNN consists of neural kernels which are structured into "blocks", while conditional computation is applied at the level of individual block. The gating strategies consider different dimensions of the network ("width-wise" or "depth-wise") as well as the order of gating ("independent" or "nested"). The training approach for TNNs essentially enforces sparsity in the kernels such that there is no catastrophic accuracy loss as runtime utilization decreases in comparison to a naively trained network. TNNs are trained by maximizing task performance metric over all the utilization settings, while optionally minimizing the computational cost for an application under conditions that can change over time.

We are contributing to the body of research on conditional computation and adaptive control (Sect. 2) with DTNN to improve model performance and address efficient inference with tight resource budgets such as memory, power, and compute capability. For vision-based systems, this is particularly important, as the workload and processing throughput are demanding and dynamic. To the best of our knowledge, we offer the following contributions in this paper:

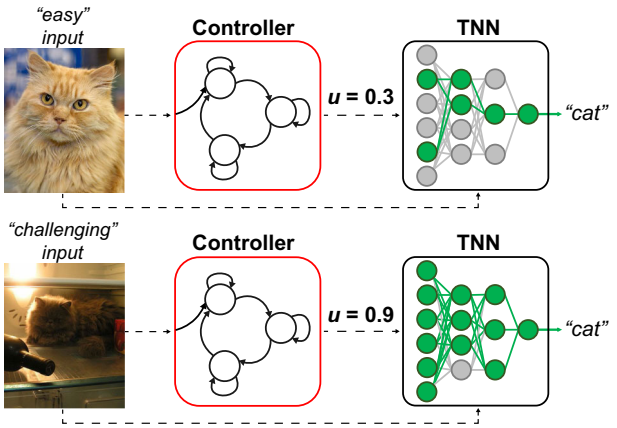
1. A novel architecture called throttleable neural networks with plug-and-play throttleable blocks comprised of convolutional or fully connected filters, and the usage of model capacity is controlled by a single utilization parameter.
2. A lightweight context-aware controller trained to regulate TNNs' performances and computing needs with learnable control policies.
3. Comprehensive experimental results on image classification, object detection and video-based hand gesture recognition tasks that demonstrate the effectiveness of TNNs, while most of the previous work only shows results for image classification.



(a) An example diagram of network with conditional gating mechanism [30, 31]. The "gater" generates the gating functions as binary masks and applies for all computation nodes.



(b) An example diagram of adaptive control of neural networks [28, 29, 32, 33]. The execution of a computation node is determined by a control node ◆. As a result, only a subset of the inference graph is executed.



(c) Our proposed dynamically throttleable neural network. Based on the contextual information of the input data, the controller decides how much utilization of TNN is needed. Different from Figure 1a, gating is controlled by one scalar, the utilization u, instead of vectors. The dynamic execution is decoupled instead of entangled as in Figure 1b. Our DTNN enables much more robust and flexible configurations.

Fig. 1 Conceptual architectures: **a** an ordinary gating network, **b** an ordinary dynamic network, **c** the proposed DTNN. A computation node ○ can represent a single kernel, a group of kernels, a layer or a group of layers based on different designs. An executed computation node is drawn as ● while an unexecuted or gated node is drawn as ○

4. A use case of DTNN for hand gesture recognition system which outperforms the vanilla architectures and all fixed utilization settings.
5. Physical power and runtime measurements on an embedded GPU that demonstrate the efficacy and practicality of TNNs.

This paper is organized as follows: in Sect. 2, we present an overview of related research. In Sect. 3, we define the DTNN system, including the TNN and contextual controller architectures and training approaches. In Sect. 4, we provide comprehensive experimental results and ablation studies that thoroughly evaluate our approach. We showcase an example of gesture recognition system that is built upon the DTNN framework. Finally, in Sect. 5, we conclude our work and contributions.

2 Related work

We focus on the research related to conditional computation and adaptive control of DNNs [34]. Design of efficient architectures [2,3,35–37], NAS [7,9,38,39], and static model compression [12–14,16–19,40–44] are different from our approach such that we have multiple operational points within a single architecture for performance and efficiency trade-off, rather than a single static model. In fact, most architectures can be easily converted into a throttleable one without losing the peak performance which will be demonstrated later in Sects. 4 and 4.6.2.

2.1 Conditional computation

Our work builds on conditional computation [49] where portions of the model are executed for prediction to reduce overall computational load. Different from static or post-training pruning, sub-models are generated on the fly to achieve dynamic model size or computation resource. An intuitive idea is the early prediction where the inference stops at an early stage of the network once a criterion or "confidence" is satisfied. Such examples include Adaptive Computation Time (ACT/SACT) [50], BranchyNet [51], and Dynamic Time Recurrent Visual Attention (DT-RAM) [52], where essentially gating is applied for all the kernels after the decision layer. Convolutional neural mixture model (CNMM) [53], as an example, is the ensemble of convolutional neural networks (CNNs) that are sampled during inference with "early-exit" classifiers. However, vision tasks in real-world environments are challenging and complex, depending only on low-level features is not practical. This work leans more toward encouraging sparse activations throughout the entire network rather than discarding high-level semantic features.

Beyond early prediction, there are other methods that reduce computations conditionally. Shazeer et al. [54] proposed the Mixture-of-Experts layer that learns to rank and select the top several sub-networks. Similarly, in [55], a gating module is proposed to select among multiple branches of networks of features for each input. NestedNet [47] constructs a nested architecture with several levels of sparsity. Stochastic Depth [56], Skipnet [48], ConvNet-AIG [33] and BlockDrop [30] are similar approaches that learn to bypass ResNet [57] blocks based on the input. Similar ideas also appear in recent work on neural architecture search, such as EfficientNet [58] that studies the model scaling of depth, width and resolution, and OFA [11] that searches sub-networks for specialized edge devices. Dynamic channel pruning [32,59,60] can also be categorized as conditional computation which performs selective convolutions during runtime.

2.2 Adaptive control

More recently, the research is moving toward dynamically configuring the network topology at runtime based on input. In this line of research, no parallel networks are explicitly defined, instead, a single sub-network is selected from the super-network by partially activating model components such as filters and layers for each input. For example, Slimmable Neural Networks [46] can scale the network width to pre-defined configurations. GaterNet [31] uses a separate gating network to generate sparse binary masks for the backbone network in an input-dependent manner. Odena et al. [61] introduce a "Composer" module to select the computational graph. D^2NN [28] uses reinforcement learning to jointly learn the parameters of computation nodes and control nodes. Similarly, Spasov et al. [29] propose a channel based selection method by casting the gating function as a multi-armed bandit problem. Ahn et al. [45] consider the dynamic network as an estimator-selector framework for multi-task learning such that the candidate network is optimized for one specialized task.

2.3 Summary

The key difference for TNNs as compared to previous work is the flexible integration of the network modules with the control decision nodes. Compared with recent work theoretically, the TNN itself provides a much more flexible model selections and applications (Table 1). TNNs subsume these models [30,46,47]. In particular, [46,47] are subsumed under our nested width-wise gating strategy with limited number of operational levels. BlockDrop [30] is only applicable to ResNet-type networks with skip-connections which can be summarized as our nested depth-wise gating strategy. In terms of adaptive control, [28,29,31,61] generate

Table 1 Theoretical comparisons of the DTNN and related work on conditional computation

	Architecture support				Reported experimental results			
	Width-wise	Depth-wise	Inference		Image classification	Object detection	Video classification	Embedded
	#OP	#OP	Static	Dynamic				
DTNN	$\mathcal{O}(2^{CL})$	$\mathcal{O}(2^{L})$	✓	✓	✓	✓	✓	✓
DEN [45]	$\mathcal{O}(2^{CL})$	$\mathcal{O}(2^{CL})$	✓		✓			
GaterNet [31]	$\mathcal{O}(2^{CL})$	$\mathcal{O}(2^{CL})$		✓	✓			
Slimmable [46]	$\mathcal{O}(N)$		✓		✓	✓		
NestedNet [47]	$\mathcal{O}(N)$		✓		✓			
Spasov and Liò [29]	$\mathcal{O}(2^{CL})$			✓	✓			
RNR [32]	$\mathcal{O}(2^{CL})$			✓	✓			
D^2NN [28]		$\mathcal{O}(N)$	✓		✓			
BlockDrop [30]		$\mathcal{O}(2^{L})$		✓	✓			
SkipNet [48]		$\mathcal{O}(2^{L})$		✓	✓			
ConvNet-AIG [33]		$\mathcal{O}(2^{L})$		✓	✓			

The proposed method supports both width-wise and depth-wise gating. It can achieve static inference using fixed utilization, or per-input dynamic inference with a learnable contextual controller without fine-tuning or retraining the throttleable neural network

#OP: the theoretical maximum number of operational points or sub-models within the architecture (the actual #OP depends on implementations).

Static: the sub-model is manually selected from several available configurations. Dynamic: the sub-model is selected dynamically based on the input.

C: the maximum number of channels among all layers.

L: the number of layers within the architecture.

N: the number of configurations within the architecture.

DTNN can have at most $\mathcal{O}(2^{CL})$ operational points when applying both width-wise and depth-wise gating within the architecture

sequences of control nodes as a form of vectors. The control and backbone networks are tightly coupled before fine-tuning training is further applied. Our approach, on the other hand, introduces the single-scalar utilization parameter to control the backbone network that is user-friendly and semantically meaningful. With the utilization parameter, the control and backbone network can be trained and optimized independently. Furthermore, our two-phase training does not require fine-tuning the network jointly. The TNN backbone can work by itself and preserve a monotonic performance without the controller. Other adaptive control approaches can not guaranteed this performance since their control is either integrated into the network or limited by complicated gating functions. Whereas most previous papers show results for image classification only, we demonstrate our approach with insights and provide results for object detection, video classification, and hardware performance.

3 Technical approach

3.1 Objective and problem setting

Our goal is to enable TNN with dynamic performance during inference while having the same or similar number of parameters with vanilla ones. Furthermore, TNN should be modular such that (1) they are easily trained with gating policies that can be fixed or learned, (2) the controller can be trained separately without retraining the TNN, (3) the framework is largely model-agnostic.

A neural network is a function $\mathsf{T}_\Phi(\mathsf{x})$ parameterized by Φ that maps an input data x to an output $\hat{\mathsf{y}}$. For example, x can be an image or a video sequence, and $\hat{\mathsf{y}}$ is the prediction of class label for a classification problem. We define a *throttleable neural network* as a function of two variables, $\mathsf{T}_\Phi(\mathsf{x}, u)$, where $u \in (0, 1]$ is a control utilization parameter that indicates how much "computational effort" the network should exert. We emphasize that u is an additional input to the network; after training is complete, the network parameters Φ are fixed but u can change.

