# REINFORCEMENT LEARNING BASED ASYMMETRICAL DNN MODULARIZATION FOR OPTIMAL LOADING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Latency of DNN (Deep Neural Network) based prediction is the summation of model loading latency and inference latency. Model loading latency affects the first response from the applications, whereas inference latency affects the subsequent responses. As model loading latency is directly proportional to the model size, this work aims at improving the response time of an intelligent app by reducing the loading latency. The speedup is gained by asymmetrically modularizing the given DNN model among several small child models and loading them in parallel. The decision about number of feasible child models and their corresponding split positions are taken care by reinforcement learning unit (RLU). RLU takes into account the available hardware resources on-device and provides the best splitting index $k$ and their positions $\vec{p}$ specific to the DNN model and device, where $\vec{P} = (p_1, p_2, ..., p_k)$ and $p_i$ is the end position of child $i$. The proposed method has shown significant loading improvement (up to $7X$) on popular DNNs, used for camera use-case. The proposed method can be used to speed up the app response. Along with the mentioned advantages, RLU driven approach facilitates for On-device personalization by separating one module only with trainable layers and loading that particular module while training on-device.

## 1 INTRODUCTION

Users frequently encounter machine learning algorithms (or DNN) via intelligent applications on mobile phone. The first interaction between user and DNN happens through phone unlocking biometric application. The phone unlocking, camera preview and similar applications go through 2 major steps pertaining predictive modeling. Step-1: Loading the pretrained model or weights on RAM. Step-2: Inferencing the users sample across the loaded model for identification (biometric), classification (Bixby vision) or segmentation. If we focus on the latency of the application, we realize that net latency is the aggregation of model loading latency (Step-1) and inference latency (Step-2), which can be represented as $T_{total} = T_{Load} + T_{Inference}$, where $T_{Total}$ is the total processing time, $T_{Load}$ is the time spent for loading the model from auxiliary memory to primary memory and $T_{Inference}$ is the inference time. It has been noticed that $T_{Load} > T_{Inference}$ in general. In order to improve the user experience, it is necessary that $T_{Total}$ should be in acceptable range and to fulfill this requirement there have been series of inventions to keep the architecture light weighted, leading to less floating point operations consequently short inference time. The process of speeding up the inference time helps in reducing the model size (in general), (Huang et al. (2019); Matsubara et al. (2019); Zhuang et al. (2020); Ben-Nun & Hoefler (2019); Rallapalli et al. (2016) and leads to small loading time. In order to extend the DNN facilities to edge devices, it is recommended to have an architecture with limited parameters. The stated requirements gave birth to several DNN like MobileNet-V1,V2,V3 (Howard et al. (2017); Sandler et al. (2018); Howard et al. (2019) SqueezeNet (Iandola et al. (2016)) etc. But it has been completely unseen to notice the work for reducing the model loading time ($T_{Load}$) specifically without disturbing the inference time and performance. The proposed work introduces a feedback driven model splitting method to split the pretrained neural architecture among tiny child architectures so that they can be loaded simultaneously leading to reduced loading latency. The proposed methodology uses reinforcement learning framework in order to split the model. This work highlights 2 hardware dependent important cases of model loading and inferencing. The cases depend on available hardware threads in the device where lesser number of hardware threads leads to occurrence of Case-II and more num-

ber of available hardware threads leads to Case-I. The switching from Case-I to Case-II is planned in order to utilize the loading time for inference. Along with the parallel operations of loading and inferencing the proposed method also facilitates for on-device training. The current ecosystem for on-device training aspires to perform the training on last few layers of the pretrained/deployed model for importing the user dependent features to provide personalized response to the users. In order to do so gradients are set as True for last few layers only, whereas gradients are False for any other layers. The proposed method keeps the mentioned fact into account and while modularizing the architecture, it ensures to have all trainable layers in last one module and sets the gradients as True. This way additional overhead of loading the full model and then controlling the gradients while backpropagation is reduced. The RL based framework is used at developer site and instead of deploying the one full model the bundle of multiple small models is deployed. Splitting is hardware dependent process and advised to be used once for each type of mobile device. The novelty of the method and contribution can be shown as: (a) The work proposes a novel methodology of modularizing the given architecture to enable parallel loading and to reduce the model loading latency. (b) The loading method proposes a novel technique to perform loading and inferencing simultaneously. (c) The model splitting work facilitates the on-device training by assigning the trainable layers as a separate module. Modularizing the DNN architecture with the proposed method for improving the loading time is a first work of its kind to the best of our knowledge.

