Master Internship Report

Report

AI on the Edge

Submitted by

Wassim Seifeddine
Masters Student
ESIEE Paris

Under the guidance of

Dr. Nadjib Achir
Associate Professor at Inria-Saclay

Dr. Cedric Adjih
Research Scientist at Inria-Saclay

Tribe Team

Institut national de recherche en informatique et en automatique
1 Rue Honoré d'Estienne d'Orves, 91120 Palaiseau

Acknowledgements

Before starting with this report. I would like to express my great appreciation for ESIEE Paris represented by Dr. Yasmina Abdeddaïm for being a great educational institute and offering such amazing opportunities for it's student.

Second I would like to thank INRIA for giving me the opportunity to do this internship.

I would like to express my great gratitude to my supervisors throughout this internship, Dr.Nadjib Achir and Dr. Cedric Adjih for providing such astounding guidance and support. They guided me through my tasks and how to accomplish them with ease and confidence.

Table Of Contents

1

Introduction and Problem Definition

The past two decades have seen tremendous changes in the way people's lives are affected by IT. One notable trend is the emergence of the Internet of Things (IoT). IoT (low-end) devices are connected to form a much larger system and introduce a large number of new applications. Both low-end and high-end devices are also continuously generating a tremendous amount of data to feed some application-based deep learning algorithms to extract and make critical decisions.Such applications can perform anomaly detection through environment sensors, AI-based camera processing, audio recognition, smart localization, etc. Even if the end-devices' capacity is continuously increasing with what is known as TinyML, unfortunately, it is still not easy to deploy the best (or even sometimes, just acceptable) deep learning algorithms on them. In this area, one specific focus is on running deep learning models, that is deep neural networks.

To meet the computational requirements of Deep Learning, a well-established approach is to leverage the Cloud computing paradigm. The end devices have to offload their data and or model to the Cloud, which raises many challenges, such as latency and scalability. To overcome this limitation, one possible approach is to push the processing or learning at the edge. This is known as Edge AI, one quickly developing field. The most straightforward

method is to offload all the computation from the end devices to the edge server. Another alternative is to consider both end-devices and edge servers by distributing the model and the computation between both sides.

In this work, we researched on ways of how to run deep learning models on the edge with minimal resource consumption. The work ranged from model compression techniques for reducing the memory, space and time needed for the model to run like pruning and quantization to more advanced techniques like conditional computation, network slimming and slicing. Another part of the research focused on the optimization problem dealing with finding the best solution for processing an input through the model, this optimization problem can have multiple constraints like time deadline, energy consumption limit while maximizing another metric. Finally we propose a novel algorithm for optimizing model execution between end device, edge devices and cloud servers. This technique will allow to optimize a single model placement between multiple hierarchical levels of execution capable devices with different computational abilities. Moreover,this technique aims to preserve high accuracy while using the required amount of computation resources in terms of energy.For time critical applications, this technique can also be constrained with a strict time deadline to ensure that the model execution finishes before.

The goal of the internship is to imagine novel strategies for efficiently running deep learning models on the edge, by smartly splitting their processing between the constrained end-devices, the edge servers, and even the Cloud if necessary. This can also include the larger but essential question of training the models themselves.

This report will be structured in the following way. First we will discuss about techniques that we learned about while reading the literature that are related to running deep learning models on resource constrained devices.Then in the second section, we will talk about our contribution to this field, the work will be discussed thoroughly step by step from the model architecture to the training procedure. Then we will discuss the optimization problem that we are trying to solve and how we approached it.After that we will present the results we got and how do we interpret them.Then we will discuss about

future work that might be done as a sequel to our contribution. Finally, we will end with a conclusion about what we learned during this internship.

2

Related Work

In this section we will discuss the currently available state of the art techniques that enable model deployment on resource constraint devices. The actual definition of resource constraint devices is generally vague, we define it by computational device d that alone is not able to execute deep learning model M *efficiently*.
These techniques range from methods that try to compress the model itself such as Model Pruning or Model Quantization to techniques that try to run the model on the device in a 'efficient way', these are categorized under conditional computation techniques. The last set of methods that we are going to discuss are methods that try to understand how to model work and use just a portion of it depending on the current needs

2.1 Model Compression

Model compression [3] refers to set of techniques that are designed in order to reduce model's pressure on the running device while keeping the model architecture.They achieve this by modifying the weights and biases matrices which consitute the main part of the model. The two most common ways for model compression are model pruning and model quantization. To understand better how these techniques work, we first have to understand what a deep learning model is made up of.

Neural network based models consist of set of layers, each layer consist of two main things, the weight matrix and the bias vector. The $m * n$ weight matrix W_l is the matrix of parameters that are learned during training and the bias vector is just a simple m size vector that is also learned during training of the model. During the computation of the model whether it is in inference phase or training phase, a $m * n$ matrix is also calculated at each layer which is the result of the multiplication of the input X_l to this layer l by it's weight matrix W_l, this layer is called the activation matrix of layer l denoted by a_l . However, after saving the model, or even when the model is not doing any computation , the activation matrix is not there as it is calculated on the fly. to generalize these matrices across the whole model, we can say that each model is made up of a set of weight matrices W, the set of bias vectors B and the set of activation matrices A.

