Cognitive Computation (2020) 12:954–966
https://doi.org/10.1007/s12559-020-09734-4

Why Should We Add Early Exits to Neural Networks?

Simone Scardapane[1] · **Michele Scarpiniti**[1] · **Enzo Baccarelli**[1] · **Aurelio Uncini**[1]

Received: 25 February 2020 / Accepted: 14 May 2020 / Published online: 17 June 2020

Abstract
Deep neural networks are generally designed as a stack of differentiable layers, in which a prediction is obtained only after running the full stack. Recently, some contributions have proposed techniques to endow the networks with *early exits*, allowing to obtain predictions at intermediate points of the stack. These multi-output networks have a number of advantages, including (i) significant reductions of the inference time, (ii) reduced tendency to overfitting and vanishing gradients, and (iii) capability of being distributed over multi-tier computation platforms. In addition, they connect to the wider themes of biological plausibility and layered cognitive reasoning. In this paper, we provide a comprehensive introduction to this family of neural networks, by describing in a unified fashion the way these architectures can be designed, trained, and actually deployed in time-constrained scenarios. We also describe in-depth their application scenarios in 5G and Fog computing environments, as long as some of the open research questions connected to them.

Keywords Deep learning · Conditional computation · Early exit · Fog computing · Distributed optimization

Introduction

The success of deep networks can be attributed in large part to their extreme modularity and compositionality, coupled with the power of automatic optimization routines such as stochastic gradient descent [73]. While a number of innovative components have been proposed recently (such as attention layers [66], neural ODEs [16], and graph modules [58]), the vast majority of deep networks is designed as a sequential stack of (differentiable) layers, trained by propagating the gradient from the final layer inwards.

Even if the optimization of very large stacks of layers can today be greatly improved with modern techniques such as residual connections [72], their implementation still brings forth a number of possible drawbacks. Firstly, very deep networks are hard to parallelize because of the gradient locking problem [49] and the purely sequential nature of their information flow. Secondly, in the inference phase, these networks are complex to implement in resource-constrained or distributed scenarios [32, 53]. Thirdly, overfitting and vanishing gradient phenomena can still happen even with strong regularization, due to the possibility of raw memorization intrinsic to these architectures [27].

When discussing overfitting and model selection, these are generally considered properties of a given network applied to a full dataset. However, recently, a large number of contributions, e.g., [3, 7, 36, 45, 68], have shown that, even in very complex datasets such as ImageNet, the majority of patterns can be classified by resorting to smaller architectures. For example, [38] shows that the features extracted by a single convolutional layer already achieve a top-5 accuracy exceeding 30% on ImageNet. Even more notably, [30, 70] show that predictions that would be correct with smaller architectures can become incorrect with progressively deeper architectures, a phenomenon the authors call *over-thinking*.

In this paper, we explore a way to tackle with all these problems simultaneously, by endowing a given deep network with multiple *early exits* (auxiliary classifiers) departing from separate points in the architecture (see Fig. 1). Having one/two auxiliary classifiers is a common technique to simplify gradient propagation in deep networks, such as in the original Inception architecture [62]. However, recently it was recognized that these multi-exit networks have a number of benefits, including the possibility of devising layered training strategies, improving the efficiency of

✉ Simone Scardapane
simone.scardapane@uniroma1.it

[1] Department of Information Engineering, Electronics and Telecommunications (DIET), "Sapienza" University of Rome, Via Eudossiana 18, 00184 Rome, Italy

the inference phase, regularizing for computational cost, just to name a few. Depending on which of these aspects takes priority on a certain application, in this paper we overview a number of techniques and ideas to design, train, and deploy this emerging class of neural network models.

Contribution of This Work

While multi-exit networks have attracted significant interest lately, the related literature is widely fragmented, with similar ideas reappearing under broadly different names (e.g., cascaded networks [45], IDK networks [70], deeply supervised models [36]). In this paper, we aim at providing a coherent and unified introduction to them, by highlighting the key challenges and opportunities related to their optimized design, training, and implementation.

To this end, this overview is broadly structured as follows. "Related Works" briefly overviews three separate research lines that are closely connected to this paper. In "Description of the Model" we describe the multi-exit neural network models. We also briefly introduce the main issues related to them, which are explored more in-depth in the following sections: designing and placing the auxiliary components ("Placement and Optimized Design of the Early Exits"), training strategies ("Training Neural Networks with Early Exits"), and multi-exit inference ("Inference in Multi-exit Networks"). Then, in "Additional Topics" we connect our discussion to some broader themes of interest, namely, biological plausibility of the training paradigms, distributed implementation on multi-tiered (e.g., networked) platforms, and the information bottleneck principle. We conclude by highlighting some open research challenges in "Open Research Challenges."

Related Works

Before moving to the main part of the overview, we briefly review three lines of works that are related to the topics we describe. This also allows us to properly define the domain of the paper.

Fast Inference in Neural Networks

Designing NN with fast execution times is a critical task, especially in resource-constrained applications (e.g., mobile and IoT). An early example is the work pursued by [75], where a very deep NN generates an endless stream of labelled examples for the adaptive training of a much smaller student network. More in general, several authors have considered lowering the implementation complexity by reducing the computational precision of the performed per-layer operations, either through compression (e.g., group sparsity), quantization, or the introduction of specific low-latency components [76, 77].

Multi-output neural networks are well suited for fast inference, but they exploit an orthogonal mechanism to the works described before. Instead of searching for a single, efficient network for processing all input patterns, they constrain the processing time to be small for the *majority* of input patterns, through the introduction of early exits (see in particular "Inference in Multi-exit Networks" later on). We avoid an overview of the relevant literature here, as it would be redundant with respect to the rest of the manuscript. We refer for this especially to "Training Neural Networks with Early Exits."

Distributed Training of Neural Networks

As we describe below, a second important motivation for multiple exits is to distribute model training and evaluation on a multi-tiered platform, such as in emerging Fog Computing (FC) applications. In fact, the general topic of the design of multi-tier FC platforms is receiving large attention in these last years, mainly due to the emerging areas of the Mobile Cloud, Pervasive Computing and Edge Computing [78, 79]. These contributions focus on the offloading of (parts of) application programs from mobile devices to nearby FC servers through the exploitation of WiFi/Cellular-based communication technologies (see, for example, [4] and references therein for an updated overview of this topic). NNs with multiple exits fit naturally into this paradigm, and we overview this more in-depth in "Distributed Implementation of Multi-exit Networks." A related topic is that of distributed training of NN architectures on networks of interrelated agents, e.g., [57]. As we discuss in "Distributed Implementation of Multi-exit Networks" and "Layer-Wise Training," some training approaches for multi-output networks naturally lend themselves to distributed implementations.

Layered Training

Having multiple early exits makes the NN a stage-wise classifier (see in particular Fig. 3a and "Layer-Wise Training"). Layerwise training approaches have always been a fundamental area of research in neural networks. In fact, some of the earliest models predating back-propagation were iterative [7, 26], and (unsupervised) layerwise strategies were instrumental in the training of some early very deep CNN models [10]. In this paper, we also survey layered optimization strategies in the context of multi-output neural networks. However, we underline that many recent lines of research on multi-layered training, such

as boosting neural classifiers [17, 24], kernel-based methods [33], progressive growing architectures [29], or clustering methods [44], do not fit into our model.

