ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Technical University of Crete
School of Electrical and Computer Engineering

Diploma Thesis

Distributed Deep Neural Network (DDNN) Analysis for Classification Tasks

Author: Konstantinos Nikolos

Thesis Committee
Thrasivoulos Spyropoulos (Supervisor)
Vasileios Digalakis
Evangelos Kalogerakis

Department of Electrical and Computer Engineering
Chania, Greece

December 2025

Abstract

The Internet of Things (IoT) already consists of a fast-growing fleet of devices scattered worldwide. A large part of this ecosystem is made up of CCTV cameras and traffic-control units producing an ever increasing flow of data. A single 4K camera produces a huge amount of data daily that needs to be processed and classified locally or remotely both of which put heavy pressure on bandwidth and computing resources. Similar challenges appear in areas like cloud gaming or cashier-less retail systems, where the demands are even tougher due to low end-to-end latency restrictions. In practice, all of these cloud–edge applications face the same dilemma: process each frame locally to save bandwidth cost, or off-load it to the cloud and gain higher accuracy in return.

One solution already in research by the community is the use of Distributed Deep Neural Networks (DDNNs). In these systems, the shallow part of the model runs on the device objecting fast low-latency responses, while the deeper part executes in the cloud producing more detailed and complex answers. Earlier work often made the off-loading decision by looking at simple confidence measures, such as calculating the entropy of the local exit or Monte-Carlo Dropout Uncertainty. However, these heuristic techniques can be misleading, especially in cases where the edge model is highly confident about its own wrong prediction. Our approach keeps the distributed-network architecture but changes the decision rule: instead of fixed heuristics, a small shallow Neural Network located on the device is trained to predict whether a sample can be safely classified locally or should be sent to the cloud for further processing. Testing on CIFAR-10 and other datasets show that this learned strategy outperforms entropy-based baselines and gets closer to the performance of an oracle router, while allowing more samples to be processed directly on the edge.

Περίληψη

Το Διαδίκτυο των Πραγμάτων (IoT) ήδη αποτελείται από έναν ταχέως αναπτυσσόμενο στόλο συσκευών διεσπαρμένων παγκοσμίως. Ένα μεγάλο μέρος αυτού του οικοσυστήματος αποτελείται από κάμερες CCTV και μονάδες ελέγχου κυκλοφορίας, παράγοντας μια ολοένα αυξανόμενη ροή δεδομένων. Μια μεμονωμένη κάμερα 4K παράγει έναν τεράστιο όγκο δεδομένων καθημερινά που χρειάζεται να επεξεργαστεί και να ταξινομηθεί τοπικά ή απομακρυσμένα, με τις δύο περιπτώσεις να ασκούν βαριά πίεση στην κατανομή εύρους-ζώνης και στους υπολογιστικούς πόρους. Παρόμοιες προκλήσεις εμφανίζονται σε τομείς όπως το cloud gaming ή τα συστήματα λιανικής χωρίς ταμία, όπου οι απαιτήσεις είναι ακόμη πιο σκληρές λόγω των περιορισμών χαμηλής καθυστέρησης από άκρο σε άκρο. Στην πράξη, όλες αυτές οι cloud-edge εφαρμογές αντιμετωπίζουν το ίδιο δίλημμα: να επεξεργαστούν κάθε καρέ τοπικά για να εξοικονομήσουν κόστος εύρους ζώνης, ή να το μεταφορτώσουν στο σλουδ και να κερδίσουν υψηλότερη ακρίβεια ως αντάλλαγμα.

Μια λύση που ερευνάται ήδη από την κοινότητα είναι η χρήση των Κατανεμημένων Βαθιών Νευρωνικών Δικτύων (DDNNs). Σε αυτά τα συστήματα, το ρηχό μέρος του μοντέλου εκτελείται στη συσκευή, στοχεύοντας σε γρήγορες αποκρίσεις χαμηλής καθυστέρησης, ενώ το βαθύτερο μέρος εκτελείται στο σλουδ, παράγοντας πιο λεπτομερείς και σύνθετες απαντήσεις. Παλαιότερες εργασίες συχνά έκαναν την απόφαση μεταφόρτωσης εξετάζοντας απλές μετρήσεις εμπιστοσύνης, όπως ο υπολογισμός της εντροπίας της τοπικής εξόδου ή το Monte-Carlo Dropout Uncertainty. Ωστόσο, αυτές οι ευριστικές τεχνικές μπορεί να είναι παραπλανητικές, ειδικά σε περιπτώσεις όπου το edge μοντέλο είναι ιδιαίτερα σίγουρο για τη δική του λανθασμένη πρόβλεψη. Η προσέγγισή μας διατηρεί την αρχιτεκτονική του κατανεμημένου δικτύου αλλά αλλάζει τον κανόνα απόφασης: αντί για σταθερές ευριστικές μεθόδους, ένα μικρό ρηχό Νευρωνικό Δίκτυο που βρίσκεται στη συσκευή εκπαιδεύεται να προβλέπει εάν ένα δείγμα μπορεί να ταξινομηθεί με ασφάλεια τοπικά ή θα πρέπει να σταλεί στο υπολογιστικό νέφος για περαιτέρω επεξεργασία. Δοκιμές στο CIFAR-10 και άλλα σύνολα δεδομένων δείχνουν ότι αυτή η εκμαθημένη στρατηγική υπεραποδίδει έναντι των βασικών γραμμών baselines που βασίζονται στην εντροπία και πλησιάζει περισσότερο στην απόδοση ενός ιδανικού δρομολογητή (oracle), επιτρέποντας ταυτόχρονα περισσότερα δείγματα να επεξεργάζονται απευθείας στο edge.

Acknowledgements

I would like to thank...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Figure 1.1: IoT Devices Network.

1.1 Motivation

The proliferation of the *Internet of Things* (IoT) has resulted in a rapidly expanding fleet of devices deployed at the network edge . This diverse ecosystem includes not only smartphones but also specialized hardware such as CCTV systems and traffic cameras, each serving distinct use cases . Edge-cloud perception, however, extends beyond a daily consumer's needs or public safety applications; IoT devices enhanced with embedded AI software are becoming significant for modern warfare [3] pushing demands and their adeptness to these scenarios. For instance, UAVs on (ISR) missions must perform *on-board visual recognition* to detect and map enemy

threats . In contested environments where communicating systems fail due to signal jamming the ability to classify locally becomes life-saving. Sending an image remotely and waiting for a response might be incredibly costly while also risks leaking information to the enemy.

A typical centralized approach of offloading all data to the cloud guarantees maximum accuracy but also activates significant delays, consumes substantial amounts of bandwidth, and increases the system's energy consumption. Conversely, relying exclusively on local processing denies the benefits of deeper, more powerful cloud models such as a detailed more accurate analysis. Both extremes present serious operational downsides, motivating the development of **Distributed Deep Neural Networks (DDNNs)** .

In a DDNN architecture, the network is distributed into a smaller, lightweight classifier that runs on the device, while a deeper more powerful one executes in the cloud . The operational rule is simple: inputs labeled as "easy" are classified locally, while "hard" ones are offloaded to the cloud. This allows IoT pipelines to conserve computational resources, bandwidth, and time.

1.2 Problem Definition

1.2.1 Deep Neural Networks

Neural and Deep Neural Networks. Artificial neural networks (ANNs), or just neural networks (NNs), are models that can be tuned with parameters to learn useful patterns from input data. In general an input is pushed through layers, performing several computations and produces an output. The core building blocks are the so-called neurons. Each neuron is simple: it takes inputs, multiplies them by weights, adds a bias, and then applies some non-linear function. This process can be written as

$$f(x) = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{1.1}$$

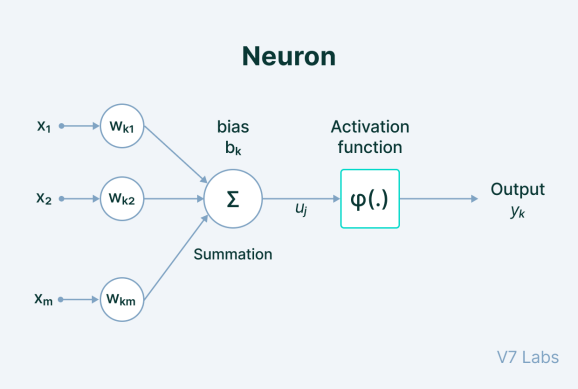
where x_i are the inputs, w_i are trainable weights, b is the bias, and $\sigma(\cdot)$ is a non-linear activation (for example, sigmoid or ReLU). Figure 1.2a shows this in a compact way. In practice, this operation is repeated many times across many neurons.

We combine neurons into layers, stacking more layers together forms a network capable of mapping complex inputs. A common neural network structure consists of an input layer, followed by more hidden layers, and finally an output layer where the prediction is produced. Neural Networks are capable of learning non-linear functions that classic machine learning models cannot easily capture.Furthermore networks with more hidden layers and neurons, consequently having more parameters called Deep Neural Networks (DNNs), can discover even very complex mappings and decision boundaries fig. 1.2b. DNNs have become extremely successfu in tasks such as computer vision, speech recognition, or natural language processing, where non linear problems need to be learned from input data [4].

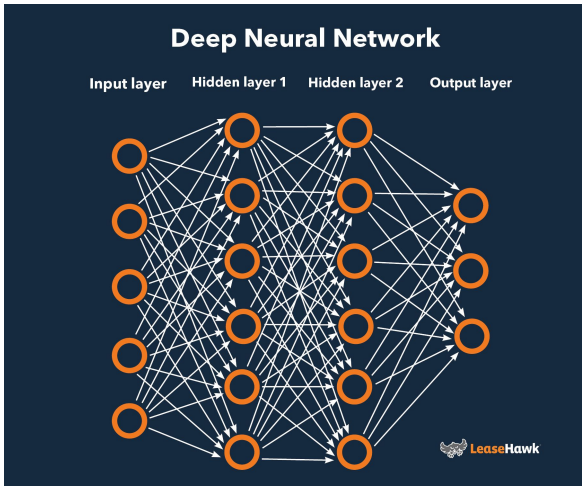
We our specifically interested in supervised learning a popular branch of machine learning where each input comes with a ground-truth label. The network tries to learn by minimizing a loss function, for example mean-squared error in regression tasks or cross-entropy in classification tasks. The training is done with gradient-based methods, the gradients are calculated efficiently using an important technique, back-propagation [5]. During this process, the weights and biases are updated constantly trying to minimize the loss function. Over many iterations, and after repeating the cycle countless times, the network gradually becomes better at making predictions.

Scope of this discussion. In the remainder of this thesis, references to neural networks and deep neural networks will, unless otherwise noted, be considered in the context of *classification*

tasks. This choice is made for simplicity since our objective is to classify images efficiently in a DDNN setup.



(a) Single artificial neuron: weighted sum plus non-linear activation.



(b) Deep neural network (MLP) with input, hidden, and output layers.

Figure 1.2: (a) Single neuron schematic. (b) Multi-layer perceptron (DNN) architecture.

1.2.2 Challenges in Deep Neural Networks

Deep neural networks (DNNs) have reached state-of-the-art performance in many areas such as computer vision, natural language processing, and speech recognition. A significant breakthrough came when a massive labeled dataset *ImageNet* was released enabling researchers to effectively train deeper and most complex networks which required huge amounts of data in order to generalize and achieve greater results. Soon enough *AlexNet* marked a huge jump in classifying accuracy using GPUs and techniques like ReLU activations and dropout [6, 7]. More deeper designs followed, such as VGG (with 16 or 19 layers) and later residual networks (ResNet) pushing performance even further, this time by exploiting depth and skip connections [8, 9, 4].

Even though these models have been very successful, several important challenges have occurred when dealing with Deep Neural Networks:

- **Inference latency and energy cost:** Deep Neural Networks achieve high performance metrics producing high quality accurate predictions but with what cost? Inference on a DNN requires executing every single layer, regardless of how simple or difficult the input is without an early exit choice. This extra computation time creates extra latency and consumes a large amount of energy. This issue is even more crucial in mobile edge devices, where power budgets are tight and resources are limited not allowing huge models. [10].

- **Overfitting and generalization:** Another disadvantage of deeper network architectures is overfitting. Having too many parameters can reduce the ability to generalize and in most cases parameterize the network specifically for training dataset.[11] et al showed that ordinary CNNs can even fit random labels or pure noise underlining the need of simpler approaches.

- **Inefficient use of depth:** Not all computational power of a Neural Network is always needed. In many cases, a clean and typical image (a cat for example) could be classified correctly after using only a few layers. On the other hand, low-quality or more complex

inputs may require deeper processing. Running the entire network on every input given wastes computation and leads to unnecessary cost.

Overthinking in Deep Networks. Another related issue is what has been called *overthinking*. In this case, the network makes a correct prediction early on, but continues processing and may waste time and energy, or (ii) even change the correct prediction into a wrong one at the final layer. Kaya *et al.* used Shallow-Deep Networks with internal classifiers and showed that CNNs overthink in the majority of inputs. In fact, they found that up to 95% of these cases end up with wasted computation and slower inference. Moreover, around 50% of the errors happen when an initially correct early prediction is flipped into a misclassification in the final layer [12].

These limitations make clear the need for methods that adapt the amount of computation to the difficulty of each input. By reducing latency and energy use without hurting accuracy, such methods can improve efficiency. This is the motivation behind *early-exit mechanisms*, which allow a network to stop inference earlier when the input is considered "easy".

1.2.3 Early-Exit Strategies

Early-exit neural networks consist of multiple layers followed by intermediate classifiers placed at various depths within the architecture. The Job of each classifier (early exit branch) is to decide if a prediction (classification) made at this point for a certain sample that comes through is "correct enough" to be the output of the neural network. Otherwise, the input is propagated to deeper layers for further computation to achieve better results.

The architecture is simple having multiple early exits shown in Figure 1.3. to a standard deep neural network and serve as intermediate checkpoints for classification. At each checkpoint the network must decide that either the current prediction is enough to classify the sample or it should be sent to the latter stages for further processing. In order to evaluate a given prediction probabilistic confidence measures are employed such as the entropy of the softmax output distribution:

Definition — *Entropy-based confidence (early exit)*

Definition. For a classifier over $|C|$ classes with predicted probabilities $p_i(x)$, the normalized entropy of x is

$$H(x) \;=\; -\frac{1}{\log |C|} \sum_{i=1}^{|C|} p_i(x)\, \log p_i(x).$$

Notation. $p_i(x)$: predicted probability for class i; $|C|$: number of classes; (normalization by $\log |C|$ bounds $H(x) \in [0, 1]$). Lower entropy implies higher confidence in the predicted class.

Formally low entropy values indicate confident predictions, whereas high values tend to uncertainty. In order to decide whether to exit early, a comparison between the entropy score and a predefined threshold that takes into account the possible accuracy loss is made . This way, early-exit mechanisms manage to resolve "easier" tasks with high confidence faster, saving computation and latency, while more complex samples are processed deeper to preserve accuracy.

Overall, early-exit strategies introduced cost reduction, cutting down wasted computation and latency. They also set the stage for more advanced ideas, such as *Distributed Deep Neural*

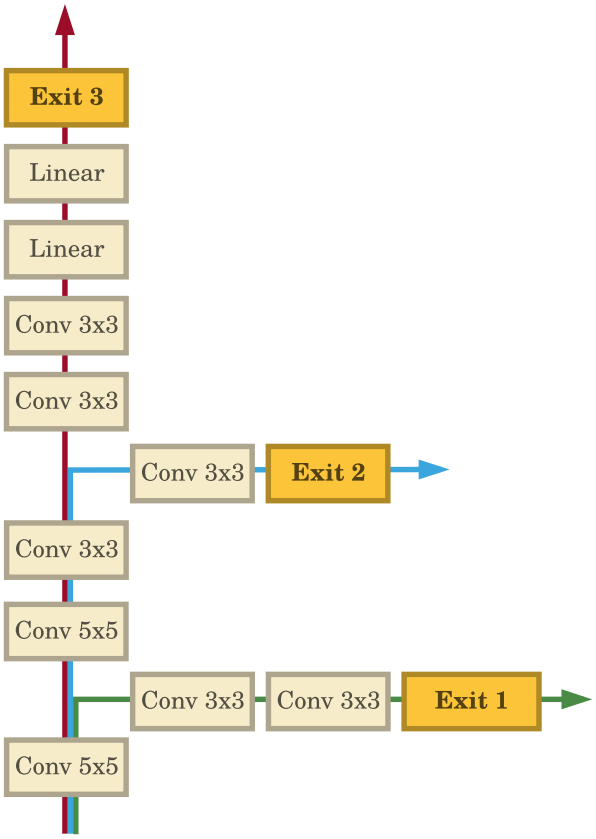


Figure 1.3: Illustration of an early-exit network with multiple side branches. Adapted from Teerapittayanon et al. [1].

Networks (DDNNs), which move beyond early classification and proceed to a more efficient edge-cloud environment.

1.2.4 Distributed Deep Neural Networks (DDNNs)

Distributed Deep Neural Networks (DDNNs) were introduced by Teerapittayanon et al. [2] as a natural extension of early-exit architectures into the device-edge-cloud hierarchy. Instead of hosting the entire model in the cloud, the network is distributed into computing hierarchies: a shallow network runs locally on the device, an intermediate edge server hosting a deeper network with more layers and of course the cloud where the more powerful computation takes place. At each stage (device, edge, cloud), an early exit branch is attached in order to perform classification on the inputs that come through while in the later stages (edge-device) the output of earlier stages serves as the input for later ones. Following the early-exit approach "easy" samples are classified early, exiting the network while more complex ones are forward to later stages for further proccessing.

This distributed architecture, illustrated in Figure 1.4, enables more complex system designs such as multiple local devices forwarding all the images into the edge, constructing a better multi-view image suitable for capturing complex patterns. This multi-input system can be further applied into having more edge devices leading to a central cloud network capable of capturing the decision.

The exit decision in DDNNs follows the same principle as the early exit networks approach: a probabilistic method such as the entropy checks the reliability of the current prediction by measuring its confidence (see for example section 1.2.3) compared to a predefined threshold.

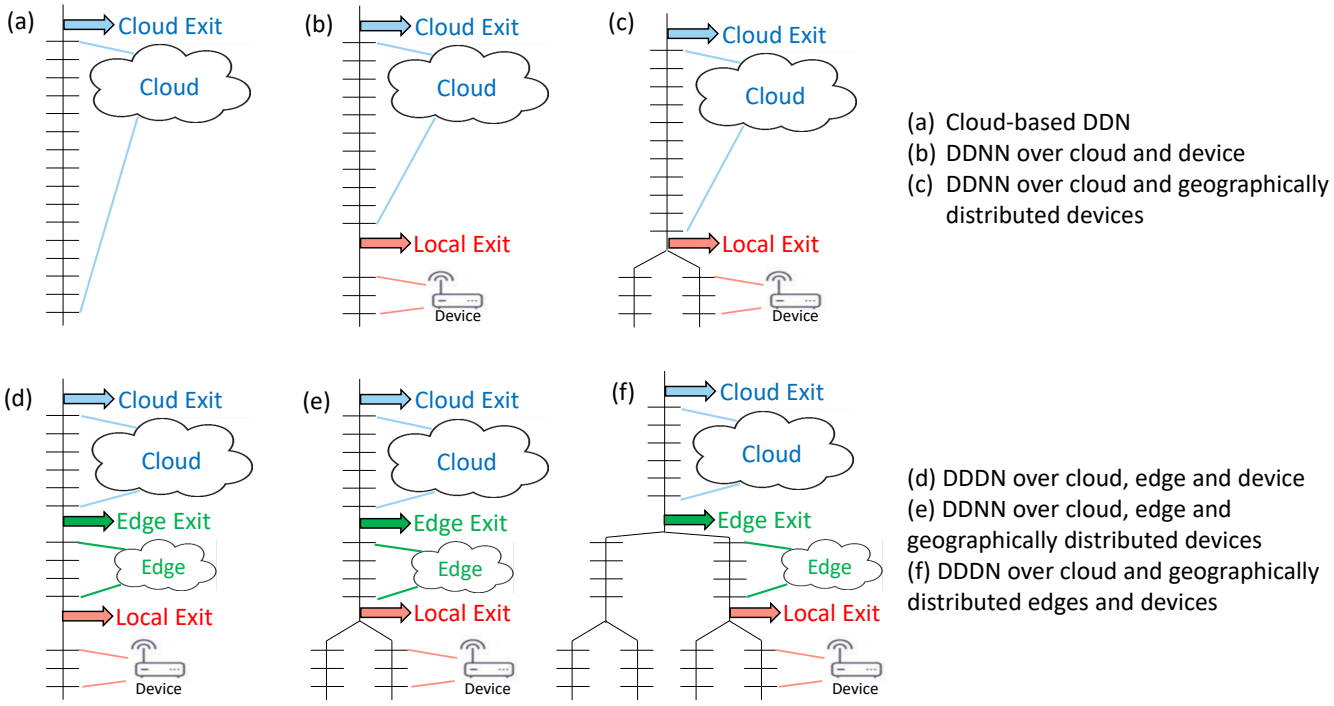


Figure 1.4: Illustration of a Distributed Deep Neural Network (DDNN) across device, edge, and cloud with early exits available at intermediate stages . Adapted from Teerapittayanon et al. [2].

Joint training and efficiency. A key contribution of the original DDNN framework is that both the local and cloud networks are trained jointly under a single loss function. This strategy is crucial for the DDNN setup taking into account both cloud-edge losses ensuring balance between networks without overshadowing the less powerful local exit. Joint Training also acts as a form of regularization for the cloud exit improving its accuracy. Finally it introduces improved efficiency for both exits training them to deal with different "difficulty" tasks while using cloud gradients in the edge loss function ensures that early processing benefits also the cloud exit and lowers the needed computation.

Limitations of probabilistic exiting decisions. Although calculating the entropy of a prediction to decide whether its accurate enough is pretty straightforward and very simple to use, on a real edge-cloud system it may lack some key insight leading to unnecessary computation and false predictions costing even more. The decision is based solely on the *edge* model's self-assessed uncertainty, making it vulnerable to cases where the local network is "confidently wrong" about its own prediction. Moreover the entropy rule as offloading decision mechanism takes into consideration only the data selected from the online present prediction of the local network without taking advantage of the predictions made by the cloud network and the accuracy of both networks which are crucial insights. We also have to consider the communication cost incurred by offloading which in cases might be more than the accuracy cost of sticking to the local approach.

Recent studies backup this argument that simple probabilistic confidence like entropy-thresholding alone may be insufficient for reliable decision making. Pomponi et al. [13] argue that simple entropy-based exits are not precise, motivating to more complex probabilistic weighting schemes, while Meronen et al. [14] report overconfidence issues in dynamic or early-exit networks that can mislead routing policies.

1.3 Illustrative UAV Offloading Scenario

To ground the discussion, we will consider the above scenario of a small UAV flying toward a contested battle zone on the Russia–Ukraine war territory. Its mission is straightforward: scan the battlefield , detect enemy assets (e.g., tanks, anti-drone stations) and define if its safe fast enough to act. In these time crucial missions, even a few milliseconds can make the difference.

Toy Setup fig. 1.5. The drone runs a lightweight CNN locally able to classify objects and can forward features to a cloud back-end more powerful CNN when further processing needed. Its offloading mechanism needs to decide where the classification output should come from. We use indicative numbers to keep the example realistic:

- Local inference: $\approx 10\,\text{ms}$ per frame, $\approx 85\%$ on the mission classes.
- Cloud inference (with offloading): extra $\approx 80\,\text{ms}$ end-to-end latency; accuracy up to $\approx 95\%$ on the same classes.
- Uplink cost: (8x); In our prototype we assume an $8\times$ cost penalty to ship figures to cloud.

Intuitively, when the scene is "easy" (clear target, good SNR), sticking to the local classifier keeps latency tiny. When it is "borderline", asking the cloud for help often adds $\sim 80\,\text{ms}$ but can prevent costly mistakes. The router's job is to tell these two cases apart, and to do it right away.

Operational challenges. Analyses of UAV deployments in battle environments underline the importance of systems that can operate autonomously, with little or no human intervention, even under electronic jamming or degraded communications [3]. While accuracy is valuable, the ability to respond independently within milliseconds is often more important. On–board inference enables such rapid reactions, whereas reliance on remote servers and human decision adds latency and exposes the system to link vulnerabilities. Reports on the Ukraine war note that drones process live video streams through local CNNs on embedded systems (e.g., RK3588) to automate decision, and at the same time use long–range, low–latency links such as ExpressLRS to transmit compact features to remote servers, supporting collaborative processing even under electronic warfare pressure [15].