2.1.1 Model Pruning

Model Pruning[3] is a model compression technique that reduces the model size by reducing weights matrices W. It is one of the model compression techniques that preserve the model architecture while changing the computation in it.The way it works is that it tried to eliminated some units in the weight matrices by zeroing them out. While creating ways to "eliminate" entries in the weight matrix might seem trivial, finding a technique that preserves the model accuracy while eliminating weight units might be challenging. The most common approach is zeroing out the weights with low magnitude. There are multiple 'ways to define what 'low magnitude' weights are, one of the most common methods is using the $l1$ norm of the weights and setting a percentage of the weights you have to prune. In other words, you calculate the $l1$ norm of all weights and then with the percentage you have p you eliminate the $(p * number\ of\ weights)$ lowest weight units. After that you compute a mask matrix which has the same size has your weight matrix with 0 where you want to cancel out the weights.

$$Weight\ Matrix\ W = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad Mask\ Matrix\ M = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Now during the execution of the model, instead of multiplying the input of the layer X_l by the weight matrix W_l directly, you first the weight matrix W_l by the mask matrix M_l which will result in a sparse matrix and then you multiple this result by X_l.One of the side effects of using this approach is that now you model approximately double in size since for each layer L you have two matrices W^l and M^l. However after finishing training the model, you can replace these two matrices by the result of their multiplication which is a sparse matrix this way your result with a much smaller model.

2.1.2 Model Quantization

Model Quantization[4] is also a model compression technique that is both is very effective and widely used. it consists of transforming the underlying precision of the weights of a model into a lower precision for faster processing and lowering the model size. What we mean by precision is how the components of the model are represented in memory by the computer. By default,in all modern everyday computers ,all the model is represented by 32-bit floats, quantization tries to represent the weights, biases and activations by a lower precision (16-bit, 8-bit and even 4-bit!) while preserving the model accuracy. A common way for quantization is called post-training quantization, where you train your model in single precision (32-bit float) and then you quantize it to a lower precision (16-bit or 8-bit integer) while trying to preserve some metrics. The process has some steps to preserve the accuracy as much as possible.

Post-Training quantization usually takes a single precision model and tries to lower the parameters (weights and biases) to a lower representation, However, lowering the representation for all the parameters in the model in the same manner might result in very bad performance, this is be-

cause the ranges of where these parameter live might change from a layer to another. One simple technical trick is to monitor these ranges and quantize each layer accordingly.The ways this is done is by attaching observers to the non-quantized model, run the model through multiple validation steps (without actually training it). This way, these observers will be able to collect the ranges for the activations of each layer and then the quantizer will quantize according to these ranges.

2.2 Conditional Computation

Conditional computation is very recent and active research field in machine learning. It comes from the idea that in most cases you do not need to run the whole network in order to get decent reliable predictions from an input x. In other words, it conditions that computation of the network based on the data point x that you are trying to predict now.
A lot of approaches are used to accomplish this. Some of the most notable ones are BranchyNets [15] which proved that almost 98% of that CIFAR10[9] does not need the whole network (VGG16[13] in their case) for accurate predictions.
SkipNets [17] and Slimmable Neural Networks [18] are also another more complex approaches for conditional computation, however these network usually require an external entity, normally a sequential decision making agent to decide what to do and where.
Conditional computation was thoroughly studied in our work due to it's simplicity and power for efficient deep learning on edge devices. In the below couple of sections, we will go over several techniques for conditional computation to describe them in details along with their advantages and disadvantages.

2.2.1 Early Exit Networks

Early Exit Networks are a subset of conditional computation models where the network is trained with multiple exit points along with the network com-

putational graph.These exit points are evaluated along with the computation of an input x and if the confidence at a certain branch is sufficient enough for the application, the network can exit without evaluating the rest of the layers. BranchyNet [15] is one of the first examples of early exit networks.They showed that adding exits to the exiting VGG16[13] network can have a lot of benefits. They conducted several experiments on the CIFAR10[9] dataset and showed that around 98% of this dataset do not need to compute all the model to get decent predictions.

Training these networks might seem complex due ot the multiple output paths however it turns out to be very simple. In BranchyNet [15] the approach for learning is jointly training on all the exit paths and optimizing the weighted summation of the loss. Training is this manner add a regularization aspect for training the network as the loss you are trying to optimize is the joint loss of all the exit points you have which will lead to less overfitting. Although theoretically you can add an exit after every layer you have in your network, this would lead to an increase in computational resources as the model has to execute every exit after every layer and this will increase a lot the network size as exits are made up of usually small layers but nevertheless they will add up. Finding the optimal number of exits and where to add them is not a easy process, it requires a lot of experimentation to test and monitor where the optimal placement should be.

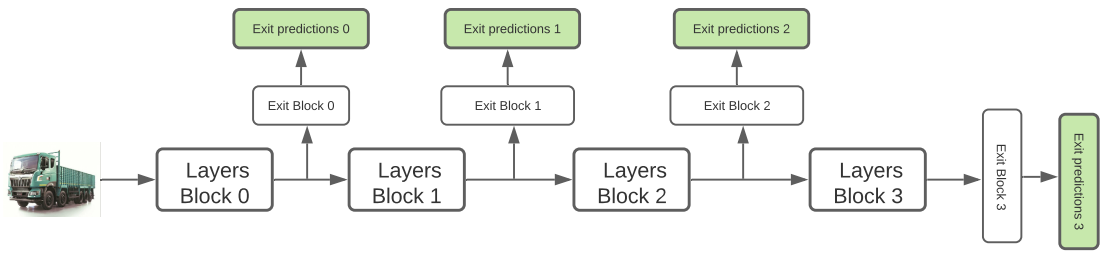


Figure 2.1: Early Exit Network with 4 exits

2.2.2 SkipNets

SkipNets[17] is another approach of early exit networks where they try to condition the computation of the network based on the current input x. However, instead of doing simple exit points, they proposed a new way to skip convolutional layer computation based on a input

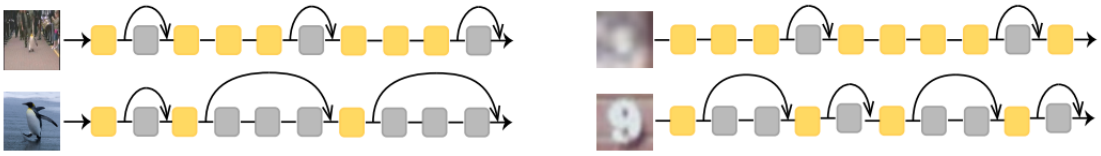


Figure 2.2: "The SkipNet learns to skip convolutional layers on a per-input basis. More layers are executed for challenging images (top) than easy images (bottom)" [17]

They added these "Skip Connections" to a Residual Network, more commonly known as ResNet[6] and they showed their approach can reduce computation between 30-90% while preserving accuracy of the original model and they out-perform most dynamic neural network techniques along static model compression techniques discussed in section 2.1.
The tricky part of their approach is the decision maker that decides which convolutional layers to skip. The main problem is that they can can not train this task jointly with the network similar to what BranchyNet[15] does because this task non-differentiable. The proposed a much more interesting approach which combines supervised learning and reinforcement learning after formulating this problem as a sequential decision making problem. In this context, they train a reinforcement learning agent to learn the optimal policy on how and where to skip the convolutional layers.