Description of the Model

The discussion in "Related Works" has framed several key concerns in the use of neural networks. We now turn to introducing formally the idea of multi-output networks, a design strategy that can be exploited to simplify training, reduce the inference cost, and distribute computation.

Basic Neural Networks

Consider a generic deep neural network $f(x)$, taking an input x and providing a prediction y. The input x can be as simple as a vector, or a more structured data type, such as an image, a sequence, a video, a graph, etc. Without loss of generality, we assume that f is composed by the cascade of L differentiable operators $f_1, \ldots, f_L$:

$$f(x) = (f_L \circ f_{L-1} \circ \ldots \circ f_1)(x), \tag{1}$$

where $\circ$ denotes function composition such that $(f_i \triangleq f_j)(\cdot) = f_i(f_j(\cdot))$. Examples of operations that can act as components in (1) include convolutional layers for images [20], batch normalization, self-attention [66], and many others.[1] We denote by h_i the output of the ith operation, so that $h_i \triangleq f_i(h_{i-1})$, and $h_L = y$ is the final output of the network. It is common to refer to h_i as the ith *embedding* generated by the network for the input x. Finally, θ_i will denote the set of adaptable parameters of the ith layer, which can eventually be empty (e.g., in the case of dropout).

Given a set of examples $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$ and a loss function $l(\cdot, \cdot)$ penalizing the wrong predictions of the network, the parameters are tuned by performing stochastic gradient descent (SGD) on the expected loss:

$$f^* = \arg\min_{\theta} \sum_{n=1}^{N} l\left(y_n, f(x_n)\right), \tag{2}$$

where $\theta = \bigcup_{i=1}^{L} \theta_i$ is the union of all trainable parameters. In this basic setup, all L layers must be evaluated before obtaining a prediction and this full processing, as we elaborated on in the introduction, might not be necessary and even downright harmful in terms of overfitting and energy consumption. Next, we describe a model that attempts to address these drawbacks by adding intermediate prediction steps along the network stack.

[1] A single f_i can also be a sequence of operations. We do not make any distinction in the paper, and we use the terms *block* and *layer* indistinguishably to refer to a single f_i.

Neural Networks with Early Exits

In order to describe this extended architecture, we begin by selecting a set of early exit indices $\mathcal{C} \subseteq \{1, \ldots, L-1\}$ corresponding to 'interesting' mid-points in the architecture. We will have more to say on how to properly select $\mathcal{C}$ in the following sections of the paper. At each mid-point $i \in \mathcal{C}$ that was selected, we feed the intermediate embedding h_i to an auxiliary classifier/regressor c_i to obtain an intermediate prediction y_i:

$$y_i = c_i(h_i), \tag{3}$$

where c_i is another (smaller) auxiliary neural network in the form (1), typically consisting of only one/two layers. We call f the *backbone* network, c_i the ith auxiliary classifier/regressor, and y_i the ith *early exit* from the backbone. This setup is depicted graphically in Fig. 1. As a result, by running the full network with all early exits we obtain a sequence of predictions y_i, $i \in \mathcal{C}$, each potentially more refined and accurate than the previous one. However, in order to make the specification of this model useful, a number of questions must be answered, forming the basis for the rest of this overview. We briefly summarize the key points below. Note that some design choices are mutually dependent (e.g., the training strategy can influence the criterion for placing the auxiliary classifiers). However, for clarity, we try to keep them conceptually separated and mention inter-dependencies when needed in the corresponding sections.

Selecting Where and How to Early Exit

Firstly, one has to select the design of the auxiliary models, the number of early exits, and their placements over the backbone network. If the multi-exit architecture is used only to counter overfitting, then a small number of early exits is generally sufficient. For example, the original Inception model described in [62] has two early exits, placed roughly at 1/3 and 2/3 of the backbone network. More in general,

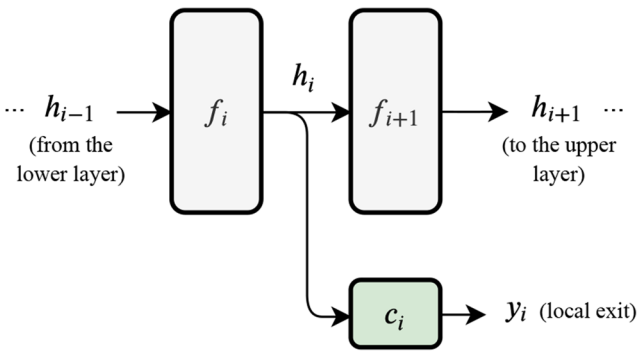

Fig. 1 Graphical depiction of a generic early exit in neural network architectures. In green, we show the auxiliary predictor

however, placing a set of early exits to satisfy energy or efficiency constraints is a combinatorial problem. This problem is formally described in "Placement and Optimized Design of the Early Exits."

Training the Network

Secondly, a proper training mechanism must be devised. Here, we partition the possible training approaches in three broad classes:

1. The overall architecture can be trained *jointly* by defining a single optimization problem that embraces all intermediate exits. This can be done by a combination of per-classifier losses [36], or by first combining the outputs and then building a single loss atop this combination [56].
2. We can adopt a *layer-wise* approach, where at each iteration we train a single auxiliary classifier together with the backbone layers preceding it, and, then, we freeze them after training [22].
3. We can first train the backbone network, and then separately train the auxiliary classifiers on top of it [67]. This can be interpreted as a very primitive form of knowledge distillation [23].

We describe these three classes, along with some design considerations, in "Training Neural Networks with Early Exits."

Exploiting Multiple Early Exits in the Inference Phase

Thirdly, once the network is trained, a suitable inference mechanism must be chosen. Once again, in the simplest case, auxiliary classifiers are used only to simplify gradient flow, and in this case their intermediate exits can be discarded [62]. Alternatively, the different predictions can be combined to obtain an ensemble almost for free [56]. In the most interesting case, however, one wants to exploit the different exits to process each input as efficiently as possible, by selecting the first auxiliary classifier that has a reasonable chance to provide the correct prediction, exploiting the full network only for cases that are extremely hard to predict [74]. In this case, it is necessary to select (and possibly train) a criterion to decide whether to exit the network or whether to continue processing over the next block. We consider this task in "Inference in Multi-exit Networks."

Formal Properties of Multi-exit Networks

Finally, we can consider the properties induced by the separate exits, in terms of, e.g., non-convexity of the optimization problem [72], convergence and generalization gaps of the different training approaches, guaranteed accuracy of the intermediate predictions, and so on. While the focus of this survey is more application-oriented, we still overview some of the most interesting results on these formal topics later on.