A concrete walk-through available at fig. 1.6. The UAV sees a fast-moving object in its FoV:

1. Local CNN produces logits, a softmax prediction, and confidence in $\sim 10\,\text{ms}$.
2. If confidence $< \theta$, we accept the local class (e.g., "tank" – "danger") and act immediately (evade, attack, report).
3. Else, we ship the *feature tensor* upstream; the cloud refines the prediction (potentially from 0.85 to 0.95 accuracy) with an added $\sim 80\,\text{ms}$.

Table 1.1: Indicative per-frame figures for the UAV scenario.

Quantity	Local (edge)	Cloud (after offload)
Latency (ms)	~ 10	~ 80 extra
Accuracy (top-1)	$\sim 85\%$	up to $\sim 95\%$
Uplink cost	1	

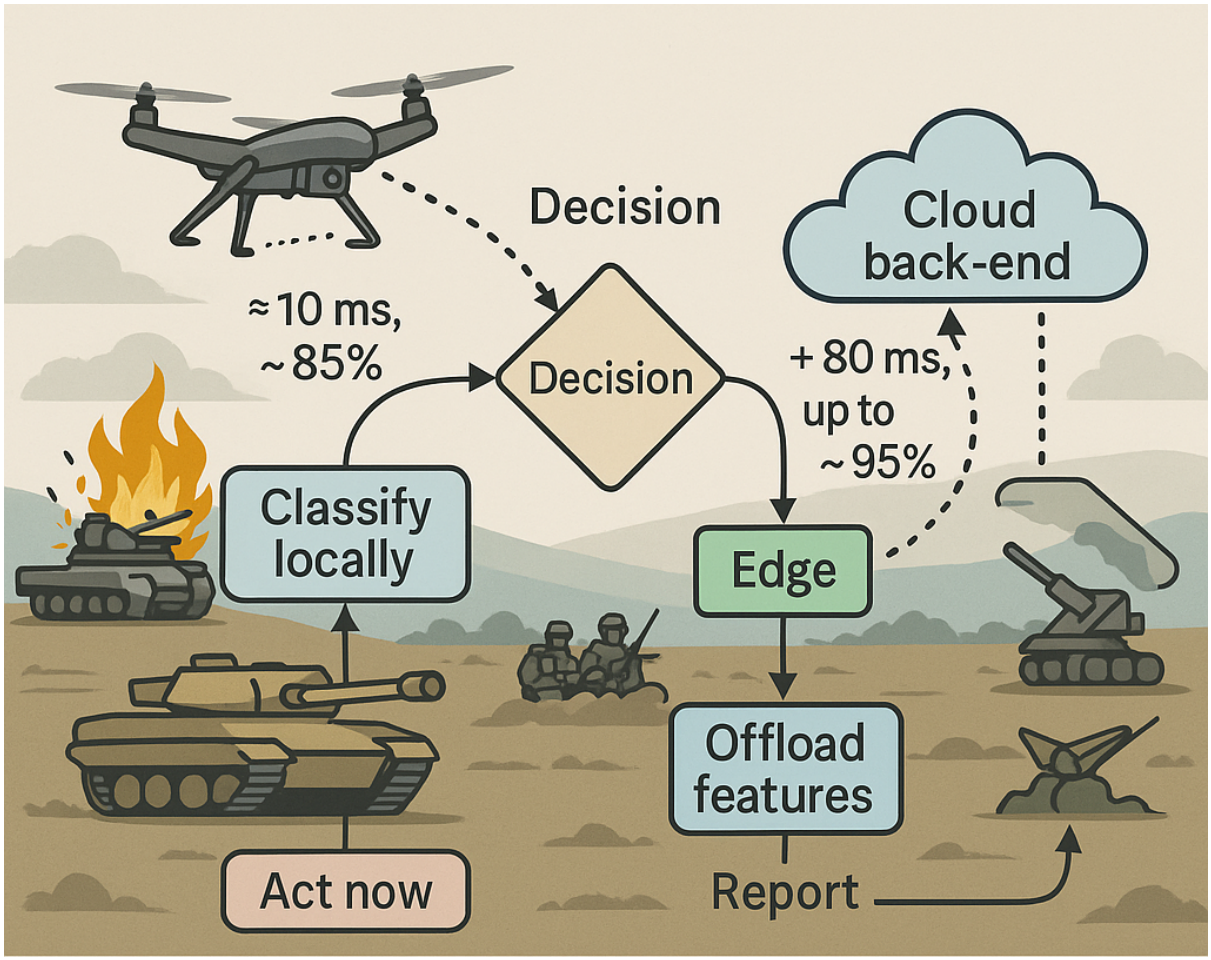


Figure 1.5: UAV offloading scenario: classify locally (∼10 ms, 85% acc.) or offload to the cloud (+ ∼80 ms, 95% acc.) depending on confidence and cost.

1.4 Proposed Solution

Objective. The objective of this thesis is to design an *offload decision mechanism* that is both effective and compact enough to run on small devices (embedded systems, smartphones, etc.) with limited memory and compute budgets. In short, we want a decision module that fits on every possible device capable of taking efficient decisions quickly in order to eliminate extra cost.

Main perspective. Our main perspective is to treat the offloading decision as an optimization problem.

Before formulating this task, we briefly recall the general definition of an optimization problem,

Definition — *Optimization Problem*

An **optimization problem** is the process of finding the values of a set of variables that minimize (or maximize) an *objective function* subject to certain constraints. In other words, given a function $f(x)$ and constraints $g_i(x) \leq 0$ or $h_j(x) = 0$, the goal is to find x^* such that

$$x^* = \arg\min_x f(x) \quad \text{subject to } g_i(x) \leq 0,\ h_j(x) = 0.$$

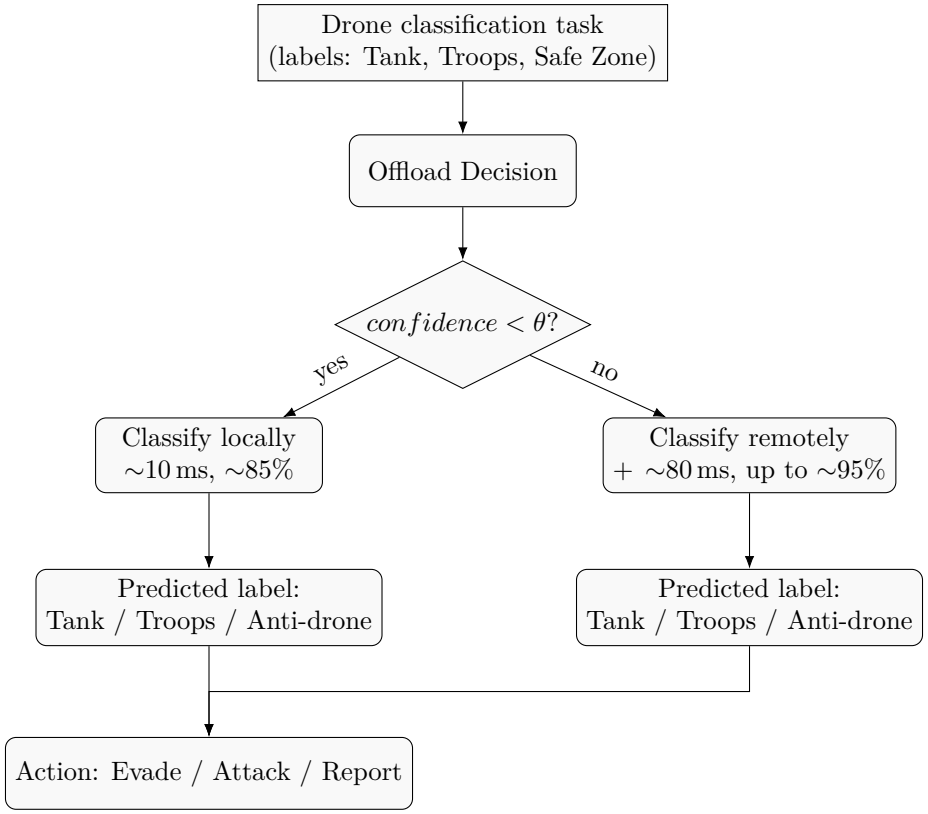


Figure 1.6: Flow of a UAV decision: entropy thresholding decides between local (fast) vs. remote (accurate) classification, followed by an action.

In our case, the optimization function aims to *minimize the total cost* of the DDNN while *maximizing the number of samples correctly classified on the edge* without requiring additional transmission or computation. Following the formula presented by Giannakas in [16] which is the paper that inspired this work, we define the joint cost as:

$$\min_{C_J} \; C_J = C_{L,x} \cdot L_0 + C_{R,x} \cdot (1 - L_0), \qquad \text{with} \quad \max L_0. \tag{1.2}$$

where $C_{L,x}$ and $C_{R,x}$ denote the local and remote inference costs for sample x, and L_0 represents the fraction of inputs processed locally. The objective is to minimize C_J by maximizing L_0, keeping as many samples as possible on the edge without degrading accuracy.

Our goal is to use the limited evidence given by the local (edge) prediction outcome to pick the best action (keep prediction vs. offload). In doing so, we explicitly aim to outperform the simple entropy rule commonly used in early Distributed Deep Neural Network systems in terms of classification accuracy—(the normalized-entropy threshold in the original DDNN paper for classification [2])—and to close part of the performance gap to an oracle router that always chooses the correct exit knowing every sample Remote and Local cost.

Proposal. We propose a lightweight "small" neural network (a learned router) that predicts whether forwarding a sample to the cloud will improve the outcome compared to accepting the local (edge) prediction. Rather than relying solely on a fixed confidence heuristic, this offloading mechanism is trained using a labeled meta-dataset derived from a simple *cost–benefit* signal (cloud-vs-local cost). The model will get the early edge prediction as an input and will produce a binary (keep-offload) decision as an output.

Contributions of this work.

- Built a DDNN image-classification network edge (local)–cloud (remote) with a clear accuracy gap between the two parts (edge smaller/weaker, cloud deeper/stronger) and a common loss function.
- Create a training dataset for the offloading mechanism router based on the DDNN performance.
- Trained a compact offload router to output a binary decision.
- Simulate extensively with multiple known 2D Dataset and benchmark against a **normalized-entropy** baseline (tuned threshold) and an **oracle** router (upper bound).

Chapter 2

Related Work

2.1 Early-Exit Architectures as a Foundation for DDNNs

As introduced in Section 1.2.3, early-exit mechanisms represent one of the first practical approaches to dynamically use computation within deep networks. They supply a standard DNN with intermediate classifiers that can produce predictions before the final layer, allowing "easy" samples to exit early while harder ones continue deeper. This early-exit strategy initially proposed in **BranchyNet** [17], demonstrated that passing through the entire network isn't mandatory for a sufficient classification. A significant number of samples can be classified at shallow layers effectively reducing inference latency and computational cost.

Following BranchyNet, **Shallow-Deep Networks** [12] spoke about the phenomenon of network "overthinking", where a correct early prediction may turn into a wrong decision from further processing as it passes through more layers . This observation motivated Kaya et all to use similar to BranchyNet confidence-based early exits that would lower the percentage of misclassification significantly.

More recently, unsupervised and probabilistic early-exit strategies have been explored. **Self-Xit** [18], **Unsupervised Early Exit in DNNs with Multiple Exits** framework [19] work on self-supervised approaches that propose self exits positions on different layers of the DNN , relying on algorithms such as NAS (Neural Architecture Search) or Multi-Armed-Bandits without labelled supervision and also set confidence threshold values needed to exit early. Although these methods can generalize to new data without retraining, they typically ignore system-level trade-offs and are not suitable for using in edge-cloud environments where latency and communication cost matters.

2.2 Split Computing Precursors to DDNNs

Hybrid Offloading for mobile image classification [20] showed that computing features locally and transmitting *feature tensors* (rather than raw images) to cloud can reduce bandwidth and energy usage by several factors without degrading accuracy. The concept of splitting a deep model between a mobile or edge device and the cloud for collaborative inference, selecting the partition point to balance local computation and transmission cost was introduced by **Neurosurgeon** [21]. Kang et all pioneered this idea by profiling each network layer's latency and data size to estimate total execution time. It proposes a lightweight scheduler to automaticity partition DNN computation between mobile devices and data centers split point that minimizes overall delay and mobile energy consumption while meeting an accuracy target.

Later, more works like **JALAD** [22],and **MeDNN** [23] pushed this direction with accuracy/latency-aware partition while applying compression to further reduce transferred activations under changing bandwidth. A more advanced work [24] performs*online* adaptation by dynamically profiling system resources and network state during inference instead of a static modeling once.

All in all, these works established the core principle of profiling the transmission and computations costs of a pipeline in order to effectively split a network.

Architecture Search and Design for Split Execution Another couple of research papers automates the design of split models using search optimization algorithms. **SplitNets** [25] and **JMSNAS** [26] aim to optimize network topology and location for edge–cloud computing, using neural architecture search to produce efficient transmission-cost splits while also taking into consideration hardware limitations.

In summary, profiling-based splitting in [21], [20], [22],[23], [24], [25], [26], [27], [28] consistently show that splitting a DNN across edge and cloud improves latency/energy versus all-local or all-cloud execution; They are natural precursors to modern DDNNs, where splitting is combined with joint training and (later) with early exits for adaptive inference. Our work extends this trajectory by applying learned, cost-aware early-exit routing within the DDNN framework.

2.3 Decision Mechanisms in DDNNs and Split Networks

The **Distributed Deep Neural Networks (DDNN)** architecture proposed by Teerapittayanon *et al.* [2] extends early-exit inference into a hierarchical, distributed framework spanning multiple end devices, edge servers, and the cloud. In their implementation, each end device (e.g., camera) hosts a shallow local shallow network that extracts intermediate feature representations. These feature activations are aggregated at the next level typically (edge) across multiple views of the same scene and used for an early local classification that will be criticized as confident or not to offload.

The offloading decision at each level is done by a *normalized entropy* confidence metric (see Section 1.2.3). Each local or edge classifier computes the entropy of its softmax output, and this value is compared against a predefined threshold: if the entropy is below the threshold (indicating high confidence), the prediction is accepted at that level; otherwise, the corresponding feature tensor is transmitted upward for further processing.

Experimental results demonstrated handling samples locally, not only reduces computation and transmission cost, but also improves overall classification accuracy of the DDNN. This work substantially influenced our research direction which will center the idea of replacing the heuristic entropy rule with an *optimized* one.

SplitEE and I-SplitEE: Multi-Armed Bandit-Based Splitting and Adaptive Offloading. Another important work would be the **SplitEE** framework [29] that combines deep neural network splitting with early-exit offloading decisions in an edge–cloud setup. The splitting process is optimized through a *multi-armed bandit* (MAB) algorithm, which explores different partition points between the edge and the cloud while evaluating transmission cost, inference latency, and confidence of the intermediate predictions at each exit. Through iterative exploration, the MAB controller selects a fixed split configuration common for all samples under a shared data distribution determining up to which layer the edge network should process inputs. Once this split point is set, each incoming sample is evaluated locally using a similar confidence metric that will keep a current prediction or offload the sample for further processing Building upon this concept, **I-SplitEE** [30] extends SplitEE by introducing *dynamic, per-sample splitting.* Instead of keeping a single partition for all inputs, I-SplitEE employs an enhanced MAB algorithm with an Upper Confidence Bound (UCB) strategy for online splitting into computation parts and adopts on each sample difficulty and current system conditions .

A closely related work and very interesting work is **DistrEE** [31], which targets distributed early exits across multiple edge devices. Instead of calculating a confidence metric on the local prediction, DistrEE compares the *feature difference* between consecutive layers or exits along the inference pipeline to assess the qualitative change of representations.

2.4 Optimization-Based Offloading

A major source of inspiration for this thesis is the work of Giannakas, Spyropoulos and other collaborators, who were the first to capture the offloading decision as an optimization problem and train an explicit network to effectively handle it instead of an heuristic technique [16]. The authors of **Fast Edge Resource Scaling with Distributed DNNs**, propose an edge–cloud architecture to handle regression-based time-series tasks under 5G constraints. They define a joint cost function that captures both prediction error and transmission penalties for each sample in order to train a separate DNN that would decide between offloading a sample or keep its early prediction minimizing the total DDNN cost.

Building on this foundation, **Ehsanian and Spyropoulos** extend this work to a multi-edge setting [32] introducing two local exits on a single edge pointwhile working with LSTM models. They also use a similar optimized rule first training an offloading mechanism for more efficient decisions.

Our present work adopts the same optimization-driven philosophy in the context of image classification, aiming to surpass the limitations of entropy-based heuristics by learning a lightweight neural rule that predicts, per sample, whether the cloud inference will yield a meaningful improvement over the edge result.

Chapter 3

Problem Setup

In this chapter, we present the overall structure and settings of the Distributed Deep Neural Network developed for our experiments. Our goal is to develop a simple hierarchical DDNN consisting of a lightweight local network (edge) and a more powerful one on the cloud. We will focus on an image classification task, so the DDNN is trained and evaluated on popular small-image benchmarks that are standard for this problem.

3.1 Background on CNNs for Image Classification

Firstly we have to mention some fundamental concepts and mathematical operations that will be used to create and set a DDNN architecture for classification tasks.

Fukushima work on *Neocognitron* was the first one mention convolution ideas using a hierarchical model with local receptive fields and subsampling as an ancestor of CNNs [33]. The first practical convolutional neural networks trained end-to-end with backpropagation were demonstrated by LeCun and colleagues on handwritten digit recognition which established a complete recipe for document/image recognition with backprop-trained CNNs [34]. The core ideas are local receptive fields, weight sharing (the same small filter applied at all spatial locations), and subsampling/pooling. Together they showed to reduce parameters and learn complex high-dimensional non-linear mappings making them suitable for image recognition task.

For completeness, we recall the standard definitions of *2D image convolution* and the closely related *2D cross-correlation* used by most deep-learning libraries (e.g., PyTorch). Both formulas are derived from the MIT online book of Goodfellow, Bengio, and Courville's *Deep Learning* (Ch. 9) [35].

Definition — *2D Convolution (mathematical)*

Notation. $I \in \mathbb{R}^{H\times W}$: input image (grayscale) with height H and width W. $K \in \mathbb{R}^{k_h\times k_w}$: 2D kernel (filter) of spatial size $k_h \times k_w$.

Definition. The 2D convolution of I with K is

$$S(i,j) \;=\; (I * K)(i,j) \;=\; \sum_m \sum_n I(m,n)\, K(i-m,\, j-n),$$

which corresponds to *flipping* the kernel and then sliding it across the input. For multi-channel images $X \in \mathbb{R}^{H\times W\times C}$ and kernels $K \in \mathbb{R}^{k_h\times k_w\times C}$, the sum also runs over channels $c = 1, \ldots, C$.

Source: Goodfellow, Bengio, and Courville, *Deep Learning*, Chapter 9 (Eq. 9.4) [35].

Definition — *2D Cross-correlation (used in CNN layers)*

Notation. $I \in \mathbb{R}^{H\times W}$: input image, $K \in \mathbb{R}^{k_h\times k_w}$: kernel. Indices i, j (output), m, n (kernel offsets). Deep-learning libraries (e.g., PyTorch `Conv2d`) implement this operation.

Definition. The 2D cross-correlation follows the same formula as convolution but *without* flipping the kernel:

$$S(i,j) \;=\; (K * I)(i,j) \;=\; \sum_m \sum_n I(i+m,\, j+n)\, K(m,n).$$

With stride/padding/dilation the offsets become

$$S(i,j) \;=\; \sum_m \sum_n I(i\cdot s_h + m\cdot d_h - p_h,\; j\cdot s_w + n\cdot d_w - p_w)\, K(m,n),$$

treating out-of-bounds indices as zeros (zero padding). For multi-channel inputs, sum also over $c = 1, \ldots, C$.

Definition — *Max Pooling (2D)*

Description. Max pooling is a downsampling operation commonly used in CNNs to reduce spatial dimensions of a feature map $x \in \mathbb{R}^{H\times W\times C}$. A window of size $k_h \times k_w$ slides across each channel with stride (s_h, s_w) and outputs the *maximum* value within the corresponding input window.

Intuition. Max pooling reduces spatial resolution while preserving the strongest local evidence acting as down-sampler in convolutional feature maps. (i) Using $s_=, s_w = k_h = k_w = 2$ (common on CNNs) reduces spatial dimension to half. (ii) With stride 1 it behaves as a local nonlinearity; (iii) with larger strides it performs subsampling.

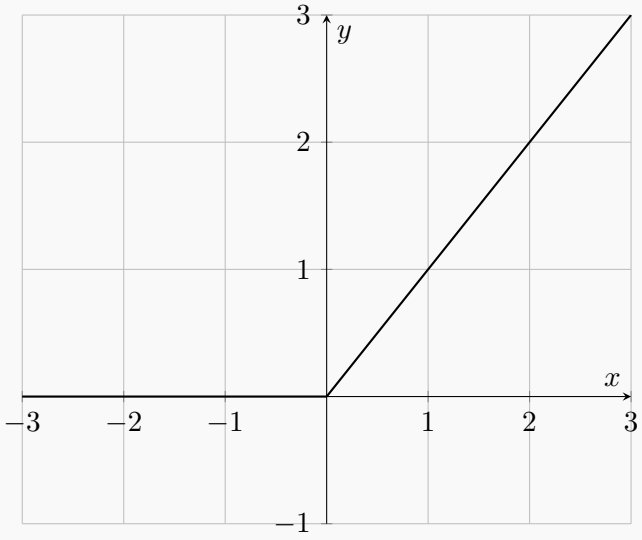
Source: Goodfellow, Bengio, and Courville, *Deep Learning*, Chapter 9 (Convolutional Networks — Pooling) [35].

Definition — *ReLU activation (Xu et al., 2015)*

Definition. The Rectified Linear Unit is

$$\mathrm{ReLU}(x) = \max(0,\, x).$$

ReLU is a non-linear activation function designed to keep the non-negative part of its argument. Its commonly used in deep learning to avoid avoid vanishing gradients and introduce sparse activation improving optimization and generalization.



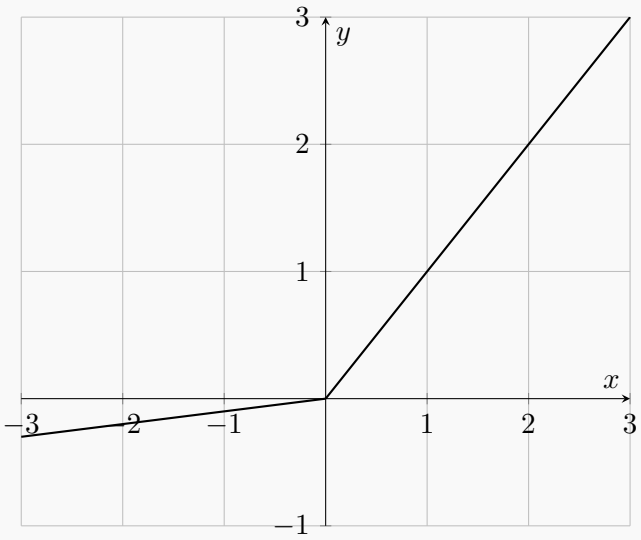
Source: Xu, Wang, Chen, and Li, *Empirical Evaluation of Rectified Activations in Convolutional Network* (2015) [36].

Definition — *Leaky ReLU activation (Xu et al., 2015)*

Definition. The Leaky Rectified Linear Unit keeps a small negative slope:

$$\text{LeakyReLU}(x) = \begin{cases} x, & x \geq 0, \\ \alpha\, x, & x < 0, \end{cases} \quad \text{with } \alpha \in (0, 1).$$

By allowing a small non-zero slope for $x < 0$, Leaky ReLU allows for some negative value neurons and reduces "dead" units observed with simple ReLU. We also use Leaky ReLU in our implementation (with α=0.01).



Source: Xu, Wang, Chen, and Li, *Empirical Evaluation of Rectified Activations in Convolutional Network* (2015) [36].

Definition — *Dropout*

Definition. During training, each hidden unit is independently set to zero with probability p and the remaining activations are scaled by $1/(1-p)$ (*inverted dropout*). This keeps the expected activation unchanged. At inference time, no dropout or scaling is applied.

Source: Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* [37].

Definition — *Data augmentation (for images)*

Definition. Data augmentation enlarges the training set by slightly transforming the inputs, improving generalization and reducing overfitting. Empirical studies show that simple geometric transforms (e.g., flips and translations) provide better training to CNNs without increasing model capacity [38]. In our experiments we use:

`RandomHorizontalFlip` → `RandomCrop(32, padding=4)` → `Normalize`.