The problem is then formulated as the minimization of the "task loss" under all utilization settings, and the general objective of training TNNs is

$$\min \mathbb{E}_{u\sim\text{Uniform}(0,1)} \left[\mathcal{L}_{\text{task}}(\mathsf{y}, \hat{\mathsf{y}}; u)\right], \tag{1}$$

where y is the ground-truth label of x, and the utilization u is uniformly drawn from [0, 1]. The loss $\mathcal{L}_{\text{task}}$ is a task-specific performance measure. We use cross-entropy loss for classification tasks, and smooth L1 loss [62] for bounding box regression in object detection tasks.

3.2 Throttleable neural networks

3.2.1 Throttleable block

We consider the building block of TNN architectures that we call *throttleable block* (TB). A TB consists of arbitrary number of filters within one convolutional and fully connected layer, or across several layers with skip-connections. Most CNN architectures can be converted into TNNs by replacing their layers with TBs.

Let $\mathbf{x}$, $\mathbf{y}$ denote the input and output features to the throttleable block t parameterized by a set of transformations $\boldsymbol{\phi} = \{\phi_i \mid i = 1, \ldots, D\} \subset \Phi$, where D is the size of the set which is called "cardinality". The transformation ϕ_i could be anything from individual neurons to entire networks, and we focus on an intermediate level of granularity. It can be arbitrary NN modules like convolutional or fully connected layers, as long as they have the same input space and their outputs can be aggregated appropriately. We define the *gating functions* $\boldsymbol{g} = \{g_i \mid i = 1, \ldots, D\}$ to determine whether to execute each transformation. Then the *throttleable block* has the functional form as

$$\begin{aligned} \mathbf{y} &= t_{\boldsymbol{\phi}}(\mathbf{x}, u) \\ &= \texttt{aggr}\left(\boldsymbol{g}(\mathbf{x}, u) \odot \boldsymbol{\phi}(\mathbf{x})\right) \\ &= \texttt{aggr}\left(g_1(\mathbf{x}, u) \cdot \phi_1(\mathbf{x}), \ldots, g_D(\mathbf{x}, u) \cdot \phi_D(\mathbf{x})\right), \end{aligned} \tag{2}$$

where $g_i(\mathbf{x}, u) : \mathbf{x} \times (0, 1] \mapsto \{0, 1\}$ is the *gating function* generating a scalar of either 0 or 1, $\odot$ denotes element-wise multiplication, and `aggr` is the *aggregation function* that maps the set of transformations to the appropriate output space by feature concatenation or summation. The gating function g_i is computed first and determines whether to execute ϕ_i in practice. For example, if $g_i = 0$, the component ϕ_i is effectively disabled and replaced with zero tensors. Our exposition focuses on a single gated module for simplicity, but in practice we compose multiple TBs in a typical TNN. Let $\mathbf{x}^{(l)}$ denote the input feature to the l-th layer, then the output feature $\mathbf{x}^{(l+1)}$ has the functional form as $\mathbf{x}^{(l+1)} = t_{\boldsymbol{\phi}^{(l)}}(\mathbf{x}^{(l)}, u)$, and $\mathbf{x}^{(l+1)}$ is directly fed as input to the next layer.

3.2.2 Gating strategies

For simplicity and comparison, we use convolutional layers for illustrating different gating strategies. Let $\boldsymbol{\phi}$ denote a convolutional layer with C filters. It can be decomposed into D groups of transformations $\{\phi_i \in \mathbb{R}^{C/D\times h\times w} \mid i = 1, \ldots, D\}$, each of which consists of at most C/D filters. Each group can be gated selectively. The decomposition and gating is performed along the first dimension (channel) of ϕ_i, and we name this gating strategy as WidthWise gating. Then we apply the non-gated transformations on the input features

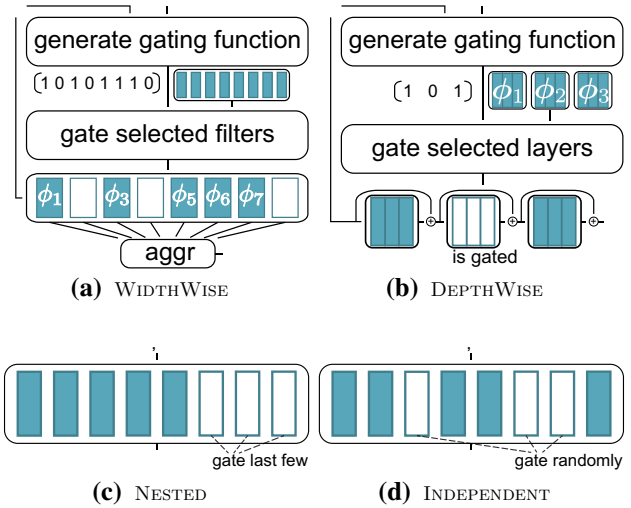


Fig. 2 Selective gating strategies. The colored blocks are activated groups while white groups are gated. **a**, **b** Gating strategies along different dimensions, while **c**, **d** have different ordering of gating

and perform aggregation following Eq. 2. An example of WidthWise gating is shown in Fig. 2a.

Optionally, we can chose to skip the whole layer or block if we have shortcut connections between layers such as in ResNet [57], DenseNet [63], and their variants, then we call this gating strategy as DepthWise gating. Formally, we consider a residual block of the form as $\mathbf{y} = \boldsymbol{\phi}(\mathbf{x}) + \mathbf{x}$ where $\mathbf{x}$ and $\mathbf{y}$ have the same shape, and a DepthWise throttleable block can be represented as

$$\begin{aligned} \mathbf{y} &= t_{\boldsymbol{\phi}}(\mathbf{x}, u) + \mathbf{x} \\ &= \boldsymbol{\phi}(\mathbf{x}) \cdot g(\mathbf{x}, u) + \mathbf{x} \\ &= \begin{cases} \boldsymbol{\phi}(\mathbf{x}) + \mathbf{x} & \text{if } g(\mathbf{x}, u) = 1 \\ \mathbf{x} & \text{otherwise.} \end{cases} \end{aligned} \tag{3}$$

In DepthWise gating, $\boldsymbol{\phi}$ can be any block that consists of either a single layer or several layers, and the gating function g is applied to the whole block. An example of DepthWise gating is shown in Fig. 2b. It is worth noting that WidthWise and DepthWise gating can also be used concurrently within the same network.

3.2.3 Mapping utilization to gating functions

For each throttleable block, the mapping $g(\mathbf{x}, u)$ from the utilization to binary gating vectors needs to be defined. We consider Independent and Nested gating strategies to determine which transformations should be gated off, or the order of gating. In previous conditional computation research, the components of each gated module are viewed as independent of one another with few constraints on their pattern of activation. This Independent gating strategy (Fig. 2d) works for each component to model different features, such as in [30,54]. For our goal of throttling over a range of set points, however, this specialization produces redundancies in the representation. We propose a different method that we call Nested gating. In the Nested strategy, the gating function $\boldsymbol{g}$ is constrained such that $g_i = 1 \Rightarrow g_j = 1 \; \forall j < i$ (Fig. 2c). In our experiments, we employ a training scheme designed to maximize the useful range of u. For each training sample, we draw $u \sim \text{Uniform}[0, 1]$. Then, for each throttleable block, we select d group of transformations to be activated, where $d = \texttt{min}(D, \lfloor u \cdot (D+1) \rfloor)$ and D is the total number of groups in the block (cardinality). For Nested gating strategy, we set $g_1, \ldots, g_d$ to 1 and $g_{d+1}, \ldots, g_D$ to 0, while for Independent gating strategy we choose d indices at random without replacement. Empirically, we observe that the Nested strategy gives superior throttling performance given the same architecture.

3.2.4 Learnable gating function

Equation 1 only considers the task performance under all the utilization settings. Optionally, we can add model complexity constraints while maximizing the task performance, and the overall loss is then defined as

$$\mathcal{L}(\mathsf{x}, u, \mathsf{y}, \hat{\mathsf{y}}) = \mathcal{L}_{\text{task}}(\mathsf{y}, \hat{\mathsf{y}}; u) + \lambda \mathcal{C}(\mathsf{x}, u), \tag{4}$$

where $\mathcal{C}$ is a function that measures the resources used (FLOPs, energy, latency, etc.) for data x at utilization u, and λ controls the balance of the two components in the loss. The utilization u is used as the ground-truth for computing the complexity. We enforce the constraint that the actual complexity of the TNN should not exceed the target complexity u by optimizing the combined loss function (Eq. 4). We experimented with variants of $\mathcal{C}$ of the two functional forms, namely the hinge penalty

$$\mathcal{C}^p_{\text{hinge}}(\mathsf{x}, u) \stackrel{\text{def}}{=} \max(0, c(\Phi; \mathsf{x}, u) - u)^p, \tag{5}$$

and the distance penalty

$$\mathcal{C}^p_{\text{dist}}(\mathsf{x}, u) \stackrel{\text{def}}{=} |c(\Phi; \mathsf{x}, u) - u|^p, \tag{6}$$

where $p \in \{1, 2\}$, and $c(\Phi; \mathsf{x}, u)$ is the complexity of a TNN. In practice, we found that using the distance penalty with $p = 1$ achieves a higher accuracy for the same resource constraint. The complexity of a TNN is defined as the macro-average of the complexities over all modules:

$$c(\Phi; \mathsf{x}, u) = \frac{1}{L} \sum_l c(\boldsymbol{\phi}^{(l)}; \mathbf{x}^{(l)}, u), \tag{7}$$

where $c(\boldsymbol{\phi}^{(l)}; \mathbf{x}^{(l)}, u)$ denotes the complexity of a block, L is the total number of blocks in a TNN, and we set $c(\boldsymbol{\phi}^{(l)}) = 1$ for non-throttleable blocks. A natural measure of the complexity of a TB is its cardinality of active transformations which is defined as

$$c(\boldsymbol{\phi}; \mathbf{x}, u) = \|\boldsymbol{\phi}\|_1^{-1} \sum_i \mathbf{1}\left(g_i(\mathbf{x}, u) = 1\right) \cdot \|\phi_i\|_1, \tag{8}$$

where $\|\boldsymbol{\phi}\|_1^{-1}$ is the total number of transformations in a TB, $\mathbf{1}(\cdot)$ is the indicator function, and $\|\phi_i\|_1$ is the number of transformations in a group of the TB.

Learning the gate controller is complicated by the "rich get richer" interaction between $\boldsymbol{g}$ and $\boldsymbol{\phi}$, in which only the subset of $\boldsymbol{\phi}$ selected by $\boldsymbol{g}$ receives training, which improves its performance and reinforces the tendency of $\boldsymbol{g}$ to select it. To address this, we adopt a two-phase training strategy similar to [50]. In the first phase, we train the "data path" with random utilization to optimize only Eq. 1 for being compatible with throttling. In the second phase, we train the gate to optimize the full objective (Eq. 4) while keeping the data path fixed.