## 2 Related Work

In order to reach up to maximum dimensions of human life, DNNs have traveled a journey meeting several modifications. As in current society mobile phone serves the maximum workload for providing ease to life by its prediction ability. There have been couple of work in the domain of speeding up the inference. Kim et al. (2017) proposed a lightweightand specially structured DNN, which is split friendly and automatically learns to split the network on the basis of certain assignment matrices. The design of the network helps resulting into fewer parameters achieving parallelism at the inference time. This way the proposed architecture is optimal in space and quick in response.Rajbhandari et al. (2019) proposed a novel approach about improving the training performanceover enormously huge parameters. The work (by Lepikhin et al. (2020)) provides a parallel way of performing training with a huge number of parameters. The proposed method has been applied for training the neural machine translation transformer with 600 billion parameters. The proposed parallel solution enables to finishthe training of translating 100 languages to the English language with 2048 V3 TPU accelerators in 4 days only. Ushakov et al. (2018) have proposed split neural network particularly for addressing the problem of executing the full neural network on resource constrained mobile device. Huang et al. (2019), in their paper proposed an approach known as Gpipe, which is based on the idea of pipelining the subsequences of the layers on different accelerators. Matsubara et al. (2019), in their work proposed a way of offloading the DNN compute work to edge servers in order to speed up the inference. In their approach a DNN is split into head model and tail model. The proposed method helps in reducing the computational load of mobile device. Shao & Zhang (2020) proposed end to end system for balancing the on-device computation using other available hardware in network. In the proposed Bottelnet++ architecture authors claim for exploiting the sparsity and fault tolerant property of the intermediate features in the deep neural architecture. Jin & Hong (2019), proposed split convolutional neural network which splits the images in the patches and convolution is applied independently on all the patches on early stage. The proposed idea helps in significantly higher training scalability, which enables the deep learning algorithms to work with limited GPU memory.

## 3 Preliminary

The section gives detail about all the existing framework/ techniques, which are used in this work.

### 3.1 Reinforcement learning (RL)

RL is based on the action-reward idea. The learning agent takes action based on certain policy and environment replies back in terms of reward corresponding to the taken action. The method is basically used in the place when true label is not available and the only possibility left is hit and

trial. The RL algorithm proceeds in a way of defining the value of each state by modeling the given problem in terms of Markov Decision Process (MDP) which defines state, action, reward, discount, and state transition probability $(S, A, R, \gamma, P)$ . Where MDP is based on Markov principles which state that the future is independent of the past given present.
**STATE (S)** is defined as the information with the agent which is used to take action. On ground of Markov principles the state of the agent is defined as $P(S_{t+1}|S_t) = P(S_{t+1}|S_1...S_t)$.Which says probability of occurrence of a state depends only on one previous state.
**ACTION (A)** can be defined as an outcome of applying a policy on the state as $A = \pi(S)$. Where in the process of reinforcement learning a policy is learnt to derive an action corresponding to attainment of maximum reward.
**STATE VALUE (V(S))** The value of state defines the total reward achievable from that state. Which can also be defined as total return at that state $S(t) = E(G(t)|S_t = s)$ Where $G(t) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}...$ and G(t) is known as return value and $\gamma$ is the discount on future reward.
**POLICY ($\pi$)** A policy is defined as a function applied on a state to get action. Policy is basically of two types eg. Deterministic, Stochastic.
**ACTION STATE VALUE ($Q(S, A)$)** It is defined as expected return from state $S$ after an action $A$ the following policy $\pi, Q(S, A) = E_\pi[G_t|S_t = S \wedge A_t = A]$. As per Bellman expectation equation: $Q_\pi(S, A) = E_\pi[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1})]|S_t = S, A_t = A$

**OPTIMAL VALUE FUNCTION** The optimal value function is defined as the function deriving maximum value at a given state among all the policies $\pi$.

- Optimal state value$V^*(S) = max^\pi(V_p i(S)$
- Optimal action value$Q^*(S, A) = max^\pi(Q_\pi(V_p i(S)))$

# 4 METHODOLOGY

This section explains the basic notations, problem formulation and proposed solution. In the following method DNN is Deep Neural Network, RLU is Reinforcement Learning Unit, modularized model, child model and module all of these three terms represent same thing.