Placement and Optimized Design of the Early Exits

To begin with, we need to devise an optimized strategy for designing and implementing the auxiliary classifiers c_i.[2] For fully connected layers, the vast majority of works consider either simple linear classifiers [63], or small neural networks with one/two hidden layers [30]. For convolutional layers, this is slightly more complex because early layers in most CNNs have a very high dimensionality, which would result in extremely high-dimensional classifiers [30]. A simple solution in this case is to consider very strong dimensionality reduction procedures before the classification step, in the form of global average pooling or pooling with very large windows.

A similar consideration has to be taken into account when designing the set of indexes $\mathcal{C}$ for the placements of the early exits. Properly selecting this set is especially important for the case in which early exits must be used to speed-up the inference phase. A very simple technique is to estimate the relative computational cost of each layer, and place the early exits to some given percentiles of the network' total cost [30].

More in general, let us suppose that γ_i denotes the computational cost of running the network up to early exit i (as measured, e.g., in total number of operations to be performed). Similarly, denote by I_i the fraction of data that exits at the ith early exit (techniques to decide when to stop are discussed later on in "Inference in Multi-exit Networks"). Focusing only on early exit i, using the auxiliary classifier improves the efficiency of the network whenever the following relation is met [52]:

$$(\gamma_{i+1} - \gamma_i)\,(I_i - I_{i+1}) > \gamma_i I_{i+1}. \tag{4}$$

Because the number of possible placements of M early exits in a network grows exponentially in the network's depth L, the optimal placement according to (4) can only be evaluated for small networks.

However, possibly sub-optimal greedy algorithms have been proposed to evaluate the placement from the origin of

[2] We use the term classifier for simplicity, but everything extends to regression and other supervised problems.

the network up to the final layer [3, 52]. The algorithm in [3], in particular, allows to define a further hyper-parameter TH $\in [0, 1]$ to fine-tune the final validation accuracy. To use the same description as in [56], we overload our previous notation and define γ_{f_i} to be the computational cost of f_i alone, and similarly for γ_{c_i}. In addition, we define a compression ratio $\text{cm}_i = \frac{I_{i+1}}{I_i}$. Then, the greedy approach in [3] keeps an early exit whenever the following relation at layer i is met:

$$(\text{TH} - \text{cm}_i)\ \gamma_{f_{i+1}} - (1 - \text{cm}_i)\,\gamma_{c_i} - (1 - \text{TH})\,\gamma_{f_i} \geq 0. \tag{5}$$

An alternative approach, described in [12], is to evaluate each early exit considering both constant auxiliary classifiers and adaptive auxiliary classifiers. Whenever the former performs better in terms of training/efficiency trade-off, we remove them from our set. General hyper-parameter optimization procedures [60] can also be used to fine-tune the entire architecture, including the eventual placement of some early exits.

Training Neural Networks with Early Exits

In this section, we consider the second problem introduced in "Description of the Model," namely, training the architecture. We describe several variants that have been proposed in the literature, and we highlight some of their formal properties. Note that, once the set $\mathcal{C}$ of the early exit placements has been designed, we can always assume that there is a one-to-one correspondence between layers and auxiliary classifiers, since in-between layers can be merged into a single layer. Because this simplifies our notation, we will make this assumption from here on. This section is organized as follows: we start with joint training strategies in "Joint Training Approaches" and "Joint Training on a Combined Output"; then, we describe layer-wise strategies in "Layer-Wise Training," and we conclude with other variants in "Further Training Approaches."

Joint Training Approaches

In the simplest setup, we can associate a loss L_i to each intermediate classifier c_i, and perform SGD on a suitable combination of all losses. We call this *joint training* (JT) or, equivalently, deeply supervised [36]. Specifically, we train the model on:

$$f^* = \arg\min \left\{ L + \sum_{i=1}^{L-1} \alpha_i L_i \right\}, \tag{6}$$

where L (called the *overall loss* in [36]) is the final loss defined in (2), L_i is an auxiliary loss (called *companion loss* in [36]) defined as:

$$L_i \triangleq \sum_{n=1}^{N} l\left(y_n, c_i(x_n)\right), \tag{7}$$

and α_i weights the contribution of the ith auxiliary classifier. We will discuss the selection of these weights later on. In general, if the auxiliary classifiers are only used to improve performance, it is customary to weight earlier classifiers less. For example, the Inception model [62] propose to set $\alpha_i = 0.3$. A schematic description of the joint training method is provided in Fig. 2.

Apart from Inception, similar approaches have been successfully applied to, among others, face recognition [61], object detection [14], image segmentation [15], pose estimation [25], saliency detection [41], metric learning [71], visual attention prediction [69], and super resolution [34, 65]. JT approaches are also popular in the case of recurrent neural networks for sequence classification/regression, in which intermediate predictions can be obtained for free by considering the outputs of the network at each time-step [40].

On the Convergence Property of the JT Approach

Interestingly, it can be shown that even a relatively simple setup as (7) can improve the convergence of the resulting network with early exits. Lee et al. [36] analyze the case where the loss is selected as the squared hinge loss (although the proof can be easily extended to other losses, such as the cross-entropy), and each loss is augmented by an ℓ_2 regularization term. In this case, they prove in [36] the following result.

Theorem 1 (Theorem 1, [36]) *Assume that* $\sum_{i=1}^{L-1} L_i$ *is* λ*-strongly convex*[3] *close to the optimal* θ^**. Let us denote by* T *the number of iterations of SGD, by* $\eta_t = 1/t$ *the decaying step size at iteration* t*, by* $\theta_T^{standard}$ *the solution obtained by SGD at iteration* T *when optimizing* (2)*, and by* θ_T^{joint} *the one obtained by optimizing* (7)*. Then, optimizing* (7) *compared with* (2) *provides a relative speed-up in convergence of* $\frac{\mathbb{E}\left[\left\|\theta^* - \theta_T^{\text{standard}}\right\|^2\right]}{\mathbb{E}\left[\left\|\theta^* - \theta_T^{\text{joint}}\right\|^2\right]} = \Theta(\exp\{\ln(T)\lambda)\})$.

The authors in [36] argue that, in practice, the Lipschitz constant λ of the companion loss is large (compared with the Lipschitz constant of the original loss), providing a

[3] A function F is λ-strongly convex if, for any a, b, we have that $F(a) \geq F(b) + \nabla^T F(b)(a - b) + \frac{\lambda}{2}\|a - b\|^2$.

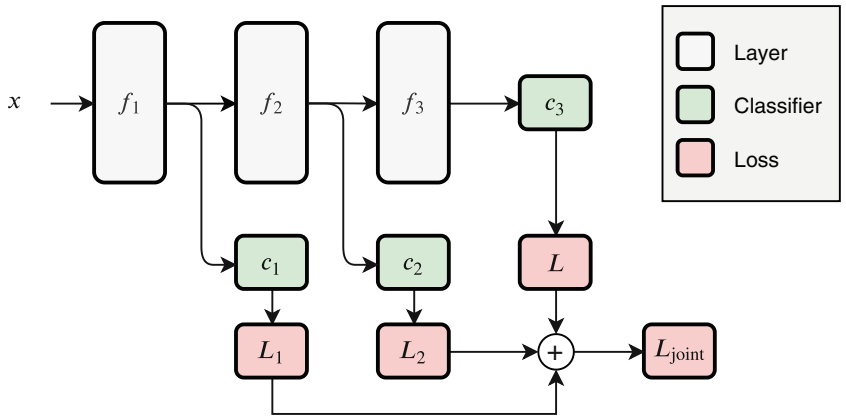

Fig. 2 Graphical description of the joint training approach. Colors are described in the legend

justification for the good empirical performance of the JT approach. However, analyzing and proving the relationship between the inner (local) and standard optimization problems remain open questions. Also, we note that this strategy could benefit from a more efficient sampling of patterns at the different exits, although this aspect has not received much attention in the literature.