3.2.5 Training the data path

The data path is referred to as the TNN for specific tasks without any form of learnable gating functions or controllers. For classification problem, the data path is the network consisting of the feature extractor and multi-class classifier. During phase 1 of training, we train the feature representations of the TNN to be robust to varying amounts of gating. Therefore, the utilization is randomly sampled from a uniform distribution (Eq. 1). The choice of how u is sampled during training is important for obtaining the desired performance profile. From an empirical risk minimization perspective, we can interpret the training-time distribution of u as a prior distribution on the values of u that we expect at test time. Naïve training without gating can be viewed as one extreme, where we always set $u = 1$. Either NESTED or INDEPENDENT gating function can be used in phase 1 training.

3.2.6 Training the gate

Besides the fixed gating functions NESTED and INDEPENDENT, we propose two learnable gating functions called CONCRETE and REINFORCE. Our objective is to learn all the gating functions $\mathrm{G} = (\boldsymbol{g}^{(1)}, \ldots, \boldsymbol{g}^{(L)})$ during the phase 2, where we freeze the data path parameters Φ and optimize the gate function parameterized by Ψ. To make the output of a gating function g_i either 0 or 1, it is modeled as a Bernoulli random variable parameterized by ψ_i:

$$g_i(\mathbf{x}, u; \psi_i) \sim \text{Bernoulli}(\mathrm{P}(\mathbf{x}, u; \psi_i)), \tag{9}$$

and our task is to learn the parameters ψ_i for minimizing $\mathcal{C}(\mathsf{x}, u)$ while maintaining the task performance. It is worth noting that the utilization for each TB could vary and the average utilization reaches u approximately. We minimize the difference of the actual utilization and the target utilization u for each TB instead of enforcing an exact utilization value. Since the complexity estimate $\mathcal{C}$ is discontinuous and non-differentiable, we need to employ a gradient estimator for training. We evaluated two existing methods of training networks with stochastic discrete neurons for this purpose.

Score function estimator The most common approach [30,48,64,65] is to treat $\boldsymbol{g}$ as the output of a stochastic policy and train it with a policy gradient method such as the score function (REINFORCE) estimator,

$$\nabla_\Psi \mathbb{E}[\mathcal{C}] = \mathbb{E}_\mathsf{x} \mathbb{E}_u [\mathcal{C} \cdot \nabla_\Psi \log \mathrm{P}(\mathrm{G}(\mathsf{x}, u; \Psi))], \tag{10}$$

where $\mathrm{P}(G(\mathsf{x}, u; \Psi))$ is the probability density of random variables G. Since each g_i is an independent Bernoulli random variable (Eq. 9), the log probability is given by

$$\begin{aligned} \log \mathrm{P}(\mathrm{G}) &= \sum_l \log \mathrm{P}(\boldsymbol{g}^{(l)}) \\ &= \sum_l \sum_i \log[g_i^{(l)} p_i + (1 - g_i^{(l)})(1 - p_i)], \end{aligned} \tag{11}$$

where $p_i = \mathrm{P}(g_i; \mathbf{x}, u, \psi_i)$ is the probability of gating determined by the learnable gating functions.

Continuous relaxations Relaxation methods soften the discrete gate vector into a continuous vector of "activation strengths". In particular, we use Concrete random variables (CONCRETE) [66] to stand in for discrete gating during training. Concrete distributions have a temperature parameter τ where the limit $\tau \to 0$ recovers a corresponding discrete distribution. The Bernoulli distribution is replaced by the binary Concrete distribution,

$$g_i \sim \text{Sigmoid}\left(\left(Z + \log \frac{p_i}{1 - p_i}\right)^{-\tau}\right), \tag{12}$$

where $Z \sim \text{Logistic}(0, 1)$. We set $\tau = 0.1$ during training to make the network differentiable, and use $\tau = 0$ during testing to recover the desired hard-gated network.

3.3 Context-aware controller

Trained TNNs take an input and a single control parameter u. However, using a fixed control parameter at runtime may not provide optimal trade-off between performance and utilization. The goal of incorporating the context-aware controller is to adjust the control parameter for TNNs dynamically, and there are many methods to achieve this functionality in a heuristic or learned manner. In this paper, we investigate

a trainable input-dependent controller via solving a contextual bandit problem. Such a controller for the hand gesture recognition task is elaborated later, which we believe offers a generalizeable example for a wide variety of vision-based applications.

3.3.1 Input-dependent contextual bandit

A simplified approach to learn the controller is to frame it as a contextual bandit problem. Hypothetically, for a classification problem, the fact that different inputs or classes require different amount of computational resources is demonstrated in [30,31]. Based on this assumption, in order to determine what is the best utilization parameter for each input, the prediction of it can be formulated as a sub-problem using contextual bandits. To derive the optimal controller, we develop the controller network F_Θ which is parameterized by weights Θ. The controller outputs the probabilities of choosing a predefined set of actions. During training, a reward is given considering the actual FLOPS and confidence of the prediction.

3.3.2 State

Given input image or video sequence x, the state representation is simply defined by the input itself as s. The goal of the controller is to receive current input signal and select actions in a way that maximizes rewards for all similar input signals. It will learn appropriate utilization parameters u for different states. The controller network will extract a dense feature representation of input, as well as produce the probabilities for taking each action. The state reflects the contextual information associated with input x.

3.3.3 Action

The action is the selection of the utilization parameter u from the action set $\mathcal{U}$. Let $K = |\mathcal{U}|$ denote the size of the action set. The actions resemble the definition of "arms" in multi-armed contextual bandit problems. The learner pulls an arm according to information provided by input x. As a result, the prediction of an action is context-dependent. The size of the action set depends on how u is discretized, namely the step between the adjacent selections of utilization. As long as the number of possible actions is determined, the action set is defined as $\mathcal{U} = \{u_k = \frac{k}{K} \,|\, k = 1, \ldots, K\}$. There can be as many as actions in the set, but K should correlate with the cardinality D of the throttleable blocks. Larger K requires larger D to differentiate TNN sub-models if the differences among actions are small. The context-aware controller is relatively small, it is much more difficult to train with larger K. For the rest of the paper, we simply set $K = 10$ since we have at most 16 filters in each TB.

3.3.4 Reward

Only the throttleable blocks that are not gated off according to u and gating strategy will be evaluated in the forward pass. The actual percentage of computations over the maximum of being full-throttled does not always equal to the utilization. Therefore, the true utilization is measured in the forward pass where the ratio of actual number over the maximum of Multiply-Accumulate Operations (MACs). One can also simply use u as an approximation for simplicity which is shown below. Given the prediction $\hat{\mathsf{y}}$ with confidence c (softmax of the prediction logits) from the TNN, the reward of taking u is defined as:

$$r(u|\mathsf{y}, \hat{\mathsf{y}}, c) = \begin{cases} \exp(1-u) \cdot (1-u) & \text{if } \mathsf{y} = \hat{\mathsf{y}} \\ -(c+\gamma_1) \cdot (u+\gamma_2) & \text{otherwise,} \end{cases} \tag{13}$$

where γ_1 and γ_2 are constant parameters for balancing the reward. Larger γ_1 gives more penalty if we use more computations, and larger γ_2 gives more penalty if the network is too confident of a wrong prediction. Their values are empirically selected so that no over-optimization occurs during training, which may make the controller only predict the same utilization value. In experiments, they are empirically set to 0.5 and 1.5 respectively. Such design of reward encourages to achieve higher accuracy while retaining low utilization. And more penalty is given if the prediction is wrong but takes a high utilization parameter.

3.3.5 Optimization

Formally, we define the policy of choosing the utilization as $\pi_\Theta(u_k|\mathsf{x}) = [\mathrm{F}_\Theta(\mathsf{x})]_k$, where Θ denotes the learnable weights of the controller network F_Θ, and $[\mathrm{F}_\Theta(\mathsf{x})]_k \in [0, 1]$ is the k-th entry of the output vector produced by the controller network which represents the probability of choosing the particular utilization u_k. Then the utilization is determined by the optimal action with the highest probability such as

$$u^* = \mathrm{argmax}_{u_k \in \mathcal{U}} \, \pi_\Theta(u_k|\mathsf{x}). \tag{14}$$

In order to encourage exploration, ϵ-greedy is applied where a random utilization is selected with probability ϵ. The contextual bandit network is trained via a policy gradient approach [67] that maximize the expectation of rewards. The loss function for training the controller is defined as

$$\mathcal{L}_\Theta = -r(u) \log \pi_\Theta(u|\mathsf{x}), \tag{15}$$

where the conditions of the reward are omitted for the simplicity of presentation. There are other designs of such a context-aware controller network. For example, we can

Algorithm 1 Context-aware TNN training

Input: Training set $\mathbb{X} = \{(\mathrm{x}_n, \mathrm{y}_n)\}_{n=1}^{N}$
Output: TNN T_Φ, Controller network F_Θ
Let $u \leftarrow u_0$, step Δu, learning rates α_1, α_2
for iteration $\leftarrow 1, 2, \cdots, \mathcal{N}_{\text{TNN}}$ **do** ▷ TNN training
  $(\mathrm{x}, \mathrm{y}) \leftarrow$ Sample data from $\mathbb{X}$
  $\hat{\mathrm{y}} \leftarrow \mathrm{T}_\Phi(\mathrm{x}, u)$
  $\Phi \leftarrow \Phi - \alpha_1 \nabla \mathcal{L}_{task}(\mathrm{y}, \hat{\mathrm{y}})$
  if reach the iteration of increasing u **then**
    $u \leftarrow u + \Delta u$
  end if
end for
Freeze parameters of T_Φ ▷ Controller training
for iteration $\leftarrow 1, 2, \cdots, \mathcal{N}_{\text{controller}}$ **do**
  $(\mathrm{x}, \mathrm{y}) \leftarrow$ Sample data from $\mathbb{X}$
  $u \leftarrow$ Forward F_Θ via $\epsilon-$greedy
  $\hat{\mathrm{y}}, c \leftarrow$ Forward T_Φ
  $r \leftarrow$ Compute via Eq. 13
  $\Theta \leftarrow \Theta - \alpha_2 \nabla \mathcal{L}_\Theta$
end for
return $\mathrm{T}_\Phi, \mathrm{F}_\Theta$

utilize more complex interactions between the physical environment and the state of our system if we frame the controller as a full Markov Decision Process. In our formulation of the controller, we treat the TNN as a fixed model, but they can also be fine-tuned by end-to-end training.

3.3.6 Training and testing

Training the TNN is a two-phase procedure as discussed in Sects. 3.2.5 and 3.2.6. The context-aware controller is considered as a learnable gating function decoupled from the TNN, hence only the phase 1 TNN training (Eq. 1) is adopted via curriculum learning [68]. Specifically, the TNN is firstly trained with a low utilization, and then with a higher utilization as training epochs increases. By starting with easier tasks and less learnable parameters, training the TNN is more stable with faster convergence. After training the TNN, we freeze its parameters and train the context-aware controller via Eq. 15. The entire procedure for training a DTNN is illustrated in Algorithm 1.

