Deep Compressive Offloading: Speeding Up Neural Network Inference by Trading Edge Computation for Network Latency

Shuochao Yao[1], Jinyang Li, Dongxin Liu, Tianshi Wang,
Shengzhong Liu, Huajie Shao, Tarek Abdelzaher
[1]George Mason University, University of Illinois at Urbana-Champaign
[1]shuochao@gmu.edu, {jinyang7, dongxin3, tianshi3, sl29, hshao5, sl29, zaher}@illinois.edu

ABSTRACT

With recent advances, neural networks have become a crucial building block in intelligent IoT systems and sensing applications. However, the excessive computational demand remains a serious impediment to their deployments on low-end IoT devices. With the emergence of edge computing, offloading grows into a promising technique to circumvent end-device limitations. However, transferring data between local and edge devices takes up a large proportion of time in existing offloading frameworks, creating a bottleneck for low-latency intelligent services. In this work, we propose a general framework, called deep compressive offloading. By integrating compressive sensing theory and deep learning, our framework can encode data for offloading into tiny sizes with negligible overhead on local devices and decode the data on the edge server, while offering theoretical guarantees on perfect reconstruction and lossless inference. By trading edge computing resources for data transmission time, our design can significantly reduce offloading latency with almost no accuracy loss. We build a deep compressive offloading system to serve state-of-the-art computer vision and speech recognition services. With comprehensive evaluations, our system can consistently reduce end-to-end latency by 2× to 4× with 1% accuracy loss, compared to state-of-the-art neural network offloading systems. In conditions of limited network bandwidth or intensive background traffic, our system can further speed up the neural network inference by up to 35× [1].

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Embedded and cyber-physical systems**.

KEYWORDS

Deep Learning, Edge Computing, Offloading, Compressive Sensing, Compressive Offloading, Internet of Things

[1]Code is available on https://github.com/CPS-AI/Deep-Compressive-Offloading

SenSys '20, November 16–19, 2020, Virtual Event, Japan

ACM ISBN 978-1-4503-7590-0/20/11…$15.00
https://doi.org/10.1145/3384419.3430898

ACM Reference Format:
Shuochao Yao[1], Jinyang Li, Dongxin Liu, Tianshi Wang, and Shengzhong Liu, Huajie Shao, Tarek Abdelzaher. 2020. Deep Compressive Offloading: Speeding Up Neural Network Inference by Trading Edge Computation for Network Latency. In *The 18th ACM Conference on Embedded Networked Sensor Systems (SenSys '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3384419.3430898

1 INTRODUCTION

Future sensing systems will be smarter and more user-friendly. They will perceive the physical environment, understand human context, and interact with end-users in a human-like fashion. Daily objects will be capable of leveraging sensor data to perform complex estimation and recognition tasks, such as recognizing visual inputs, understanding voice commands, tracking objects, and interpreting human actions. This raises important research questions on how to endow low-end embedded (usually mobile) devices with the appearance of intelligence despite their resource limitations.

Thanks to recent advances in deep learning, state-of-the-art neural networks achieved significant accuracy improvements in a broad spectrum of areas, including computer vision [24, 40], speech analysis [4, 21], language processing [6, 16], and mobile sensing [46, 47, 49]. However, high execution time and energy consumption remain the major barriers to large-scale deployment of deep learning services on lower-end embedded and/or mobile sensing devices. Despite recent progress in compressing neural networks for reducing resource demands [10, 22, 48, 50], the computational requirements of deep learning models remain prohibitive for lots of low-end devices.

Offloading data to a more computationally capable node is a potential solution to facilitate ubiquitous deep neural network services on otherwise computationally-limited devices. By partitioning neural network models and transferring inputs or intermediate data to nearby (edge) servers, inference can be entirely or partly offloaded, which eases the burden on local end-devices [17, 30]. However, transferring data between the mobile/embedded sensing device and the edge server takes up a long time in most existing offloading pipelines, creating a need for optimization to fit latency-sensitive applications. This challenge motivated much recent research.

One potential system solution is to decide the optimal offloading point in a neural network based on current computing resources and network conditions [17, 30]. The intuition here is that some intermediate layers in the neural network may have smaller sizes. Selecting these layers as offloading points can reduce data transmission time. However, the intermediate data sizes of the first several layers are still large. In practice, we have to run a considerable portion of the model locally to reach a bandwidth-efficient offloading point, which diminishes the offloading benefit. Another design settles on an inferior but efficient local model to cut down the frequency of offloading requests [39]. This method, however, trades inference accuracy for speed-up, resulting in up to a 10% accuracy loss. There are also application-specific systems that control the overall latency carefully [34]. Unfortunately, these system designs are integrated with the application and are not directly applicable across application domains.