2.2.3 Throttable Neural Networks

Throttable Neural Networks, specifically, Dynamically Throttleable Neural Networks (TNN) [10] is also a subset of conditional computation that aims to reduce the computational load on of the network while maintaining the

model performance. Using the modularity property of neural networks, they divided a network into blocks which they are TNN blocks. These block are plug-and-play modules that can be executed or not based on the decision of an external entity. This external decision making entity is a reinforcement learning agent that they call "Context-Aware Controller" which outputs a simple number u called the utilization parameter which is used by the neural network as a metric on how much layers it should turn off.

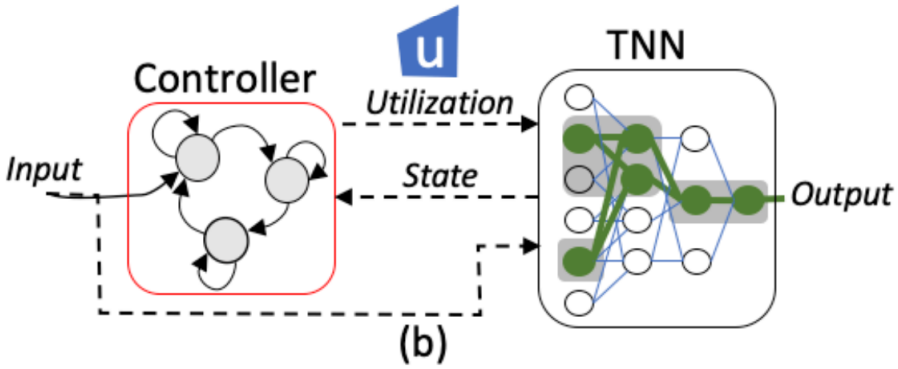


Figure 2.3: "Proposed dynamically throttleable network" [10]

2.3 TinyML

2.3.1 Introduction

TinyML refers to a type of machine learning that aims to put machine or even deep learning models on tiny devices. These devices can be your smartphone, Google Home, Smart Fridge, basically everything that can do some sort of computation and can benefit from machine learning is under the umbrella of TinyML. Everything we discussed so far in this report can be considered part of TinyML as all the techniques are way to running deep learning models in faster more efficient way. However, TinyML comes with a handful of software and hardware utilities that are aimed for the same cause making it necessary to be part of this report. We will go over some software libraries and packages

and briefly describe what are they developed for, then we will go over some hardware such as computational accelerators that companies are designing to put AI on the edge. But before we dive into the technicalities of tinyML, we need to understand it's applications.

2.3.2 Applications

TinyML comes with a variety of applications, some of the them are in-use now in our daily life and others are still not very applicable due to the computational constraint of small embedded devices.

- *Wake Word Detection*: Probably the easiest application to understand is the one that we use daily. When you say "OK Google" or "Hey Siri" or "Hey Alexa" your device will automatically turn on and start listening for more commands, however the important aspect is that the device should listen and recognize its "wake word" in a split of a second.Also, while listening to the "wake word" the device should not drain the battery by staying on all the time, this is where TinyML comes into place.

- *Intrusion Detection*: Nowadays, you do not expect your security cameras only to record what's going on, you expect them to warn you if someone is trying to break into your house. To do that the cameras should be always trying to figure out if the picture they are taking has someone inside, this is called image classification and object detection which is a very computationally expensive task. One way to solve this is that the cameras send their video footage to a cloud server where the actual machine learning is done but this obviously has a LOT of security, privacy and cost issues, the best solution is that the machine learning model runs directly on the device in psuedo real-time. This is a classical application of TinyML.

- *Next word prediction*: When you are using your phone to chat with people, you expect your keyboard to give our predictions of word or phrase you might say next. Also, you expect your keyboard to "adapt"

to your way to talking and learn vocabulary that are closely related to your and start suggesting them but at the same time you will not appreciate if your phone is sending everything you type to a cloud server that is owned by a tech giant that has no respect for your privacy, you want everything to be done on your phone offline, which is what Google is working on with their Gboard app[5]. This is also a classical application of TinyML.

2.3.3 Software

In order to have deep learning models on edge devices, you need to have the software tool-kits to enable that. There is a plethora of such tool-kits that we will discuss some in this section.

- *Tensorflow Lite for microcontrollers*: Coming from the creators of the Tensorflow machine learning framework[1] this library is "designed to run machine learning models on microcontrollers and other devices with only few kilobytes of memory". There is a plethora of supported platforms and boards. The usual workflow you train the model in Tensorflow, then you convert it to a Tensorflow Lite model, after you convert it to a C code using the tools provided to you by the Tensorflow Lite for microcontrollers framework. After that you can deploy and run it on microcontrollers with the interpreter provided.

- *Edge Impulse*: Edge Impulse is one of the leading companies working on putting deep learning models on edge devices. They provide a tool-kit that integrates well with their ecosystem. This tool-kit will allow you to convert Tensorflow models into very light-weight models that you can deploy of a handful of development boards. The advantages of using Edge Impulse over building your model with Tensorflow Lite for microcontrollers is that you get a monitoring tool that will allow you to have an overview of how your model is performing.