One of the advantages of DTNN is that the controller is replaceable after deployment. There are many methodologies to design controllers, such as heuristic rule-based approaches [69,70], data-driven model-based approaches using reinforcement learning (RL) [28], etc. Heuristic rule-based approaches are intuitive, handcrafted, but not context-aware. Data-driven model-based approaches are capable of making context-aware decisions. For example, the controller is implemented as a much smaller neural network than the data network that takes in the input and predicts a proper control parameter. This is another advantage of a single controllable parameter as it is more feasible to learn.

4 Experiments

4.1 Experimental setup

To examine the generality of our TNN concept, we implemented throttleable convolutional and fully connected layers to directly replace the vanilla layers. Then, we created throttleable variants of several popular CNN architectures. All experiments are implemented with the same default architecture and training hyper-parameters for each dataset unless explicitly mentioned. We use suffix to indicate specific gating strategies such as "-W" for WIDTHWISE, "-D" for DEPTHWISE, "-N" for NESTED and "-WN" for WIDTHWISE NESTED gating, *etc*let@tokeneonedot. All experiments are implemented using PyTorch [71] (versions 0.3.1, 0.4, 1.0 and 1.3), and run on Nvidia GTX 2080 Ti or GPUs. We generate all of the performance curves by evaluating each model on the full test set using fixed values of u. We refer to the vanilla model that do not have gating applied during training as the baseline. Each data point in each chart is the result of a single evaluation run for a single instance of the trained model.

VGG-W VGG [72] is a typical example of a "single-path" CNN. We apply WIDTHWISE gating with concatenation aggregation to groups of convolutional filters in each layer and combine the outputs by concatenation. Because VGG lacks skip-connections, we enforce that at least one group must be active in each layer, to avoid making the output zero.

ResNet-D We also experimented with a depthwise-gated version of standard ResNet with summation aggregation, similar to BlockDrop and SkipNet [30,48]. In this architecture, a throttleable block is converted from an entire ResNet block that can be skipped when gated off.

ResNeXt-W ResNeXt [73] is a modification of ResNet [57] that structures each ResNet block into groups of filters then aggregates the results by summation. We created a widthwise-gated version of ResNeXt ("ResNeXt-W") by considering each group as a throttleable block.

DenseNet-D/-W In the DenseNet architecture [63], each dense block contains multiple narrow layers that are combined via concatenation. These narrow layers make natural units for gating. We view this architecture as both widthwise and depthwise gating due to the nature of dense blocks and skip connections.

C3D-W The throttleable block used in C3D is of the NESTED widthwise gating strategy with concatenation aggregation, which is applied to 3D convolutional and fully connected layers. We only apply the widthwise gating along the channel dimension, and it is worth noting that it can also be applied along the temporal dimension.

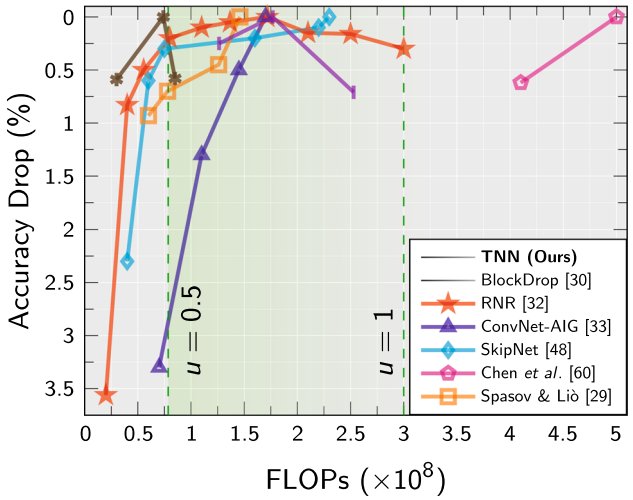


Fig. 3 Comparisons of relative accuracy drop (%) w.r.t. the peak accuracy on CIFAR-10 for DenseNet-WN with recent dynamic computation methods. Green shaded area denotes the utilization range of [0.5, 1] for the TNN, which has 3×10^8 FLOPs at $u = 1$ and 0.79×10^8 FLOPs at $u = 0.5$

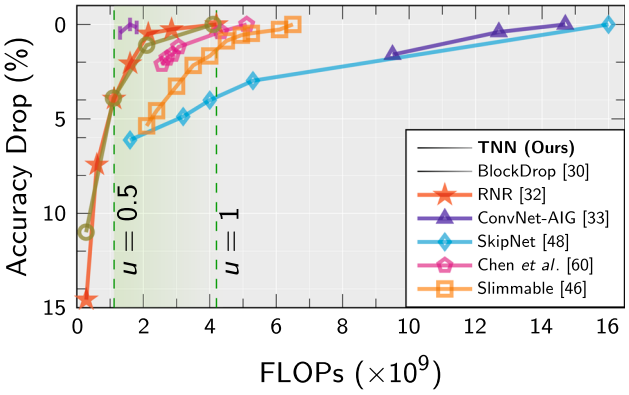


Fig. 4 Comparisons of relative accuracy drop (%) w.r.t. the peak accuracy on ImageNet for ResNeXt-WN with recent dynamic computation methods. Green shaded area denotes the utilization range of [0.5, 1] for the TNN, which has 4.2 GFLOPs at $u = 1$ and 1.1 GFLOPs at $u = 0.5$

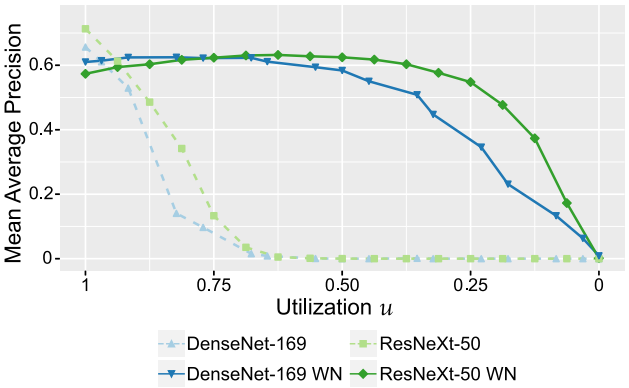


Fig. 5 TNNs are robust to test-time dropout for object detection on VOC2007 using Faster R-CNN with throttleable "backbones"

4.2 Image classification task

4.2.1 CIFAR-10

We evaluate the proposed TNNs on CIFAR-10 [74] and ImageNet [75] datasets following the standard settings of dataset split and preprocessing [57,72,74,75], and report the corresponding top-1 classification accuracy.

The TNNs are based on following architectures: DenseNet-WN, the DenseNet-BC (a compressed DenseNet) with 3 dense blocks with a growth rate $k = 12$, where each dense block is of WIDTHWISE NESTED gating with cardinality $D = 16$; ResNeXt-WN, the ResNeXt-50 architecture as described in [73] with cardinality $D = 16$ in each of the 4 stages; VGG, the VGG-D architecture truncated to the first 3 convolution stages followed by a 4096-dimensional fully connected layer, where all three convolutional stages and the fully connected layer are throttleable with cardinality $D = 16$. The learnable gating function (REINFORCE or CONCRETE) is implemented as a multilayer perceptron (MLP) network (FC $\rightarrow$ ReLU $\rightarrow$ FC) that maps the control input u to gate vectors $\boldsymbol{g}$ for each throttleable block. We show results for the C^2_{dist} complexity penalty (Eq. 6) where $p = 2$ and $\lambda = 10$.

We compare DenseNet-WN with some state-of-the-art models described in Sect. 2 on CIFAR-10. Due to the diversity of the ideas, implementations, and reported metrics, we only use methods that report computation complexity (e.g., FLOPs) and accuracy across multiple operational points for comparison. The FLOPs for other methods are directly obtained from their papers. For every operational point of each method, we compute its FLOPs and accuracy drop between the corresponding peak accuracy, namely the increase in error. It is important to note that utilization controls the ratio of active filters instead of ratio of FLOPs (low-level filters will have more FLOPs). The FLOPs of other methods are obtained from their papers. As shown in Fig. 3, our throttleable DenseNet achieves the most robust performance across a wide range of utilization (only a few operational points are drawn for clear visualization) . We also have the largest number of operational points that can be adjusted under different computational constraints in a single model without fine-tuning. For all dynamic inference models [29–33,48,60], retraining or fine-tuning is required to obtain each individual model shown in Fig. 3.

4.2.2 ImageNet

Our second set of experiments examines image classification on the 1000-class ImageNet dataset [75] based on DenseNet-169, ResNeXt-50, and ResNet-50 architec-

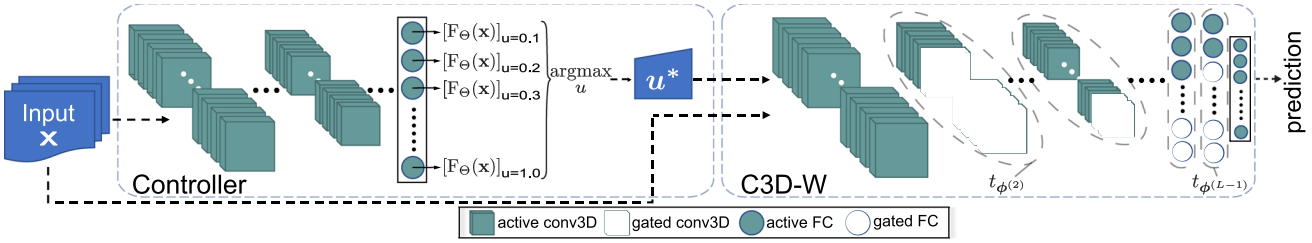


Fig. 6 A DTNN framework consisting of a lightweight context-aware controller and WIDTHWISE throttleable 3D convolutional neural network (C3D-W) for video-based hand gesture recognition. The first layer $t_{\phi^{(1)}}$ of C3D-W is non-throttleable

tures. For ImageNet, we use pre-trained weights to initialize the data path, then fine-tuned the weights with gating. In these experiments, we consider WIDTHWISE and DEPTHWISE NESTED gating. For the DEPTHWISE NESTED strategy, we repeatedly iterate through the stages of the network from output to input and turn on one additional block in each stage, unless the proportion of active residual blocks in that stage exceeds u, and stop when the total utilization exceeds u.

We compare ResNeXt-WN with methods described in Sect. 2 on ImageNet, and perform the same comparison as described in Sect. 4.2.1. As shown in Fig. 4, our TNN model still achieves the best trad-off between accuracy and efficiency across a wide range of utilization. ResNeXt-WN achieves its peak accuracy of 75.66% at utilization $u = 1$ (4.2G FLOPs), and 71.72% at $u = 0.5$ (1.13G FLOPs). As comparisons, Slimmable [46] achieves similar peak accuracy of 76%, but degenerates more with only four pre-defined operational points; the best pruned model from Chen et al. [60] achieves 68.62% accuracy with 1.6G FLOPs, where our model outperforms it by 3.1% while having 0.47G less FLOPs.