Source: Pérez and Wang, *The Effectiveness of Data Augmentation in Image Classification using*

Deep Learning (2017) [38].

Modern deep CNNs were popularized by AlexNet, which combined deeper convolutional stacks with GPU training and advanced techniques that made the DNN extremely efficient: ReLU activations, data augmentation, and dropout regularization [7, 39, 37].

This combination delivered a large accuracy jump on ImageNet and effectively defined the modern deep CNN recipe. In our DDNN, we adopt the same principles with a *lightweight* convolutional stack at the edge (to meet compute budgets) and a *deeper* cloud model for harder samples.

- **ReLU** Non-Linear activation function that combats vanishing gradients by using a non-saturating positive branch, promote *sparsity* in activations (many zeros, acting like mild regularization). Leaky ReLU further reduces "dead" neurons by keeping a small negative slope.
- **Data augmentation.** Simple data transformation (random crops, flips) expand the effective training distribution and reduce overfitting.
- **Dropout on FC layers.** Reduces co-adaptation of high-level features and improves generalization by averaging many thinned networks.
- **Max pooling** — downsamples feature maps by taking local maximum, enlarges the effective receptive field, and makes the network less sensitive to small shifts

3.2 Network Architecture

3.2.1 Layer Structure

The distributed model in our system shown on Figure 3.1 leverages 2D Convolutional Neural Networks (CNNs), which are well-suited for image classification tasks due to their ability to capture spatial hierarchies in visual data. The network is split into a local network and a cloud network, each responsible for different aspects of feature extraction and classification.

Our architecture is inspired by earlier works from [2], [16], [32] that proposed a edge-cloud architecture consisting of a local CNN network with early-exit, followed by a more powerful one having more layers located in the cloud.

As shown in Figure 3.2 (adapted from [2]), the DDNN can aggregate signals from *multiple end devices* (e.g., different camera angles) through a local aggregator and produce an *early* decision at the edge. If this decision is deemed confident, inference terminates locally; otherwise, the sample is forwarded to the cloud exit.

In our work, for simplicity we adopt a *single* edge device (one local network), while keeping the same edge–cloud split proposed in prior DDNN studies [2, 16]. Following the pipeline illustrated in Figure 3.3, an input image is first processed by a local CNN (*Local Feature Extractor*) $\rightarrow$ a small fully connected head (*Local Classifier*) that yields an early prediction. An offloading rule then evaluates the confidence of this local prediction; if not confident, instead of sending the raw image, the edge forwards the *latent feature tensor* to the cloud CNN+FC for further processing. Receiving the already processed the already processed features by the local network reduces computation needed from the cloud based one effectively lowering latency cost. A more detailed step-by-step view of this per-sample routing appears later in our *Data Flow* section (Figure 3.5).

For our specific problem, we aimed to design a network that allows for clear decision boundaries between local and cloud processing. Ideally, the cloud network should be able to classify harder samples with at least **10**% overall higher accuracy than the local network, ensuring a meaningful performance gap that justifies offloading decisions. In our case, we achieved a **15**%

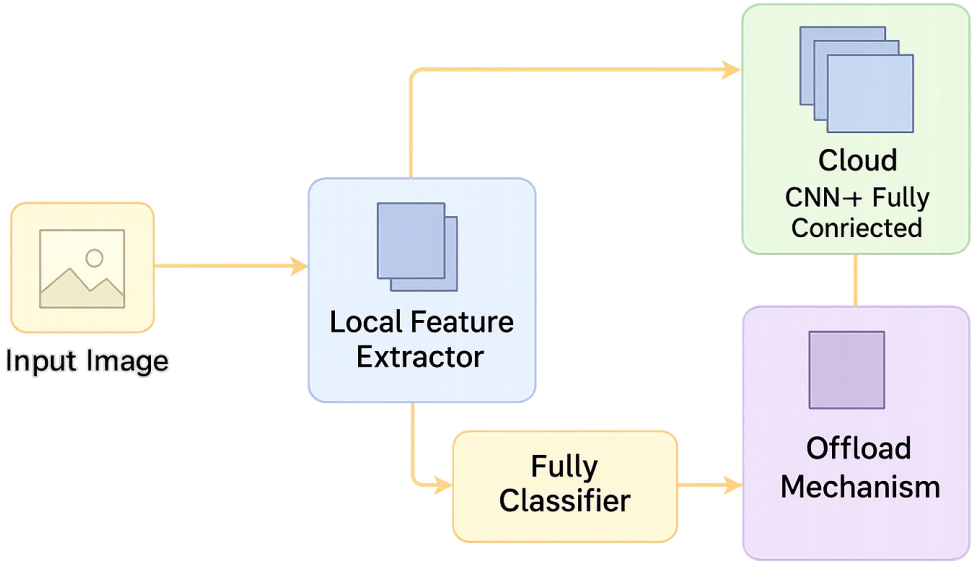


Figure 3.1: Overview of a DDNN network consisting of a small CNN feature extractor, local-cloud exits and the offloading mechanism

accuracy difference, making it possible to quantitatively evaluate whether our offloading mechanism correctly assigns samples to the most appropriate classifier. Our cloud backbone follows an AlexNet-style design (five convolutional layers followed by three fully connected layers) [7], which is known to achieve strong accuracy on datasets substantially harder than MNIST and is a good fit for our problem setting. This approach ensures that our model follows a well-established design, making it reliable and capable of deployment in a real-world data constrained scenario.

Local Network

The local network consists of two components, the *Local Feature Extractor* and the *Local Classifier*.

Local feature extractor (edge). We adopt a compact CNN stack tailored to small images (CIFAR-10), aiming to keep parameters and latency low:

$$\text{Conv}(3\to32,\ 3\times3)(\mathit{Dropout}(0.2)) \ \to\ \text{BN} \ \to\ \text{LeakyReLU} \ \to\ \text{MaxPool}(2)$$

This stage extracts the first feature maps and reduces resolution via max pooling.

Local classifier (early exit). On top of the extractor we attach a lightweight head to allow early decisions for easy samples:

$$\text{Flatten}(32\times16\times16\to8192) \ \to\ \text{FC}(8192\to64) \ \to\ \text{LeakyReLU} \ \to\ \text{FC}(64\to10).$$

This stage follows the early-exiting idea popularized by BranchyNet [1] and enables low-latency local predictions.

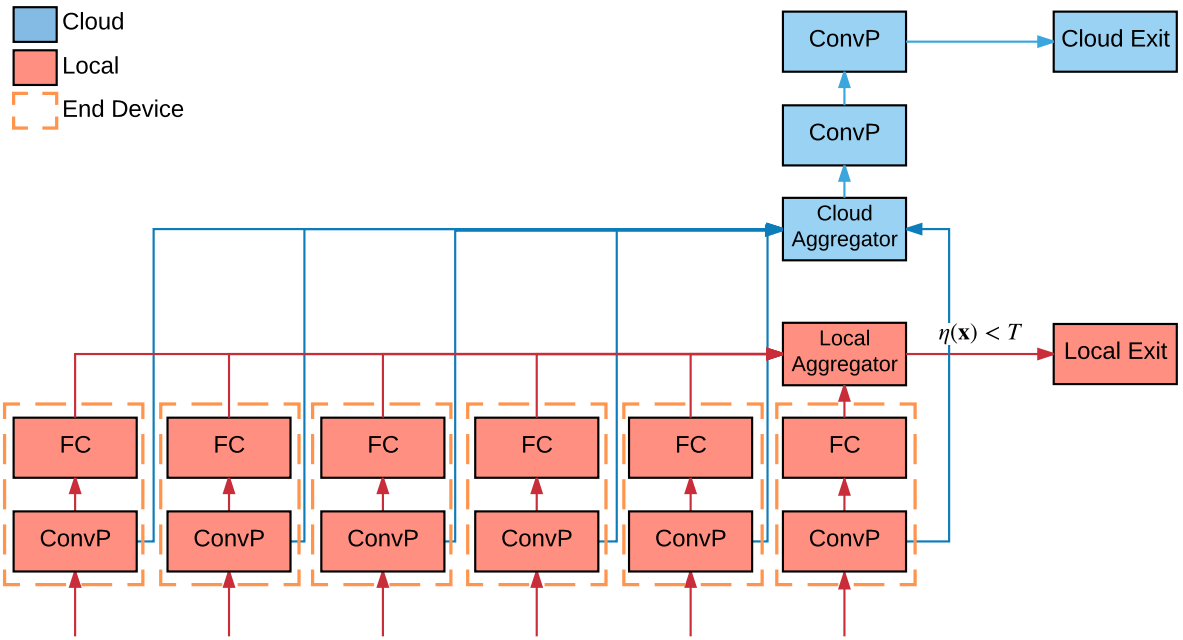


Figure 3.2: Distributed DNN with multiple end devices and an early local exit, adapted from [2].

Cloud model (remote). The cloud branch is deeper/wider to boost accuracy on difficult inputs:

$$\text{Conv}(32\to64,\ 3\times3,\ )(\textit{Dropout}(0.2)) \to \text{BN} \to \text{LeakyReLU} \to \text{MaxPool}(2),$$

$$\text{Conv}(64\to128,\ 3\times3,\ )(\textit{Dropout}(0.2)) \to \text{BN} \to \text{LeakyReLU} \to \text{MaxPool}(2),$$

$$\text{Conv}(128\to256,\ 3\times3,\ )(\textit{Dropout}(0.2)) \to \text{BN} \to \text{LeakyReLU}, \quad \text{Conv}(256\to256,\ 3\times3,\ )(\textit{Dropout}(0.2)) \to \text{BN}$$

$$\text{Flatten}(256\times2\times2\to1024) \to \text{FC}(1024\to256) \to \text{LeakyReLU} \to \text{FC}(256\to64) \to \text{LeakyReLU} \to \text{FC}(64\to10).$$

Overall, the edge path prioritizes latency and communication savings, while the cloud path maximizes accuracy; both are jointly trained to ensure balance between their learning.

3.2.2 Dataset Selection

In this work, we utilize the CIFAR-10 dataset , which consists of 60,000 color images divided into 10 distinct classes, with each image being 32x32 pixels in size. The dataset is split into 50,000 images for training and 10,000 images for testing. In our implementation, we further split 5,000 images from the training set to serve as the validation set, leaving 45,000 images for actual training. We use CIFAR-10 for most experiments and plots to ensure fair and stable comparisons. Unless otherwise stated, all reported results are on CIFAR-10.

We chose this dataset as it presents a more challenging classification problem compared to simpler datasets, such as MNIST, due to the higher complexity and variability of the images while still being a popular and easy to use dataset. The increased difficulty allows for a clearer demonstration of the advantages of the cloud network over the local classifier in terms of classification accuracy. Below is an overview of the CIFAR-10 dataset fig. 3.4

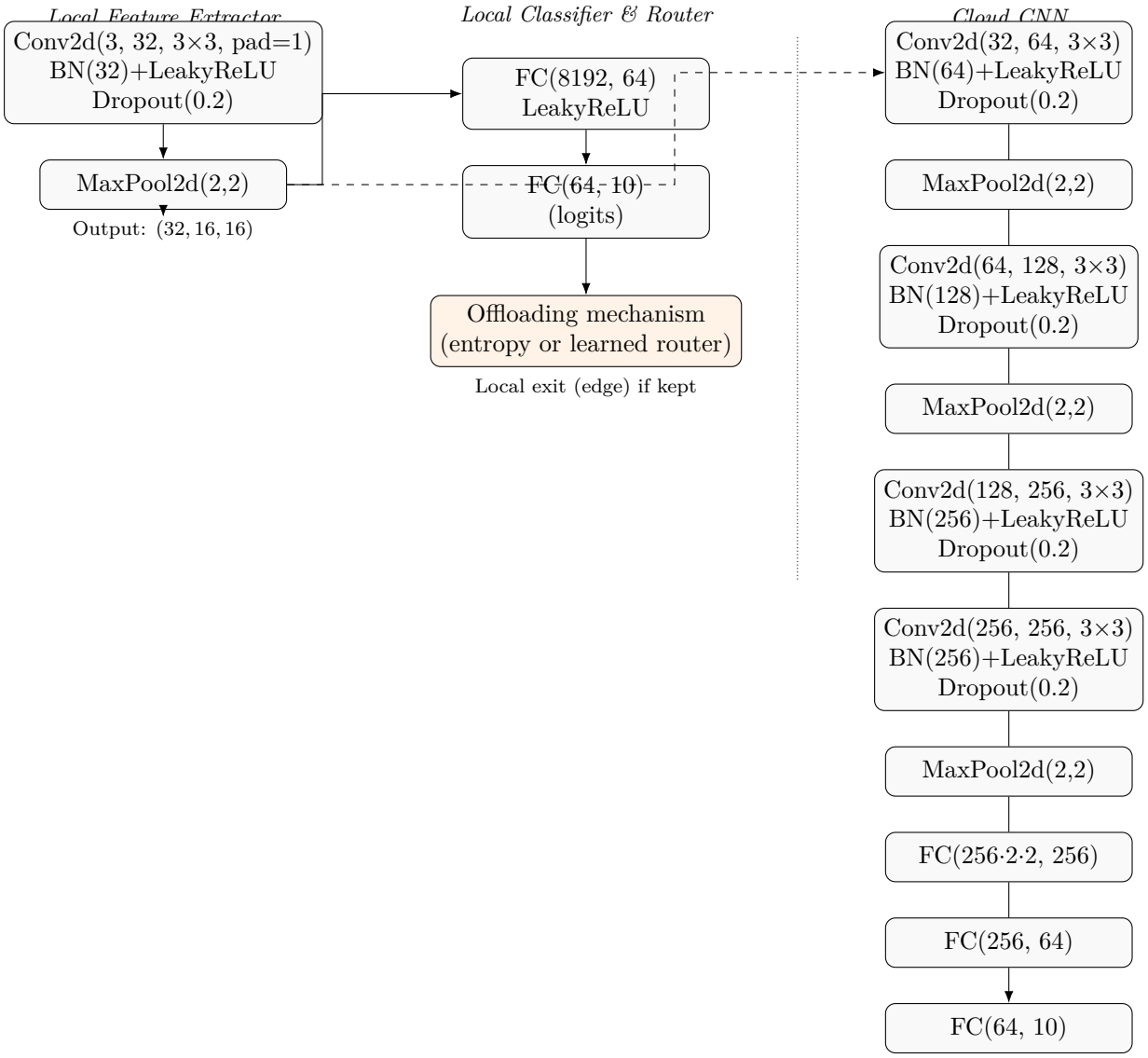


Figure 3.3: Distributed Deep Neural Network (DDNN) architecture as used in our work. Left: local feature extractor. Middle: local classifier and offloading mechanism (entropy threshold or learned router). Right: cloud CNN.

3.3 Inference

Definition — *Softmax*

Definition. For logits $\mathbf{y} \in \mathbb{R}^{|C|}$, the softmax produces class probabilities

$$\sigma(\mathbf{y})_i = \frac{e^{y_i}}{\sum_{j \in C} e^{y_j}}, \qquad i \in C,$$

$\sigma(\mathbf{y})_i \in (0,1)$, $\sum_{i \in C} \sigma(\mathbf{y})_i = 1$.
The softmax function maps a vector y of C real-valued logits into a probabilitiy distribution over C classes, where its probability is proportional to the exponential of its corresponding input component.

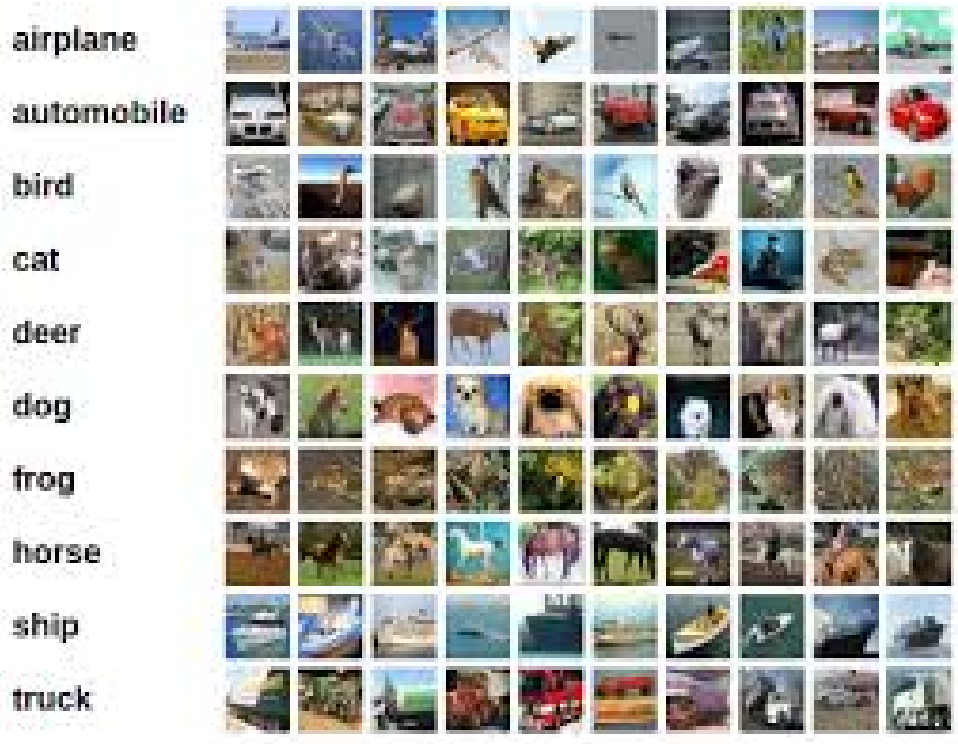


Figure 3.4: Overview of Cifar-10 Dataset.

Definition — *Cross-Entropy (classification)*

Single-sample. For a ground-truth class $y \in C$ and predicted probabilities $\mathbf{p} \in (0,1)^{|C|}$ with $\sum_{i \in C} p_i = 1$,

$$\ell_{\mathrm{CE}}(\mathbf{p}, y) = -\log \mathbf{p}[y].$$

Dataset Estimation . For $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{N}$ and model probabilities $\mathbf{p}_\theta(\cdot \mid x_k)$,

$$\mathcal{L}_{\mathrm{CE}}(\theta) = \frac{1}{N} \sum_{k=1}^{N} \ell_{\mathrm{CE}}(\mathbf{p}_\theta(\cdot \mid x_k), y_k).$$

3.3.1 Data Flow

Figure 3.5 illustrates the per-sample flow in our DDNN network. An input image x first passes through the *Local Feature Extractor* $f_E(\cdot)$, which produces the latent representation $z = f_E(x)$ as defined in (1). This latent vector z captures compact feature information extracted at the edge and subsequently feeds (i) the *Local Classifier* $g_L(\cdot)$, producing the local logits $\mathbf{y}_L = g_L(z)$ according to (2), and (ii) the *Offloading Mechanism*, which decides whether the sample should exit locally or be sent to the cloud.

Throughout this work, the Offloading Mechanism consumes the local logits as input—specifically $\mathbf{y}_L$ *from* (2)*—from which it derives the confidence score*

If the offloading score indicates sufficient local confidence, the system returns the local prediction $\hat{y}_L = \arg\max_i \mathbf{p}_L[i]$ obtained from $\mathbf{p}_L = \mathrm{softmax}(\mathbf{y}_L)$ as in (3). Otherwise, only the compact latent features z are forwarded to the *Cloud* CNN+FC, modeled as $g_R(\cdot)$, which continues inference by computing $\mathbf{y}_R = g_R(z)$ and the corresponding probabilities $\mathbf{p}_R = \mathrm{softmax}(\mathbf{y}_R)$ also defined in (2)–(3). The cloud then produces the final prediction $\hat{y}_R = \arg\max_i \mathbf{p}_R[i]$.

Quantity	Expression	Eq.
Latent features	$z = f_E(x; \theta_E)$	(1)
Local exit logits	$\mathbf{y}_L = g_L(z; \theta_L)$	(2)
Cloud exit logits	$\mathbf{y}_R = g_R(z; \theta_R)$	(2)
Local probabilities	$\mathbf{p}_L = \mathrm{softmax}(\mathbf{y}_L)$	(3)
Cloud probabilities	$\mathbf{p}_R = \mathrm{softmax}(\mathbf{y}_R)$	(3)
Local CE loss	$\ell_L(x, y) = \ell_{\mathrm{CE}}(\mathbf{p}_L, y)$	(4)
Cloud CE loss	$\ell_R(x, y) = \ell_{\mathrm{CE}}(\mathbf{p}_R, y)$	(4)

Table 3.1: Feed-forward variables and where they are defined.

For each sample $(x) \in \mathcal{D}$, the edge extractor f_E produces a latent feature tensor

$$z \;=\; f_E(x) \in \mathbb{R}^{d_z}. \tag{1}$$

The local head g_L and the cloud head g_R map z to logits

$$\mathbf{y}_L \;=\; g_L(z) \in \mathbb{R}^{|C|}, \qquad \mathbf{y}_R \;=\; g_R(z) \in \mathbb{R}^{|C|}. \tag{2}$$

We obtain class probabilities via the softmax *as defined in* Definition 3.3:

$$\mathbf{p}_L = \mathrm{softmax}(\mathbf{y}_L), \qquad \mathbf{p}_R = \mathrm{softmax}(\mathbf{y}_R). \tag{3}$$

The per-sample classification losses are cross-entropies *as in* Definition 3.3:

$$\ell_L(x, y) = \ell_{\mathrm{CE}}(\mathbf{p}_L, y) \;=\; -\log \mathbf{p}_L[y], \qquad \ell_R(x, y) = \ell_{\mathrm{CE}}(\mathbf{p}_R, y) \;=\; -\log \mathbf{p}_R[y]. \tag{4}$$

Feed-forward definitions.

3.4 Training

This training setup corresponds to the data flow illustrated in Figure 3.6, which shows the complete DDNN pipeline during training, including the joint loss computation based on both local and cloud outputs.

$$\min_{C_J} \; C_J = C_{L,k} \cdot L_0 + C_{R,k} \cdot (1 - L_0), \qquad \text{with} \quad \max L_0. \tag{3.1}$$

where $C_{L,k}$ and $C_{R,k}$ denote the local and remote inference costs for sample k, and L_0 represents the fraction of inputs processed locally. The objective is to minimize C_J by maximizing L_0, keeping as many samples as possible on the edge without degrading accuracy.

3.4.1 Loss Function

In our DDNN we jointly train the edge-cloud network under a single loss, so that the earlier exit learns to map features correctly for the cloud exit while also being able to handle "easy" samples. This follows a well-established practice in split/distributed network, early-exit networks train intermediate heads together with the backbone exit [1, 13, 40, 31, 32], and split/DDNN systems co-train the edge and cloud parts to minimize loss while preserving accuracy [2, 41, 31]. In our setting, joint training effectively passes gradients and weights of the cloud network to the local one increasing the edge performance while also acting as a regularizer for the remote exit y_R

and benefiting its classification accuracy too. An effect also observed in related DDNN studies [2, 16, 42].

Per-sample joint loss (with weights). As in (4) local and cloud per sample cost are $\ell_L(x, y) = \ell_{\text{CE}}(\mathbf{p}_L, y)$, $\ell_R(x, y) = \ell_{\text{CE}}(\mathbf{p}_R, y)$

We define the joint loss as:

$$\ell_{\text{joint}}(x, y; \theta_E, \theta_L, \theta_R) = w\,\ell_L(x, y) + (1-w)\,\ell_R(x, y) = -\,w \log \mathbf{p}_L[y] - (1-w) \log \mathbf{p}_R[y], \quad w \in [0, 1]. \tag{6}$$

The weight w controls the edge–cloud trade-off: larger w favors the local exit , while smaller w prioritizes the remote exit's training.

Dataset objective. Averaging over $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^N$ yields

$$\mathcal{L}_{\text{joint}}(\theta_E, \theta_L, \theta_R) = \frac{1}{N} \sum_{k=1}^{N} \Big[w\,\ell_L(x_k, y_k) + (1-w)\,\ell_R(x_k, y_k) \Big] = w\,\mathcal{L}_L + (1-w)\,\mathcal{L}_R, \tag{7}$$

where $\mathcal{L}_L = \frac{1}{N} \sum_k \ell_L(x_k, y_k)$ and $\mathcal{L}_R = \frac{1}{N} \sum_k \ell_R(x_k, y_k)$.

Notation clarification. Throughout this work, we distinguish between:

- ℓ_L, ℓ_R, $\mathcal{L}_{\text{joint}}$: **training losses** (cross-entropy terms minimized during optimization);
- $C_{L,k}$, $C_{R,k}$, C_J: **inference costs** (classification cost per sample).