Joint Training on a Combined Output

A variant of the JT approach is described in [31]. Instead of merging the auxiliary losses, we can adaptively merge the *predictions* of the network as:

$$\hat{y} = \alpha f(x) + \sum_{i=1}^{L-1} \alpha_i c_i(x), \tag{8}$$

where the weights α and α_i can be trained simultaneously with θ. Several variants can be devised where, for example, weights are constrained to be positive, or they are processed through a softmax function to arrive at a convex combination.

More in general, we can devise a range of options to adaptively combine the different auxiliary predictions. For example, [64] analyze max-pooling and average-pooling to combine the different predictions with a fixed (non-trainable) operation. Alternatively, [56] endows each early exit with an additional binary classifier $g_i(x) \in [0, 1]$ to decide how much the current prediction should be trusted. In this case, the global output can be defined recursively as:

$$\hat{c}_i(x) = g_i(x)\, c_i(x) + (1 - g_i(x))\, \hat{c}_{i+1}(x), \tag{9}$$

where the base case is the final prediction $\hat{c}_L(x) = f(x)$. The combination process can also be implemented through an additional network component (e.g., an LSTM or recurrent network), providing a further degree of freedom to the system. This last setup is related to the use of boosting methods to incrementally build the network's architecture [17, 48].

Layer-Wise Training

The insertion of auxiliary local classifiers opens up interesting alternatives to the standard joint training described above. In particular, we can devise a *layer-wise training* (LT) strategy by separately optimizing each layer in the architecture. Note that supervised and unsupervised LT methods were explored repeatedly in the initial attempts at building deeper architectures [10, 35]. Only recently, however, have they become a successful tool for training them, e.g., through boosting [17, 24], clustering [44], and kernel similarity [33].

In the context of multi-exit networks as described in this paper, [22] introduced the general idea under the name "forward thinking," while [45] calls it "deep cascade learning." We depict it visually in Fig. 3a.

According to Fig. 3a, the LT approach proceeds as follows:

1. Start by training the first auxiliary classifier $c_1(f_1(x))$, depicted as *Training #1* in Fig. 3a. Considering the notation of "Joint Training Approaches," this is equivalent to minimize L_1 individually.
2. Freeze the previous layer, by replacing all inputs x_i in the training set with the corresponding embedding $f_1(x_i)$.
3. Train the second layer $c_2(f_2(\cdot))$, by solving L_2 only with respect to the weights θ_2.
4. Repeat steps (2)–(3) for as many layers as needed.

The advantage of this method is that, since each layer is relatively small (depending on the size of each f_i and c_i), regularization problems such as vanishing and exploding gradients should be less prominent under this training approach [22]. An extension to semi-supervised problems with growing networks is presented in [68].

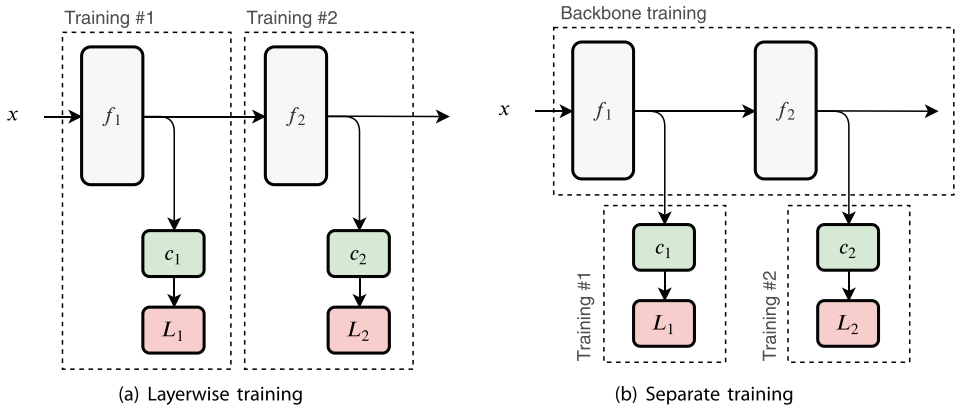

Fig. 3 Graphical depiction of the **a** layerwise training approach and **b** separate training approach. Colors are the same used in Fig. 2

On the Formal Properties of the LT Approach

A preliminary formal analysis of the LT approach is provided in [8], and later extended in [7]. Roughly speaking, their works show that, if each layer-wise optimization process incurs an error that is upper bounded by ε, the final predictions given by the network will have an error with a similar upper bound given by $\mathcal{O}(L^2\varepsilon)$. They also show, empirically and through an informal analysis, that the approach will naturally lead to increasing the linear separability of the data for each layer that is added. Notably, [7] is the first work to show that LT can scale to large-scale problems, such as image classification on ImageNet. Additional formal considerations are provided in [3] showing that, under suitable considerations on the blocks composing the architecture, there always exists a solution in which adding a new layer does not worsen the empirical error of the model.

Further Training Approaches

Besides joint training (Joint Training Approaches and Joint Training on a Combined Output) and layer-wise training (Layer-Wise Training), the flexibility of the architectures analyzed in this paper lend themselves to a variety of alternative training approaches, which we briefly describe in the sequel. First of all, several authors [12, 30, 51, 67] have considered a variation of LT, in which the backbone network is trained separately from each of the auxiliary classifier. This is depicted visually in Fig. 3b. This kind of approach is especially suitable for describing and analyzing situations in which the backbone network is provided beforehand, and we desire to turn it into a multi-exit architecture. It also allows for easy experimentation on the selection of the auxiliary classifiers, which can be trained and removed quickly according to the rules-of-thumb introduced in "Placement and Optimized Design of the Early Exits." Finally, it allows for the use of non-neural auxiliary classifiers [67], such as using decision trees or support vector machines for the intermediate predictions.

Some strategies mix different approaches. For example, freezeout [13] is a variation of the JT approach, in which layers are iteratively 'freezed' by annealing their learning rates to zero, providing a trade-off between the accuracy of training everything jointly with a partial speed-up similar to a layer-wise training. Specifically, denote by $\eta_i(t)$ the learning rate of layer i at iteration t. Given a budget of training iterations T, we select a set of freezing points t_i equispaced in $[0, T]$. Then, the learning rate of layer i for $t < t_i$ is given by:

$$\eta_i(t) = 0.5\,\eta_i(0)\left(1 + \cos\left(\frac{\pi t}{t_i}\right)\right), \tag{10}$$

while $\eta_i(t) = 0$ for $t > t_i$. An example with $L = 3$, $T = 150$, and $\eta_i(0) = 0.1$ is shown in Fig. 4.