## 4.1 PROBLEM DEFINITIONS

If a given pretrained model $M$ has $N$ blocks such that $M = Block_1||Block_2, ...||Block_N$, we have to find total $k$ segments of the given architecture along with their position $\vec{P} = 0, P_1, P_2, ..., P_k$ such that $M_i = Block_{P[i-1]}||...||Block_{P[i]}$

$$T_{Load}(M_1) = T_{Load}(M_2) = ... = T_{Load}(M_k) \tag{1}$$

$$if L_i \in M, gradient(L_i) = True => L_i \in Block_M \tag{2}$$

$L_i$ is the $i^{th}$ layer and $i \in (N - m, N)$ where Eq. (1) signifies condition of true loading parallelism and Eq. (2) signifies the need to keep trainable layers at the last module. In general, by keeping additional constraints while splitting the model among smaller modules helps in solving two problems. However, it has been noticed that Eq. (1) works under certain tolerance such that $T_{Load}(M_1) = T_{Load}(M_2) \pm \delta ... = T_{Load}(M_k) \pm \delta$.

## 4.2 PROPOSED SOLUTION

As model loading is the necessary step for any app which adheres to some sort of intelligence or learning ability. The DNN model is the additional overhead for the apps and limits its usage for real time applications. Despite the fact that AI based algorithms are better in their approximation over geometrical methods which makes them invariant to (occlusion, illumination etc.) shortcomings of geometrical methods. The only concern with AI algorithms are their additionally imposed overhead, for instance if an app takes $T_1, T_2$ time in loading and processing respectively, then its DNN powered equivalent version will take $T_1 + \delta t_1$ in loading and $T_2 + \delta t_2$in processing input sample through the app, additionally imposing overall response time by $\delta t1 + \delta t2$. The increase in response time is responsible for affected user experience, which restrains the AI algorithms to be used over geometry driven algorithms. In this paper, the solution of the addressed problem is given in terms of RLU

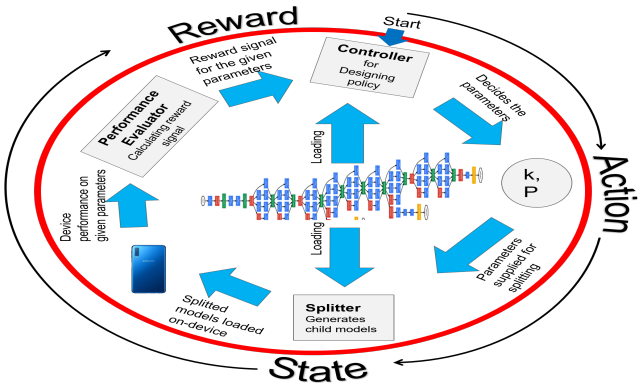


Figure 1: Reinforcement Learning framework to attain parameters

framework and execution policy. In the process of solving the problem of loading latency, it is quite understandable to realize its direct dependency on disk space of the object. This intuitive idea directs us to come up with the method of splitting the given DNN among tiny models as a vector $\vec{M} = [m_1, m_2..., m_k]$. As we already know that smaller the model lesser will be loading latency and if they are loaded in parallel with each other, then over all loading latency will depend only on the biggest among the existing tiny models. Therefore, in this work we propose to split the models using Reinforcement Learning and then provide an execution policy, which helps in parallel loading (case#1) and parallel loading in parallel with inference (case#2).

#### 4.2.1 Reinforcement Learning Unit

In the proposed work reinforcement learning is used for automating the model splitting process. This work proposes the asymmetric splitting of the given DNN architecture. Whereas similarity is maintained in terms of loading latency based on the hypothesis, which says that loading time for the pool of models is the maximum loading time taken by the heaviest model in the pool. Hence, it is proposed to split the model in a way that all the child models should take similar loading time. Reinforcement learning provides the number of splitting and splitting index corresponding to each of the concerned DNN. The problem environment can be formulated using standard MDP (Markov Decision Process) with $(S, A, P, R, )$, where $S$ is a state space, $A$ is action space, $P$ is the transition matrix, $R$ is reward function and  is discounted value.