An elegant category of solutions that come closest to ours leverages learning-based data compression techniques, such as autoencoder structures [3, 35], to compress data locally for offloading and then reconstruct it on the server side. These techniques can compress data features into tiny sizes while attaining a high-fidelity reconstruction. However, they result in a *symmetric* split of processing burden among the encoder and decoder sides. This is suboptimal due to the inherent asymmetry in computing resources and power among the end-device and edge server sides. In contrast, the main contribution of this paper lies in an *asymmetric* encoder/decoder framework that incurs much less overhead on the (resource-limited) end-device, putting most of the burden on the server side. As shown in the evaluation, this asymmetry results in significantly improved end-to-end latency.

We call the general framework proposed in this paper, *Deep Compressive Offloading*. It substantially reduces offloading latency, while contributing only a negligible computational overhead on local end-devices, and incurring almost no degradation in inference accuracy. More specifically, by taking the computational capabilities of local and edge devices into consideration, we design an imbalanced "autoencoder" with a lightweight encoder to compress data into tiny sizes on a local device and a relatively complex decoder to reconstruct the data on the edge server, while offering a recovery guarantee. One potential choice of the imbalanced autoencoder is to apply the well-known and theoretically grounded compressive sensing [7, 13] ideas. If the data are sparse in a domain, compressive sensing can encode (*i.e.*, compress) the data with a simple measure matrix and decode (*i.e.*, reconstruct) the data with an optimization-based method. Unfortunately, unmodified compressive sensing has two significant drawbacks, making it an inappropriate choice for our imbalanced encoder-decoder design.

First, compressive sensing requires prior knowledge of the transformation that can sparsify the target data. For those well-studied data types, such as images or voice, we can obtain prior knowledge from related research. However, our system will offload not only well-studied data types but also intermediate neural network features, whose sparse properties are unknown. Second, the reconstruction of compressive sensing requires slow iterative optimization methods, such as iterative soft thresholding and gradient projection [14, 18]. Although edge servers are far more powerful than local devices, it still takes several seconds to reconstruct a mid-size ($224 \times 224 \times 3$) image. If we adopt traditional compressive sensing recovery algorithms at the decoder, the reconstruction process will become a new offloading bottleneck.

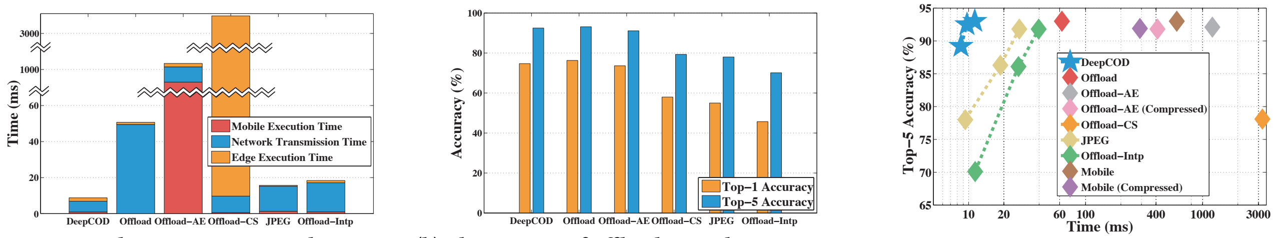
To overcome these two drawbacks, deep compressive offloading replaces the optimization-based decoder with a trainable generative neural network. Instead of relying on sparsity, a generative neural network can serve as the implicit prior constraint for decoding data [11]. The generative neural network learns a map from the low-dimensional data space to the targeted data distribution. Therefore, we can reconstruct the encoded data through a single run of the generative model and can get rid of the slow iterative algorithms, including gradient descent [11] and iterative optimizations [14, 18]. Furthermore, traditional compressive sensing uses random measurement matrices, which is known to be suboptimal [45]. Deep compressive offloading designs the encoding part as a learnable but lightweight single-layer convolution. It can further automatically learn the optimal transformation for compressing the data for offloading with low computational overhead on local devices.

Designing the offloading encoder and decoder as neural networks does not mean giving up recovery guarantees of compressive sensing. Following the theoretical analysis of compressive sensing [11, 13], deep compressive offloading imposes the Restricted Isometry Property (RIP) and Lipschitz continuity on the encoder and the decoder respectively, which ensures satisfaction of recovery guarantees in our offloading system.