4.3 Object detection

We experiment TNNs for object detection task on the PASCAL-VOC2007 dataset [76]. To create a throttleable object detector, Faster R-CNN [77] is adopted without changing its hyper-parameters and the backbone network is converted into a TNN. We use WIDTHWISE NESTED DenseNet-169 and ResNeXt-50 in this experiment. Following the approach of [57] for combining ResNet with Faster R-CNN, we use the TNN as the feature extractor, followed by a region proposal network (RPN) and a detection "head" for object proposal classification and regression. The vanilla models are trained on ImageNet and then fine-tuned on VOC2007. The throttleable models are pre-trained TNNs on ImageNet and fine-tuned on VOC2007 with uniform sampling of u.

We evaluate the above-mentioned models and report the mean average precision (mAP [.5, .95]) in Fig. 5. Similar to the results on image classification, we observe that the baseline methods degenerate quickly when any gating is applied and reduce to zero at $u = 0.55$ approximately. For throttleable DenseNet and ResNeXt, the detection performance is well maintained. DenseNet-169-WN achieves 0.61 mAP at $u = 1$ and 0.58 mAP at $u = 0.5$, saving approximately 60% FLOPs with only 0.03 mAP drop. ResNeXt-50-WN achieves even higher mAP in a broader range of utilization (0.6 mAP at $u = 0.375$). Though the TNN have little lower peak MAP, it degrades more gracefully and achieves exceptional trade-offs between performance and computations. TNNs achieve higher tolerance to test-time dropout because of learning robust features under different utilization levels.

4.4 Video-based hand gesture recognition

To showcase the TNN architecture for more complex vision applications, we implemented a TNN architecture based on C3D [78] to perform hand gesture classification referred as C3D-W. It takes a fixed number of frames and a control parameter as input and outputs the gesture classification results. The controller is a deep contextual bandit network that follows the design discussed above. The proposed DTNN framework is shown in Fig. 6.

C3D-WN: We implement a C3D-W architecture with WIDTHWISE NESTED gating strategy. Experiments are performed for the hand gesture recognition task on the 20BN-JESTER dataset [79]. There are a total of 148,092 videos of which 118,562, 14,787 and 14,743 are in training, validation and test set, respectively. Since the ground-truth of test dataset is not publicly available, we use the validation dataset for testing and report the results. The class distribution of the dataset is well balanced except for the class `Doingotherthings` which represents activities other than hand gestures. We resize the input video spatial size (height × width) to 100 × 160, and sample 16 continuous frames with random starting index from each video. For C3D-W training, we use four Nvidia RTX 2080 Ti graphic cards with the batch size of 20 for each GPU. Adam optimizer [80] is applied with an initial learning rate of 5×10^{-5}, $\beta_1 = 0.5$ and $\beta_2 = 0.999$. The learning rate is reduced by 10 with the patience of 3 epochs if the loss is not decreasing. The

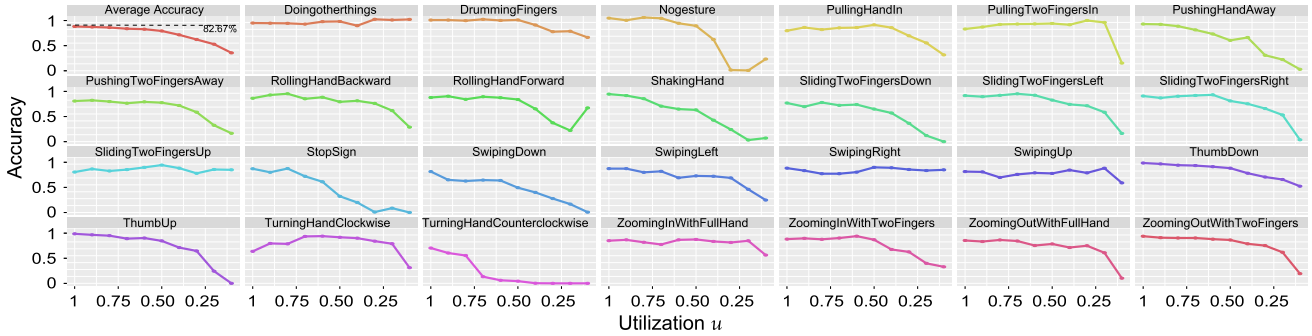
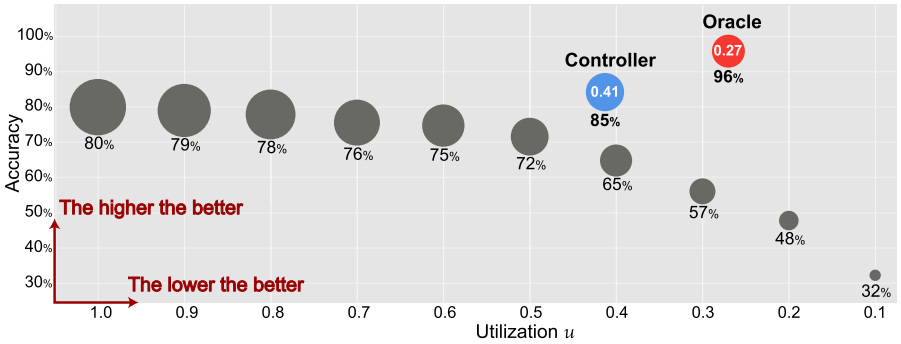
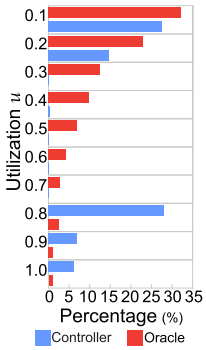


Fig. 7 Classification accuracy on validation set over utilization parameter u for each gesture class. The top-left facet shows the average accuracy of 81.10% across all classes ,while a vanilla C3D achieves an accuracy of 82.67%



(a) Utilization *vs*. accuracy for the learned controller, performance oracle, and fixed u configurations. The area of a circle denotes the average utilization, and the accuracy for each configuration is shown below the circle.



(b) A comparison of utilization distributions.

Fig. 8 Hand gesture recognition results on 20BN-Jester validation set (as test set). With the context-aware controller, DTNN achieves the best accuracy–computation trade-off comparing to TNN with fixed utilization

network is trained for 20 epochs with the curriculum learning schedule where u is set to 0.1, and then increased by 0.1 every two epochs. No further fine-tuning is carried out for training the TNN.

4.4.1 Context-aware controller

We experiment a 3D version of ShuffleNet architecture [3,81] as the controller network. The specifications of the DTNN are presented in the Appendix Tables 4 and 5. The design of the controller network is computation-efficient such that the total number of parameters is only 0.955M with 0.255G MACs. Note that the controller network is not a TNN architecture. For controller training, we use a smaller batch size of 16 for handling both the data path and controller network at the same time. We use RMSprop optimizer [82] with a learning rate of 1×10^{-7}. The controller is trained for 10 epochs while freezing the parameters of the data path.

Figure 7 shows the various utilization levels for each gesture class. We verify our earlier hypothesis (Sect. 3.3.1) that some classes require more resources than others. For instance, the classification performance remains almost constant for each tested utilization parameter u for `Doing otherthings`, while it is linear with u for `Shakinghand`. This demonstrates that we can use less resources (e.g., set $u = 0.1$) and still achieve high classification accuracy for `Doingotherthings`. Practically, DTNN can run with low utilization ("idle") when detecting non-gesture activities, and with higher utilization when detecting gestures ("throttle"). Given this trained TNN, the classification results of each test video for every u parameter are collected. Then a performance oracle is derived by collecting the lowest u where the TNN makes the correct prediction for each testing data. The oracle has an accuracy of 96.14% and an average utilization of 0.27. It is important to note that the performance oracle is just an ideal case for a particular TNN which is impossible to achieve.

The effectiveness of this context-aware controller is confirmed with Fig. 8a. The learned controller achieves an accuracy of 85.24% with an average utilization of 0.41, which outperforms the vanilla C3D by 2.57%. It manifests that the controller learns the input-specific utilization for each input. In Fig. 8b, by comparing utilization distributions with the oracle, the controller learns a sparse selection of u where $u = 0.1$, $u = 0.2$ and $u = 0.8$ are chosen more often. Although the controller has 0.14 more utilization on average

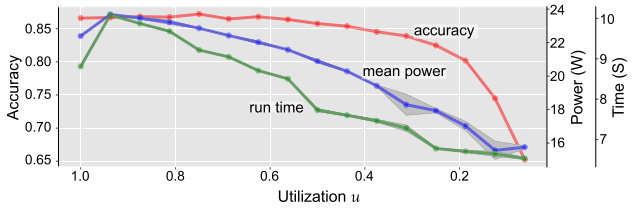


Fig. 9 Measured throttling performance on NVIDIA Jetson AGX Xavier

and 10.90% accuracy drop compared to the oracle, it outperforms all the fixed utilization configurations as well as the vanilla C3D. It further demonstrates how the system can operate within different utilization levels with a learned control policies. Regarding the time of processing a 16-frame clip, we compute the number of parameters, MACs and inference time (clip per millisecond) as shown in Table 6. The controller adds a very little computation cost (0.52% more MACs), but reduces much more overall computations by dynamically predicting per-input utilization for the TNN.

4.5 Hardware implementation

We examine the performance of TNN on embedded systems. We implement and measure the computational benefits of TNNs on a NVIDIA Jetson AGX Xavier processor [83], using a throttleable VGG-WN model trained on the CIFAR-10 dataset. Relative savings in runtime and power rather than actual peak values are shown because the results highly depend on the provided hardware mechanisms. We replace the convolution operations in VGG with throttleable 2D convolution by tensor slicing described in Algorithm 2 where $\mathbf{W}$ and $\mathbf{b}$ are the weights and bias for the convolution kernel. To perform a gated convolution, we form truncated weights and bias tensors by removing channels corresponding to nonactive groups, then perform an ordinary convolution with the truncated weights, and finally restore the output to its original shape by padding with zeros. In the case of NESTED gating, the implementation is highly efficient, requiring only two tensor slice operations and one concatenation.

Figure 9 shows accuracy, mean power draw, and runtime for classifying the entire CIFAR-10 test set on the Xavier GPU in "MAXN" power mode. The shaded regions show standard deviation of three experiments. Salient aspects of this result is the linear ramp down of the power and runtime while accuracy remains high at lower utilization settings. Compared to no throttling where $u = 1$, the TNN at $u = 0.1$ uses 70% of the power, 74% of the runtime, and 52% of the total energy (103J vs 197J). We anticipate similar gains in TNN inference efficiencies for other TNN models described in this paper. The key is having an effective control policy to guide the utilization settings to match application needs.