**State Space** A state is defined as information used to determine what happens next in terms of action. The state can formally be stated as $S_t = f(H_t)$ where $s$ is state, $H$ is the history and $t$ is the time stamp. In the present work state is defined as a $\vec{S} = (C_{split}, I_{split}, n_p, n_r)$ where $C_{split}$ is known as split count and tells about total number of splits required to be done, $I_{split}$ defines the current index of split, $n_p$ is the total number of processed nodes and $n_r$ is the remaining nodes to be processed next.

**Assumptions:a** The nodes in the graph are static in nature and any of the RL process/action does not affect their relative position in the graph.

**Assumptions:b** The DNN architecture, treated as graph is prone to split at any point resulting into two sub graphs, which are independent models given the input.

**Action Space** The action space of the concerned problem deals with possibility of all the options for decision, thereby it is the Cartesian product of the total number of possible splits $C_{split}$ and $I_{split}$ as shown in Eq.(3).

$$A = C_{split} X I_{split} \tag{3}$$

**Reward function** The reward is given for the agents decision on splitting count and splitting index. If $s_1$ is the child model generated by taking action $A$, such that if $\tau = Loading_latency(s_1)$ , then reward parameter is defined in Eq. (4).

$$r_s = \begin{cases} 0 & \text{if } \tau = T/C_{split} \pm \epsilon \\ -1 & \text{otherwise} \end{cases} . \tag{4}$$

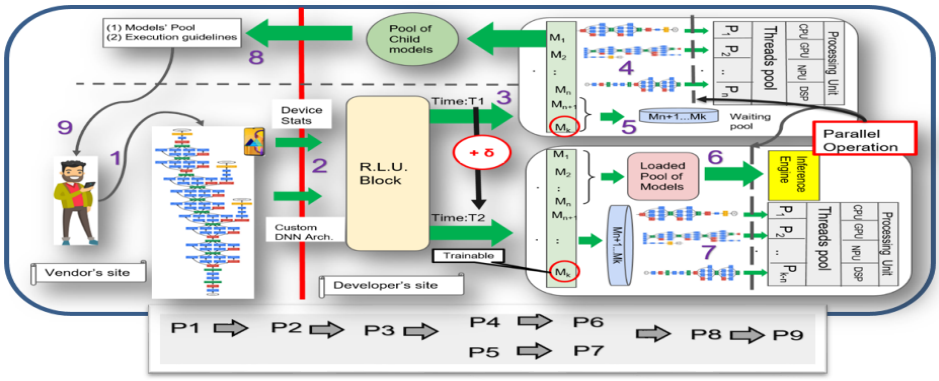


Figure 2: Architecture of proposed methodology

Where $T$ is the total loading time of the full model. The reward parameter reinforces the behavior of having split which should lead to a sub model having load time at max equal to the load time of a proportionally equivalent decomposed model. This method drives towards achieving true parallelism.

**Optimization Action-State value** The most appropriate policy to reach the target point is obtained by iterative process, where each iteration leads to update the action state value $Q(S, A)$, which is commonly known as Q-value.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A')) - Q(S_t, A_t) \tag{5}$$

where $A' \sim \pi(.|S_t)$ is the action sampled from the available option at state $S_t$, $\gamma$ is the discount value and $\alpha$ is the learning rate. The initial value of $Q(S, A)$ is maintained in the form of look-up table. The entries in the action-state value table corresponds to the $Q$ value of action $A$ taken on state $S$. In the proposed work the stated function is performed by Controller in RLU (Fig. 1). The function of Reinforcement Learning Unit can be understood well by looking at the Fig. 1, where it is worth noticing that initial values of splitting count and index $(k, p)$ is given to the model splitter which provides the output in form of child models. Child models performance is calculated by loading on device and outcome is shared with performance evaluator and equivalent reward value is calculated. The reward value is supplied to the controller, which updates the state value and again participates in next iteration with better decision.