The training losses quantify classification accuracy error, whereas the provisioning costs will be used to train the offloading mechanism.

3.4.2 Adam optimizer

Joint training of the DDNN components is optimized using the Adam optimizer proposed at [43], a widely adopted algorithm for stochastic gradient-based (SGD) optimization. Adam uses momentum to accelerate the gradient descent process by incorporating an exponentially weighted moving average of both past gradients m_t and heir squared values v_t. Combining both techniques provide a more balanced and efficient optimization process.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2.$$

where g_t is the current gradient, β_1 and β_2 are exponential decay rates, and typical values are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

For initialization bias-corrected estimates are applied (since both m_t and v_t start at zero):

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

The model parameters are then updated using these corrected estimates as:

$$\theta_{t+1} = \theta_t - \alpha\,\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where α is the learning rate and ϵ is a small constant (typically 10^{-8}) to avoid division by zero. The learning rate α controls the step size in parameter updates; for deep neural networks, a common choice is $\alpha = 0.001$, which we also adopt in our experiments unless otherwise stated.

Adam is straightforward to implement, computationally efficient, and requires minimal memory showing strong performance across Deep-Learning architectures and tasks.

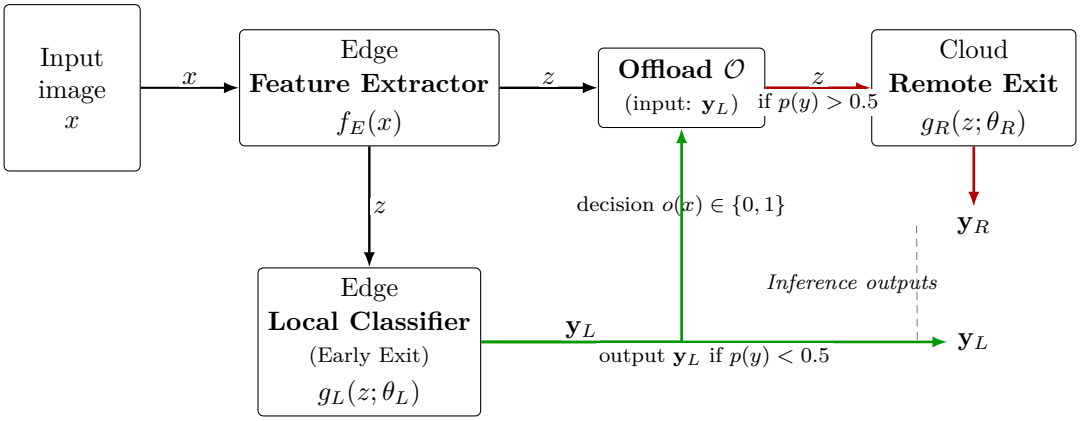


Figure 3.5: Inference data flow. The edge extractor produces $z = f_E(x;\theta_E)$ from input image x. The local classifier (Early Exit) computes $\mathbf{y}_L = g_L(z;\theta_L)$; the offloading mechanism *consumes* $\mathbf{y}_L$ to decide $o(x)$ and, if $o(x)$=1, *forwards* the latent z to the cloud, which outputs $\mathbf{y}_R = g_R(z;\theta_R)$.

3.4.3 Loss weight initialization

Following previous DDNN work, we implement a joint training loss function combining both cloud and local loss using scalars weights to balance their contributions.

In the TSNM framework of Giannakas et al [16], the joint cost is defined as

$$C_J = w\, f(y_L, d; \theta_0, \theta_1) + (1 - w)\, f(y_R, d; \theta_0, \theta_2)$$

assigns a weight $w \in [0, 1]$ to the local exit, with $1 - w$ applied to the remote exit. While they mention that further experiments regarding the weight values showed that training with $w > 0$ regularizes the cloud output y_R, improving test accuracy compared to the centralized case ($w = 0$), and that the best trade-off was observed around $w = 0.7$. By contrast, Teerapittayanon et al [2] reported that the initialization of w did not significantly affect performance, and thus adopted neutral or random values during training.

In our implementation, we follow a similar stochastic initialization strategy: at the beginning of each training run, the local loss weight $w_L \sim \mathcal{U}(0, 1)$ while the cloud weight is set as its complement

$$w_R = 1 - w_L. \tag{3.2}$$

We remind that the joint loss function is defined as:

$$\ell_{\text{joint}}(x, y; \theta_E, \theta_L, \theta_R) = w\, \ell_L(x, y) + (1-w)\, \ell_R(x, y) = -\, w \log \mathbf{p}_L[y] - (1-w) \log \mathbf{p}_R[y], \quad w \in [0, 1]. \tag{6}$$

We also have to mention than in training the DDNN with a local weight $w_L = 0$ corresponds to having a completely centralized Deep Neural Network with a single exit and any further optimization.

In order to backup this approach empirically tested the DDNN performance with different weight configurations multiple times while using the entropy-based offloading rule and verified stability across classification percentages (see Fig. 6.17), confirming robustness to the weighting scheme.

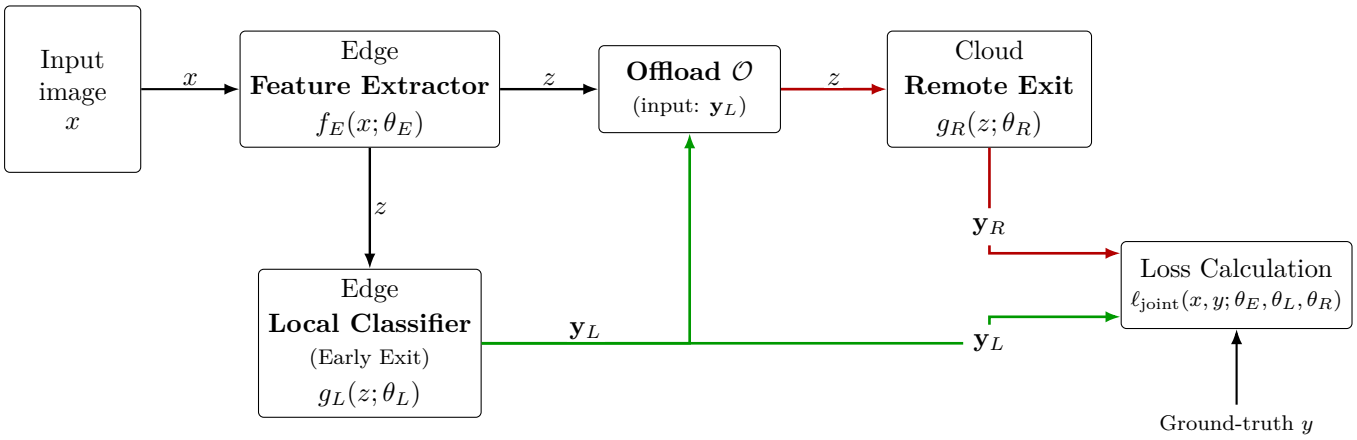


Figure 3.6: Training data flow and joint loss. From input image x, the edge extractor produces $z{=}f_E(x;\theta_E)$; the local classifier (Early Exit) outputs $\mathbf{y}_L$, and the cloud outputs $\mathbf{y}_R$. Both outputs feed the *Loss Calculation* box, where the joint objective $\ell_{\text{joint}}(x,y;\theta_E,\theta_L,\theta_R)$ is computed using also the ground-truth label y.

Chapter 4

Problem Solution

4.1 Offloading Mechanism $\mathcal{O}$

Definition (Offloading mechanism $\mathcal{O}$). We define $\mathcal{O}$ as an *online* mechanism located at the edge responsible for deciding whether the inference output will be produced via the early exit or the remote one. For every image sample x coming through the DDNN pipeline, $\mathcal{O}$ observes only edge-available information (e.g., local features, logits, or early $\mathbf{y}_L$ prediction probabilities) and outputs

$$o_x \in \{0, 1\}, \qquad o_x = 0 \text{ (resolve locally)}, \quad o_x = 1 \text{ (offload to cloud)}.$$

Over a horizon of N samples, the fraction of them resolved locally is

$$L \;=\; \frac{1}{N}\sum_{t=1}^{N} o_x \;\in [0, 1]. \tag{4.1}$$

Offloading Objective. The operator seeks offloading decisions that *minimize* the overall cost, while ensuring that at least a target fraction L_0 of samples are handled locally,

$$L \geq L_0. \tag{4.2}$$

L_0 is set by the provider in order to tune the edge demand and bandwidth usage

$$L_0 \in \{0, 1\}$$

Let $C_{L,x}$ and $C_{R,x}$ denote the per-sample provisioning cost if sample k is decided locally or remotely, respectively. The offloading mechanism $\mathcal{M}$ aims to identify a selected number of "difficult" samples (out of T total) for which the difference

$$B_x = C_{L,x} - C_{R,x}, \tag{4.3}$$

takes the smallest possible values. Ideally, all selected samples would satisfy $B_x \gg 0$, meaning that the remote (cloud) inference offers a clear benefit over the local one, thus justifying offloading. In practice, this depends on the intrinsic difficulty of each sample and on the available edge resources. The operator can therefore manually tune L_0 according to the desired balance between latency, bandwidth, and computational cost of the deployed DDNN setup.

Mechanism $\mathcal{O}$ has to solve this decision process *online*, using only local information available at the edge, and dynamically adjust its offloading policy to the specific characteristics of the current classification task and network conditions.

The metric B_k defined in Eq. (4.3) will later be used as the basic data provided to train the optimized offloading rule in Section 4.2.1.

4.1.1 Offloading Rule with Entropy

As mentioned on the Relative Research (chapter 2) multiple previous early-exit and distributed deep neural networks settings use a typical heurstic confidence meter as an offloading mechanism in order to quickly decide between an early prediction and further processing the input. Teerapittayanon et al used the (normalized) entropy of the local probability vector $\mathbf{p}_L$ (see Def. 1.2.3) to quantify the uncertainty of an early prediction both in [17, 2].

Entropy measures how concentrated or spread the predicted class distribution is, thus indicating how confident the local classifier is about its prediction. Low entropy means one class dominates with significant percentage over the others and the sample can *exit locally*; high entropy means the mass is spread across classes and the sample should be *forwarded* to the cloud for a more reliable decision. Operationally, the offloading rule compares entropy to a threshold in order to decide.

A similar approach was made on TSNM [16] considering a regression task, where entropy over discrete classes is not applicable. Instead, they used Monte Carlo (MC) dropout to estimate confidence: Multiple stochastic forward passes using dropout produce a sample of outputs, and the predictive uncertainty is estimated from the output variance.

Since we implement an image recognition network we will use the entropy confidence rule in our results as a reference to compare its accuracy with our Optimized Confidence Rule.

To ensure a fair comparison between entropy and our mechanism, we sweep the entropy threshold so that the fraction of locally classified samples (*local exit rate*) closely matches that achieved by our optimized rule; we then report accuracy/latency under matched offload budgets.

4.2 Optimized Offloading Mechanism

4.2.1 Optimized Rule

In this section, we introduce the proposed Optimized Offloading Mechanism. The key idea is to take advantage of the information obtained from the already trained DDNN and translate the offloading task as an optimization one to derive data-driven offloading decisions, rather than relying solely on uncertainty-based heuristics. After the joint training stage , all DDNN components — namely the *Edge Feature Extractor* the *Local Classifier*, the *Cloud Exit* — and their weights are considered fixed and optimized for classification tasks. At this point, we can switch the model into inference mode and analyze its behavior on training data while using both local and cloud exit in order to obtain their classification costs. For each input sample x, we record the corresponding *local* and *remote* provisioning costs $C_{L,x}$ and $C_{R,x}$ and consider them as known inputs for the offline training of the Optimized Rule. We remind that the offloading mechanism $\mathcal{O}$ outputs a single binary value $o_x \in \{0, 1\}$ for every sample being processed detecting the exit for classification and that at least target fraction of $L0$ (provider settings) samples should be processed locally.

Using these fixed and known cost values, the offloading problem can be expressed as offline optimization task where our goal is to minimize the average joint classification cost of the DDNN across all N samples, subject to a local processing constraint defined by L_0.

Optimized Offloading Problem:

Symbol	Description
o_x	Binary offloading decision variable for each input sample x, where $o_x = 0$ denotes local process
$C_{L,x}$	Local classification cost for sample x.
$C_{R,x}$	Remote classification cost for sample x.
N	Total number of samples in the dataset.
L_0	Target minimum fraction of samples that must be processed locally (latency constraint).

Table 4.1: Variables used in the Optimized Offloading Problem.

$$\begin{aligned} \min_{\{o_x\}} \quad & \frac{1}{N} \sum_{x=1}^{N} \Big(C_{L,x}\, o_x + C_{R,x}\, (1 - o_x) \Big) \\ \text{subject to} \quad & \frac{1}{N} \sum_{x=1}^{N} o_x \geq L_0 \\ & \sum_{x=1}^{N} o_x = N \end{aligned} \tag{4.4}$$

where:

Optimization variables. *Note.* The above optimization formulation refers to the *offline training* phase of the offloading mechanism, where every cost value is known, while on *inference* the mechanism has to operate online and typically predict these numbers.

4.2.2 Data Labelling

After defining the optimization goal for our offloading mechanism we have to create a labeled dataset for training. We proceed to label the training data using the above per sample x variable that quantifies the *benefit of offloading* a particular sample. The idea is to simulate an oracle rule that already knows the provision costs and takes optimal decisions.

$$B_x = C_{L,x} - C_{R,x}, \tag{4.5}$$

The variable b_x represents the difference between the local and cloud classification costs for sample x. A positive value of b_x indicates that the local cost exceeds the cloud cost, meaning that the sample would be more efficiently classified through the cloud network, while a negative value suggests that the local network already performs better. The local and cloud classification costs, $C_{L,x}$ and $C_{R,x}$, are computed from the predicted probabilities of the correct class by the local and cloud classifiers, respectively. Specifically, we define:

$$C_{L,x} = 1 - p_{L,y}, \qquad C_{R,x} = 1 - p_{R,y}, \tag{4.6}$$

where $p_{L,y}$ and $p_{R,y}$ represent the predicted probabilities for the true class y by the local and cloud networks, respectively. This formulation suggests that the more confident a network is in predicting the correct class i.e.*the higher the probability assigned*), the lower the associated classification cost will be.

For instance, if either the local network predicts the correct class y with probability 0.85, then its local classification cost will be:

$$C_{L,x} = 1 - 0.85 = 0.15.$$

Symbol	Description
B_x	Difference between local and remote classification costs (*benefit*).
B^*	Threshold separating "easy" (local) and "hard" (remote) samples.
$C_{L,x}$	Local classification cost, computed as $1 - p_{L,y}$.
$C_{R,x}$	Remote classification cost, computed as $1 - p_{R,y}$.
$p_{L,y}$, $p_{R,y}$	Predicted probabilities for the correct class by local and cloud networks.
L_0	Target local ratio, defining the desired fraction of locally processed samples.

Table 4.2: Definitions used in the oracle-based labelling process.

Conversely, if the cloud network predicts the same class with confidence 0.95, its corresponding cost is:

$$C_{R,x} = 1 - 0.95 = 0.05.$$

In this case, $B_x = C_{L,x} - C_{R,x} = 0.10 > 0$, indicating that the cloud network provides a lower expected cost and should therefore process the sample.

In order to decided between *easy* (local) and *hard* (remote) samples, we introduce a threshold value b^* that represents the classification boundary between these two categories. B^* is defined as the L_0 percentile of the sorted B_x values, the higher the desired percentage of local processing, the larger the portion of samples that must be labeled as "easy."

Formally, all samples whose benefit values satisfy $B_x < B^*$ are labeled as *easy* (process locally), whereas those with $B_x \geq B^*$ are labeled as *hard* (offload to the cloud). During the inference period of the DDNN we perform a single forward pass, storing the associated B_x value for every sample paired together and calculate the desired B^* .

$$\text{choose } o_x = \begin{cases} 0, & \text{if } B_x \leq B^*, \quad \text{(resolve locally)} \\ 1, & \text{if } B_x > B^*, \quad \text{(offload to cloud).} \end{cases}$$

Algorithm 1 — Data Labelling (Oracle-based)

1. *Input:* target local rate $L_0 \in [0, 1]$, per-sample costs $\{C_{L,x}\}_{x=1}^{N}$ and $\{C_{R,x}\}_{x=1}^{N}$.
2. Compute benefit values $B_x \leftarrow C_{L,x} - C_{R,x}$ for all samples x.
3. Sort $\{b_x\}$ in ascending order.
4. Determine the critical threshold $b^* \leftarrow \text{percentile}\big(L_0,\ \{B_x\}\big)$.
5. Pair each sample x with its corresponding benefit value b_x.
6. Assign labels based on the oracle rule:

$$\text{o}(x) = \begin{cases} 0, & \text{if } B_x < B^* \text{ ("easy", local)} \\ 1, & \text{if } B_x \geq B^* \text{ ("hard", remote)} \end{cases}$$

7. *Return:* the labeled dataset $\{(b_x, x)\}_{x=1}^{N}$.

Note that here $C_{L,x}$ and $C_{R,x}$ represent classification *costs* at inference time, and are distinct from the training losses ℓ_L, ℓ_R used in Eq. (6). The training losses are optimized during learning, whereas the provisioning costs are fixed measurements of the deployed model's runtime behavior.

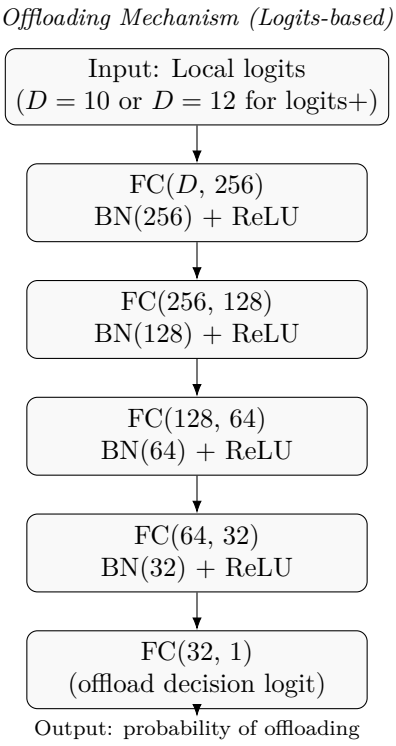


Figure 4.1: Architecture of the logits-based optimized offloading mechanism.

4.2.3 Layer Structure

Before finalizing the network design of the offloading mechanism, several input formats were explored, each aiming to improve overall DDNN accuracy through more informed offloading decisions and to outperform the baseline entropy-based routing method. We tested three different input approaches regarding the representation the mechanism should receive as input based on the data provided at the edge: the raw input image, the early feature maps extracted at the edge, and the final local classifier outputs (logits). Testing (see **??**), proved that the logits-based input format was the most effective in terms of overall DDNN accuracy.

Feeding the mechanism with logits provides a compact and efficient representation of the local classifier's prediction,while allowing the model to learn patterns from the classifier's confidence distribution. Logits are low-dimensional numerical inputs, significantly lighter than handling raw images or high-dimensional feature maps, resulting in a smaller and faster model suitable for on-device deployment. Battery consumption is also reduced since the inference is faster due to lower complexity calculations while maintaining higher accuracy than the entropy-based baseline.

The final layer design is summarized below.

$$\begin{aligned}
&\text{FC}(D \to 256) \;\to\; \text{BN}(256) \;\to\; \text{ReLU},\\
&\text{FC}(256 \to 128) \;\to\; \text{BN}(128) \;\to\; \text{ReLU},\\
&\text{FC}(128 \to 64) \;\to\; \text{BN}(64) \;\to\; \text{ReLU},\ \text{FC}(64 \to 32) \;\to\; \text{BN}(32) \;\to\; \text{ReLU} \;\to\; \text{FC}(32 \to 1).
\end{aligned}$$

The Optimized Rule classifier outputs just a single value representing the offload decision and is trained with binary cross-entropy (with logits) using oracle labels.

4.2.4 Training

The dataset used for training the offloading mechanism is the labeled meta-dataset introduced in Section 4.2.2. Each training sample is represented by the local classifier's logits (or logits+ variant) and its corresponding offloading label, rather than by the raw input image or early feature maps. In this way the offloading mechanism is able to learn an offloading pattern that will output a single probability indicating the offload decision $o_x \in \{0, 1\}$.

Since the offloading decision is a binary classification problem (local exit vs. cloud exit), the appropriate loss function is the binary cross-entropy (BCE).

Definition — *Binary Cross-Entropy Loss*

Definition. For a binary classifier with true labels $y_i \in \{0, 1\}$ and predicted probabilities $\hat{p}_i \in [0, 1]$, the average binary cross-entropy loss over N samples is defined as

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)\Big].$$

Notation. y_i: ground-truth oracle label (1 for *offload*, 0 for *local*); $\hat{p}_i$: predicted probability of offloading (sigmoid of output logit); N: total number of training samples; $\mathcal{L}_{\text{BCE}}$: average binary cross-entropy loss. The lower the value , the stronger the alliance between predictions and oracle labels.

During training, the offloading mechanism receives the logits-input, passes it through the fully connected network as described in Figure 4.1, and computes the single output logit.

The sigmoid of this logit is compared to the oracle label $y \in \{0, 1\}$ using the BCE loss.

The model is optimized using Adam stochastic-gradient algorithm Section 3.4.2 and early stopping based on validation accuracy of offloading decisions.

Notation. Given that the meta-dataset is constructed using the threshold B^* based on the latency requirement L_0, a change in L_0 changes also B^* and thus the labels. As a result, a new *Optimized Rule* must be (re)trained whenever L_0 is modified.

4.2.5 Inference

As shown in Figure 3.5, the inference process of the DDNN using the optimized offloading rule operates as follows.

First, the input image x is processed by the *Edge Feature Extractor*, creating the latent feature tensors z, Latent variable z is passed to the *Edge Local Classifier*, producing an early prediction $\mathbf{y}_L$.

Then the logits produced during this early prediction are forwarded to the optimized offloading mechanism $\mathcal{O}$, together with the latent representation z.

The offloading mechanism takes as input only the local logits $\mathbf{y}_L$ and outputs a single probability value $p(y) \in (0, 1)$, which determines the exit of the DDNN. If $p(y) < 0.5$, the local prediction is accepted and the sample follows the green path in Figure 3.5, producing the final output $\mathbf{y}_L$. Otherwise, if $p(y) \geq 0.5$, the latent vector z follows the red path: and gets offloaded to the cloud network, which performs deeper processing through the *Remote Exit* to produce $\mathbf{y}_R$.

It is worth noting that the optimized rule requires only a single forward pass through the edge network to make its decision. When an early exit is selected (i.e., $p(y) < 0.5$), the cloud network is bypassed entirely, resulting in reduced computation time, and improved efficiency at inference.

4.3 Case study: UAV Offloading scenario

4.3.1 UAV Scenario Review using DDNN

Let us go back to hypothetical UAV surveillance example introduced earlier (Figure 1.5) after getting familiar with the proposed models to review how the DDNN and the optimized offloading mechanism apply in a realistic Edge-Cloud setting. In this scenario the *edge* device is the autonomous drone itself: it carries a camera that continuously captures the forward area and runs a lightweight local network that makes an early prediction whether the zone is *clear* or *dangerous* by classifying visible objectives (enemy troops, vehicles, other drones). The *remote* network is located at the headquarters several kilometers away owing a deeper more powerful network, able to detect and classify more complex and "difficult" scenarios providing with a safer but more time costly response.

A small, offloading mechanism deployed on the drone consumes the local classifier logits and decides, per frame, whether the local prediction is sufficiently reliable or whether the latent representation should be forwarded to the barracks for deeper processing. Real-life constraints such as jamming by adversarial devices, increased up-link cost and energy consumption due to long distance transmission required for offloading, or even running the risk of the radio channel to intercepted leaking valuable information often require rapid on-board decisions to avoid exposing the drone to danger. For these reasons, only a fraction of the captured frames can be safely and effectively handled by the remote site in real time requiring that at least a certain percentage of samples L_0 classified locally in order to promote faster decisions.