Algorithm 2 WIDTHWISE NESTED Conv2D

Function: Conv2D-WN($\mathbf{x}$, u, $\mathbf{W}$, $\mathbf{b}$)
Let $C \leftarrow$ OUT_CHANNELS($\mathbf{W}$)
Let $n \leftarrow \lceil u \cdot C \rceil$
$\hat{\mathbf{W}}, \hat{\mathbf{b}} \leftarrow \mathbf{W}[0:n], \mathbf{b}[0:n]$ ▷ Slice parameters
$\hat{\mathbf{y}} \leftarrow$ CONV2D($\mathbf{x}$, $\hat{\mathbf{W}}$, $\hat{\mathbf{b}}$)
$\mathbf{z} \leftarrow$ ZEROS($n - C$, size($\hat{\mathbf{y}}$))
$\mathbf{y} \leftarrow$ CONCATENATE($\hat{\mathbf{y}}$, $\mathbf{z}$) ▷ Pad with zeros
return $\mathbf{y}$

4.6 Analysis

TNNs enable a general framework for conditional computation, whereby the overall computational load and model accuracy can be determined dynamically at inference time. In this section, we offer in depth analysis of our results to best evaluate the proposed architecture and methodology.

4.6.1 Ablation study on gating strategies

The experimental results on CIFAR-10 are shown in Fig. 10. The most noticeable result is that all TNNs are much more robust to various utilization configurations, while the naïve models degrade dramatically (less than 50% accuracy when applying 25% dropout). By comparing the three architectures, DenseNet-DW achieves the best peak accuracy of 91.19% at $u = 0.8125$, and maintains the accuracy over 91% in the utilization range of [0.5, 1]. The other two TNN architectures also demonstrate strong and consistent performance in the same utilization range.

Among all gating strategies, NESTED gating substantially outperforms all variations over INDEPENDENT for all 3 architectures. The difference is especially pronounced for VGG, and we attribute this to that VGG or similar architectures learn more "entangled" representations than architectures with skip connections, which could make it more sensitive to exactly which transformations are gated off. For depth-wise gating, the performance difference between applying NESTED and INDEPENDENT is smaller than it is for width-wise. This observation indicates that there are more dependencies among features when using width-wise gating. Depth-wise gating is more tolerant of losing features when applying INDEPENDENT due to the short connection between adjacent TBs.

Among models with INDEPENDENT gating strategy, learnable gating models (REINFORCE [84] and CONCRETE [66]) are consistently better than random gating. By training with INDEPENDENT gating, the TNN is robust to different levels of utilization. With learnable gating functions, the learned gating pattern achieves better performance by allocating computation non-uniformly across different stages of the network (shown in Fig. 11). Blocks in the later stages (higher-numbered) are used preferentially over components in earlier

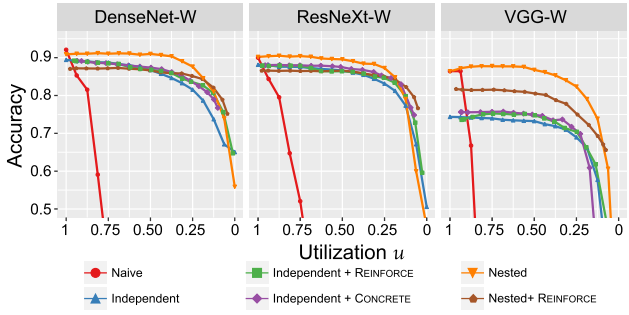


Fig. 10 Comparison of classification accuracy with different gate control methods for three standard CNN architectures on the CIFAR-10 dataset

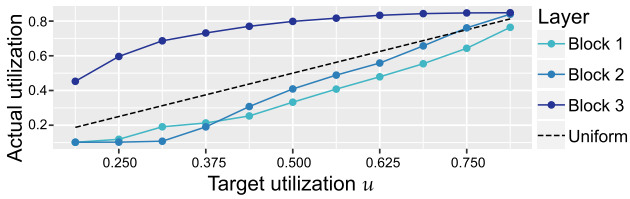


Fig. 11 Learned gating pattern for selected blocks of DenseNet-DW on CIFAR-10 with the REINFORCE training. The dotted line shows uniform utilization

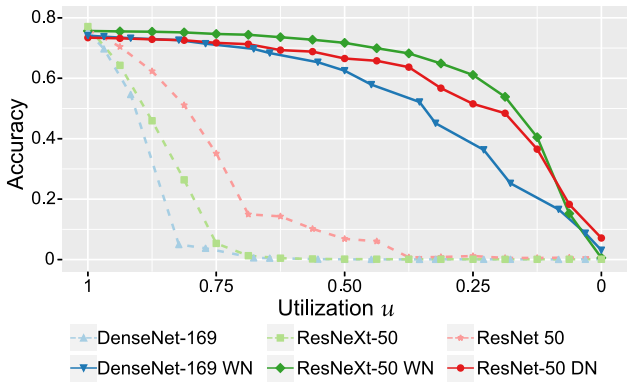


Fig. 12 Comparisons of results between throttleable and vanilla architectures for image classification on ImageNet-1K

stages. Note that the learned gating functions do not cover the entire range of possible utilization [0, 1]. The useful range of u is larger for higher λ and for complexity penalties with $p = 1$. We observe that learnable gating functions do not outperform fixed gating since it is hard to train with very few learnable parameters using MLPs. Thus, we derive the context-aware controller for improving the results by decoupling the control from TNNs.

The ImageNet classification results are shown in Fig. 12. All TNNs are smoothly throttleable through the full range of utilization whereas the pre-trained models degrade rapidly with increased throttling. The ResNeXt-50 model is the best in terms of both peak accuracy (75.66% at $u = 1$) and area-under-curve. It maintains at least 71% accuracy in the utilization range of [0.5, 1].

4.6.2 Full-throttle TNN versus Vanilla architecture

To evaluate and emphasize the effectiveness of TNNs, we summarize and show the results comparisons on CIFAR-10 between the TNNs and the corresponding vanilla architectures in Table 2. We consider the vanilla architectures as the baselines, and all the throttleable variants are applied with NESTED gating strategy without learnable gates. Applying a 50% dropout on the vanilla architecture (Vanilla $u = 0.5$) will result in catastrophic accuracy drop. Instead, TNNs at $u = 0.5$ only have a relative decrease within 1.1%. As for the full-throttle models, we observe an increase of 0.21% for ResNeXt-W. Remarkably, TNNs can achieve the peak performance at a lower utilization instead of full-throttling. The peak accuracy of TNNs is competitive or even superior to the baselines'. For example, the accuracy on CIFAR-10 for VGG-W at $u = 0.75$ has 1.36% accuracy improvement, and for ResNeXt-W at $u = 0.88$ improves by 0.51%. The trivial performance difference between the full-throttle TNN and baseline reveals that most CNN architectures can be converted into throttleable ones while maintaining the peak performance.

It can be observed that generic TNN architectures achieve better performance. VGG consists of several basic convolutional layers; ResNeXt consists of groups of convolutional layers; and DenseNet consists of convolutional layers that are connected with previous layers. Without more complex designs of gating strategies, we expect that VGG and ResNeXt perform better than DenseNet as shown in Table 2. It is worth noting that no fine-tuning or data augmentation is applied, and we can always fine-tune a trained TNN at any level of utilization. More importantly, the task performance can be retained or even improved with lower utilization as demonstrated in Table 2.

4.6.3 Controller efficacy

For experiments discussed in Sect. 4.4. The controller network only has 3% parameters and 0.05% MACs compared with the TNN. The detailed architectures and their computational costs are shown in the Appendix A.2.

4.6.4 Advantages and potential applications

One important benefit of the proposed DTNN is to leverage different groups of features in a single architecture for robust prediction, which is controlled dynamically by a single utilization parameter. Decoupling DTNN into two modules alleviates the training difficulty, and enables more flexible architectures and applications. In a TNN, how much to throt-

Table 2 Comparisons of accuracy (%) on CIFAR-10 between full-throttle TNNs and vanilla architectures

	Vanilla	Vanilla $u = 0.5$	TNN $u = 0.5$	TNN $u = 1.0$	Peak	u
DenseNet-W	92.09	0.10	90.99	90.89	91.19	0.81
	(0.00)	*(91.99)*	*(1.10)*	*(1.20)*	*(0.90)*	
ResNeXt-W	89.99	0.10	89.55	90.20	90.50	0.88
	(0.00)	*(89.89)*	*(0.44)*	**(0.21)**	**(0.51)**	
VGG-W	86.43	0.09	86.77	86.39	87.79	0.75
	(0.00)	*(86.34)*	**(0.34)**	*(0.04)*	**(1.37)**	

For each architecture, the second row shows the relative change (**increase** or *decrease*) compared to the vanilla baseline. The peak performance of a TNN could be achieved with a lower utilization

tle and how to throttle are also disentangled while existing approaches [28,30–32,48] focus on more complex methods that are not practical in real-world deployment.

Having a single control signal u also allows us to enable dynamic throttling to additional constraints beyond those are presented during training. For example, a deployed application may run differently based on environmental conditions (such as battery charge, illumination levels, temperature, *etc*let@tokeneonedot), as a result, may require alternative operating conditions for the TNN. Because of decoupled design of DTNN, we can still throttle TNN based on application inputs. For example, a system with low battery charge may impose a lower utilization u, and thereby dynamically adjust the quality of services based on system capability. Moreover, the controller behavior can be changed to any other handcrafted or learnable policies at any time. This modularized design of DTNN offers a user-friendly and domain-agnostic learning system for a wide range of real-world applications such as the presented video-based hand gesture recognition, object detection and tracking, video analytics and monitoring.

5 Conclusion

In this paper, we presented a novel run-time dynamically *throttleable* neural network (DTNN), as an adaptive model with flexible topology whose performance can be varied dynamically to produce a range of trade-offs between task performance and resource consumption. A DTNN is composed of a throttleable neural network and a contextual controller for dynamically adjusting TNN's inference path. We designed TNNs using throttleable blocks that can be activated and deactivated during inference time. A separately trained context-aware controller which is capable of input-dependent resource management was implemented. Comprehensive results on image classification and object detection show that TNNs can be effectively throttled across a range of operational points, while having peak accuracy comparable to their vanilla architectures. The experimental results on hand gesture recognition task demonstrate that the proposed DTNN achieves dynamic execution of TNNs with a context-aware controller, outperforming the vanilla architecture and all fixed configurations of utilization.

Appendix A

A.1 FLOPs calculation

We use the codes from [85] for computing the FLOPs. Table 3 summarizes how to compute the FLOPs for some common layers:

For computing FLOPs of a convolutional layer in the TNN, we count how many activated kernels within each layer, and the FLOPs are calculated as 2 × No. activated kernels × kernel shape × output shape. For fully connected layers, the input size will change, and the FLOPs are calculated as 2 × activated input size × output size.