**Algorithm 1** RL_Framework(Pretrained DNN: $M, s$)

$Block_1||Block_2||...||Block_N = M$
**for** $i = 1$ to $N$ **do**
  $T_{Ld}[i] = LoadingTime(Block_i)$
  $T_{in}[i] = InferenceTime(Block_i)$
**end for**
$\vec{P}, k = RLU(T_{ld}, T_{In}, Block_1||Block_2||...||Block_N, s)$
$\{\vec{P}$ is the split positions$\}$
**for** $i = 1$ to $k$ **do**
  **if** $i == k - 1$ **then**
    $M_i = Block_{P[i]-s}||...||Block_{P[i]}$
  **else**
    $M_i = Block_{P[i-1]}||...||Block_{P[i]}$
  **end if**
**end for**
**return** $M_1, M_2..., M_k$

### 4.2.2 Execution Policy of the Proposed Method

The given pretrained model with $N$ blocks is modularized among $k$ child models as explained in previous sub section and shown in Fig. 1. There are two decisive steps in the approach (1). Total number of splits. (2). Split positions. The architecture of the proposed method can be seen in Fig. 2.

**Algorithm 2** Execution(Models: $\vec{M}, Threads : \vec{T}, In$)

CASE:1
**if** $len(\vec{T}) > len(\vec{M})$ **then**
  **for** $i = 1$ to $\vec{M}$ **do**
    $\vec{T}[i] = M_i$ Parallel Loading
  **end for**
**end if**
$INFERENCE(\vec{T}, In)$
CASE:2
**if** $len(\vec{T}) < len(\vec{M})$ **then**
  **for** $i = 0$ to $len(\vec{T})$ **do**
    $\vec{T}[i] \leftarrow M_i ParallelLoading$
  **end for**
  **for** $i = len(\vec{T}) + 1$ to $len(\vec{M})$ **do**
    $\vec{T_i} \leftarrow M_i$ Parallel Loading
    **for** $i = 1$ to $len(\vec{T})$ **do**
      $x = M_i(In)$Inference
      $In = x$
      **if** $i == (\vec{T} - 1)$ **then**
        **return** (x)
      **end if**
    **end for**
  **end for**
**end if**

It can be noticed that the process is triggered with the users requirement of converting model $M$ to $M'$ to attain reduced loading latency by using RLU. The RLU helps in deciding the number of child models and their indexes for a given device. In general, if number of generated child models are fewer than available H/W threads then all of the child models can be loaded in parallel. However, in few cases number of child models could be equal to or more than available H/W threads. In these cases, because of having high degree of loading overhead, loading time increases abnormally, which can be controlled by restraining the degree of parallel loading. The stated concept is shown as process P3: T1 in architecture diagram (Fig.2), where it can be seen that child models $M_1$ to $M_n$ are used for parallel loading as per availability of available H/W threads. However, it is recommended to have number of sub models lesser than available H/W threads in order to achieve the better performance.

**Process- P3:T1** This process is responsible to manage the segregation of models in two categories each taken care by process P4 and P5 respectively. The child model circled with red oval is a last indexed child model with live gradients, which supports for finetuning. **Process-P4** The child models of category-A are loaded in parallel on H/W threads. **Process-P5** It collects the remaining child models in waiting pool. Process P5 is executed in parallel to process P4. **Process- P3:T2** As shown in Fig. 2, process initiates $\delta$ time after P3:T1. By this time, the models which were plugged-in to the H/W threads are in loaded state and rest of the child models are in a state ready to be loaded. **Process-P6, P7** Both of these process proceeds in parallel when already loaded pool of child models are plugged-in to compute unit for inferencing by process P6 and models waiting in pool are loaded by process P7. The child models collected after these processes are bundled together in process **P8** and are supplied back to vendor along with execution scheme by process **P9**. The implementation methodology of the proposed approach can be referred from Algorithm 1. The loading time and inference time of each block is calculated in line number 3, 4. The latency stack along with the model (set of computation blocks) is given to the Reinforcement Learning Unit (RLU). The function of RLU (Q -Learning) is to consider load timings of each block and provide the optimal number of split value ($k$) and their most appropriate position (true parallelism as reward). Where it should be worth noticing that the latency varies as per the loading hardware. Algorithm 2 gives the methodological detail about parallel loading of all the split models and inference across all of them. There are two probable cases. **Case1** When number of split models are less than available threads, then it will be a trivial case of loading all of the models simultaneously and will help in achieving true parallelism. **Case2** Showcases the aspect, when number of split models are more than available threads. In this case only few models (same as number of available threads) can be