4.3.2 Example latency–accuracy benefit

We adopt the indicative per–frame latency costs presented on Figure 1.1: edge time T_{edge}=10 ms and increased cloud time T_{cloud}=80 ms. If a fraction L_0 of frames is kept locally and the rest is offloaded, the average per–area latency for a given N total captures is approximated by

$$\bar{T}_A(L_0, N) \;\approx\; L_0\, T_{\text{edge}} \;+\; (1 - L_0)\,(T_{\text{edge}}{+}T_{\text{cloud}}).$$

Symbol	Description
$\bar{T}_A(L_0, N)$	Average per-area latency as a function of L_0, N.
L_0	Fraction of frames kept locally (local exit rate).
T_{edge}	Per-frame latency for local (edge) inference.
T_{cloud}	Additional per-frame latency for cloud offload.
N	Number of frames captured for area A.

Table 4.3: Variables used in the average latency approximation.

Example: with L_0=0.5 , we obtain $\bar{T}_A(0.5) \approx 0.5 \cdot 10 + 0.5 \cdot (10{+}80) = 5 + 45 = 50$ ms, whereas the remote–only policy ($L_0 = 0$) would need $10 + 80 = 90$ ms to produce a response.

For accuracy, using the indicative rates $\text{Acc}_L{\approx}85\%$ (edge) and $\text{Acc}_R{\approx}95\%$ (cloud), a conservative mixed estimate is

$$\text{Acc}_{\text{mix}}(L_0) \;\approx\; L_0\,\text{Acc}_L \;+\; (1 - L_0)\,\text{Acc}_R.$$

Thus for L_0=0.5 we get $\text{Acc}_{\text{mix}}(0.5) \approx 0.5 \cdot 85\% + 0.5 \cdot 95\% = 90\%$.

Concluding. We present this simple approximation to highlight the practical impact of the DDNN approach instead of using a simple remote exit. A mixed edge-remote classification

strategy achieves a 50ms average response time, a significant improvement over the 90ms cloud-only latency, while this theoretical calculation results in a 5% accuracy reduction (from 95% to 90%), an acceptable trade-off for such a large latency gain.

More importantly, this Acc_{mix} is a conservative baseline calculated using classification accuracy values based on independent edge-remote performances. Joint training process (see Equation (7)) used in DDNNs specializes the local classifier on simpler inputs and the cloud classifier on more complex ones. As shown in our results (Chapter 6) and supported by prior DDNN research [2, 16], applied joint training and specialization acts a source of regularization for the remote exit often increasing the accuracy of both individual exits compared to models trained in isolation. Our **Optimized Rule** is designed to substantially improve this result by effectively separating "easy" from "hard" samples to offload, achieving even higher classification accuracy $\geq \text{Acc}_{\text{mix}}$.

Therefore, while the UAV scenario is illustrative, the principle is clear: Using the Optimized Rule as the offloading mechanism for DDNN not only provides significant latency reduction but also improves the accuracy trade-off, outperforming standard heuristics like entropy by 1-2% as shown on (Chapter 6), a critical margin in classification tasks.

Chapter 5

Implementation Details

5.1 Issues overview

Dataset Notation
Unless otherwise stated, all experiments and plots in this section are reported on CIFAR-10 Dataset.

In this chapter, we will discuss the key experimental decisions that led to our final model structure. Before arriving at the proposed solution, we investigated several alternative approaches for the offloading mechanism design, trying to surpass the accuracy of baseline entropy method leading to a successful one. We have to mention these implementation alternatives since they are crucial in understanding the key difficulties raised when training such a network for offloading decisions.

Specifically our main issues involve experimenting with the training data given as input, their form (e.g., raw features vs. logits), directly influencing the offloading mechanism's architecture and the strategy for creating and labelling the meta-dataset used to train the Optimized Rule.

We defer the discussion of fine-grained hyperparameter tuning (e.g., specific dropout rates or learning rates) to chapter 6. Such tuning is discussed only after the final model's structure has been established.

5.2 Experimenting with Local Feature Maps as Input

5.2.1 Initial Shallow Optimized Rule Setting

Before arriving at the final architecture described in Section 4.2.1, we experimented with several input formats for the optimized offloading mechanism $\mathcal{O}$.

Our initial approach was adopted from the reference work on distributed DNNs for resource scaling [16], which successfully utilized the latent feature maps as the primary input for its learned offloading rule using only a simple Multi-Layer Perceptron (MLP) while mentioning that they also tried different input forms such as raw features or logits y_L but the most successful one was the latent vector approach. A high-dimensional latent vector z (the output of the *Local Feature Extractor*) would contain richer, more comprehensive information about the sample compared to the final condensed decision produced at the edge, while also carrying already processed feature maps, reducing computational time.

Despite the rationale behind it, our experiments showed that this feature-based implementation using a simple Multi-Layer Perceptron (MLP) failed to generalize. As documented in fig. 6.13, the performance of this CNN-based offloading mechanism was extremely poor. It failed to learn the underlining the need cost-benefit patterns and performed similarly to a **Random** offloading Rule, falling short to the heuristic entropy rule.

5.2.2 Deeper Offloading Mechanism Architecture

This failure, necessitated adopting significantly deeper and more complex architecture for the offloading mechanism itself. Instead of a simple Multi-Layer Perceptron, we adopted some convolution (CNN) layers to process the spatial dimensions of the feature maps, followed by several fully-connected layers. The CNN-based approach did provide better results, successfully moving beyond the random baseline performance closing the bag between with the entropy rule but failed to match it as shown in fig. 6.14.

5.2.3 Challenging Overfitting

However, this more complex model introduced significant **overfitting** issues. As seen in fig. 6.15 Train vs Test data accuracy differ significantly, and further attempts to optimize the architecture by increasing/decreasing neurons/layers and tune regularization techniques such us dropout and data augmentation did not yield any meaningful progress.

These results implied that it wasn't about the models performance, since it had the capacity to learn leading us to search other potential sources such as the complexity of incoming data and difficulty of the learning task.

5.2.4 Raw Images as Input

A similar attempt using the raw input images directly, objecting in providing a more detailed input than the already processed features—which also required a deep CNN architecture yielded nearly identical poor performance as shown on fig. 6.16. It's clear that in both cases either using Local Features or using the raw sample image as input the Deeper CNN Ofload mechanism fails to capture more efficient patterns performing lower than the baseline entropy rule. Moreover, analysis on the MetaDataset Labelling showed that for a significant number of samples (3.6%) fig. 5.1 the task of labelling a sample for offloading is unclear and harder than expected with a cost benefit difference of $|B_x| < 0.01$ further underlining the difficulty of learning from an input image.

5.2.5 Final Logits Based Input

Our mechanism was required to learn a complex mapping from high-dimensional vector (either raw pixels or feature maps) to a single, binary classification label ($o_x \in \{0, 1\}$) derived from the B_x classification cost.

This task appears more challenging than the objective presented on paper [16]. Although Giannakas et al work also trained a learned rule, its input was not such high-dimensional, consisted of time-series data used to predict resource allocation values, a simpler regression task.

Based on this empirical evidence, we revised our input form approach and selected an alternative "simpler" format. We moved from the high-dimensional, complex inputs to the most abstract and low-dimensional representation available at the edge: the 10-dimensional **logits vector** ($\mathbf{y}_L$) produced by the *Local Classifier*. This low-dimensional input proved to be more effective and led to our finalized architecture detailed in Section 4.2.1.

5.3 Meta-Dataset Labelling Strategy

The performance of the optimized offloading mechanism $\mathcal{O}$ is highly dependent on the quality of its training meta-dataset. In order for the model to learn the correct patterns a consistent dataset is needed without noisy or misleading signals.

We experimented with two distinct labelling strategies for this meta-dataset:

1. **Simple Cost-Based Rule:** The initial approach described in section 4.2.2, where a cost benefit value B_x is used to label samples for offloading (label=1).This cost-based labelling rule is also extensible, a real-world deployment could easily incorporate additional costs, such as uplink transmission energy, into the B_x calculation.

2. **Oracle Labelling Rule:** The theoretically optimal approach, detailed in section 6.1.2 labels a sample for offload when the cloud classifier is correct and the local one is wrong. Conversely, it assigns a "local" label when the cloud classifier is wrong and the local one correct. It uses the cost aware B_x only in cases of a tie. This strategy is more straightforward prioritizing classification *correctness* first picking the optimal choice every time, thereby mimicking the behavior of a true oracle rule. Although this approach is less adaptive and not as cost-aware.

Firstly since the model's goal is to perform optimally we have to make sure our data-labelling aligns with the oracle-labelling allowing our model to learn similar patterns. The above figure fig. 5.1, generated from a single test run (which is representative of the typical metrics observed, with minimal change but not an average, for clarity), analyzes the consistency of our labelling strategies to ensure the meta-dataset is not "noisy".

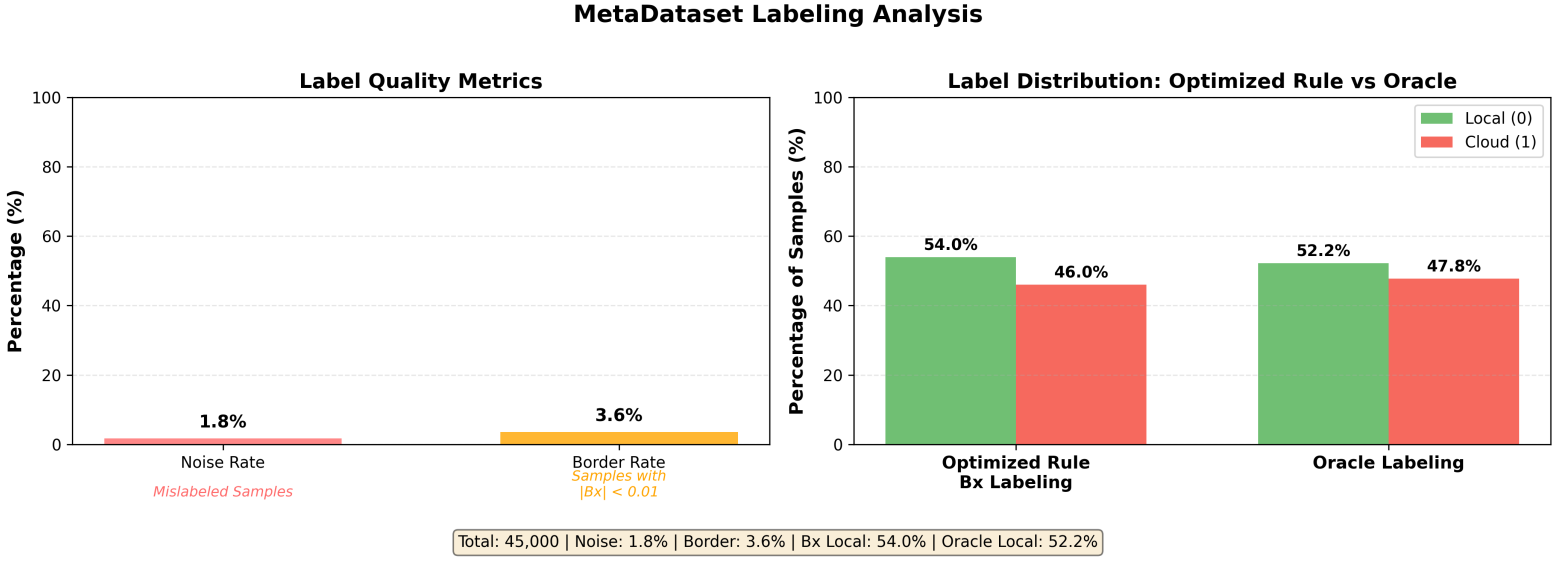


Figure 5.1: Analysis of "Label Noise": the percentage of samples where the simple B_x rule disagrees with the true oracle decision.

The "Noise rate" metric quantifies the percentage of samples where the simple B_x-Cost rule (Strategy 1) **disagrees with** the correctness-based Oracle Labelling Rule (Strategy 2). A low "Noise rate" (e.g., 1.8% as shown in the figure) is acceptable and confirms that our chosen Oracle Labelling Rule (see Section 6.1.2) is well structured providing a clean signal for the mechanism to learn.

Furthermore, we measure the "Border rate" (3.6%) which represents a critical subset of "borderline" samples where the benefit of offloading is not clear ($|B_x| < 0.01$). These samples are inherently difficult to learn as there is no clear pattern. A tiny, insignificant cost difference might favor the cloud, causing an unnecessary offload even if the local prediction was equally correct, thereby wasting transmission energy. It might be beneficial to further analyze and experiment with these samples in order to use them for the networks benefit instead of acting as noise.

Finally, the "Local Rate" (52.4%) shows the oracle's resulting decision distribution, indicating a balanced split.

We further validate our cost-based labelling with a clear plot comparing both data labelling methods (Original cost-based vs Oracle) for creating the meta-dataset and training the offloading mechanism. Figure 5.2 compares the two performances in a single indicative test illustrating similar results with the Oracle Labelling rule confirming that our "theoretically" better approach holds achieving similar performance value with minimal accuracy difference as penalty for following our Optimized-Rule approach. Overall DDNN Accuracy presents the performance of DDNN using the Optimized Rule offloading mechanism trained with each data-labelling respectfully, while Offload validation accuracy represents the percentage of correct decisions the Optimized Rule made.

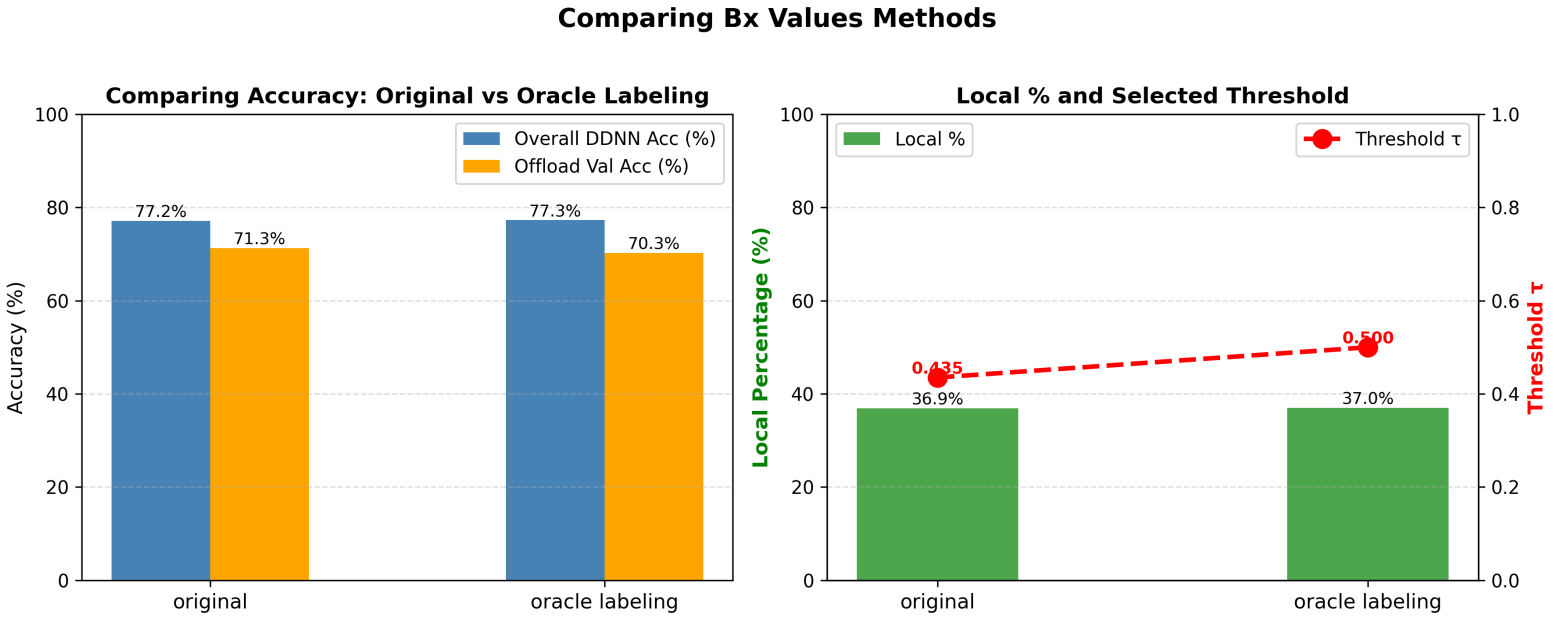


Figure 5.2: Performance comparison of the Optimized Rule trained with the simple b_k-only rule versus the Oracle Labelling Rule.

Chapter 6

Results

Overview This chapter presents the empirical evaluation of our proposed DDNN and its optimized offloading mechanism. We first define the baseline models used for comparison and the key performance metrics. We then present our findings, starting with the standalone performance of the DDNN components and moving to the end-to-end system results.

Dataset Notation
Unless otherwise stated, all experiments and plots in this section are reported on CIFAR-10 Dataset.

6.1 Preliminaries and Baselines

To evaluate our proposed *Optimized Rule*, we compare it against several standard baselines and heuristic methods. All models (unless specified) use the same jointly-trained DDNN weights to ensure a fair comparison.

6.1.1 Baseline and Comparison Models

- **Cloud-Only(Remote):** A centralized network where all samples ($L_0 = 0\%$) are processed by the remote (cloud) exit, $\mathbf{y}_R$ with a given cost termed as C_R. It represents the maximum achievable accuracy of the DDNN architecture, since the more powerful model is used always with greater learning capacity.

- **Edge-Only (Local):** A DDNN keeping all samples locally ($L_0 = 100\%$) and processed *only* by the shallow (local) exit, $\mathbf{y}_L$. Its cost is termed as C_L, it represents the fastest possible inference (lowest latency) but is expected to achieve less classification accuracy in more complex samples.

- **Random:** A DDNN whose decision to keep a sample locally is an i.i.d. Bernoulli random variable with success probability L_0 . For a given number of N samples it is expected to keep locally an average local rate L_0. The joint average cost per sample will range between the local cost value C_L and the remote one C_R according to the local rate L_0 tasked.

- **Oracle:** An ideal router that knows beforehand the classification costs for both networks and makes the optimal decision every time. It uses the *Oracle Labelling Rule* in order to offload "hard" samples discussed in (Section 6.1.2). It serves as the theoretical upper bound for overall DDNN classification accuracy and can even exceed the Cloud-Only baseline in sample cases where the local model predicts correctly in contrast to a cloud misclassification (i.e., $C_{L,x} < C_{R,x} \iff B_x < 0$).

- **DDNN (Entropy Rule):** The primary heuristic baseline. Typical DDNN architecture using normalized entropy(defined in Section 1.2.3) as confidence mechanism to decide between offloading samples and keeping them locally. We aim to outperform this baseline and move closer to the oracle's performance.
- **DDNN (Optimized Rule):** Our proposed solution. This is the jointly-trained DDNN guided by the lightweight, learned offloading mechanism $\mathcal{O}$ described in detail in Section 4.2.1.

Table 6.1: Summary of Baseline Models and Offloading Decision Strategies.

Baseline Model	**Offloading Decision Mechanism**
Cloud-Only (Remote)	Always offload (Local Rate $L_0 = 0\%$)
Edge-Only (Local)	Never offload (Local Rate $L_0 = 100\%$)
Random	Offload with probability: $p_x = 1 - L_0$
Oracle	Offload when: $o_{\text{oracle}}(x) = 1$ (Correctness-first)
DDNN (Entropy Rule)	Offload when: $H(\mathbf{p}_y(L)) > \theta_{\text{entropy}}$
DDNN (Optimized Rule)	Offload when: $\mathcal{O}(x) = 1$ (Cost-aware)

6.1.2 Oracle Decision Rule

As described earlier, we define an optimal router using the oracle decision rule. It has a clear advantage, knowing all classification costs and outcomes in advance and always selects the classifier (local or cloud) that yields the correct result. Specifically, the oracle follows this rule:

- If only the cloud predicts correctly, the sample is offloaded to the cloud.
- If only the local predicts correctly, the sample is kept local.
- In tiebreak scenarios where both classifiers predict correctly/incorrectly, the decision is based on which network assigns a higher probability to the correct class (i.e., has the lower cost) using the already known $B_x \geq B^*$ equation.

Let $\mathbf{y}_L$ and $\mathbf{y}_R$ be the output vectors from the local and cloud classifiers, respectively. In this case $\hat{y}_L = \arg\max(\mathbf{y}_L)$ and $\hat{y}_R = \arg\max(\mathbf{y}_R)$ are their predicted labels, and y the true ground-truth label for sample x. Cost-benefit terms B_x and B^* are defined in Section 4.2.2.

The oracle decision $o_{\text{oracle}}(x)$ (where $1 = \text{Local}, 0 = \text{Cloud}$) is defined as follows:

$$o_{\text{oracle}}(x) = \begin{cases} 1 & \text{(Clear Cloud Win)}, \quad \text{if } (\hat{y}_L \neq y) \wedge (\hat{y}_R = y) \\ 0 & \text{(Clear Local Win)}, \quad \text{if } (\hat{y}_L = y) \wedge (\hat{y}_R \neq y) \\ 1 & \text{Tie-break Cloud Win}, \quad \text{if } (\hat{y}_L = y \wedge \hat{y}_R = y) \vee (\hat{y}_L \neq y \wedge \hat{y}_R \neq y) \wedge (B_x \leq B^*) \\ 0 & \text{Tie-break Local Win}, \quad \text{if } (\hat{y}_L = y \wedge \hat{y}_R = y) \vee (\hat{y}_L \neq y \wedge \hat{y}_R \neq y) \wedge (B_x > B^*) \end{cases}$$

6.1.3 Performance Metrics

In our simulations we will focus on two metrics in order to evaluate the performance of each model:

Overall Classification Accuracy (%). This is our main metric for correct prediction. It measures the final, end-to-end accuracy of the overall DDNN. For a given test set, we count how many samples were correctly classified, regardless of whether the decision was made by the local exit ($\mathbf{y}_L$) or the remote one ($\mathbf{y}_R$), as determined by the active offloading mechanism.

Local Classification (%). This is our main metric for efficiency. It represents the fraction of total samples that were chosen from the offloading mechanism to be processed locally. A higher local percentage implies significant savings in latency and energy since there is no offloading to the cloud. However as more "hard" samples are forced to be resolved locally the more incorrectly will be and beyond a certain point, this sacrifice in accuracy becomes significant.

We define a successful offloading mechanism the on that achieves both a high Overall Classification Accuracy while also keeping a significant number of samples to be resolved locally. Therefore, our result plots will almost always show Overall Accuracy (y-axis) as a function of the Local Classification percentage (x-axis) to visualize this trade-off.
Note: For more detailed analysis, we will also evaluate the *offloading mechanism* itself by measuring if its binary predictions (local-cloud) match the "ground truth" labels of the meta-dataset.

6.1.4 Baseline Performance

The rationale behind using a Distributed Deep Neural Network instead of a centralized DNN is to gain meaningful performance upgrade by offloading samples to the cloud. In order to validate this result we present the standalone overall DDNN accuracy of the Edge-only ($L_0 = 100\%$) and Cloud-Only ($L_0 = 0\%$) baselines after joint training and testing them.

As shown in Figure 6.1, the jointly-trained local classifier achieves a significantly lower accuracy than the cloud one. This performance gap (e.g., ~10-15% in our CIFAR-10 experiments) is essential for the DDNN architecture as it confirms that the cloud provides a substantial performance benefit and justifies an offloading decision. It is important to note that these are the standalone performances *after* joint training. As mentioned before the joint process usually improves the accuracy of the standalone networks, implying that our DDNN architecture would outperform a solo-trained local and cloud network too.

It is also important to mention that a DDNN using an optimal router (the **Oracle** baseline) can achieve a higher accuracy than both the standalone local model (which is expected) and the standalone cloud-only model. This phenomenon occurs because there are samples where the local network is correct, but the cloud network is wrong (i.e., $C_{L,x} < C_{R,x}$). The Oracle knows this and keeps these samples local, boosting its overall score. As we demonstrated in our meta-dataset analysis (Section 5.3), the Oracle labelling rule, which prioritizes correctness, keeps a large fraction of samples locally (e.g., 52.4% in Figure 5.1). This proves that the local network provides a unique classification view often better in simpler samples, not just a "weaker" version of the cloud, making the DDNN's mixed strategy theoretically superior to a cloud-only approach as shown in Figure 6.1.

6.2 Optimized Rule Performance

The final model takes as input a tensor with 10 logits produced from the local classifier, a clear low-dimension input that can be mapped and analyzed with only a few Fully-Connected Layers, thus leading to compact small MLP network. This approach is expected to at least catch or even surpass the entropy's baseline performance since it utilizes the same data as the entropy confidence metric does.