Compared with traditional data reconstruction and compressive sensing problems, our system has one additional advantage: It can leverage the fact that the goal of the decoder *is no longer to perfectly recover original data* before encoding. Rather, the goal is to decode the encoded data to *achieve the best inference results*. Therefore, we can borrow additional knowledge from the original deep learning service, using intermediate feature maps of its deep learning model as additional supervisions to our offloading encoder and decoder training. Notice that the proposed compressive offloading does not require any additional changes in the original deep learning service, including its model structure and parameters. Thus, we can easily apply our design of deep compressive offloading as a unified solution to a general deep learning service without domain knowledge. Moreover, we can train our encoder and decoder without any labeled data, which further simplifies the deployment in practice.

We integrate the ideas and theoretical underpinnings of deep compressive offloading into a practical system, called DeepCOD. The system has a performance predictor and a runtime partition decision maker to find the optimal partition point for offloading. DeepCOD provides a general offloading function for any deep learning service. We implement this system on Android mobile devices and a Linux edge server with GPUs. We choose two widely deployed applications with corresponding state-of-the-art neural network models to evaluate our offloading system, namely, image recognition with ResNet-50 [24] on *ImageNet ILSVRC2012 dataset* [15], and speech recognition with Deep Speech [23] on *Librispeech dataset* [38]. We deploy and evaluate DeepCOD with the combinations of two types of mobile devices, an edge server with two types of GPUs, and two types of wireless connections with various additional bandwidth constraints. In all deployments, DeepCOD consistently achieves $\times 2$ to $\times 4$ end-to-end offloading latency reduction with at most 1% accuracy loss for image and speech recognition services when compared with the state-of-the-art offloading systems and model compression techniques for deep neural networks [3, 30–32, 35, 48]. Under conditions of limited network bandwidth or intensive background traffic, DeepCOD can further speed up the neural network inference time by up to $\times 35$.

In summary, we propose a general offloading technique, called deep compressive offloading, including theoretical analysis, system design, and implementation. From the theoretical and empirical perspectives, deep compressive offloading can substantially speed



(a) Decompose latency into network transmission, mobile and edge execution time.

(b) The impacts of offloading techniques on inference accuracy.

(c) Deep compressive offloading achieves Pareto optimality (the time axis in log scale).

Figure 1: A case study of image recognition application with ResNet-50 model. Google Pixel is connected to an edge with Titan V through 450Mbps WiFi. Images on the mobile device are offloaded to the edge server with various offloading techniques.

up edge offloading of deep learning services with almost no impact on model inference results.

2 MOTIVATION: SYSTEM PERFORMANCE

It has been widely recognized that transferring data between mobile devices and edge servers is the bottleneck in offloading deep learning services [17, 30, 34, 39]. In this section, we start with a practical example and empirical measurements to investigate the challenge and opportunity of speeding up such offloading.

In this experiment, we deploy a 1000-category image recognition service with ResNet-50 on Google Pixel phone. The mobile phone is connected to an edge service with Nvidia Titan V GPU through a 450Mbps WiFi connection. The size of the testing images is 224×224. We measure the end-to-end latency of image classification with seven different inference designs, including both edge offloading and on-device processing techniques.

On-device processing is one of the most widely adopted solutions for deploying deep learning on embedded/mobile devices. As illustrated in Figure 1c, the ResNet-50 model, denoted as "Mobile", takes more than 650 ms on the mobile device. Even when we apply the state-of-the-art system-aware compression technique, FastDeepIoT [48], the compressed ResNet-50 model, denoted as "Mobile (Compressed)", still takes more than 250 ms to finish a single inference. Therefore, embedded and mobile devices naturally call for offloading solutions to speed up neural network inference and enable low-latency applications.

The vanilla offloading operation, denoted as "Offload" in Figure 1, transfers input data or intermediate representations to the edge server without additional manipulation. Based on system and network profiling, the choice of offloading input image is the optimal offloading decision under a standard WiFi environment. As shown in Figure 1a, although traditional offloading can reduce end-to-end latency to around 60 ms, most of the time is consumed by transferring the image from the mobile phone to the edge server. Therefore, the network transmission time is the bottleneck of offloading, which offers new opportunities for speeding up.