To conclude this section, we also mention that some authors have considered reinforcement learning strategies to train the overall architecture [21].

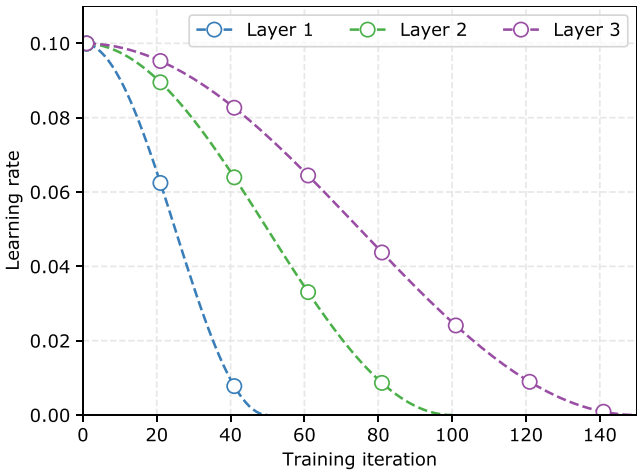

Fig. 4 Example of progressively freezing out layers as in [13]

Inference in Multi-exit Networks

We now focus on the topic of performing inference in architectures with multiple auxiliary classifiers. Note that, in some cases, this is trivial. For example, in architectures such as Inception [62], where the auxiliary classifiers are only used for improving the training phase, one can discard them and use $f(x)$ as the single final exit. In other cases, such as the aggregated output from (8) or (9), the network automatically provides a joint prediction that merges all the intermediate ones.

In the most interesting case, however, we can assume that every input x has an optimal depth corresponding to one specific prediction $c_i(x)$. In this case, we prefer to process each input up to the corresponding exit and, then, stop the forward propagation. In fact, similar schemes have been shown to provide significant improvements in inference time [63, 64], and they can also prevent overfitting. In particular, [30] describes a phenomenon it calls *overthinking*, in which a prediction $c_i(x)$ on the test set is correct, but later predictions turn out to be wrong.

For classification problems, a common setup in this case is to consider the confidence of the network on its own prediction, and exploit this to decide whether to early exit the architecture. According to this setup, denote by $c_{it}(x)$ the prediction of the ith classifier on the tth class, and by C the total number of classes. The normalized entropy of the prediction is:

$$H\left[c_i(x)\right] \triangleq \frac{1}{\log\left(C\right)} \sum_t c_{it}(x) \log\left(c_{it}(x)\right). \tag{11}$$

The value $H\left[c_i(x)\right]$ is always normalized between 0 and 1. By comparing $H\left[c_i(x)\right]$ against a user-selected threshold β_i, we can make the final decision on whether to exit the architecture. This strategy was popularized by BranchyNet [63, 64]. The drawback of this approach is that a separate threshold must be selected for each auxiliary classifier, in order to ensure that the average accuracy of the model does not degrade. Wang et al. [70] describe other alternatives, including using the most-confident prediction instead of the entropy, or learning the confidence score on top of $c_i(x)$ with a separate trainable function $g_i(c_i(x))$. The gating approach described in (9) is a specialization of this idea.

Alternatively, [3] introduces an iterative approach to fine-tune a single threshold given a desired accuracy on the level, based on a gradient-like approach exploiting the error between the current level of accuracy and the desired one. The strategy is also proved in [3] to be convergent.

Regularizing for Computational Cost

One interesting side benefit arising from the utilization of early exits we study here is that we can regularize the resulting networks for *efficiency*. Searching for efficient architectures is a wide problem in the literature on neural model selection [54]. Having multiple auxiliary classifiers allows for an unprecedented opportunity of optimizing the accuracy/efficiency trade-off for each input [56].

To this end, denote by δ_{ij} an indicator function describing whether the ith training pattern was processed by the jth auxiliary classifier. In addition, denote by ϵ_j the cost of executing the jth auxiliary classifier as compared with the $(j-1)$th one. Cost here is a generic quantity, that we can define to simply consider the number of operations executed by the auxiliary classifier (in this case $\epsilon_j = \gamma_{c_j}$ as defined in "Placement and Optimized Design of the Early Exits)," or a more abstract definition, which can include the communication costs in a distributed environment [74]. The average cost of f is, then, defined as follows:

$$C[f] \triangleq \sum_{i=1}^{n} \sum_{j=1}^{L} \delta_{ij} \cdot \epsilon_j. \tag{12}$$

Relation (12) can now be used in any procedure to select the hyper-parameters of the model, in order to explicitly trade-off the average inference cost of the neural network. Wang et al. [70] call (12) the "cascaded computational cost."

In addition, supposing that the exit strategy $g_i(x)$ is differentiable (such as the entropy measure described previously), similarly to (9), we can define a soft-approximation to (12) as [56]:

$$C[f] = \sum_{i=1}^{n} \sum_{j=1}^{L} p_{ij} \epsilon_j, \tag{13}$$

where:

$$p_{ij} = g_j(x_i) \cdot \prod_{l=1}^{j-1} (1 - g_l(x_i)). \tag{14}$$

Since the above expression is differentiable, we can use it as an alternative to the classical ℓ_p regularization to provide an explicit trade-off concerning accuracy and efficiency of the network, as described by the average computation cost incurred for a single example.

In the case of a pre-trained backbone network, [12] explores training separately each auxiliary classifier with a constraint on computational time. The global problem is, then, reduced to a sequence of weighted binary classification problems, where the weights depend on the costs. This is extended in [60], where the entire network design (including multiple hyper-parameters) is optimized through a Bayesian optimization procedure.

Additional Topics

In this section we detail three interesting applicative fields for the multi-output architecture described in the paper. We start with the topic of biological plausibility of deep networks and their training algorithms in "Biological Plausibility of Multi-exit Networks," move on to a distributed implementation over multi-tiered or distributed environments in "Distributed Implementation of Multi-exit Networks," and conclude with an overview of layer-wise training in the context of the information bottleneck principle in "The Information Bottleneck Principle."

Biological Plausibility of Multi-exit Networks

Multi-exit architectures in their LT formulation provide an interesting perspective on biological plausibility of neural network training [46]. In fact, backpropagation in classical neural networks is generally considered to be biologically implausible for multiple reasons [11], i.e., (i) the need to store the outputs for each intermediate layer f_i, to reuse it in the backward pass; (ii) the symmetry of weight matrices between forward and backward passes; and (iii) the perfect synchronization required by the process. Point (ii) has been explored in-depth in the literature on feedback alignment (and similar) [5, 39, 49], in which it is shown that the transpose of the weight matrix can be replaced by a (fixed) random matrix during the backward pass while still allowing for convergence of the algorithm. This approach, however, still requires a global error to be back-propagated from the final output layer, thus failing to satisfy points (i) and (iii).