As shown in Figure 6.2 the classification accuracy of the blue line representing the Optimized Rule surpasses the orange one labeled as the entropy baseline along the entire x-axis . In other

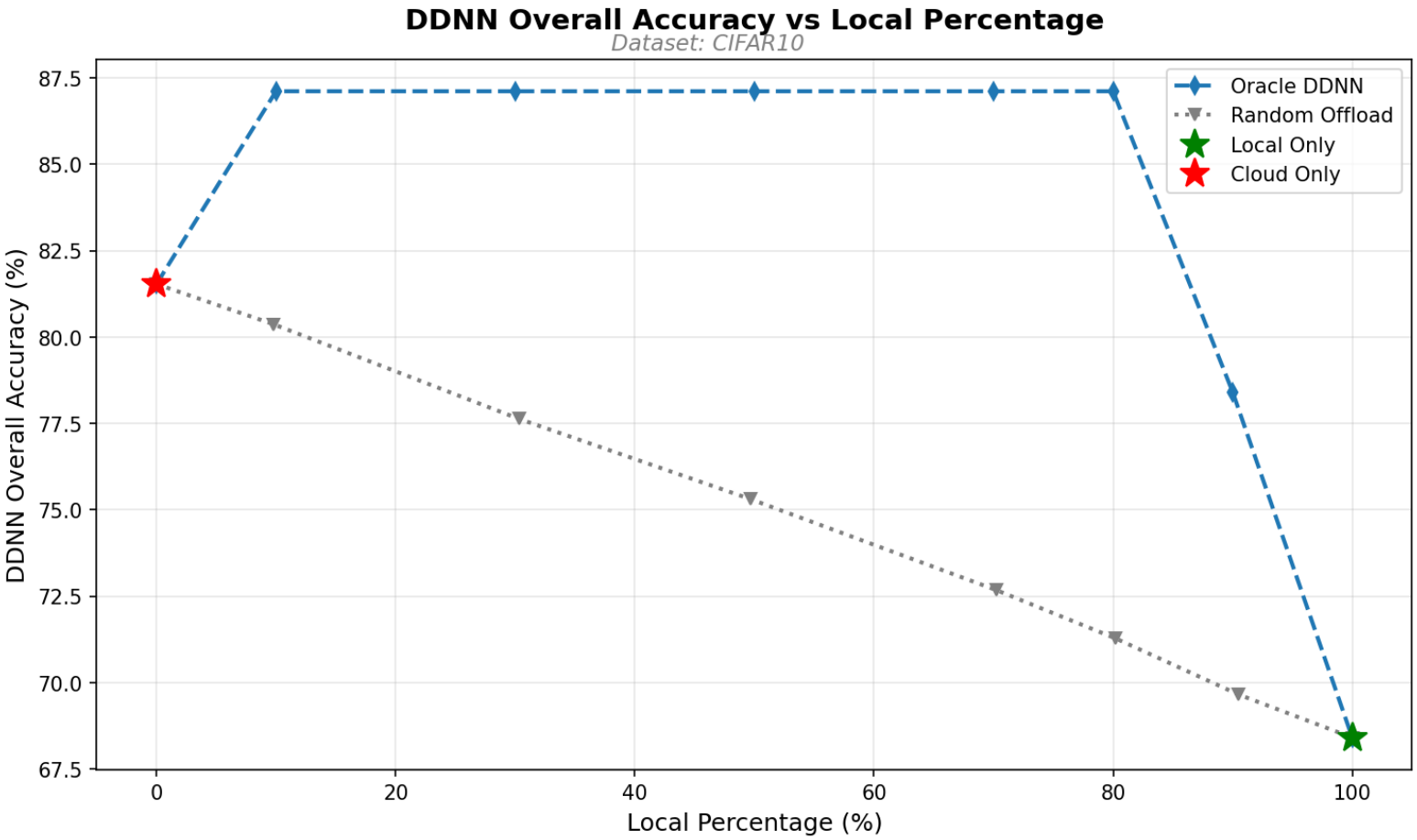


Figure 6.1: Standalone classification accuracy of the local (Edge-Only) vs. remote (Cloud-Only) exits.

words when using the proposed Optimized Rule as an offloading mechanism for the DDNN we are outperforming (across the entire range of local percentages) the already used heuristics proposed in previous DDNN works on classification settings. The difference might not seem large, but taken into consideration the simplicity of our models we can anticipate much more benefit on a real-world large scale DDNN even in these performance gaps of 1-2 % . We might have achieved our primary objective of this research but its clear that although the gap between performances of our offloading mechanism and the oracle ones is reduced there is still much for improvement that will be discussed in the later sections.

The logits-based optimized rule has several key advantages:

- It is extremely **compact and fast**. Since it is a simple MLP processing only 10 inputs, it has a very small parameter count and is significantly faster to execute on an edge device than an equivalent mechanism based on high-dimensional features or raw images.
- As shown in Figure 6.2, its classification accuracy successfully **surpasses the standard entropy baseline** across the entire range of local percentages. This was a primary objective of this research: to develop a learned offloading system that outperforms the common heuristics.
- Our rule successfully **narrows the performance gap** to the theoretical Oracle, demonstrating that a learned approach is superior to a fixed heuristic.

While this is a strong result, it is also somewhat expected. The entropy rule is a fixed heuristic (a single calculation) based on the 10-logit vector. Our approach essentially trains a

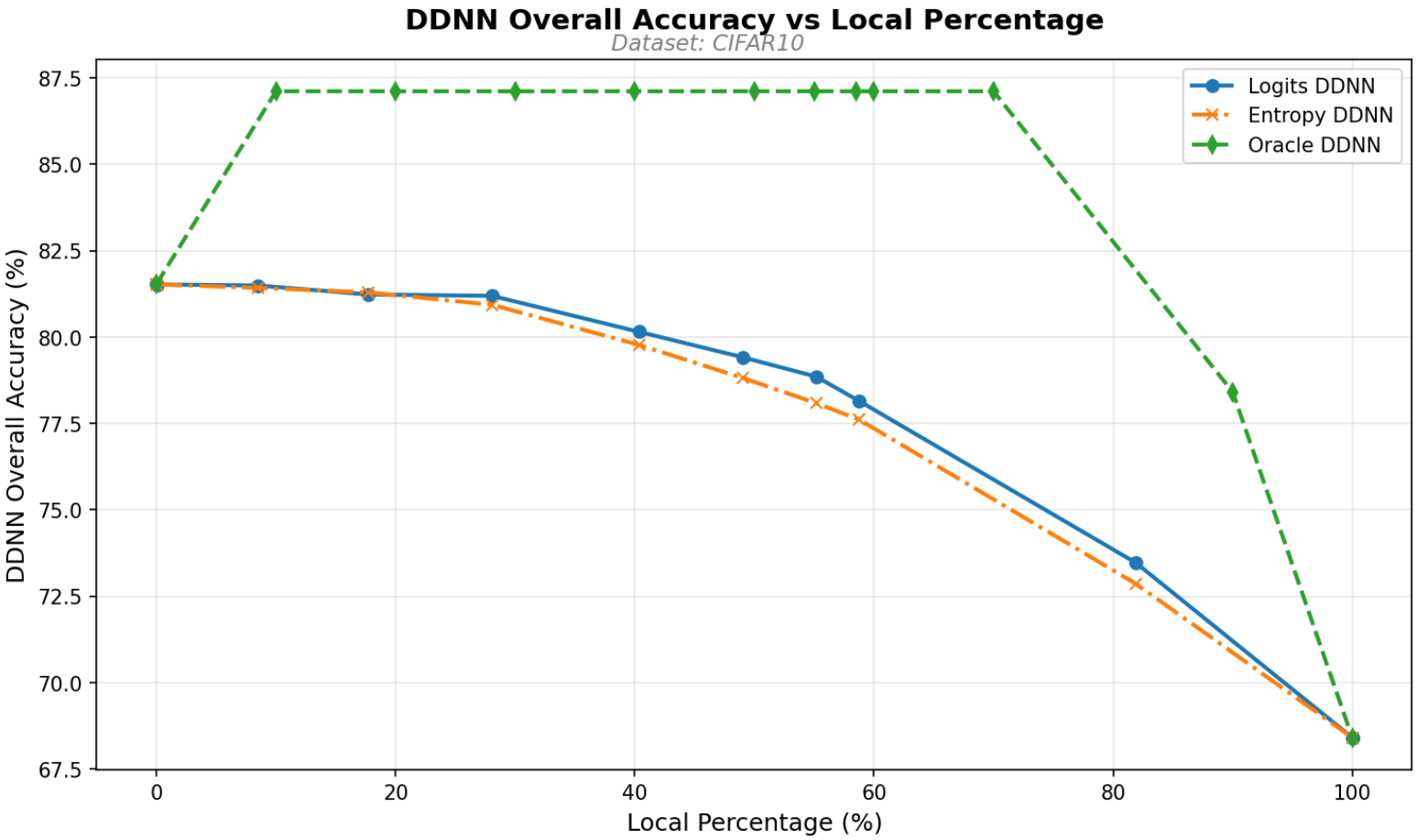


Figure 6.2: Performance of the Optimized Rule (logits-based) compared to the standard Entropy heuristic and the theoretical Oracle baseline.

small neural network to find the *optimal* possible mapping from that same 10-logit vector to the correct offload decision. This provides a simple and effective foundation with room for future improvement, which we discuss in the Future Work chapter.

6.3 Optimized Rule Training and Tuning Tests

After finalizing a logit vector as the input format discussed previously we experimented with the internal architecture of the offloading mechanism $\mathcal{O}$ and tried different fine tuning parameters for regularization.

6.3.1 Depth

With a low dimension of only 10 logits (numerical values) , it makes sense to use a shallow MLP Network comprising of 3-5 Fully-Connected layers without any Convolution ones since there are no high-dimensional features to process. We tested several depths presenting their performance on Figure 6.3. The Overall DDNN accuracy difference between alternative depths is small indicating that there is no need for deeper o structures since a 5-layer network seems to be sufficient enough. Testing with 8 layers showed similar performance.

6.3.2 Dropout Regularization

For regularization purposes we applied both Batch Normalization on every layer and Dropout. Again our shallow network would need smaller Dropout values in the range of $p = 0.1$ to $p = 0.2$. We tested several dropout rates as shown in Figure 6.4 to confirm our initial approach and found out that excessive dropout rates can destabilize the network and negatively impact the learning process. For example using a dropout rate of 0.6 resulted in changing the offloading pattern observed with the other values since the offloading mechanism kept only 35 % of samples locally while the other had a much larger number of 43% on average. According to that local percentage drop the accuracy gain is misleading and lowers the performance of the DDNN.

6.3.3 Training Hyperparameters

We present all the hyper parameters used for training the Optimized Rule network summarized on the above table Table 6.2. We also used Data augmentation on the initial DDNN dataset for further regularization. Notice that we do not refer to fine tuning the DDNN, but the offloading mechanism network since our research objective is to optimize the offloading decision.

We haven't incorporated an exhaustive weight decay values testing since we used a *ReduceLROnPlateau* scheduler during training reducing the learning rate if validation accuracy stagnates and implemented early stopping when over fitting occurs returning back to the best validation accuracy saved model.

Table 6.2: Hyperparameters for the Optimized Rule Offloading Mechanism.

Hyperparameter	**Value / Strategy**
Architecture	5-Layer MLP
Input	10-dimensional logits vector
Dropout (p)	0 – 0.2
Initial Learning Rate (α)	0.001
Weight Decay	1e-4
LR Scheduler	ReduceLROnPlateau (on validation accuracy)
Optimization	Early Stopping (based on validation accuracy)

6.4 Inference Time Analysis

In order to evaluate the added latency required to run our Optimized Rule, we measured the inference time of the DDNN using our Optimized Rule as the offloading decision mechanism compared to using Entropy and Random baselines.

Methodology After fully training the DDNN, we completed 5 independent test runs over the entire CIFAR-10 test set (10,000 samples) for each of the three offloading methods while tracking only the forward pass time. Since every test was run on a lapton with an NVDIA 1650 GPU, we included a standard deviation time attributed to normal changes in GPU clock speeds and different load management.

Results According to fig. 6.5 our offloading mechanism DDNN averaged a per sample latency of$3.79ms$ with a minimal of $0.39ms$ compared to the simple entropy baseline. Over the entire test data set of 10.000 samples this amounts a total difference of only 3.9 seconds for a significant

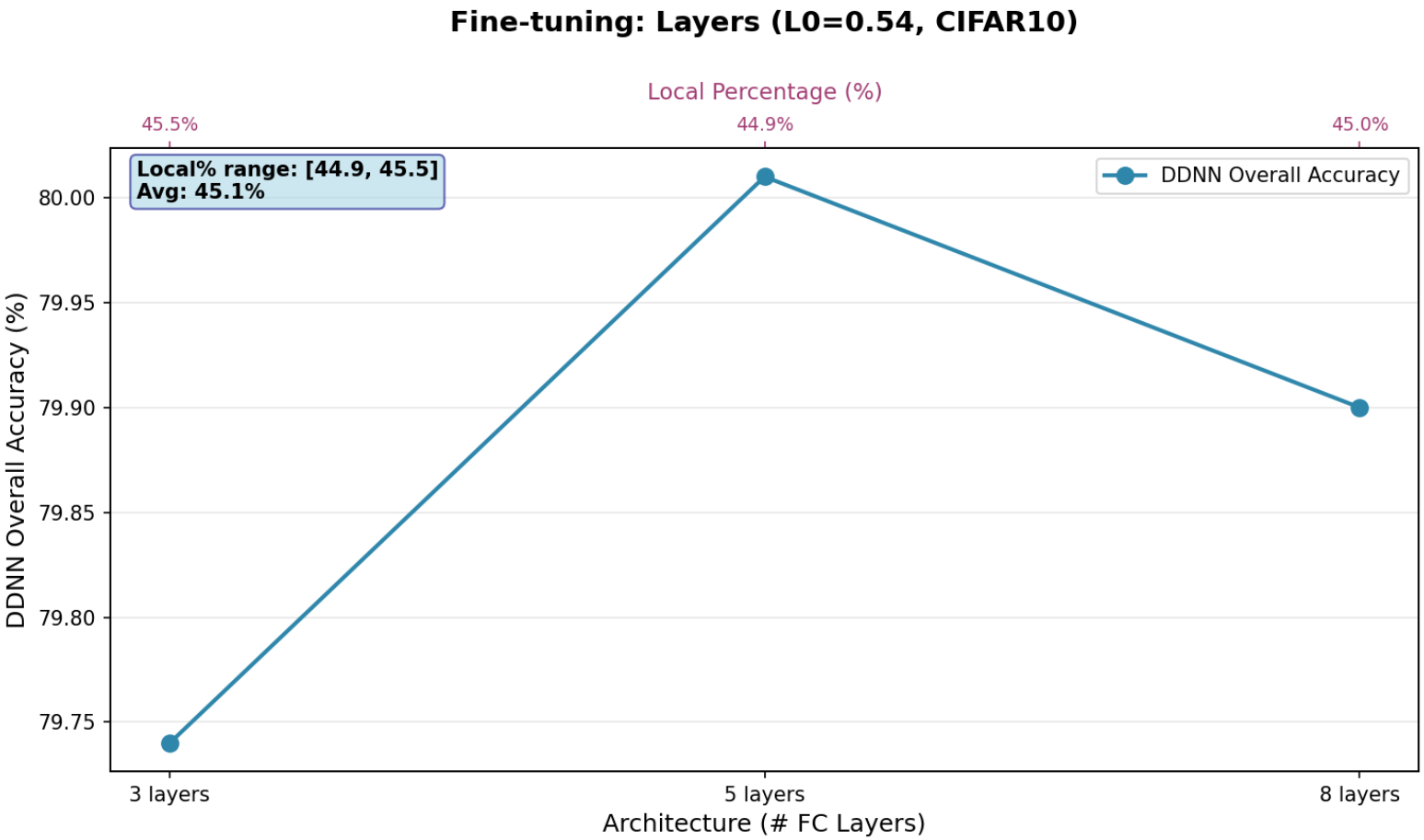


Figure 6.3: Performance of Optimized Rule using different number of layers networks.

classification accuracy upgrade of 0.6%. However, a random offloading decision without any further calculation averages 3.26ms with a total accuracy of 74.9%, opting in using an our offloading mechanism for an upgrade of 4.9% in accuracy adding a latency of 0.53ms per sample is over justified.

In addition to a minimal latency our compact MLP offloading mechanism comes with a limited storage usage and negligible battery consumption too.

6.5 Generalization to Other Datasets

THIS SECTION IS NOT DONE WE WILL ADD MORE TESTS AND ANALYSIS

In this section, we test our Optimized Rule on other popular classification datasets to demonstrate its robustness and consistent performance

First, we test the model on the **CIFAR-100** dataset containing 60,000 (50,000 for training and 10.000 for testing) expecting similar relative results outperforming the entropy, but with a significant overall drop in accuracy. The CIFAR-100 dataset is a much harder task, containing 100 classes instead of 10. The results are presented in Figure 6.6.

We also experiment with an easier in terms of classification difficulty dataset the SVHN. Similar to the MNIST dataset, SVHN contains real world street digit images with 10 classes 1 for each digit adding to a total 73257 digits for training and 26032 digits for testing.

In order to further prove the robustness of our network we tested our network on a more difficult classification task with the Cinic10 dataset. It includes the 60.000 Cifar10 images and a selection of the ImageNet database images (downsampled to 32x32), containing in total 90,000

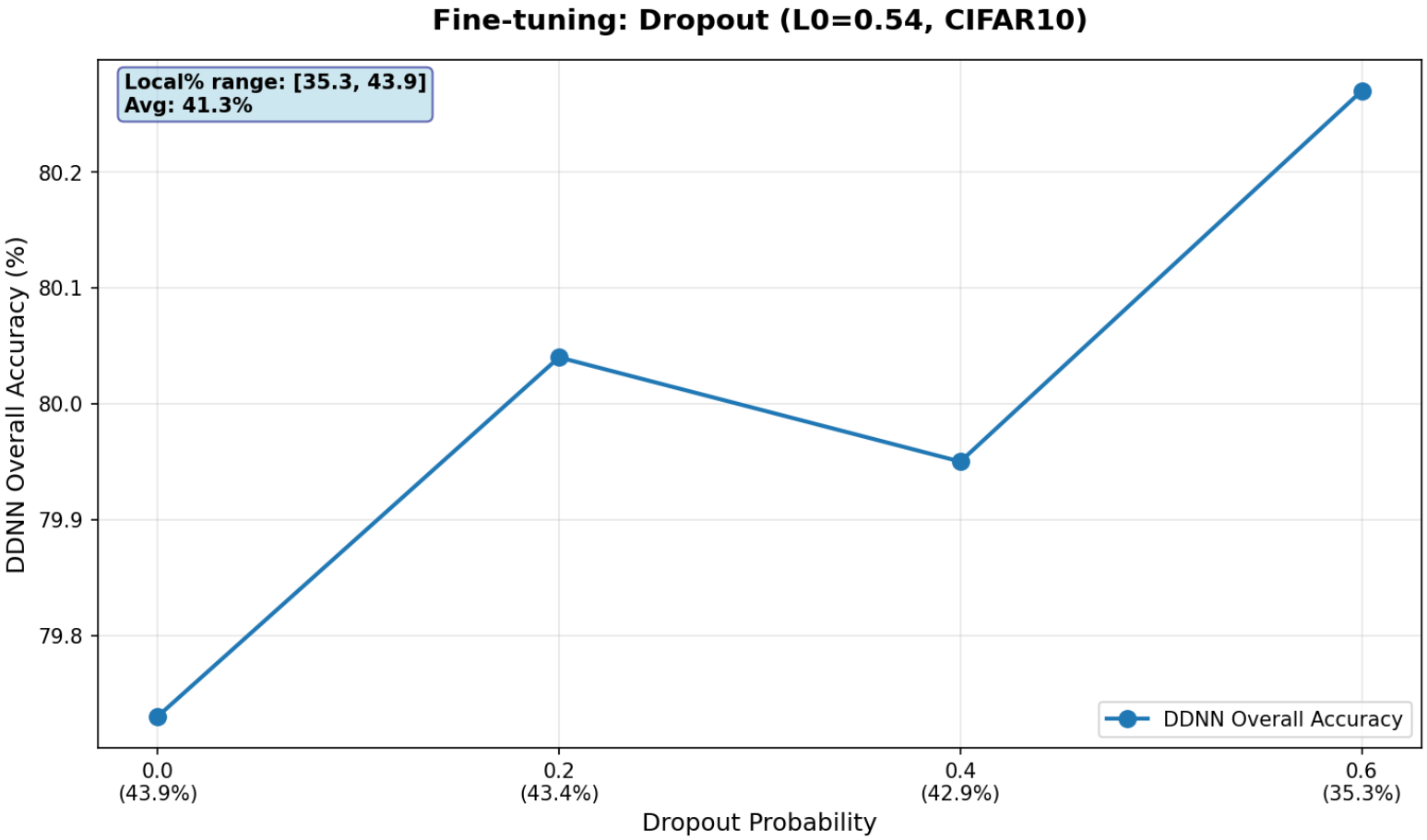


Figure 6.4: Performance of Optimized Rule using different dropout values for training.

images for each subset of train, test, validation. Results on fig. 6.9 showed similar performance between the Optimized Rule and the baseline Entropy used as offloading methods of the DDNN.

6.6 Training Analysis

Further testing of the Optimized Rule requested monitoring the training and validation accuracy of the meta-datasets across epochs in a single training process. of training and validation meta-datasets created across. Presented on Figure 6.10 we can observe that the training and validation accuracy curves remain very close together across the whole training procedure indicating a stable model with no significant overfitting.

Another significant observation is that the training curve does not reach 100 % accuracy even with more epochs. These results suggest that although the model generalizes correctly, the ability to learn more complex patterns and achieve higher accuracy is limited. Since there is sufficient capacity for the model to learn, it's bottleneck may lie in the limited information provided on the input.

Following this hypothesis in order to further close the gap with the Oracle's performance we must enrich the input signal by adding more useful information for the network to capture. This motivated the "Logits+" input format that will be tested in the following sections.

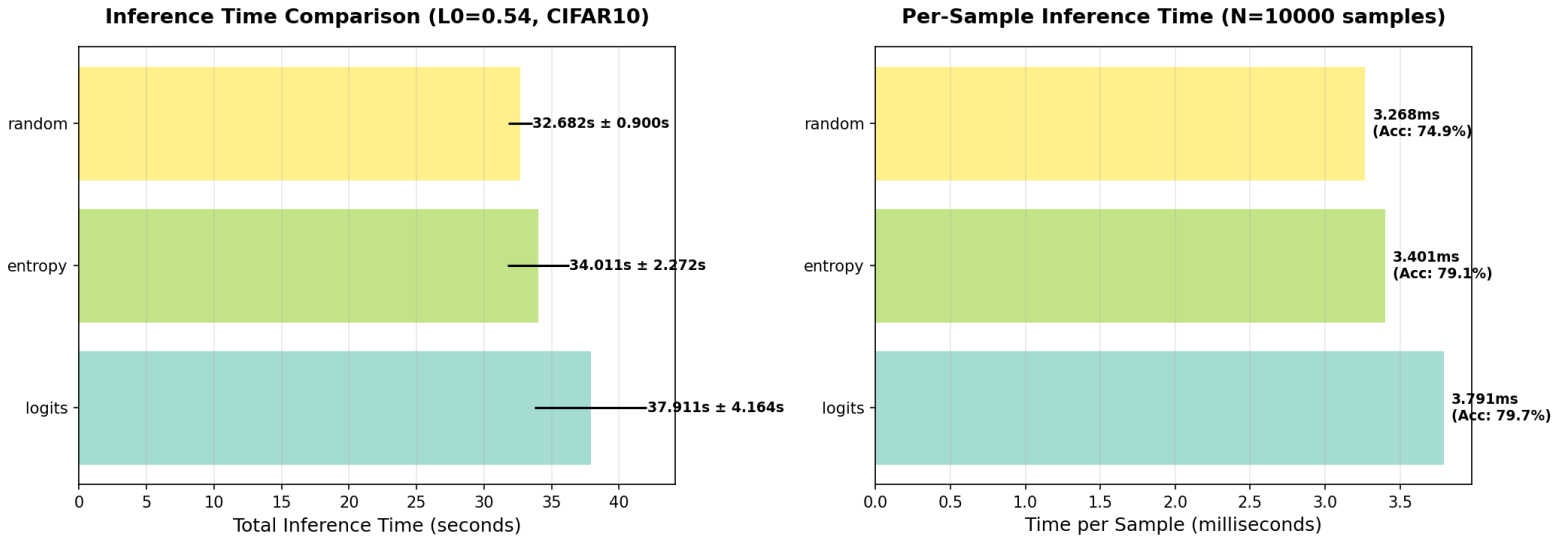


Figure 6.5: Inference time comparison (Total and Per-Sample) for the DDNN using different offload mechanisms at a fixed $L_0 \approx 54\%$. Accuracy values are shown on the right plot.