- *PyCoral*: PyCoral is a library provided by Google's Coral that enables you to communicated to the Coral's boards to run deep learning mod-

els on them. Even though it is limited only to board from the same company, the impact of this company in the TinyML community is big enough.

2.3.4 Hardware

Apart from the software part of TinyML, you will need to run these efficient deep learning models on some devices. Although technically you can run them on any device with computational capability, companies are designing and producing specialized hardware that are optimized to accelerate the execution of these models while being on the low end of energy consumption and cost. Some examples of such hardware are discussed below

- *Google Coral*: "Coral is a complete toolkit to build products with local AI. Our on-device inferencing capabilities allow you to build products that are efficient, private, fast and offline." They provided accelerators on boards that you can buy for relatively cheap price. The main winning point of this company and their boards is that they are one of the very few companies that provide TPU enabled devices. They also integrate well with existing tools like Tensorflow[1] and Tensorflow Lite. Also, as discussed in the software section 2.3.3 they provide their own Python[16] library to communicate with the board.

- *Arduino Tiny Machine Learning Kit*: Coming from the famous single-board microcontrollers Arduino, Arduino Tiny Machine Learning Kit is aimed on helping people prototype with sample machine learning applications and help them learn about TinyML. The board can sense movement, acceleration, rotation, temperature, humidity, barometric pressure, sounds, gestures, proximity, color, and light intensity and includes a camera model. The kit comes with a price tag that makes it very accessible to most people.

- *NVIDIA Jetson Nano*: NVIDIA Jetson Nano is a product from the most famous GPU maker NVIDIA. NVIDIA Jetson Nano is a small,

powerful computer that lets you run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts. it comes with a small GPU that enables you to accelerate deep learning models making it on the more powerful spectrum in it's family. This Kit also comes with a enormous community support the developer documentation to help you experiment and deploy your TinyML applications with ease.

2.4 Conclusion

As we saw in the previous sections, there is a lot of work that is being researched on how to deploy machine learning models on resource constraint devices which gives us an ideas of how important this field is. However the focus was on not directed on how to run one model efficiently across multiple devices. For example, FastVA[14] focuses on how to balancer the execution of severals models with different capabilities across multiple devices depending on the need of the user. BranchyNet[15] and SkipNets [17] did not use the modularity characteristic of neural networks to divide the network on multiple devices. To our understanding, all previous work for "AI on the edge" did not tackle this problem.

3

Dynamic Hierarchical Neural Network Offloading

3.1 Introduction

After discussing about related work that is done for putting AI on the egde, it is time to discuss about our contribution. We divide our contribution into two folds. The first is that we introduce a novel algorithm for model placement across multiple hierarchical devices, second is that we added a new decision possibility for early exit network to make it more suitable for edge scenarios.

Our work focuses improving the use of early exit networks for scenarios where you have a hierarchy of computational devices. Until now,to our most understanding, the execution of such networks has been limited to one device and the set of decisions that you have at every exit point is either to Exit the network with the current accuracy or to continue running the network utilizing the energy available on the execution device. Our work aims at dividing a single model to deploy it on multiple devices and introduce a new action, which is to offload the computation to another more capable computational device.This action along with the novel design of early exit networks allows us to divide the model across multiple devices while aiming at maximizing the accuracy and minimizing the cost of running the model.

On each device, the action space remains the same until the top level devices where offloading is not an option. The set of actions *stay, offload, exit* are available throughout the network not just at the start. This design allows models to be very modular where you can split a model between two and more devices.

One natural consequence for our approach on previous work that tackled the idea of offloading computation to other devices like [14] is added privacy by default. In previous work done in offloading, the decision was either to offload the data before executing the model or running the model on the device.This approach will hinder privacy which might be critical if the data contains personally identifying information since you are offloading the data as it is from the start of the model. Our approach along with the use of early exit networks, allows the model to offload the computation in the intermediate layers thus offloading data that does not make sense for adversarial attacks on s small scale.

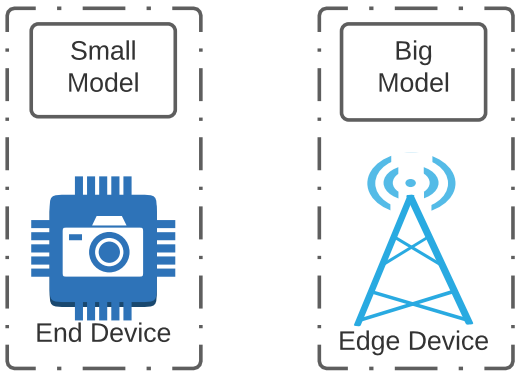


Figure 3.1: Previous approaches had 2 different models on different device depending on the device capability

In the following sections we will discuss our contribution in details starting from the model design and architecture of our neural network then moving on to the training procedure that we took to achieve good performance which we will discuss in the first results section. Then since our contribution also

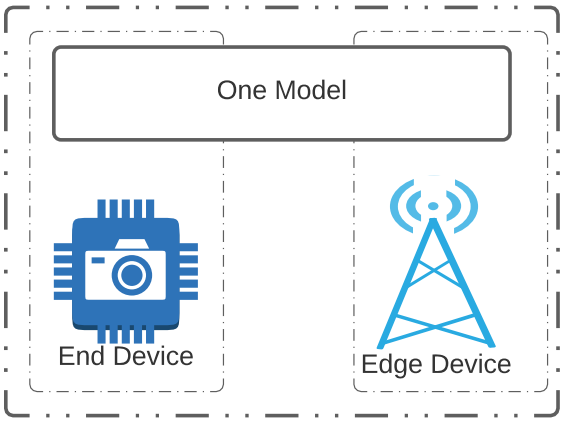


Figure 3.2: Our approach aims at having one model divided into multiple devices

contains an optimization problem that we have to solve, we will discuss and formulate it in details, after that we will briefly go over reinforcement learning which is the technique that we used to solve our optimization problem. Then we will discuss our approach in solving it along with the results we got.