Table 1: Comparative performance of the proposed method and relative positions of child models on-device with Arm-NN platform using Tensorflow (.pb) graph file

| **Model** | **Position** | | | | **Latency (ms)** | |
|---|---|---|---|---|---|---|
| | **Child1** | **Child2** | **Child3** | **Child4** | **Original** | **Proposed** |
| **Low light Photography** | 0-115 | 115-133 | 133-145 | 145- | 228 | 90 |
| **Mobilenet-V1** | 0-250 | 250-325 | 325-351 | 351- | 212 | 88 |
| **Inception-V3** | 0-650 | 650-1000 | 1000-1100 | 1100- | 960 | 430 |

Table 2: Pytorch-Beginning and Ending location of each child segment

| **Models** | **Position of child model#** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** |
| **Inception-V3** | 0-9 | 10-12 | 13-15 | 16-end | – | | - | - | - | - | - |
| **Resnet-152** | 0-3 | 4 | 5 | 6 | 7 | 8-end | - | - | - | - | - |
| **Squeezenet** | 0-5 | 6-9 | 10-13 | 14-end | – | - | - | | - | - | - |
| **Densenet** | 0-4 | 5 | 6 | 7 | 8.0-8.16 | 8.17-8.32 | 8.33-8.48 | 9 | 10.0-10.15 | 10.16-10.32 | 11-end |

loaded at a time and rest of the models will have to wait for the completion of the first loading. Lines 13-19 in Algorithm 2 show that already loaded model can be exploited for inference in parallel with the loading of the second pool of models.

#### 4.2.3 On-device Personalization

Many of the phone applications can be tuned according to the intended user in order to provide improved experience. Usually, these facilities are made available using cloud, where usage pattern is captured using transfer learning. The proposed work is developed to provide ease for on-device training so that personalized experience can be created without the need of sending sensitive information out of device. As it is quite evident that transfer learning works impressively by providing the customized training to the last layers of a DNN model. The proposed method intentionally keeps the trainable layers at last child model. The separation of trainable layers (gradients are True) from frozen layers (gradients are false) makes it easy for the framework to perform training.

## 5 Experiments

In this work, we have performed the experiments of splitting the models on six DNN architectures, which are Camera model (low light photography), Mobilenet-V1, Inception-V3, Resnet152, Squeezenet and Densenet201.
**(a). Modelset** This section gives the detail about the used models.
**Low light Photography Model** Photographic model is the customized camera based deep learning DNN architecture (a form of Resnet) which is used for low light photo capturing. The DNN architecture is extensively used across several handheld devices for photo clarity.
**Mobilenet-V1** Mobilenet model Howard et al. (2017) is an open source contribution from Google and has been a first choice for mobile application.
**Inception-V3** Inception-V3 (Xia et al. (2017)) introduces a novel concept of Inception block, which solves the problem of selecting the filter size and uses three sizes (1x1, 3x3, 5x5) in one block.
**Resnet-152** This work (He et al. (2016)) proposes the residual learning framework particularly for easy training of the deep network. The network is made deep as 152 layers, which is 8 times deeper than its peers VGGNet but lower in complexity.
**Squeezenet** In SqueezeNet (Iandola et al. (2016)) authors proposed a design to squeeze the knowledge of a neural network for resource constraint environment by following a couple of strategies.
**Densenet** Densenet (Huang et al. (2017)) is developed by exploiting the idea of residual connections with the varying scale of DNN. This way Desnenet used to have several connections from earlier

Table 3: Pytorch- Actual and Proposed performance on 5 observations

| Models | Performance | Observations | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| Inception-V3 | Actual | 4.2253 | 4.3935 | 3.8204 | 3.8085 | 4.2440 |
| | Proposed | .6328 | .7718 | .6736 | .7080 | .6690 |
| Resnet-152 | Actual | 7.2572 | 7.3213 | 7.2389 | 7.3375 | 7.3037 |
| | Proposed | .9990 | 1.1890 | .9436 | 1.0145 | 1.2538 |
| Squeezenet | Actual | .3617 | .3284 | .3775 | .3709 | .3909 |
| | Proposed | .2977 | .2795 | .3029 | .2821 | .2857 |
| Densenet | Actual | 2.1816 | 2.2205 | 2.5217 | 2.6003 | 2.7886 |
| | Proposed | .7409 | .6977 | .5535 | .7397 | .7690 |

layers to way long situate layers.
**(b). Experimental Framework** The experiments are performed on ArmNN-Tensorflow and Pytorch framework.
**PyTorch** In Pytorch [1] specific layers up to a certain index point, are selected and saved as a child model. The generated child model is saved as a checkpoint file. The performance of the child model against the full model is witnessed on a Linux powered Desktop PC having Nvidia 1080Ti GPU.
**Arm NN-Tensorflow** We have experimented using Arm-NN GPU. We used Tensorflow 2.2 as a source of generating the pretrained weights of child models. The execution environment is Samsung Galaxy S20 with Snapdragon 865 processor having 8 GPU threads using Open-CL kernels exploiting one thread for loading one model.