6.7 Enriching the Input with Logits+

Based on the analysis in Section 6.6, which indicated that our model's performance was limited by the input information , we proceeded to test an enhanced input format called "Logits+".

The rational behind it is to provide more informative signals to the mechanism by adding two new values to the 10-logit vector and detect possible performance improvement: We calculate the above values for each sample and add them to the already produced feature tensor.

1. The normalized entropy of the local prediction.
2. The margin (gap) between the top-2 predicted probabilities.

This results in a 12-dimensional input tensor . We hypothesized that by adding this information, the offloading mechanism could learn more complex patterns and improve its accuracy.

The results in Figure 6.11 showed that The Logits+ input method outperformed all previous approaches alongside with the entropy baseline, further closing the gap to the Oracle's performance, confirming an input-info bottleneck.

6.8 Test Data Analysis

According to Section 5.3, we observed a small percentage of "noisy" samples ($< 2\%$) in the training metadata, where the B_x cost-based labelling disagreed with the strict Oracle Rule. We also noted a percentage of samples associated with borderline decision costs ($B_x < 0.01$) as potential risks for the learning process. In this section, we investigate the magnitude of these sample categories during inference on test data by measuring their DDNN Miss-classification rates.

Figure 6.12 right panel captures the variety of samples representing a whole test data of the DDNN using the Optimized Rule as the offloading mechanism.

- **Noise Rate (25.9%):** The percentage of samples where the Optimized Rule's offloading decision differed from the Oracle's optimal choice.
- **Border Rate (6.1%):** The percentage of "borderline" samples where the benefit of offloading versus keeping local is unclear, defined by a very small cost difference ($|B_x| < \tau$, with $\tau = 0.01$).

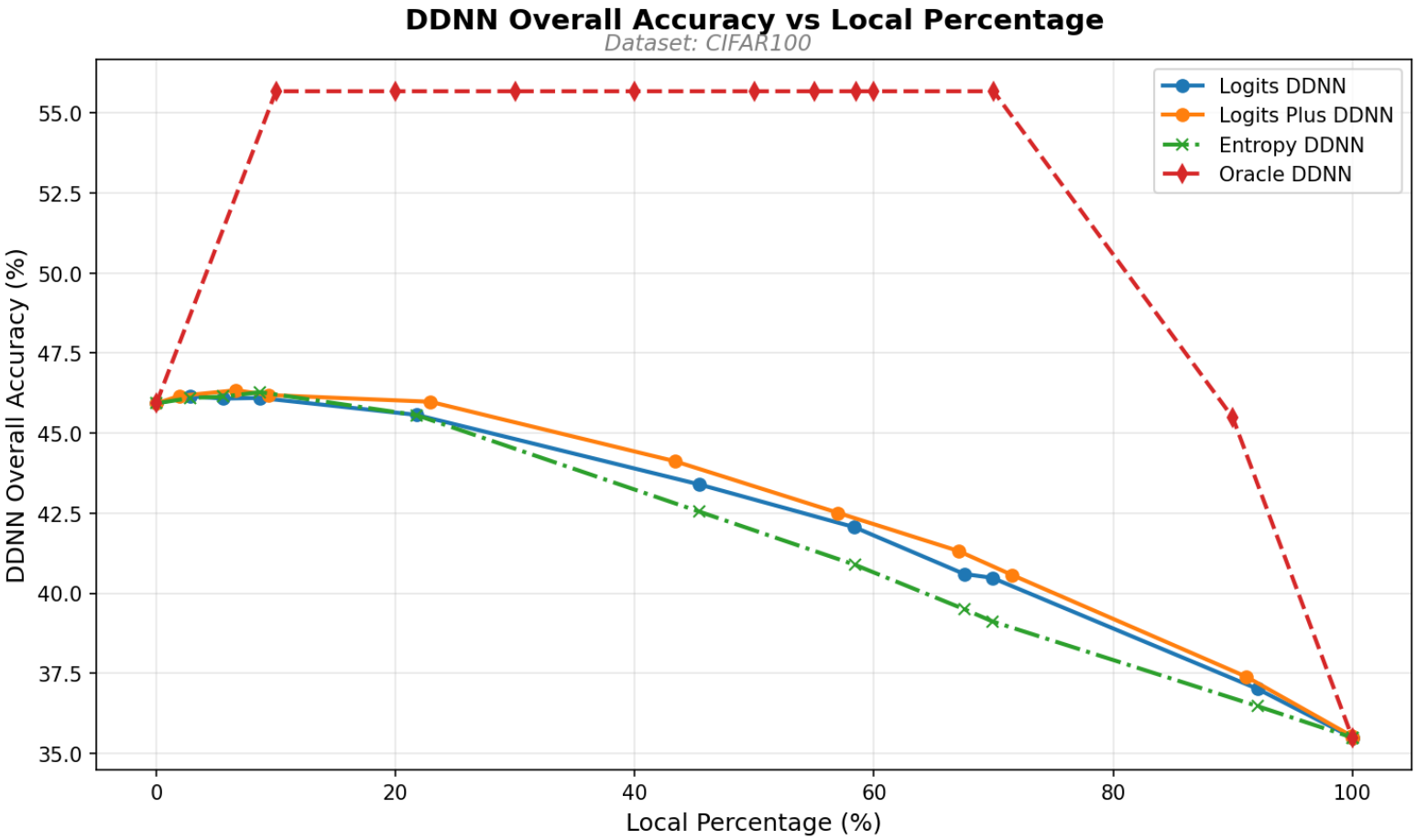


Figure 6.6: Performance of Optimized Rule on the CIFAR-100 dataset.

Figure 6.7: Performance of Optimized Rule on the GTSRB dataset.

- **Normal Samples:** The remaining majority of samples where the decision is clearer.

On the left panel we represent the DDNN **Miss-classification Rate** for each specific sample category captured.

We observe a significant Noise Rate of 25.9% on offloading decision in the test data compared to the metadataset labelling depicted on fig. 5.1, which is expected since the learned offloading mechanism cannot match the optimal Oracle decision making. Its according misclassification rate of 69.1% reveals that the majority of disagreement cases led to an error, while the remaining percentage of 30.9% samples represent "tie" cases where both networks predicted correctly not affecting total accuracy.

Crucial information comes from the 6.1% of borderline samples proving to be highly problematic since 1/5 of them led to a misclassification. This is significantly higher than the misclassification rate of "normal" samples (only4.5%) confirming samples lying near the cost decision boundary (($|B_x| < \tau$) confuse the offloading mechanism and are statistically more likely to induce errors on inference.

Since borderline samples, inherent to the dataset, affect both training (as seen in Section 5.3) and inference time, leading to more misclassification we could filter them from the dataset to further improve classification rates. This practice might applied in future work, although its time costly and not easily transferable to real world scenarios wit unseen data.

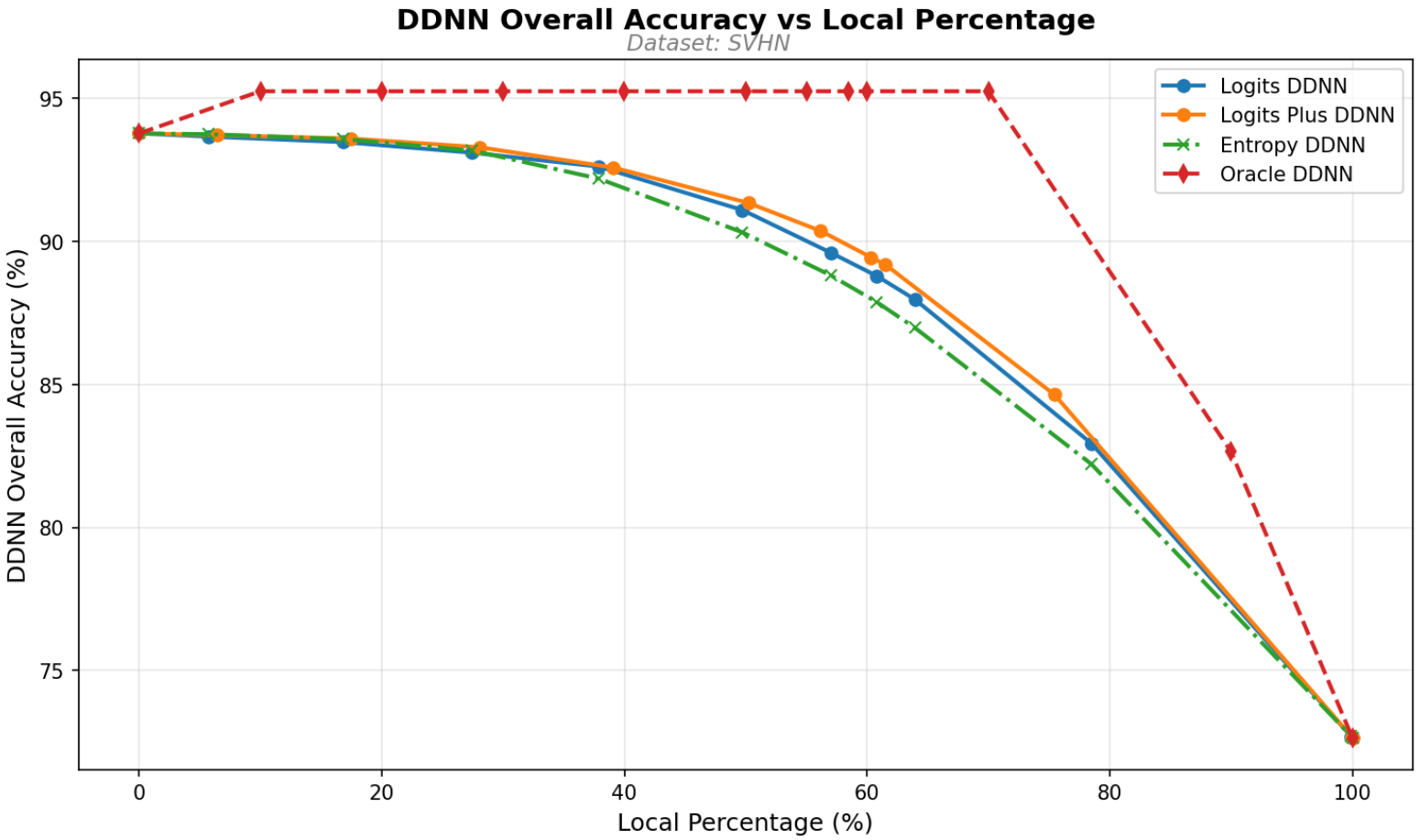


Figure 6.8: Performance of Optimized Rule on the SVHN dataset.

6.9 Mentioned Plots

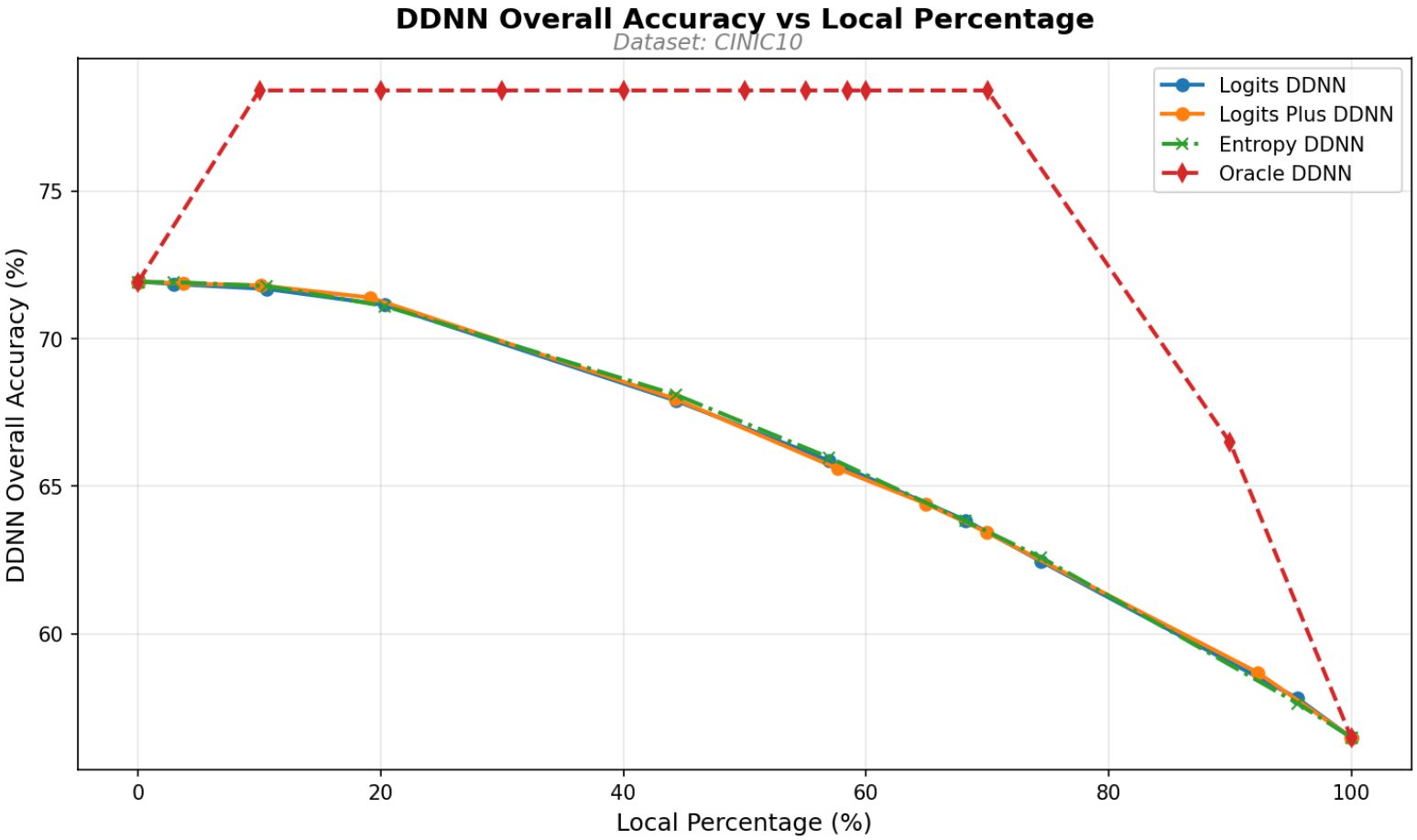


Figure 6.9: Performance of Optimized Rule on the Cinic10 dataset.

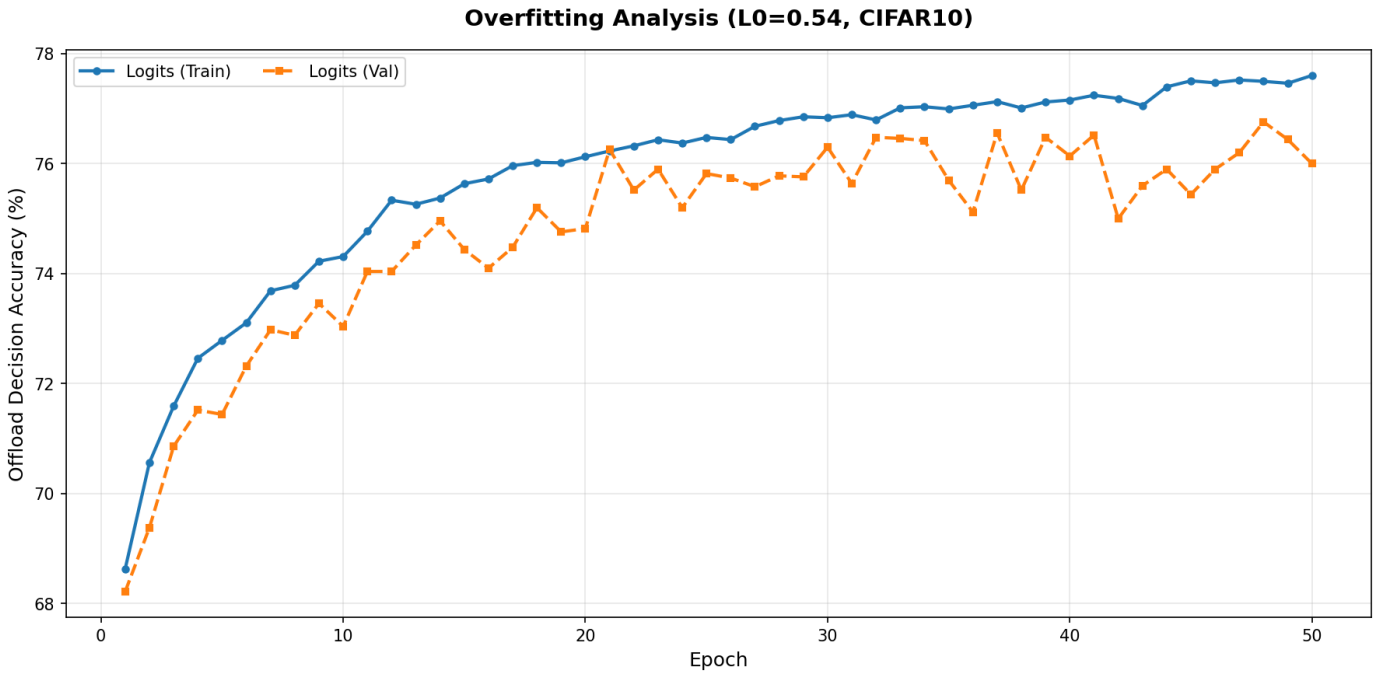


Figure 6.10: Training vs Validation accuracy of the mata-datasets for the Optimized Rule.

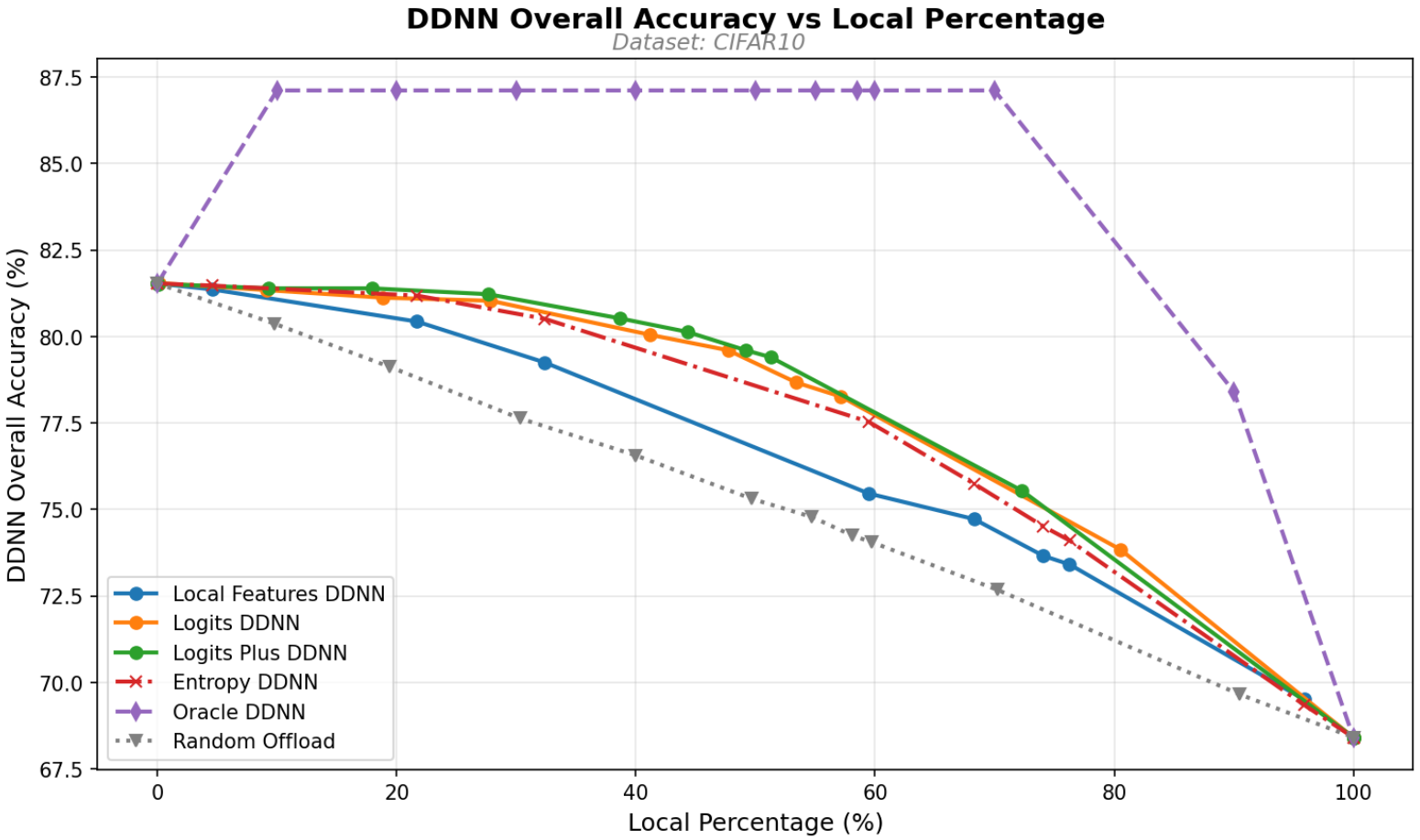


Figure 6.11: Performance of Optimized Rule using a logits+ input.

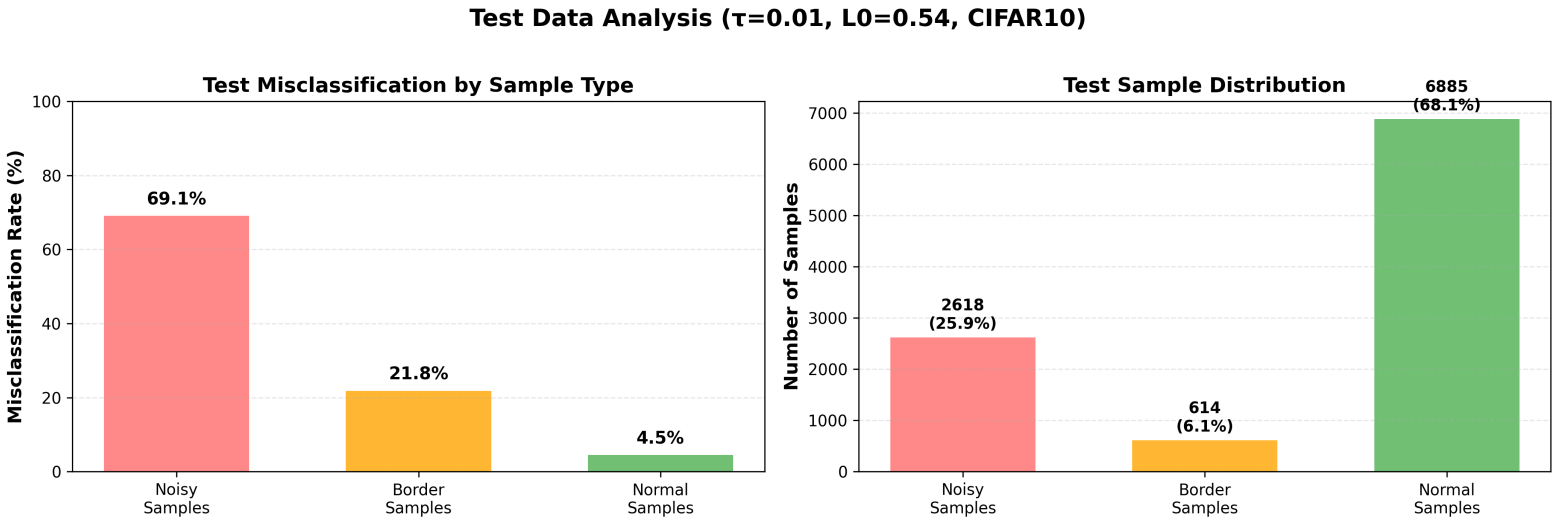


Figure 6.12: Test Data Analysis comprising of the test data categories distribution paired with their respective impact on the results.