3.2 WassimNet

3.2.1 WassimNet Design

Following the introduction of early exit in networks in [15], we implemented our network with the same design. The network, named WassimNet is composed of two main parts. The *feature extractor* part which is responsible of extracting hidden features in an image is composed of a set of Convolutional, Max pooling, Batch Normalization layers with a ReLU as the activation function.The second part which we call *the classifier*, which takes as input the output of the *feature extractor* and tries to classify these patterns into the appropriate class. *The classifier* is composed of one Linear layer that maps a vector of size 512 which is the output of the *feature extractor* to an n sized vector where n is the number of classes we have. The 'exits' of the model

has the same architecture as the classifier part, this is why in this report we will consider the output of the model as the last exit and we will count it as an exit. We tried different combination of how and where to add exit points on the model. We ended up adding 5 exits on different levels of the model graph. The first exit is on layer 3 and the last one is on layer 25. The model architecture is similar to VGG network[13]. The layer combination (Convolutional,BatchNormalization,Relu) are stacked one after another. We used a stack of 16 of these layers which we decided is a good number to balance model complexity and accuracy. We chose this architecture because we know that the VGG network[13] works well for image classification and is simple enough to play around with, although we did not test it thoroughly on other architectures like ResNet[6], it should work just fine. The main idea is finding the ideal placement of the exit points which, to our understanding, there is no automatic ways to it, you have to manually place them and test.The below is a small sketch of what the final model is.

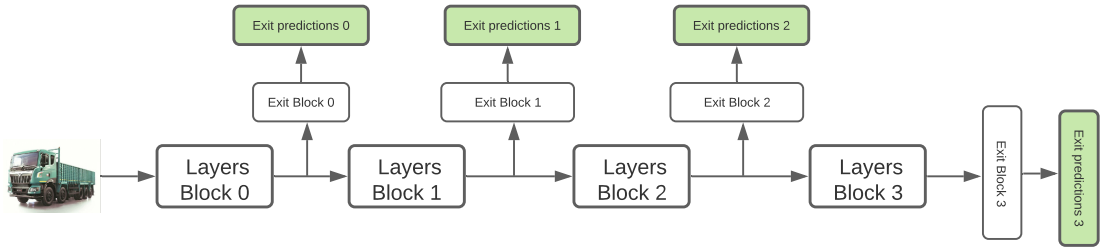


Figure 3.3: Early Exit Network with 4 exits

3.2.2 Model Training

The model, WassimNet was trained and evaluated on the CIFAR10[8] which is made up of 50,000 32x32x3 images for training and 10,000 32x32x3 images for testing. This images are distributed among 10 different classes ranging from cars to trucks and horses.

For the training procedure. We used the recommended training setup for VGG[13] network from PyTorch[12]. The optimizer used is Adam [7] with

a learning rate of 0.0003. The optimizer also had a weight decay factor of 0.001. The training was done on one of INRIA GPU machines which hosts a powerful A100 NVIDIA GPU with around 40GB of VRAMS. This amount of VRAM allowed us to have a relatively big batch size of 4096 images on training and $8,192$ on validation. Due to this bach size the training was fast, a full training process with around 30 epochs took around 20 minutes, which gave us a fast feedback loop that allowed us to iterate and finetune the model faster. The different thing in training early exit networks[15] is that each exit outputs a vector of predictions (or logits) and this vector comes with it's own loss due to the error in predictions. Usually in *normal* deep neural networks, you have one output layer which has these predictions, so you calculate the loss and back propagate the loss back to all of your layers using the chain rule to adjust the weights using a gradient descent technique.However, here you we have multiple exit, hence output layers, we had to calculate the loss across all of these exits, sum these losses and back propagate it.One very important detail to care about is that for each exit, you have to back propagate the loss using the chain rule only to the layers that contributed to the calculation of the output of this exit which might become extremely complex. Fortunately for us, PyTorch[12] has this functionality built in using it's autograd engine which for each layer's output, keeps track of the operations that led to it, this made calculating the losses across multiple exits invisible to us which saved us a lot of time.
For monitoring the training,we utilized a service called Weights and Biases[2] that monitors the whole model including the gradients, activations, weights, and other custom metrics that you want to record. This allowed us to have an easy to access overview of how the model if performing under different hyper-parameter choices.

3.2.3 Results

In this section we will present the results of training WassimNet3.2.1 on CIFAR10 dataset that we described in previous sections.

The results we got were very good, our goal was not to achieve state of the art of the CIFAR10[8] but to train a network with multiple exits that we can use for solving the problem at hand with decent accuracy so that we can validate our approach. We measured the training accuracy, training loss, validation accuracy, validation loss on every exit in the network. We present the results below.

Exit Number	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
0	0.6921	78.664	1.072	65.8
1	0.4519	87.78	0.8281	74.44
2	0.02463	99.972	0.8131	79.31
3	0.0006934	100	0.7755	82.59
4	0.00001995	100	1.081	82.51
5	0.0001207	100	0.8971	82.48

Table 3.1: Results for training

3.3 Optimization Problem

3.3.1 Introduction

After having the network trained as described in the previous section 3.2.2, we need another component for our system to be complete, which is the component that should decide what action to take at every exit. We will explain the purpose of this system in detail in this section and in the next section we will explain the approach we took to get solve it and then we will present our results.

At each exit point, the system has to choose an action from the set {*Stay, Exit, Offload*}. *Stay* meaning that the network should continue to be executed on the same device. *Exit* meaning that the network should exit execution completely and returns the current results if any. *Offload* meaning that the current device should delegate executing the model to a higher device in

the device hierarchy and stop executing it locally. Even though this problem might seem simple, it is much complex because of decision of what to do depends on multiple internal and external factors which we will explain below.