Mostafa et al. [46] explore a variation of the LT approach, in which each auxiliary classifier is set as $c_i(x) = Mf_i(x)$, where M is a random matrix, which is eventually replaced by a separate random matrix K (of the same size) during the backward pass. This approach can still obtain good results in training (albeit not at the level of a classical LT training), while being implementable through highly compact update equations involving only local error terms, thus drastically improving the overall biological plausibility of the model.

How to extend this framework to improve the classification results closer to the state-of-the-art while not sacrificing the biological plausibility remains an open research question. For example, [50] further replaces the true label at each auxiliary exit with a randomly projected version, while at the same time increasing the supervised loss with a separate one that forces the embeddings of the network to be similar whenever the corresponding projected outputs are similar. In [9], the authors investigate this approach for parallel optimization of the network, where a set of intermediate buffers is added to ensure the possibility of decoupling a layer from the previous ones. Based on the literature on target propagation [37], the authors in [28] propose a further extension to decouple the different updates, in which the targets for the different early exits are computed adaptively from the original output y_i by an additional set of autoencoder networks.

Distributed Implementation of Multi-exit Networks

Many recent computation frameworks, ranging from 5G cellular networks to IoT sensors and fog computing, are built upon the idea of multi-tier computations, in which decisions can be taken both at the edge of the networks or delegated to other, more powerful decision centers that incur some communication latency [53, 74]. In these works, it is common to design systems in which separate neural networks, of different complexities, can run on the different tiers or on different computational units [42, 47].

Multi-exit neural networks fit naturally inside these emerging classes of technologies, by distributing different auxiliary classifiers over the different tiers, as depicted in Fig. 5. In this case, an estimation of the cost of communicating across the channel can be included both in the design of the early exits, as described in "Placement and Optimized Design of the Early Exits," or in the model selection/regularization phase, as described in "Regularizing for Computational Cost" [38].

An interesting question concerns the additional implementation of a distributed training phase [57], which is necessary whenever the separate tiers have to adapt to new data without the possibility of communicating everything to a centralized controller. LT methods lend themselves naturally to an iterative implementation, in which every tier trains its own model before sending the new set of input features to the next tier. Alternatively, more sophisticated distributed optimization strategies, possibly taking into account also communication constraints, are an open interesting research area [4, 6].

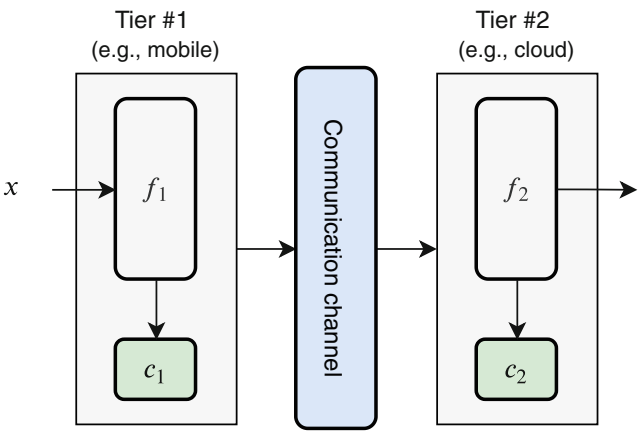

Fig. 5 Distributed implementation of a multi-exit neural network on separate tiers of an underlying distributed computing platform

The Information Bottleneck Principle

The last application context that we consider is that of the information bottleneck (IB) principle [1]. The IB was proposed as a general principle to explain the expressive power of deep networks, relying on information theoretic principles. Interestingly, one possibility for the actual implementation and test of the IB principle results in a different training scenario for multi-exit neural networks [18].

Slightly overloading our notation, denote by X a random variable describing the inputs to the network, by Y a second random variable describing the corresponding targets, and by F_i a third random variable describing the embeddings generated by layer f_i. In addition, denote by $I(\cdot,\cdot)$ the mutual information between two random variables. Roughly, the IB principle states that neural networks work because their intermediate representations tend to simultaneously maximize their mutual information with the target, while at the same time minimizing their mutual information with the input. Stated differently, they tend to compress information as much as possible while keeping the most predictive bits of information. Stated more formally, the IB principle states that intermediate representations tend to minimize:

$$\mathcal{L}_i = I\left(Y, F_i\right) - \beta\, I\left(X, F_i\right), \tag{15}$$

where β is a trade-off parameter.[4] While the IB principle has received ample empirical validation [55, 59], it is common to evaluate and/or apply it only to a *single* layer of the network.

Elad et al. [18] propose to exploit the IB principle as a training criterion for neural networks, wherein the network is trained according to a LT approach, substituting the local loss term L_i in (7) with an approximation of (15). Interestingly, this approximation requires the addition of multiple auxiliary classifiers to the network, acting as discriminators between embeddings corresponding to separate classes. Alternative supervised and unsupervised training strategies have also been proposed in a number of papers [19, 43].

Open Research Challenges

In this paper, we provided a self-contained, unified introduction to the design, implementation, and training of neural network with multiple early exits. As we argued, these models have a number of favorable properties, including a higher resilience to gradient problems, as long as a large flexibility to more advanced training/inference strategies. They also provide a viable entry point to the study of more biologically plausible alternatives to classical back-propagation, and they are more naturally suited to an implementation in multi-tiered computational frameworks such as the emerging fog computing one [2].

Apart from the multiple minor points discussed in the paper, we identify three main areas where research is needed in this context.

Challenge 1: How Powerful Are Multi-output Neural Networks?

The formal properties of these more general network architectures are only recently starting to be investigated, as we described in this paper, leaving ample margins to, e.g., quantify the relative generalization gap of each component. In addition, while these techniques have been applied multiple times to CNNs and, in smaller measure, to RNNs, their usefulness remains to be tested on more modern neural layers, such as graph convolutional networks, and transformer architectures. Finally, how to properly consider both architectural constraints, accuracy, and communication costs when implementing the networks in a distributed environment is still an open question.

Challenge 2: Full Integration in FC/IoT Environments

In principle, multi-output NNs promise to be a perfect match for distributed, multi-tiered systems given by the convergence of several recent paradigms, most notably Fog computing [4]. A number of challenges, however, need to be solved in order to see this convergence in practice, including elastic, asynchronous training strategies, optimal placement of the early exits, efficient inference, and communication costs, and so on. Several of these challenges are described and reviewed in [3].

Challenge 3: What Comes After Having Multiple Exits?

More in general, deep learning has risen to prominence thanks to the extreme modularity and flexibility of automatic differentiation frameworks combined with modern programming languages. We believe that architectures such as those explored in this paper represent a natural next step in moving away from purely feedforward implementation to more complex, multi-exit, multi-resolution ones. In this sense, an immediate extension of the ideas presented here is to consider multiple networks, conditionally connected on one another, performing inferences or routing information to one another [12].

[4] Note that, in practice, the term in (15) is infinite for deterministic networks with continuous inputs. However, it can be replaced with a noisy approximation [55].

Compliance with Ethical Standards

Conflict of Interest The authors declare that they have no conflict of interest.