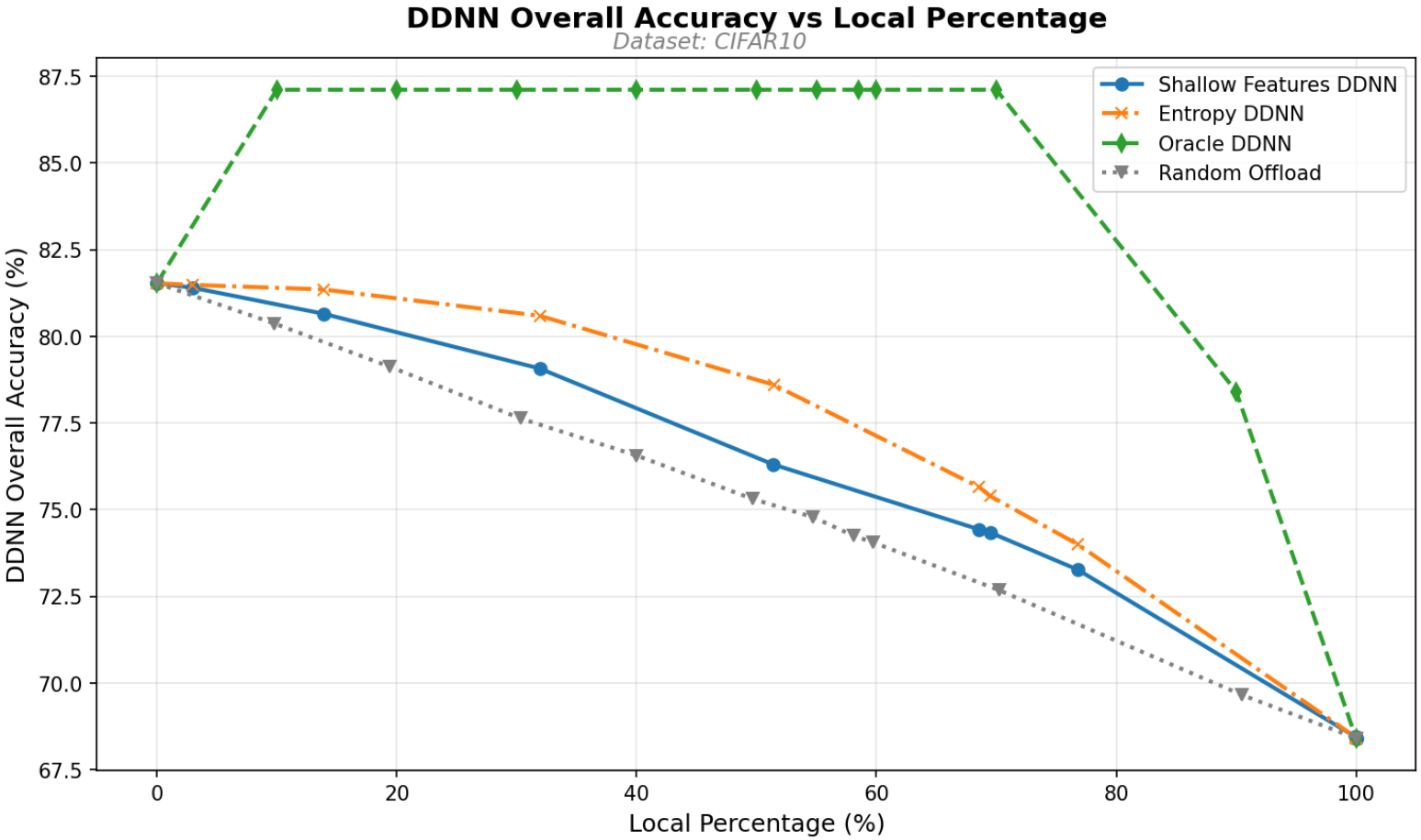


Figure 6.13: Performance of Offloading Mechanism using Local Features as input with a Shallow MLP Network.

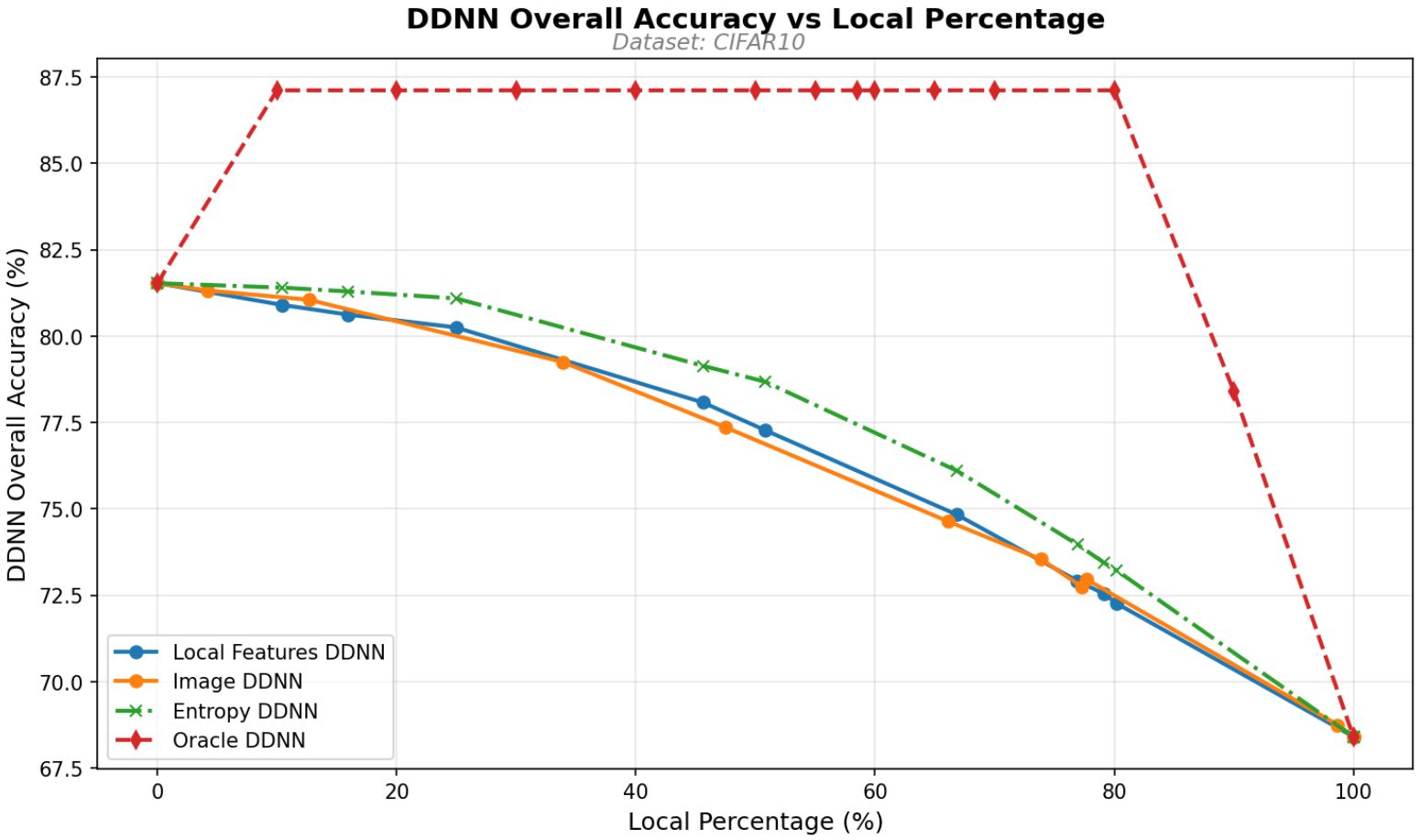


Figure 6.14: Performance of Deeper CNN Offloading Mechanism using Local Features as input.

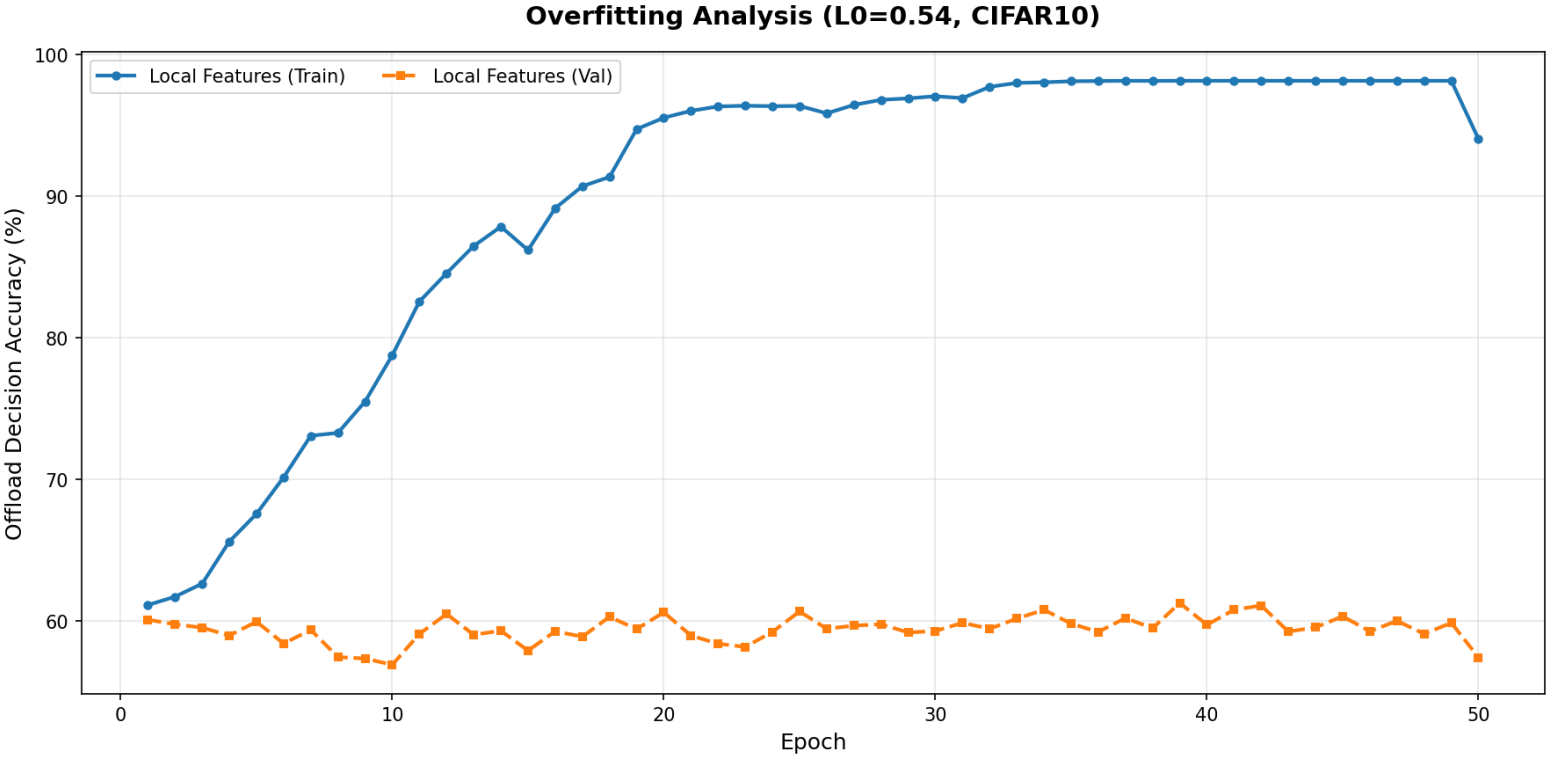


Figure 6.15: Train vs Validation on Deeper CNN Optimized Rule implying Overfitting.

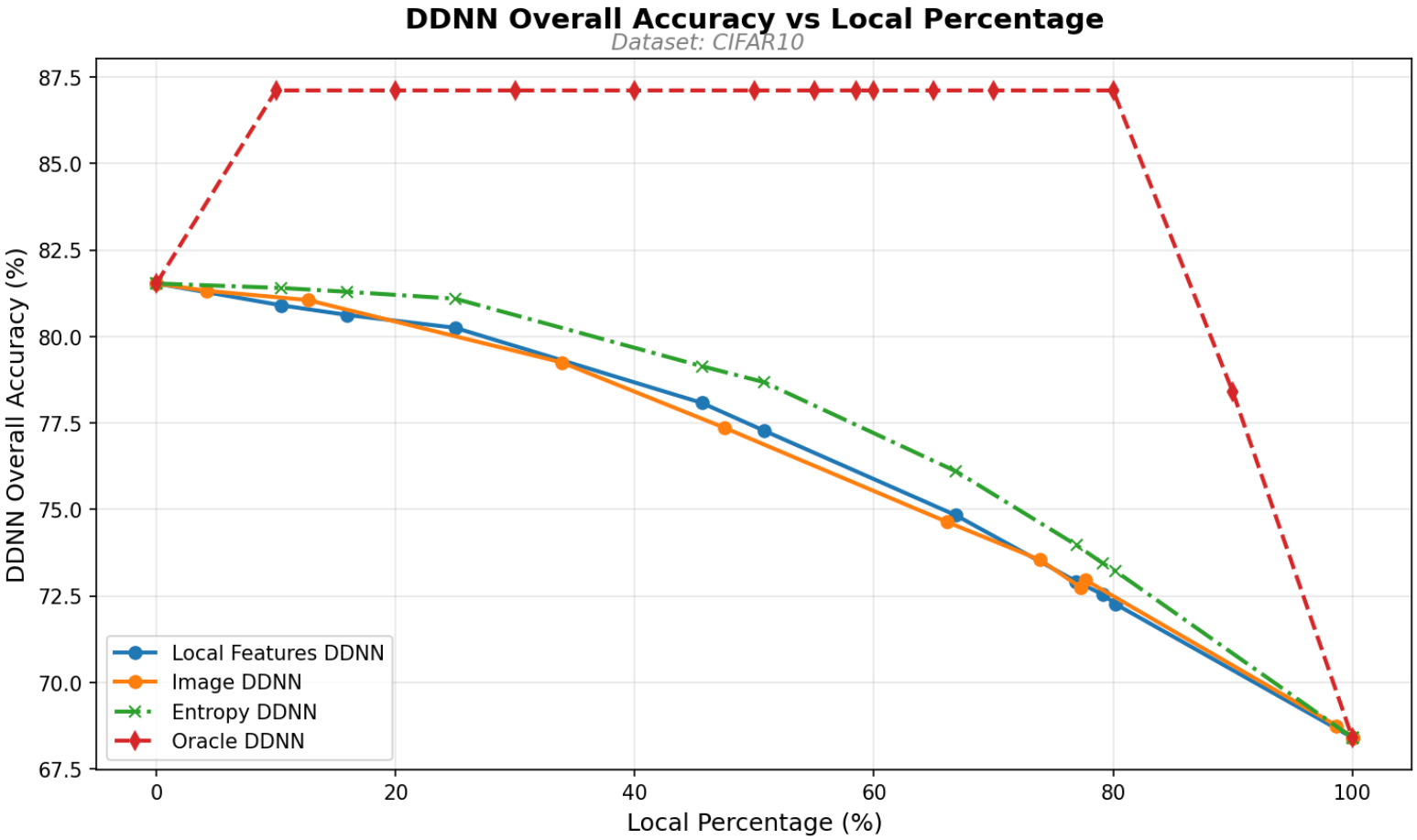


Figure 6.16: Testing local features input vs raw images accuracy performance.

Figure 6.17: Performance across different local weights using entropy-based offloading.

Chapter 7

Future Work and Next Steps

Based on the limitations and observations derived from our empirical testing, we mark several directions for potential future research. In this chapter, we discuss potential improvements for further investigation in specific areas in order to enhance the performance of the DDNN offloading mechanism.

1. **Enriching the Input Format.** A primary area for investigation lies in the input format used to train the offloading mechanism. Currently, our Optimized Rule is trained using only a 10-dimensional input vector consisting of the Local Classifier's logits. This is a significantly lower-dimensional input compared to the approach in the reference work by Giannakas et al.[16], which utilized larger latent feature sets.

   Our training analysis suggests an information bottleneck between the input provided to the Optimized Rule and its performance. As shown in Figure 6.10, the training and validation accuracy curves remain close, indicating no significant overfitting. Furthermore, increasing the model's depth did not yield better results, as seen in Figure 6.3. Since the model has sufficient learning capacity but fails to reach higher accuracy, we can conclude that the performance limitation of the Optimized Rule comes from the lack of sufficient information in the 10-logit input.

   We already validated this hypothesis on Section 6.7 experimenting with the "Logits+" mode. Adding just two extra signals—entropy and the difference between the top-2 predicted probabilities— proved to be more efficient outperforming previous approaches, confirming bottleneck. Future work could experiment with adding a compressed representation of the raw input image or extracting specific high-value features. However, such an approach requires careful experimentation to find an efficient representation that wont increase significantly the needed capacity and computational cost for the model. It would also necessitate redesigning the Offloading Mechanism's architecture to effectively consume the additional information and transform it to more complicate learning patterns.

2. **Adjusting the loss function.** Given a possibly imbalanced dataset in the sense of "easy" (local) and "hard" (cloud) samples, we may consider replacing the current binary cross-entropy (BCE) loss with a weighted BCE that accounts for the unbalanced distribution of labels. Another option might involve experimenting with focal loss, which emphasizes harder-to-classify samples.

3. **Refining the Cost Function and Data Labelling.** Already mentioned in the meta-dataset analysis (Section 5.3) and Figure 5.1, the current cost-based labelling (B_x) results in a significant percentage of "borderline" samples. These samples, characterized by negligible offload to cloud values ($|B_x| < \tau$), complicate the learning process by adding unclear offloading boundaries and in most cases are responsible for misclassification as demonstrated in Section 6.8.

In fact a possible effective adjustment would be to redefine the definition of the benefit variable B_x to produce a more separable distribution reducing the density of samples near the decision boundary. Simple adjustments such as incorporating the already used entropy of the decision and probabilities metrics might be successful. In addition more complicate metrics such as offloading transmission cost added could help push samples toward a clearer "easy" or "hard" label.

Simple filtering the meta-dataset before training to get rid of borderline samples could be beneficial in short term but it is not a robust solution. Instead the goal is to develop a cost function capable of distinguishing these borderline cases.

4. **Dynamic Joint Training.** Currently, the offloading mechanism is trained offline using a static meta-dataset generated from a single inference pass of the fully trained DDNN. As a result any noise generated during the single DDNN inference is permanently embedded on the meta dataset used to train the Optimized Rule offloading mechanism affecting its learning and performance. In order for the Optimized Rule to better generalize future work could explore a dynamic co-training strategy where the Optimized Rule is trained simultaneously with the DDNN. By co-training the offloading mechanism can adapt continuously to the evolving cost function rather than rely on fixed values. The DDNN will be trained in isolation for a certain number of epochs in "warm-up" phase to ensure feature extraction stability, after which the offloading mechanism would be activated, and both networks would be optimized simultaneously by tracking and minimizing their respective losses.

Bibliography

[1] S. Teerapittayanon, B. McDanel, and H. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2016, pp. 2464–2469.

[2] ——, "Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. Atlanta, GA, USA: IEEE, Jun. 2017, pp. 328–339. [Online]. Available: http://ieeexplore.ieee.org/document/7979979/

[3] M. L. Cummings, "Artificial intelligence and the future of warfare," Chatham House, The Royal Institute of International Affairs, London, UK, Research Paper, Jan. 2017, international Security Department and US and the Americas Programme. [Online]. Available: https://www.chathamhouse.org/sites/default/files/publications/research/2017-01-26-artificial-intelligence-future-warfare-cummings-final.pdf

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2009, pp. 248–255.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 25, 2012.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations (ICLR)*, 2015, arXiv:1409.1556.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[10] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[11] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," 2017, arXiv:1611.03530.

[12] Y. Kaya, S. Hong, and T. Dumitraş, "Shallow-deep networks: Understanding and mitigating network overthinking," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, ser. PMLR, vol. 97, 2019, pp. 3301–3310, arXiv:1810.07052.

[13] J. Pomponi, S. Scardapane, and A. Uncini, "A probabilistic re-interpretation of confidence scores in multi-exit models," *Entropy*, vol. 24, no. 1, p. 1, 2021. [Online]. Available: https://www.mdpi.com/1099-4300/24/1/1

[14] L. Meronen, E. Rahtu, and J. Kannala, "Fixing overconfidence in dynamic neural networks," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2024, pp. 2487–2496. [Online]. Available: https://openaccess.thecvf.com/content/WACV2024/papers/Meronen_Fixing_Overconfidence_in_Dynamic_Neural_Networks_WACV_2024_paper.pdf

[15] M. Samus and O. Artemenko, "The impact of artificial intelligence in the drones' war in ukraine," New Strategy Center, Tech. Rep., 2025. [Online]. Available: https://newstrategycenter.ro/wp-content/uploads/2025/02/The-Impact-of-Artificial-Intelligence-in-the-Drones-War-in-Ukraine.pdf

[16] T. Giannakas, D. Tsilimantos, A. Destounis, and T. Spyropoulos, "Fast Edge Resource Scaling With Distributed DNN," *IEEE Transactions on Network and Service Management*, vol. 22, no. 1, pp. 557–571, Feb. 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10854806/

[17] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks," Sep. 2017, arXiv:1709.01686 [cs]. [Online]. Available: http://arxiv.org/abs/1709.01686

[18] P. Aksenov, O. Slizovskaia, V. Shakhuro, and I. Gurevych, "Selfxit: An unsupervised early exit mechanism for deep neural networks," *Pattern Recognition Letters*, vol. 169, pp. 93–101, 2023.

[19] X. Liu, W. Zhang, Y. Han *et al.*, "Unsupervised early exit in dnns with multiple exits," *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[20] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge, "A hybrid approach to offloading mobile image classification," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2014, p. 8375–8379.

[21] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 615–629.

[22] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu, "Jalad: Joint accuracy- and latency-aware deep structure decoupling for edge-cloud execution," *arXiv preprint arXiv:1812.10027*, 2018. [Online]. Available: https://arxiv.org/abs/1812.10027

[23] E. Li, Z. Zhou, and X. Chen, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," *IEEE Transactions on Mobile Computing*, vol. 18, no. 11, pp. 2593–2606, 2018.

[24] J. Chen and X. Ran, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE Transactions on Mobile Computing*, vol. 19, no. 11, pp. 2596–2609, 2019.

[25] Y. Kang, Z. Zhang, Y. Liu, X. Jin, and F. Yan, "Splitnets: Designing neural architectures for efficient distributed computing on head-mounted systems," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 10 174–10 183.

[26] Z. Yang, X. Li, X. Chen, J. Xu, and D. Niyato, "Jmsnas: Joint model split and neural architecture search for learning over mobile edge networks," *IEEE Transactions on Mobile Computing*, 2022.

[27] W. Zhao, Y. Liu, L. Zhang, and J. Chen, "Improving device-edge cooperative inference of deep learning via 2-step pruning," *IEEE Internet of Things Journal*, vol. 9, no. 8, pp. 6008–6020, 2022.

[28] S. Gao, Y. Xu, Z. Li, and T. Qiu, "Joint dnn partitioning and task offloading in mobile edge computing via deep reinforcement learning," *IEEE Transactions on Mobile Computing*, 2023.

[29] D. J. Bajpai, V. K. Trivedi, S. L. Yadav, and M. K. Hanawal, "Splitee: Early exit in deep neural networks with split computing," 2023. [Online]. Available: https://arxiv.org/abs/2309.09195

[30] F. Xia, X. Li, Y. Zhang, C. Ding, S. Huang, Y. Xie, and Y. Lin, "I-splitee: Accelerating deep inference for emerging mobile applications with fine-grained early exiting," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021, pp. 529–542.

[31] X. Peng, X. Wu, L. Xu, L. Wang, and A. Fei, "DistrEE: Distributed early exit of deep neural network inference on edge devices," Feb. 2025. [Online]. Available: http://arxiv.org/abs/2502.15735

[32] A. Ehsanian and T. Spyropoulos, "Multiple edge-exit approach for slice resource allocation with distributed deep neural networks," 06 2025, pp. 2364–2369.

[33] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[35] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: https://www.deeplearningbook.org/

[36] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015. [Online]. Available: https://arxiv.org/abs/1505.00853

[37] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[38] L. Pérez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *arXiv preprint arXiv:1712.04621*, 2017. [Online]. Available: https://arxiv.org/abs/1712.04621

[39] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, 2010.

[40] Q. Xing, M. Xu, T. Li, and Z. Guan, "Early exit or not: Resource-efficient blind quality enhancement for compressed images," in *European Conference on Computer Vision (ECCV)*, 2020, vol. 12361, pp. 275–292.

[41] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "SPINN: Synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking (MobiCom)*. London, United Kingdom: ACM, Sep. 2020, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3372224.3419194

[42] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-supervised nets," in *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR, 2015, pp. 562–570.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations (ICLR)*, 2015, arXiv:1412.6980.