The goal of the system we are building is to help deploy neural network based machine learning models on edge device, which are usually resource constrained in terms of memory and energy, also sometimes there are time constraints which means that they have to return and finish the execution before a certain time deadline,this is why when the system has to take the decision on whether to Stay, Exit or Offload, it has to take into consideration multiple factors such has the energy left on the device, the time deadline,the network bandwidth available now to the device. To explain these conditions more clearly, We will discuss some questions that the system has to answer before taking any of the three decisions.

- *Stay*:
  - Does the device have enough energy to execute to the next exit ?
  - Does the accuracy achieved so far good enough or if we execute until the next exit the accuracy will he higher ?
  - Is the current prediction correct ? or maybe the model is confident but wrong class is chosen
  - If we offload, can we get more accuracy with less energy consumption?
- *Exit*:
  - Should I exit because of enough accuracy or time deadline is approaching ?
  - Should I exit even though I have low accuracy or maybe offloading is better ?
  - How confident am I about the prediction
- *Offload*:

- Does the device have enough energy to offload the data now ?
- Can I offload the current data with the network bandwidth available without depleting my energy?
- If i offloaded, will I still respect the time deadline ?

As we can see, tens or maybe hundreds of deeply nested questions can be asked before a good decision is chosen, solving such a problem with conditions and if statements would not be a good solution as it would get impossibly complex and hard to prove. For this reason we formulate this problem as an Optimization problem.

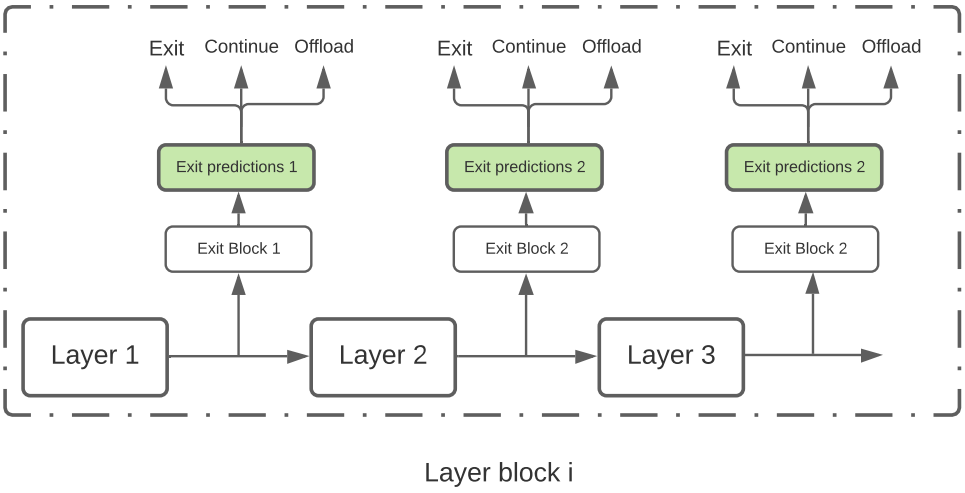


Figure 3.4: Sample Exit point

3.3.2 Formulation

In order to solve an optimization problem, one of first steps and the most important one is to formulate it precisely. In order to do that you have to define what are you trying to optimize, what are your limitations, and what are the degrees of freedom you can use. Another very important thing to decide is whether you optimization problem is single objective or multiple

objective, meaning that are you trying to maximize/minimize multiple variables or a single variable. What we decided to optimize is a function that takes as parameters the accuracy and the cost and is contrained by the time and energy deadline.

$$\begin{aligned} \max \ & objective\ function\ f(cost, accuracy) \\ \text{s.t. } & time\ consumed\ <\ time\ deadline \\ & energy\ consumed\ <\ energy\ deadline \end{aligned} \tag{3.1}$$

Once you have the formulation done, it is time to think about what are the components I am dealing with and define their properties.
In our case our main factor that drove how we approach the problem is the accuracy. In neural network domain, predictions is defined as the softmaxed output of the last layer of the neural, this is then a vector of size n where n is the number of classes we are trying to predict. This means that at every exit e_i, we have the following components of the state.

- Predictions: A vector p of size n where p_i states how confident the model is that this data is of class i at e_i
- Energy consumption until e_i
- Time consumed until exit i
- Network bandwidth available now

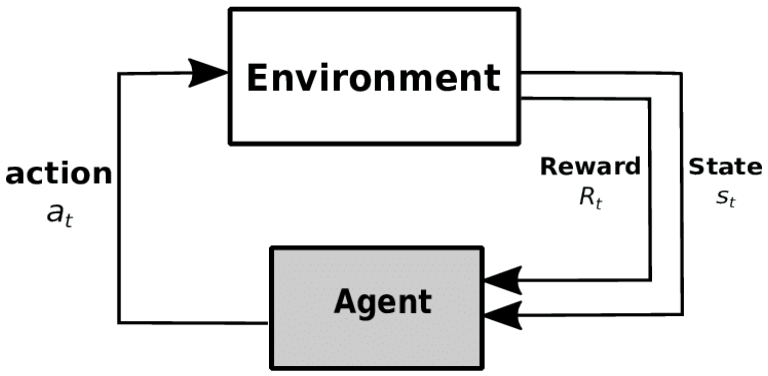
It is also crucial to understand the nature of the states, mainly whether they are continuous or discrete, finite or infinite. In our case the states are continuous infinite mainly due to the fact the the vector of predictions vn can range is bounded in $[0, 1]$ but the possible combination of the elements is infinite. When the number of states is infinite, it narrows down the possible approaches to solve it, for example, in continuous states situation, tabular Q-learning is not feasible as keeping a table of the possible states with their Q-values becomes computationally inaccessible. An approach that is robust

enough the proven to solve continuous state problems is deep Q-learning or in short DQN[11] which we will describe in the next section.