## 6 Results and Discussion

The proposed method has been tested on edge device platform and on desktop platform with popular machine learning framework like Arm NN and Pytorch respectively. Table I shows the relative loading performance of the proposed method with existing method (State-of-the-art), where latency improvement can be seen with all the tested models. Table I also contains placement of splits in original model corresponding to each child. Table II shows the beginning, end position of each child model in Pytorch environment. In the representation $a - b$, $a$ is the beginning and $b$ is the end position, whereas $x.y - x.z$ shows beginning position from sub block $y$ to end position at sub-block $z$, within the main block $x$. Table III presents the comparative performance of the actual (state-of-the-art, benchmark Pytorch) results and proposed results. Significant loading time improvement (in second) (up to 7X in InceptionV3, Resnet152) with the proposed method is witnessed in all observations. In the results shown in Table-I it is clear to observe that number of splits are fixed as 4 for Arm-NN environment. The crucial decision about splitting count is a device dependent factor. As each DNN model is designed independently, therefore while splitting the model, it is recommended to follow the design architecture before splitting.

## 7 Conclusion

This paper presents a RL driven novel asymmetric model splitting based parallel loading method, which leads to significant improvement (up to 7X) in DNN loading. The proposed work has been verified with wide range of DNN architectures operated with Desktop/Nvidia GPU on Pytorch environment and in Arm-NN environment using Tensorflow based frozen model (.pb). The proposed method also provides support for On-device training for customized experience to the user. The DNN modularization method reduces the loading time substantially thereby improving the user experience while interacting with any intelligent app powered by DNN. The proposed method can be used to improve the response time (loading and switching) of low light camera, selfie focus etc. In general, the proposed method can be used with any computer vision based deep learning models in the exact form. In future the ability of the proposed method can be tested with all the use case DNN models in order to leverage it as an ubiquitous method to work with any application which uses DNN.

[1] http://pytorch.org/tutorials /beginner/saving_loading_models.html

## REFERENCES

Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1314–1324, 2019.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in neural information processing systems*, pp. 103–112, 2019.

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

Tian Jin and Seokin Hong. Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 835–847, 2019.

Juyong Kim, Yookoon Park, Gunhee Kim, and Sung Ju Hwang. Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization. In *International Conference on Machine Learning*, pp. 1866–1874, 2017.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Yoshitomo Matsubara, Sabur Baidya, Davide Callegaro, Marco Levorato, and Sameer Singh. Distilled split deep neural networks for edge-assisted real-time systems. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pp. 21–26, 2019.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training a trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2019.

S Rallapalli, H Qiu, A Bency, S Karthikeyan, R Govindan, B Manjunath, and R Urgaonkar. Are very deep neural networks feasible on mobile devices. *IEEE Trans. Circ. Syst. Video Technol*, 2016.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

Jiawei Shao and Jun Zhang. Bottlenet++: An end-to-end approach for feature compression in device-edge co-inference systems. In *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6. IEEE, 2020.

Yury A Ushakov, Petr N Polezhaev, Aleksandr E Shukhman, Margarita V Ushakova, and MV Nadezhda. Split neural networks for mobile devices. In *2018 26th Telecommunications Forum (TELFOR)*, pp. 420–425. IEEE, 2018.

Xiaoling Xia, Cui Xu, and Bing Nan. Inception-v3 for flower classification. In *2017 2nd International Conference on Image, Vision and Computing (ICIVC)*, pp. 783–787. IEEE, 2017.

Di Zhuang, Nam Nguyen, Keyu Chen, and J Morris Chang. Saia: Split artificial intelligence architecture for mobile healthcare system. *arXiv preprint arXiv:2004.12059*, 2020.