Table 3 FLOPs calculation for common layers

Layer	FLOPs
Convolutional layer	2 × No. kernels × kernel shape × output shape
Fully connected layer	2 × input size × output size
2D Pooling layer	2 × No. output channels × output size

A.2 Detailed architectures

Table 4 presents the exact specification of the C3D-W used for hand gesture recognition. The conv-block consists of a 3D convolutional layer with ReLU activation. Stride of all convolutional layers is $1 \times 1 \times 1$. Stride of max-pool-1 is $1 \times 2 \times 2$,

Table 4 Detailed architectures of the DTNN for video-based hand gesture recognition on the Jester dataset

Layer	Kernel size	C	D	Output size
conv1	3 × 3 × 3	64	1	16 × 100 × 160
MaxPool	1 × 2 × 2	64	1	16 × 50 × 80
conv2	3 × 3 × 3	128	16	16 × 50 × 80
MaxPool	2 × 2 × 2	128	1	8 × 25 × 40
conv3	3 × 3 × 3	256	32	8 × 25 × 40
conv4	3 × 3 × 3	256	32	8 × 25 × 40
MaxPool	2 × 2 × 2	256	1	4 × 12 × 20
conv5	3 × 3 × 3	512	32	4 × 12 × 20
conv6	3 × 3 × 3	512	32	4 × 12 × 20
MaxPool	2 × 2 × 2	512	1	2 × 6 × 10
conv7	3 × 3 × 3	512	32	2 × 6 × 10
conv8	3 × 3 × 3	512	32	2 × 6 × 10
MaxPool	2 × 2 × 2	512	1	1 × 3 × 5
fc1	–	512	16	1 × 1 × 1
fc2	–	512	16	1 × 1 × 1
fc_class	–	27	1	1 × 1 × 1

Sizes are expressed as *depth* × *height* × *width*.
C denotes the output number of channels.
D denotes the cardinality of a TB, and D=1 suggests a non-throttleable layer

and other max-pool layers are 2 × 2 × 2. Padding of all conv-blocks is one, and padding of all max-pool layers is zero. In total, C3D-WN has 2.654G FLOPs of non-gated operations, and 94.752G FLOPs of throttleable operations. The detailed architecture of the controller is illustrated in Table 5.

The network begins with a convolutional layer followed by 16 ShuffleNet units grouped into 3 stages (conv2_x to conv4_x). Each ShuffleNet unit is a residual block where the residual branch consists of one 1 × 1 × 1 group convolution, channel shuffle operation, 3 × 3 × 3 depthwise convolution [1], and 1 × 1 × 1 group convolution. A cost comparison of the TNN and controller is shown in Table 6, indicating that the controller is much more computationally efficient.

A.3 Class distribution of 20BN-JESTER

The class distribution of the 20BN-JESTER training set is shown in Fig. 13.

Table 6 Cost comparison between the data path network (C3D-WN) and controller (3D-ShuffleNet)

	# params. (M)	FLOPs (G)	Speed (cpms)
C3D-W	31.865	97.406	137.55
3D-ShuffleNet	0.955	0.510	17.03

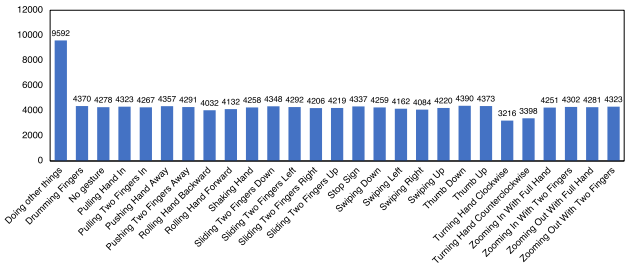


Fig. 13 Class distribution of 20BN-JESTER training set

Table 5 Contextual controller architecture based on 3D-ShuffleNet

Stage*	Kernel size	Stride	Repeat	Output size
conv1	3 × 3 × 3	1 × 2 × 2	1	24 × 16 × 50 × 80
MaxPool	3 × 3 × 3	2 × 2 × 2	1	24 × 8 × 25 × 40
conv2_x	–	2 × 2 × 2	1	240 × 4 × 13 × 20
	–	1 × 1 × 1	3	
conv3_x	–	2 × 2 × 2	1	480 × 2 × 7 × 10
	–	1 × 1 × 1	7	
conv4_x	–	2 × 2 × 2	1	960 × 1 × 4 × 5
	–	1 × 1 × 1	3	
AvgPool	1 × 4 × 5	1 × 1 × 1	1	960 × 1 × 1 × 1
fc_action	–	–	1	10 × 1 × 1 × 1

* each stage from conv2_x to conv4_x consists of one or several ShuffleNet units (the number of repetition is shown in the 4th column)

References

1. Chollet, F.: Xception: deep learning with depthwise separable convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1251–1258 (2017)
2. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: inverted residuals and linear bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4510–4520 (2018)
3. Zhang, X., Zhou, X., Lin, M., Sun, J.: Shufflenet: an extremely efficient convolutional neural network for mobile devices. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6848–6856 (2018)
4. Ma, N., Zhang, X., Zheng, H.T., Sun, J.: Shufflenet v2: practical guidelines for efficient cnn architecture design. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 116–131 (2018)
5. Yang, L., Qi, Z., Liu, Z., Liu, H., Ling, M., Shi, L., et al.: An embedded implementation of CNN-based hand detection and orientation estimation algorithm. Mach. Vis. Appl. **30**(6), 1071–1082 (2019)
6. Han, K., Wang, Y., Tian, Q., Guo, J., Xu, C., Xu, C.: GhostNet: more features from cheap operations. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1580–1589 (2020)
7. Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.J., et al.: Progressive neural architecture search. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 19–34 (2018)
8. Liu, H., Simonyan, K., Yang, Y.: DARTS: differentiable architecture search. In: International Conference on Learning Representations (ICLR) (2019). https://openreview.net/forum?id=S1eYHoC5FX
9. Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., et al.: Mnasnet: platform-aware neural architecture search for mobile. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2820–2828 (2019)
10. Cai, H., Zhu, L., Han, S.: ProxylessNAS: direct neural architecture search on target task and hardware. In: International Conference on Learning Representations (ICLR) (2019). https://openreview.net/forum?id=HylVB3AqYm
11. Cai, H., Gan, C., Wang, T., Zhang, Z., Han, S.: Once-for-all: train one network and specialize it for efficient deployment. In: International Conference on Learning Representations (ICLR) (2020). https://openreview.net/forum?id=HylxE1HKwS
12. Yu, R., Li, A., Chen, C.F., Lai, J.H., Morariu, V.I., Han, X., et al.: Nisp: pruning networks using neuron importance score propagation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 9194–9203 (2018)
13. Molchanov, P., Mallya, A., Tyree, S., Frosio, I., Kautz, J.: Importance estimation for neural network pruning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 11264–11272 (2019)
14. He, Y., Liu, P., Wang, Z., Hu, Z., Yang, Y.: Filter pruning via geometric median for deep convolutional neural networks acceleration. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4340–4349 (2019)
15. Wu, H., Tang, Y., Zhang, X.: A pruning method based on the measurement of feature extraction ability. Mach. Vis. Appl. **32**(1), 1–11 (2021)
16. He, Y., Lin, J., Liu, Z., Wang, H., Li, L.J., Han, S.: Amc: automl for model compression and acceleration on mobile devices. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 784–800 (2018)
17. Tung, F., Mori, G.: Similarity-preserving knowledge distillation. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV), pp. 1365–1374 (2019)
18. Li, T., Wu, B., Yang, Y., Fan, Y., Zhang, Y., Liu, W.: Compressing convolutional neural networks via factorized convolutional filters. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3977–3986 (2019)
19. Li, Y., Gu, S., Mayer, C., Gool, L.V., Timofte, R.: Group sparsity: The hinge between filter pruning and decomposition for network compression. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 8018–8027 (2020)
20. Hashemi, S., Anthony, N., Tann, H., Bahar, R.I., Reda, S.: Understanding the impact of precision quantization on the accuracy and energy of neural networks. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), IEEE, pp. 1474–1479 (2017)
21. Zhu, C., Han, S., Mao, H., Dally, W.J.: Trained ternary quantization. In: International Conference on Learning Representations (ICLR). OpenReview.net (2017). https://openreview.net/forum?id=S1_pAu9xl
22. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., et al.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2704–2713 (2018)
23. Gong, R., Liu, X., Jiang, S., Li, T., Hu, P., Lin, J., et al.: Differentiable soft quantization: bridging full-precision and low-bit neural networks. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV), pp. 4852–4861 (2019)
24. Dong, Z., Yao, Z., Gholami, A., Mahoney, M.W., Keutzer, K.: Hawq: Hessian aware quantization of neural networks with mixed-precision. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV), pp. 293–302 (2019)
25. Jimmy Ba, B.F.: Adaptive dropout for training deep neural networks. In: Advances in Neural Information Processing Systems (NeurIPS), pp. 3084–3092 (2013)
26. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. (JMLR) **15**(1), 1929–1958 (2014)
27. Riquelme, C., Tucker, G., Snoek, J.: Deep Bayesian bandits showdown: an empirical comparison of bayesian deep networks for Thompson sampling. In: International Conference on Learning Representations (ICLR) (2018). https://openreview.net/forum?id=SyYe6k-CW
28. Liu, L., Deng, J.: Dynamic deep neural networks: optimizing accuracy-efficiency trade-offs by selective execution. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
29. Spasov, P.L.: Dynamic neural network channel execution for efficient training. In: British machine vision conference (BMVC) (2019)
30. Wu, Z., Nagarajan, T., Kumar, A., Rennie, S., Davis, L.S., Grauman, K., et al.: Blockdrop: dynamic inference paths in residual networks. In: Proceedings of the IEEE/CVF Conference on Conference on Computer Vision and Pattern Recognition (CVPR), pp. 8817–8826 (2018)
31. Chen, Z., Li, Y., Bengio, S., Si, S.: You look twice: GaterNet for dynamic filter selection in CNNs. In: Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR), pp. 9172–9180 (2019)
32. Rao, Y., Lu, J., Lin, J., Zhou, J.: Runtime network routing for efficient image classification. IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI) **41**(10), 2291–2304 (2018)