Ethical Approval This article does not contain any studies with human or animal subjects performed by any of the authors.

References

1. Amjad RA, Geiger BC. Learning representations for neural network-based classification using the information bottleneck principle. In: IEEE transactions on pattern analysis and machine intelligence; 2019.
2. Baccarelli E, Naranjo PGV, Scarpiniti M, Shojafar M, Abawajy JH. Fog of everything: energy-efficient networked computing architectures, research challenges, and a case study. IEEE Access. 2017;5:9882–910.
3. Baccarelli E, Scardapane S, Scarpiniti M, Momenzadeh A, Uncini A. Optimized training and scalable implementation of conditional deep neural networks with early exits for fog-supported iot applications. Inf Sci. 2020;521:107–43. https://www.sciencedirect.com/science/article/pii/S0020025520301249.
4. Baccarelli E, Scarpiniti M, Momenzadeh A. Ecomobifog–design and dynamic optimization of a 5g mobile-fog-cloud multi-tier ecosystem for the real-time distributed execution of stream applications. IEEE Access. 2019;7:55565–608.
5. Baldi P, Sadowski P, Lu Z. Learning in the machine: random backpropagation and the deep learning channel. Artif Intell. 2018;260:1–35.
6. Barbarossa S, Sardellitti S, Di Lorenzo P. Communicating while computing: distributed mobile cloud computing over 5g heterogeneous networks. IEEE Signal Process Mag. 2014;31(6):45–55.
7. Belilovsky E, Eickenberg M, Oyallon E. Greedy layerwise learning can scale to imagenet. In: Proceedings of the 36th International Conference on Machine Learning (ICML); 2018.
8. Belilovsky E, Eickenberg M, Oyallon E. Shallow learning for deep networks. 2018. https://openreview.net/forum?id=r1Gsk3R9Fm.
9. Belilovsky E, Eickenberg M, Oyallon E. Decoupled greedy learning of CNNs. arXiv preprint arXiv:1901.08164. 2019.
10. Bengio Y, Lamblin P, Popovici D, Larochelle H. Greedy layer-wise training of deep networks. In: Advances in neural information processing systems; 2007. p. 153–60.
11. Betti A, Gori M, Marra G. Backpropagation and biological plausibility. arXiv preprint arXiv:1808.06934. 2018.
12. Bolukbasi T, Wang J, Dekel O, Saligrama V. Adaptive neural networks for efficient inference. In: Proceedings of the 34th International Conference on Machine Learning (ICML); 2017. p. 527–36. JMLR. org.
13. Brock A, Lim T, Ritchie JM, Weston N. Freezeout: accelerate training by progressively freezing layers. arXiv preprint arXiv:1706.04983. 2017.
14. Cai Z, Fan Q, Feris RS, Vasconcelos N. A unified multi-scale deep convolutional neural network for fast object detection. In: European Conference on Computer Vision. Springer; 2016. p. 354–70.
15. Chen LC, Yang Y, Wang J, Xu W, Yuille AL. Attention to scale: scale-aware semantic image segmentation. In: Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition; 2016. p. 3640–9.
16. Chen TQ, Rubanova Y, Bettencourt J, Duvenaud DK. Neural ordinary differential equations. In: Advances in neural information processing systems; 2018. p. 6571–83.
17. Cortes C, Gonzalvo X, Kuznetsov V, Mohri M, Yang S. Adanet: adaptive structural learning of artificial neural networks. In: Proceedings of the 34th International Conference on Machine Learning (ICML); 2017. p. 874–83.
18. Elad A, Haviv D, Blau Y, Michaeli T. The effectiveness of layer-by-layer training using the information bottleneck principle. 2018. https://openreview.net/forum?id=r1Nb5i05tX.
19. Elad A, Haviv D, Blau Y, Michaeli T. Direct validation of the information bottleneck principle for deep nets. In: Proceedings of the 2019 IEEE International Conference on Computer Vision workshops (ICCV); 2019.
20. Goodfellow I, Bengio Y, Courville A. Deep learning. Cambridge: MIT Press; 2016.
21. Guan J, Liu Y, Liu Q, Peng J. Energy-efficient amortized inference with cascaded deep classifiers. arXiv preprint arXiv:1710.03368. 2017.
22. Hettinger C, Christensen T, Ehlert B, Humpherys J, Jarvis T, Wade S. Forward thinking: building and training neural networks one layer at a time. arXiv preprint arXiv:1706.02480. 2017.
23. Hinton G, Vinyals O, Dean J. Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531. 2015.
24. Huang F, Ash J, Langford J, Schapire R. Learning deep resnet blocks sequentially using boosting theory. In: Proceedings of the 35th International Conference on Machine Learning (ICML); 2018.
25. Insafutdinov E, Pishchulin L, Andres B, Andriluka M, Schiele B. Deepercut: a deeper, stronger, and faster multi-person pose estimation model. In: European Conference on Computer Vision. Springer; 2016. p. 34–50.
26. Ivakhnenko AG, Lapa V. Cybernetic predicting devices. Tech. rep., Purdue University. 1966.
27. Jastrzebski S, Kenton Z, Arpit D, Ballas N, Fischer A, Bengio Y, Storkey A. Three factors influencing minima in SGD. arXiv preprint arXiv:1711.04623. 2017.
28. Kao YW, Chen HH. Associated learning: decomposing end-to-end backpropagation based on auto-encoders and target propagation. arXiv preprint arXiv:1906.05560. 2019.
29. Karras T, Aila T, Laine S, Lehtinen J. Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196. 2017.
30. Kaya Y, Hong S, Dumitras T. Shallow-deep networks: Understanding and mitigating network overthinking. arXiv preprint arXiv:1810.07052. 2018.
31. Kim J, Kwon Lee J, Mu Lee K. Deeply-recursive convolutional network for image super-resolution. In: Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2016. p. 1637–45.
32. Klaine PV, Nadas JP, Souza RD, Imran MA. Distributed drone base station positioning for emergency cellular networks using reinforcement learning. Cogn Comput. 2018;10(5):790–804.
33. Kulkarni M., Karande S. Layer-wise training of deep networks using kernel similarity. arXiv preprint arXiv:1703.07115. 2017.
34. Lai WS, Huang JB, Ahuja N, Yang MH. Deep laplacian pyramid networks for fast and accurate super-resolution. In: Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition; 2017. p. 624–32.
35. Larochelle H, Bengio Y, Louradour J, Lamblin P. Exploring strategies for training deep neural networks. J Mach Learn Res. 2009;10:1–40.
36. Lee CY, Xie S, Gallagher P, Zhang Z, Tu Z. Deeply-supervised nets. In: Artificial Intelligence and Statistics; 2015. p. 562–70.
37. Lee DH, Zhang S, Fischer A, Bengio Y. Difference target propagation. In: Joint european conference on machine learning and knowledge discovery in databases. Springer; 2015. p. 498–515.