In order to reduce the transmission time, an on-hand solution is to decrease the resolution of the image on the local device and to interpolate the compressed image on the edge server, denoted as "Offload-Intp". As shown in Figure 1a, by compressing the size of the image to 4%, we can reduce the end-to-end latency to around 15 ms. However, the reduction of time is at the cost of inference accuracy. "Offload-Intp" suffers 20% accuracy loss as shown in Figure 1b.

Compressive sensing, denoted as "CS", is a sophisticated method for compressing and reconstructing data, which can mitigate the negative impact on inference accuracy. However, due to the suboptimality of its sparsity assumption, it still suffers around 10% accuracy loss with an image compression ratio of 25. Moreover, the slow iterative reconstruction of CS becomes the new bottleneck of the end-to-end latency. Since we are transferring visual image, we can also apply domain knowledge, using JPEG for image compression. However, it suffers around 14% accuracy loss with a compression ratio of 25. Furthermore, JPEG only works for visual image, which has bad compression performance on the intermediate visual representations in neural network as well as data in other domains, as we will show latter.

Recent advances in deep-learning-based data compression techniques offer a data-driven solution to reduce the communication load. We implement a state-of-the-art data compression technique for the offloading system, called "Offload-AE" [35], which adopts the AutoEncoder structure. Although Offload-AE can significantly reduce the transmission time and have a minor impact on accuracy, Offload-AE has a complicated encoder model, which consumes more than 900ms on the mobile device, as shown in Figure 1a. Moreover, we further compress the encoder in Offload-AE based on FastDeepIoT [48] to reduce overhead on the mobile device. The resulted system, called "Offload-AE (Compressed)", still takes over 400 ms, as shown in Figure 1c.

The above difficulty in finding a good trade-off point calls for a solution that can significantly reduce network transmission time with almost no compromise on accuracy and with negligible computational overhead. Our proposed system, called "DeepCOD" attains those goals. DeepCOD marries solid compressive sensing theory with flexible deep-learning-based modeling to solve these practical challenges in offloading. As shown in Figure 1c, compared with all other general-purpose techniques, deep compressive offloading can achieve the Pareto optimality by attaining the best inference accuracy with the least amount of end-to-end latency (of around 10ms in this example). By leveraging the asymmetry in computational resource across the mobile/embedded device and edge server, deep compressive sensing achieves the best speed-up with the least accuracy loss. We will introduce its design, theoretical analysis, and system implementation in the following sections.

3 DEEP COMPRESSIVE OFFLOADING

We introduce technical details of deep compressive offloading in this section. The overall design of deep compressive offloading is illustrated in Figure 2, which includes a lightweight encoder on the mobile side to compress the data to transfer, and a decoder on the edge server side to reconstruct the transferred data. Notice that such design works for any offloading point (*i.e.*, regardless of how to partition neural networks among the mobile/embedded device and the edge server). We will postpone the discussion of the best offloading point to the end of this section.

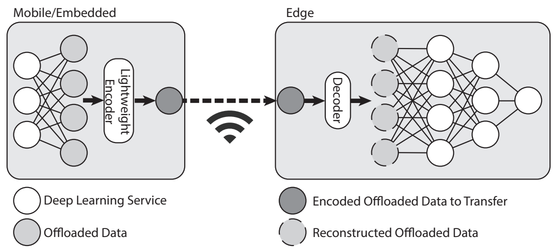


Figure 2: The Deep Compressive Offloading designs with a lightweight encoder on the local device to compress data and a decoder on the edge server to reconstruct.

We first introduce background on compressive sensing and its recent extension with deep generative neural networks. Next, we formulate deep compressive offloading and our design that ensures recovery guarantees. Then, we investigate the proper way to enhance the performance of deep compressive offloading by distilling knowledge from deep learning services. Finally, we introduce other offloading-supporting components, including quantization, entropy encoding, and dynamic offloading point selection.

3.1 Compressive Sensing

The target of compressive sensing is to reconstruct an unknown vector $\mathbf{x} \in \mathbb{R}^n$ through observing linear measurement with the possible added noise, which can be formulated as:

$$\mathbf{y} = \mathbf{E}\mathbf{x} + \eta, \tag{1}$$

where $\mathbf{E} \in \mathbb{R}^{m \times n}$ is the measurement matrix, and η is the measurement noise. Typically, we want to reconstruct the data $\mathbf{x}$ with much fewer observations $\mathbf{y}$, *i.e.*, $m \ll n$. Even without the noise, it is impossible to solve $\mathbf{x}$ for such an under-determined problem. Therefore, we need to impose certain prior knowledge on the solution $\mathbf{x}$, assuming it to be natural and simple in an application-dependent way. One widely accepted assumption is sparsity. However, finding the sparest solution of an under-determined problem is still NP-hard. Fortunately, the elegant compressive sensing theory proved that convex optimization could recover the true sparse vector $\mathbf{x}$ if the matrix $\mathbf{E}$ satisfies conditions such as the Restricted Isometry Property (RIP) or the related Restricted Eigenvalue Condition (REC) [13]. A random matrix is an example that meets RIP, which is widely used as the measurement matrix. It guarantees that minimizing the recovery error