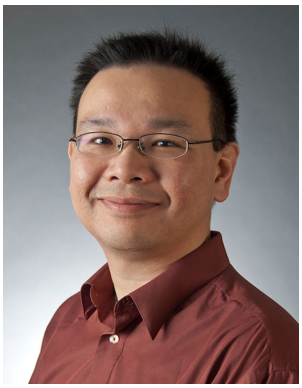
33. Veit, A., Belongie, S.: Convolutional networks with adaptive inference graphs. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 3–18 (2018)
34. Han, Y., Huang, G., Song, S., Yang, L., Wang, H., Wang, Y.: Dynamic neural networks: a survey. IEEE Trans. Pattern Anal. Mach. Intell. **30**, 9345 (2021)
35. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: imagenet classification using binary convolutional neural networks. In: European Conference on Computer Vision (ECCV), Springer, pp. 525–542 (2016)
36. Tan, M., Le, Q.: EfficientNet: rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning (ICML), pp. 6105–6114 (2019)
37. Chen, H., Wang, Y., Xu, C., Shi, B., Xu, C., Tian, Q., et al.: AdderNet: Do we really need multiplications in deep learning? In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1468–1477 (2020)
38. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. In: International Conference on Learning Representations (ICLR) (2017). https://openreview.net/forum?id=r1Ue8Hcxg
39. Wu, B., Dai, X., Zhang, P., Wang, Y., Sun, F., Wu, Y., et al.: Fbnet: hardware-aware efficient convnet design via differentiable neural architecture search. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 10734–10742 (2019)
40. Nayak, P., Zhang, D., Chai, S.: Bit efficient quantization for deep neural networks (2019). arXiv preprint arXiv:1910.04877
41. Dinh, T., Melnikov, A., Daskalopoulos, V., Chai, S.: Subtensor quantization for mobilenets. In: Bartoli, A., Fusiello. A. (eds.), European Conference on Computer Vision Workshops (ECCVW), vol. 12539, Lecture Notes in Computer Science, Springer, pp. 126–130 (2020). https://doi.org/10.1007/978-3-030-68238-5_10
42. Wiedemann, S., Müller, K.R., Samek, W.: Compact and computationally efficient representation of deep neural networks. IEEE Trans. Neural Netw. Learn. Syst. **31**(3), 772–785 (2020)
43. Gysel, P., Pimentel, J., Motamedi, M., Ghiasi, S.: Ristretto: a framework for empirical study of resource-efficient inference in convolutional neural networks. IEEE Trans. Neural Netw. Learn. Syst. (TNNLS) **29**(11), 5784–5789 (2018). https://doi.org/10.1109/TNNLS.2018.2808319
44. Ghamari, S., Ozcan, K., Dinh, T., Melnikov, A., Carvajal, J., Ernst, J., et al.: Quantization-guided training for compact TinyML models. CoRR (2021). arXiv:2103.06231
45. Ahn, C., Kim, E., Oh, S.: Deep elastic networks with model selection for multi-task learning. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV), pp. 6529–6538 (2019)
46. Yu, J, Yang, L., Xu, N., Yang, J., Huang, T.: Slimmable neural networks. In: International Conference on Learning Representations (ICLR) (2019). https://openreview.net/forum?id=H1gMCsAqY7
47. Kim, E., Ahn, C., Oh, S.: NestedNet: Learning nested sparse structures in deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 8669–8678 (2018)
48. Wang, X., Yu, F., Dou, Z.Y., Darrell, T., Gonzalez, J.E.: Skipnet: learning dynamic routing in convolutional networks. In: Proceedings of the European Conference on Computer Vision (ECCV), pp. 409–424 (2018)
49. Bengio, Y.: Deep learning of representations: looking forward. In: International Conference on Statistical Language and Speech Processing. Springer, pp. 1–37 (2013)
50. Figurnov, M., Collins, M.D., Zhu, Y., Zhang, L., Huang, J., Vetrov, D., et al.: Spatially adaptive computation time for residual networks. In: Proceedings of the IEEE/CVF Conference on Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, pp. 1790–1799 (2017)
51. Teerapittayanon, S., McDanel, B., Kung, H.: Branchynet: Fast inference via early exiting from deep neural networks. In: International Conference on Pattern Recognition (ICPR), IEEE, pp. 2464–2469 (2016)
52. Li, Z., Yang, Y., Liu, X., Zhou, F., Wen, S., Xu, W.: Dynamic computational time for visual attention. In: International Conference on Computer Vision Workshops (ICCVW), IEEE, pp. 1199–1209 (2017)
53. Ruiz, A., Verbeek, J.: Adaptative inference cost with convolutional neural mixture models. In: The IEEE International Conference on Computer Vision (ICCV) (2019)
54. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., et al.: Outrageously large neural networks: the sparsely-gated mixture-of-experts layer. In: International Conference on Learning Representations (ICLR) (2017). https://openreview.net/pdf?id=B1ckMDqlg
55. Teja Mullapudi, R., Mark, W.R., Shazeer, N., Fatahalian, K.: Hydranets: specialized dynamic architectures for efficient inference. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 8080–8089 (2018)
56. Huang, G., Sun, Y., Liu, Z., Sedra, D., Weinberger, K.Q.: Deep networks with stochastic depth. In: European Conference on Computer Vision (ECCV) (2016)
57. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE/CVF Conference on Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
58. Pham, H., Guan, M., Zoph, B., Le, Q., Dean, J.: Efficient neural architecture search via parameter sharing. In: International Conference on Machine Learning (ICML), pp. 4092–4101 (2018)
59. Gao X, Zhao Y, Łukasz Dudziak, Mullins R, zhong Xu C. Dynamic channel pruning: feature boosting and suppression. In: International Conference on Learning Representations (ICLR) (2019). https://openreview.net/forum?id=BJxh2j0qYm
60. Chen, Z., Xu, T.B., Du, C., Liu, C.L., He, H.: Dynamical channel pruning by conditional accuracy change for deep neural networks. IEEE Trans. Neural Netw. Learn. Syst. (TNNLS) **32**(2), 799–813 (2021). https://doi.org/10.1109/TNNLS.2020.2979517
61. Odena, A., Lawson, D., Olah, C.: Changing model behavior at test-time using reinforcement learning. In: International Conference on Learning Representations Workshops (ICLRW) (2017)
62. Girshick, R.: Fast r-cnn. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1440–1448 (2015)
63. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings of the IEEE/CVF Conference on Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2261–2269 (2017)
64. Bengio, Y., Léonard, N., Courville, A.: Estimating or propagating gradients through stochastic neurons for conditional computation (2013). arXiv preprint arXiv:1308.3432
65. Peng, J., Bhanu, B.: Closed-loop object recognition using reinforcement learning. IEEE Trans. Pattern Anal. Mach. Intell. (TPAMI) **20**(2), 139–154 (1998)
66. Maddison, C.J., Mnih, A., Teh, Y.W.: The concrete distribution: a continuous relaxation of discrete random variables. In: International Conference on Learning Representations (ICLR) (2017)
67. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems (NeurIPS), pp. 1057–1063 (2000)
68. Bengio, Y., Louradour, J., Collobert, R., Weston, J.: Curriculum learning. In: International Conference on Machine Learning (ICML), pp. 41–48 (2009)

69. Tann, H., Hashemi, S., Bahar, R., Reda, S.: Runtime configurable deep neural networks for energy-accuracy trade-off. In: Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, ACM, p. 34 (2016)
70. Ganapathy, S., Venkataramani, S., Sriraman, G., Ravindran, B., Raghunathan, A.: DyVEDeep: dynamic variable effort deep neural networks. ACM Trans. Embed. Comput. Syst. (TECS) **19**(3), 1–24 (2020)
71. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al.: PyTorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. (NeurIPS) **32**, 8026–8037 (2019)
72. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: International Conference on Learning Representations (ICLR) (2015)
73. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. In: Proceedings of the IEEE/CVF Conference on Conference on Computer Vision and Pattern Recognition (CVPR), pP. 5987–5995 (2017). https://github.com/facebookresearch/ResNeXt
74. Krizhevsky, A., Hinton, G.: Learning multiple Layers of Features from Tiny Images. University of Toronto, Department of Computer Science (2009). https://www.cs.toronto.edu/~kriz/cifar.html
75. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., et al.: Imagenet large scale visual recognition challenge. Int. J. Comput. Vis. (IJCV) **115**(3), 211–252 (2015)
76. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The PASCAL Visual Object Classes Challenge (VOC2007) Results (2007). http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html
77. Ren, S., He, K., Girshick, R., Sun, J.: Faster r-cnn: Towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems (NeurIPS) (2015)
78. Tran, D., Bourdev, L., Fergus, R., Torresani, L., Paluri, M.: Learning spatiotemporal features with 3d convolutional networks. In: Proceedings of the IEEE International Conference on Computer Vision (ICCV), pp. 4489–4497 (2015)
79. twentybn.: The 20BN-jester Dataset V1. Version: 1.0. Accessed: 8.1.2019. https://20bn.com/datasets/jester
80. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: International Conference on Learning Representations (ICLR) (2015)
81. Kopuklu, O., Kose, N., Gunduz, A., Rigoll, G.: Resource efficient 3d convolutional neural networks. In: Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops (ICCVW), pp. 0–0 (2019)
82. Hinton, G., Srivastava, N., Swersky, K.: Neural networks for machine learning lecture 6a overview of mini-batch gradient descent (2012)
83. NVIDIA.: NVIDIA Jetson AGX Xavier module. https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit
84. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach. Learn. **8**(3–4), 229–256 (1992)
85. Facebook.: fvcore. GitHub. https://github.com/facebookresearch/fvcore/blob/main/fvcore/nn/flop_count.py

Hengyue Liu received the B.S. degree from the Beijing University of Posts and Telecommunications, Beijing, China, and M.S. in electrical engineering from the University of Southern California. He is currently working towards his Ph.D in computer vision at the Visualization and Intelligent Systems Laboratory (VISLab), University of California, Riverside, CA, USA. His research interests include object detection, scene graph generation, and mobile vision.

Samyak Parajuli received B.S. in Computer Science and Data Science from the University of California, Berkeley. He is currently pursuing Master's degree in artificial intelligence at the University of California, Berkeley. His research interest lies broadly in optimizing deep learning algorithms for real-world applications.

Jesse Hostetler received his Ph.D. in computer science from Oregon State University. He is currently a research scientist at SRI International. His research interests are in reinforcement learning and adjacent spaces, especially in settings where data or computation are limited.



Sek Chai received the Ph.D. degree in electrical engineering from Georgia Institute of Technology, Atlanta, GA, USA. He is currently the CTO and co-founder of Latent AI. Previously, he was a technical director at SRI International, where he led R&D programs (government and commercial). Prior to SRI, he developed imaging / video solutions for next-gen mobile devices and home broadband products at Motorola Labs. His research interests includes embedded machine learning, computer vision, and computer architecture.

Bir Bhanu received B.S. (with Hons.) from IIT-BHU; M.E (with Distinction) from BITS (Pilani); S.M. and E.E. in electrical engineering and computer science from Massachusetts Institute of Technology, Cambridge, MA; Ph.D. in electrical engineering from the University of Southern California, Los Angeles, CA and M.B.A. from the University of California, Irvine, CA. He is the Bourns endowed University of California Presidential Chair in Engineering, the Distinguished Professor of electrical and computer engineering and the Founding Director of the interdisciplinary Center for Research in Intelligent Systems (1998–2019) and the Visualization and Intelligent Systems Laboratory (1991-) at the University of California, Riverside (UCR), CA. He is the Founding Professor of electrical engineering with UCR and served as its first Chair (1991–1994). He has been the cooperative Professor of computer science and engineering (since 1991), bioengineering (since 2006) and mechanical engineering (since 2008). Recently, he served as the Interim Chair of the Department of Bioengineering from 2014 to 2016. He also served as the Director of the National Science Foundation graduate research and training program in video bioinformatics with UCR. Prior to joining UCR in 1991, he was a Senior Honeywell Fellow with Honeywell Inc. He has published extensively and has 18 patents. His research interests include computer vision, pattern recognition and data mining, machine learning, artificial intelligence, image processing, image and video database, graphics and visualization, robotics, human–computer interactions, and biological, medical, military, and intelligence applications. Dr. Bhanu is a Fellow of IEEE, AAAS, IAPR, SPIE, NAI, and AIMBE.