38. Leroux S, Bohez S, De Coninck E, Verbelen T, Vankeirsbilck B, Simoens P, Dhoedt B. The cascading neural network: building the internet of smart things. Knowl Inf Syst. 2017;52(3):791–814.
39. Lillicrap TP, Cownden D, Tweed DB, Akerman CJ. Random synaptic feedback weights support error backpropagation for deep learning. Nat Commun. 2016;7(1):1–10.
40. Lipton ZC, Kale DC, Elkan C, Wetzel R. Learning to diagnose with lstm recurrent neural networks. arXiv preprint arXiv:1511.03677. 2015.
41. Liu N, Han J. Dhsnet: deep hierarchical saliency network for salient object detection. In: Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2016. p. 678–86.
42. Lo C, Su YY, Lee CY, Chang SC. A dynamic deep neural network design for efficient workload allocation in edge computing. In: Proceedings of the 2017 IEEE International Conference on Computer Design (ICCD). IEEE; 2017. p. 273–80.
43. Löwe S, O'Connor P, Veeling B. Putting an end to end-to-end: gradient-isolated learning of representations. In: Advances in neural information processing systems; 2019. p. 3033–45.
44. Malach E, Shalev-Shwartz S. A provably correct algorithm for deep learning that actually works. arXiv preprint arXiv:1803.09522. 2018.
45. Marquez ES, Hare JS, Niranjan M. Deep cascade learning. IEEE Trans Neural Netwo Learn Syst. 2018;29(11):5475–85.
46. Mostafa H, Ramesh V, Cauwenberghs G. Deep supervised learning using local errors. Front Neurosci. 2018;12:608.
47. Nan F, Saligrama V. Adaptive classification for prediction under a budget; 2017. p. 4727–37.
48. Nitanda A, Suzuki T. Functional gradient boosting based on residual network perception. arXiv preprint arXiv:1802.09031; 2018.
49. Nøkland A. Direct feedback alignment provides learning in deep neural networks. In: Advances in neural information processing systems; 2016. p. 1037–45.
50. Nøkland A, Eidnes LH. Training neural networks with local error signals. In: Proceedings of the 36th International Conference on Machine Learning (ICML); 2019. p. 4839–50.
51. Panda P., Sengupta A., Roy K. Conditional deep learning for energy-efficient and enhanced pattern recognition. In: Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE; 2016. p. 475–80.
52. Panda P, Sengupta A, Roy Ks. Energy-efficient and improved image recognition with conditional deep learning. ACM J Emerg Technol Comput Syst (JETC). 2017;13(3):1–21.
53. Park J, Samarakoon S, Bennis M, Debbah M. Wireless network intelligence at the edge. Proc IEEE. 2019;107(11):2204–39.
54. Pham H, Guan MY, Zoph B, Le QV, Dean J. Efficient neural architecture search via parameter sharing. arXiv preprint arXiv:1802.03268. 2018.
55. Saxe AM, Bansal Y, Dapello J, Advani M, Kolchinsky A, Tracey BD, Cox DD. On the information bottleneck theory of deep learning. J Stat Mech: Theory Exp. 2019;2019(12):124020.
56. Scardapane S, Comminiello D, Scarpiniti M, Baccarelli E, Uncini A. Differentiable branching in deep networks for fast inference. In: Proceedings of the 2020 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP); 2020.
57. Scardapane S, Di Lorenzo P. A framework for parallel and distributed training of neural networks. Neural Netw. 2017;91:42–54.
58. Schlichtkrull M, Kipf TN, Bloem P, Van Den Berg R, Titov I, Welling M. Modeling relational data with graph convolutional networks. In: European Semantic Web Conference. Berlin: Springer; 2018. p. 593–607.
59. Shwartz-Ziv R, Tishby N. Opening the black box of deep neural networks via information. arXiv preprint arXiv:1703.00810; 2017.
60. Stamoulis D, Chin TW, Prakash AK, Fang H, Sajja S, Bognar M, Marculescu D. Designing adaptive neural networks for energy-constrained image classification. In: Proceedings of the 2018 International Conference on Computer-Aided Design (CAD); 2018. p. 1–8.
61. Sun Y, Wang X, Tang X. Deeply learned face representations are sparse, selective, and robust. In: Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition; 2015. p. 2892–900.
62. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2015. p. 1–9.
63. Teerapittayanon S, McDanel B, Kung HT. Branchynet: fast inference via early exiting from deep neural networks. In: Proceedings of the 2016 23rd International Conference on Pattern Recognition (ICPR). IEEE; 2016. p. 2464–9.
64. Teerapittayanon S, McDanel B, Kung HT. Distributed deep neural networks over the cloud, the edge and end devices. In: Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE; 2017. p. 328–39.
65. Tong T, Li G, Liu X, Gao Q. Image super-resolution using dense skip connections. In: Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV); 2017. p. 4799–807.
66. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Advances in neural information processing systems; 2017. p. 5998–6008.
67. Venkataramani S, Raghunathan A, Liu J, Shoaib M. Scalable-effort classifiers for energy-efficient machine learning. In: Proceedings of the 52nd Annual Design Automation Conference; 2015. p. 1–6.
68. Wang G, Xie X, Lai J, Zhuo J. Deep growing learning. In: Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV); 2017. p. 2812–20.
69. Wang W, Shen J, Deep visual attention prediction. IEEE Trans Image Process. 2017;27(5):2368–78.
70. Wang X, Luo Y, Crankshaw D, Tumanov A, Yu F, Gonzalez JE. Idk cascades: fast deep learning by learning not to overthink. arXiv preprint arXiv:1706.00885. 2017.
71. Yuan Y, Yang K, Zhang C. Hard-aware deeply cascaded embedding. In: Proceedings of the IEEE International Conference on Computer Vision; 2017. p. 814–23.
72. Zhang H, Shao J, Salakhutdinov R. Deep neural networks with multi-branch architectures are less non-convex. arXiv preprint arXiv:1806.01845. 2018.
73. Zhong G, Jiao W, Gao W, Huang K. Automatic design of deep networks with neural blocks. Cogn Comput. 2020;12:1–2. https://link.springer.com/article/10.1007/s12559-019-09677-5.
74. Zhou Z, Chen X, Li E, Zeng L, Luo K, Zhang J. Edge intelligence: paving the last mile of artificial intelligence with edge computing. Proc IEEE. 2019;107(8):1738–62.
75. Bucilu C, Caruana R, Alexandru N-M. Model compression. In: Proc. 12th ACM SIGKDD international conference on knowledge discovery and data mining; 2006. p. 535–41.
76. Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y. Binarized neural networks. In: Advances in neural information processing systems; 2016. p. 4107–15.

77. Rastegari M, Ordonez V, Redmon J, Farhadi A. Xnor-net: imagenet classification using binary convolutional neural networks. In: European conference on computer vision. Springer; 2016. p. 525–42.
78. Othman M, Madani SA, Khan SU, et al. A survey of mobile cloud computing application models. IEEE Commun Surv Tutor. 2013;16(1):393–413. IEEE.
79. Mach P, Becvar Z. Mobile edge computing: a survey on architecture and computation offloading. IEEE Commun Surv Tutor. 2017;19(3):1628–1656. IEEE.