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \|\mathbf{y} - \mathbf{E}\mathbf{x}\|^2, \tag{2}$$

under the constraint that $\mathbf{x}$ is sparse, leads to accurate reconstruction $\hat{\mathbf{x}} = \mathbf{x}$ with high probability [13]. In practice, the sparsity constraint of $\mathbf{x}$ can be replaced by sparsity in a set of basis Φ. DCT, Fourier, and wavelet are common choices. Since Φ is a linear transformation, it does not affect the recovery guarantee.

However, the sparsity constraint is not an optimal and universal assumption for data within various applications. Therefore, compressive sensing is not a perfect design choice for reducing offloading transmission latency of an arbitrary IoT service. Recently, pre-trained generative neural networks have been explored as powerful but implicit alternatives to sparsity constraints for compressive sensing [11]. These pre-trained generative neural network G_θ can easily adapt to the target data distribution and add structural constraints during reconstruction. The generator G_θ usually maps a random hidden vector to the data space

$$\mathbf{x} = G_\theta(\mathbf{z}). \tag{3}$$

Instead of explicitly adding sparsity constraints, constraints are now implicitly controlled by the structure and parameters of the generator. The reconstruction process (2) now becomes minimizing

$$\hat{\mathbf{z}} = \arg\min_{\mathbf{z}} \|\mathbf{y} - \mathbf{E}G_\theta(\mathbf{z})\|^2, \tag{4}$$

where $\mathbf{x}$ in (2) is now $G_\theta(\mathbf{z})$. In order to maintain the recovery guarantee, measurement matrix $\mathbf{E}$ needs to satisfy a new condition called Set-Restricted Eigenvalue Condition (S-REC) [11], which is a generalization of REC. In addition, in order to acheive small reconstruction error with a reasonably good compression ratio, the generator G_θ needs to be an L-Lipschitz function, where smaller L grants the reconstruction with fewer measurements y [11].

3.2 Deep Compressive Offloading

However, as we mentioned in Section 2, compressive sensing [14, 18] and its extension with pre-trained generative neural networks [11] fail to work properly on "lossless" offloading latency reduction. The main reasons are twofold. On the one hand, the random measurement matrix and pre-trained generative neural network cannot perfectly fit the application-specific data distribution, causing a noticeable loss in inference accuracy. On the other hand, all aforementioned (traditional and deep-learning based) compressive sensing reconstructions are slow. Both iterative optimization (2) [14, 18] and iterative backpropagation over the pre-trained generative model (4) [11] require thousands of iterations to achieve a reasonably good result. These methods are far too slow for online reconstruction in offloading. In the following sections, we reformulate the theory and design of deep compressive offloading to overcome these challenges and provide practical deployments to validate our designs.

3.2.1 *Trainable Compressive Offloading Modules.*

We shift the computation load of reconstruction from online iteration steps to offline training, by setting the hidden vector $\mathbf{z}$ of the generative model to be the measurement $\mathbf{y}$. At the same time, we replace the pre-defined random measurement matrix $\mathbf{E}$ by a trainable kernel $\mathbf{E}_\phi$. Therefore, we are training an encoder $\mathbf{E}_\phi$ and a decoder $G_\theta(\cdot)$ that can jointly compress and reconstruct the data during the offloading. Our offline training objective function is

$$\arg\min_{\theta,\phi} \left\|\mathbf{x} - G_\theta(\mathbf{E}_\phi \circledast \mathbf{x})\right\|^2 \tag{5}$$

where $\circledast$ denotes the convolution operation, θ, and ϕ are sets of learnable parameters for decoder and encoder.