3.4 Reinforcement Learning

3.4.1 Introduction

Reinforcement Learning or RL for short is a general framework that deals with the interaction between an environment and an agent. In other words, it trains the agent to learn how to act in an environment through the feedback it gets from its actions. It can be used for example to train a robotic arm how to through a basketball, or teach an agent how to play atari games[11]. The interesting idea behind reinforcement learning is that how don't have to tell the agent how a certain task is achieved, you allow the agent to act freely in the environment and through positive reward (reinforcements) and negative rewards (punishments) the agent should learn a policy on how the behave so that it achieves the highest possible rewards. Therefore, the tricky part in solving a reinforcement learning problem is how to formulate your reward function such that it can be beneficial for the agent to learn the optimal behavior.



In a more technical sense, Reinforcement learnig can be formulated as follows

At each timestep t the agent observes a state s_t from the environment, according to this observation, the agent takes action a_t on the environment.

After this action a_t the environment reacts with a state s_{t+1} and reward r_{t+1}. In usual RL situations, the agent goals is to maximize the rewards he is getting from the environment by choosing actions that he thinks will lead to the maximum reward.
There is a plethora of algorithms that are designed to solve such problems, some are limited to finite states scenarios and others can worked under infinite states. The algorithm we decided to use is Deep Q Networks[11] or DQN in short.

3.4.2 Deep Q-Learning

Deep Q-Learning[11] or DQN for short is an general approach for solving Reinforcement learning problems. It is first published by Google's DeepMind AI research lab and applied on learning to play atari games. The breakthrough that his approach had was that it was trained on just the image of the atari game as input. No hardcoded rules on how the game works. Since DQN is a core component of our contribution we will discuss it in details in this section

DQN comes from the model-free family of Reinforcement learning algorithms, this means that for learning how to act in the environment, the agent does not need to learn how the environment works. Before DQN, most RL applications relied on engineers carefully designing the features representation that the agent will operate on, this created a huge bottleneck in the performance on such systems as it is directly proportionate to the quality of the handcrafted representations. In DQN, you do not have to do any feature engineering, you can just feed in the raw input to the agent and he should learn the optimal policy.

The main part of the algorithm is the *Experience Replay* which stores a 5-tuple "experiences" that the agent encounters, these experiences are then sampled and trained on every n episodes. This component can be described as our training dataset that is collected while the agent experience with the environment.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent
  end for
end for

One of the main and most important part of any reinforcement learning problem is the optimization objective which is more commonly known as the *reward function*. This function is what your agent will be using to learn how to adjust it's policy. It should be encapsulate all the goals that you are trying to optimize. For us, we are trying to optimize the function that is composite of the cost of running the model and the accuracy of the model. After some experiments, we use the below reward function.

$$Reward = -(\alpha * energy\ cost) + (\beta * prediction\ is\ correct) \tag{3.2}$$

Where α is set to 1 and β is the trade-off you want between accuracy and energy efficiency. The β parameter is application related and is an input for solving the optimization problem.

3.5 Solving the Optimization Problem

After having the optimization problem formulated and choosing the method to solve it. It is now the time to actually solve it to get the optimal policy

function

3.5.1 Training Setup

As mentioned in the section 3.4.2 our DQN is made up of neural network to be trained thus this means that we need training data.
The training data for this DQN is collected during the training on WassimNet discussed in 3.2.2. The algorithm for collecting the training data is as follow:

Algorithm 2 Training data collection for DQN training

```
Only applied on the last training epoch
Input: Number of images N to collect data for
for image = 1, N do
    Pass image to WassimNet
    Get predictions, losses, across all exits in WassimNet
    Save them into disk
end for
```

The DQN is then trained for 10,000 episodes across all the training data. after the end of every episode, which means that either the agent decided to exit or offload of he reached the end of the network, the reward is calculated and sent back to the agent as a signal for the agent know how to update it's policy. We also used Weights and Biases[2] to monitor the training the DQN agent and we noticed that the training loss is decreasing which is one of the indicators that the network is training correctly.

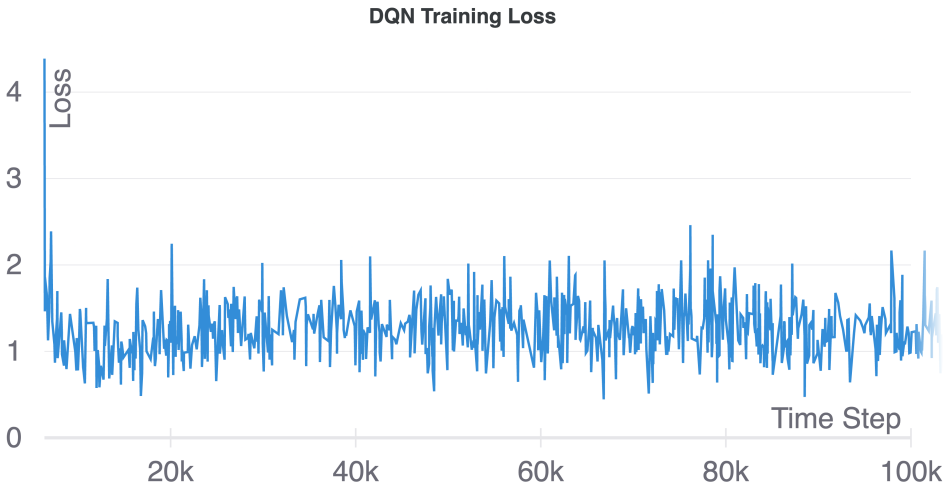


Figure 3.5: DQN training loss monitored by Weights and Biases[2]

4

Results

After training WassimNet 3.2.2 and solving the optimization problem using a DQN[11] it is time to check our results and see if our approach works. In order to do that we had to calculate the parameter β in our reward function so that we expect the model to exit at a certain point, if it does then we can say that our approach works.In other words, if we calculate β between exits 2 and 3 and use this β to train our DQN, then we expect the DQN we prefer to exit between these two exits. To calculate β and since beta is related to whether the model is correct or not. you can just take the summation of the losses between any two exits and set β has the reciprocal of this value.

$$\beta_{i,i+1} = \frac{1}{(loss_i - loss_{i+1})} \tag{4.1}$$

From our dataset we found that β between exit one and exit two is 0.1785. So we trained the DQN and we noticed that the DQN prefers to exit between these two exits. This is shown by the below figure where the mean episode length is 1.5 which means that every image on average is passing by one and two exits.

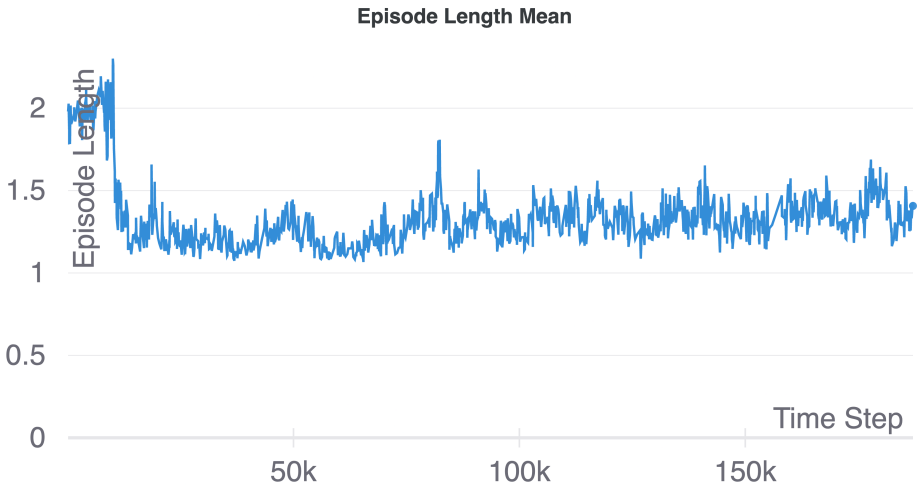


Figure 4.1: $\beta = 0.1785$

In order to validate our results more and not to be deceived by chance, we conducted multiple experiments which multiple β's between multiple exits to make sure that our approach stays valid.Below is another experiment with $\beta = -0.0021$ which refers to the β between exit three and four

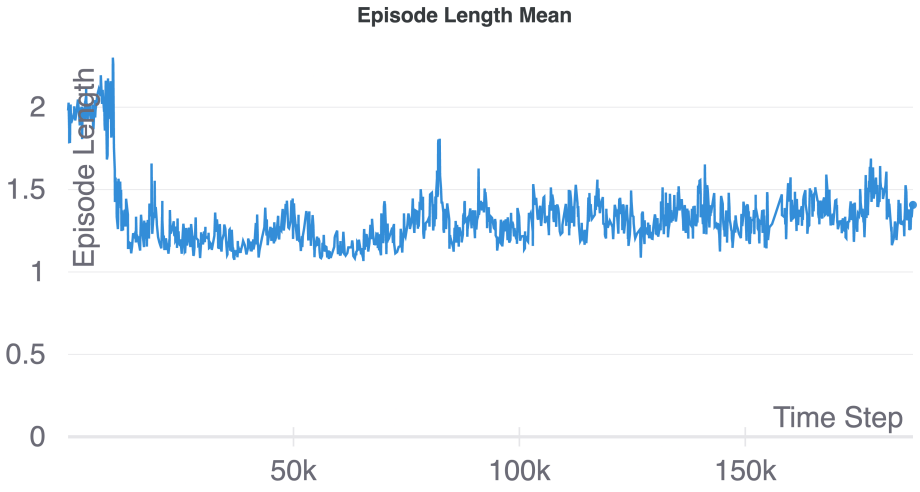


Figure 4.2: $\beta = -0.0021$

5

Conclusion and Future Work

In this report we presented a novel approach for model splitting across multiple devices, our contribution lies in a new framework that would allow users to use a single model split into multiple devices having multiple computational capabilities instead of using multiple models based on the capabilities of the hosting device. We also introduced an optimization problem that should be solved in order to get the optimal policy in order to decide when to *Stay*, *Exit*, or *Offload*. We solved this optimization problem using a DQN[11] and got results that validated our approach.

A lot of other work can be done with this approach. Unfortunately, due to time constraints, we could not do every. We were interested in testing this approach on real-world data and checking its efficacy. Another addon that we wanted to try is to weight the classes of the neural network thus making this optimization problem harder. In other words the user can select which classes he want more accuracy for and which classes he can tolerate low accuracy.This addon might be helpful in intrusion detection systems where you do not care equally if your camera detected a cat or if it detected a human being. Another thing we wanted to try is inter-device routing similar to Skipnets[17] as discussed in section2.2.2. This would make the optimization problem harder and more interesting where instead of three actions {*Stay, Exit, Offload*} and agent will have another action *Skip* where he has to skip

to a convolutional layer that he chooses. This would make the approach more dynamic and powerful.

I am really grateful I got the chance to work on such a project with two amazing supervisors. This was my first experience with doing machine learning research in a lab and because of how much I enjoyed it I decided to pursure a PhD in a similar field.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[3] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.

[4] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021.

[5] Andrew Hard, Kanishka Rao, Rajiv Mathews, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *CoRR*, abs/1811.03604, 2018.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[8] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[9] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[10] Hengyue Liu, Samyak Parajuli, Jesse Hostetler, Sek M. Chai, and Bir Bhanu. Dynamically throttleable neural networks (TNN). *CoRR*, abs/2011.02836, 2020.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[13] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[14] Tianxiang Tan and Guohong Cao. Fastva: Deep learning video analytics through edge processing and npu in mobile. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 1947–1956, 2020.

[15] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks, 2017.

[16] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[17] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks, 2018.

[18] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks, 2018.