Once we have trained the encoder and decoder, we can deploy them on the local device and edge side respectively. The encoding process $\mathbf{y} = \mathbf{E}_\phi \circledast \mathbf{x}$ has at most the same amount of computation as the traditional compressive sensing encoding $\mathbf{y} = \mathbf{E}\mathbf{x}$, when both of them are given the same compressing ratio. Next we can reconstruct the data with a one-shot inference of the decoder, $\hat{\mathbf{x}} = G_\theta(\mathbf{y})$, which is around 1000 times faster than the previous online reconstruction method (2) and (4). With the help of offline encoder-decoder training, we can dramatically reduce the time of data reconstruction without extra computational burdens on the data encoding part. Simultaneously, training encoder and decoder makes them fit better to the offloaded data distribution, which in return helps to improve the final inference accuracy.

3.2.2 Theoretical Analysis Ensures both Speed-up and Accuracy.

However, without enforcing mathematical properties on encoder and decoder that ensure the recovery guarantee of compressive sensing, training the encoder and decoder freely with object function (5) cannot attain good performance on offloaded data reconstruction. As we mentioned before, the encoder $\mathbf{E}_\phi$ has to meet the Set-Restricted Eigenvalue Condition (S-REC) and the decoder $G_\theta(\cdot)$ needs to be an L-Lipschitz function. Next, we discuss the way to impose these two properties on the encoder and decoder during training.

First, we introduce our solution for training encoder $\mathbf{E}_\phi$ to meet S-REC. According to the definition [11], the S-REC requires that for any two vectors $\mathbf{x}_1$ and $\mathbf{x}_2$, if they are significantly different, then the corresponding measurements with transformation $\mathbf{E}_\phi$ should also be significantly different, which can be formulated as

$$\left\|\mathbf{E}_\phi \circledast (\mathbf{x}_1 - \mathbf{x}_2)\right\| \geq \gamma \left\|\mathbf{x}_1 - \mathbf{x}_2\right\| - \delta, \tag{6}$$

where $\gamma > 0$ and $\delta > 0$. δ is an additive slack term. If $\gamma \rightarrow 1$, the decoder can reconstruct perfectly with fewer measurements generated by the encoder with high probability. In order to achieve a good compression ratio with a recovery guarantee (*i.e.*, reducing transmission latency without hurting inference accuracy), we require our encoder to be isometric, *i.e.*, $\gamma = 1$ in S-REC (6).

Since convolution can be reformulated as matrix multiplication, we can add regularization on the trainable convolution kernel $\mathbf{E}_\phi$ to force the convolution to be isometry. First, we turn the kernel into a 2D array [28], *i.e.*, $\mathbf{E}_\phi \in \mathbb{R}^{h\times w\times c_i\times c_o} \Rightarrow \mathbf{E'}_\phi \in \mathbb{R}^{h\cdot w\cdot c_i\times c_o}$, forming convolution as matrix multiplication. Next, as a linear transformation, we can impose the isometry property by forcing the linear transformation matrix to be a semi-orthogonal matrix [1]. Therefore, we add an orthogonal regularization to the encoder convolution kernel $\mathbf{E'}_\phi$, while training encoder,

$$\arg\min_{\phi} \left\|\mathbf{E'}_\phi^\intercal \mathbf{E'}_\phi - \mathbf{I}\right\|, \tag{7}$$

where $\mathbf{I}$ is the identity matrix. Notice that the regularization (7) still works when we design the encoder to be a single fully-connected layer. In this paper, we choose the convolution operation to reduce the computations and the number of parameters to learn.

Second, we discuss a way of ensuring that the neural network decoder G_θ is an L-Lipschitz function. According to previous theoretical analysis [11], neural network decoder G_θ needs to be an L-Lipschitz function, where L is called the Lipschitz constant. Assume that $\mathbf{y}_1$ and $\mathbf{y}_2$ are two encoded offloaded data items, then the decoder G_θ needs to meet

$$\left\|G_\theta(\mathbf{y}_1) - G_\theta(\mathbf{y}_2)\right\| \leq L \left\|\mathbf{y}_1 - \mathbf{y}_2\right\|. \tag{8}$$

Furthermore, a decoder with a smaller Lipschitz constant can reconstruct perfectly from fewer measurements from the encoder with high probability. Again, to reduce network latency without hurting inference accuracy, we have to constrain a neural-network-based decoder to have a small Lipschitz constant L.

Since deep compressive offloading is intended to be a general solution to a broad category of applications, we do not want to constrain the decoder to be a particular type of neural network. Such constraint simplifies the analysis of their Lipschitz constant but prohibits other neural networks that may fit some applications and their data distribution better. Thus, bounding a neural network to have an arbitrary design with a Lipschitz constant L is challenging. Fortunately, there exist other related works that target controlling the Lipschitz constant of neural networks.

Recent research on Generative Adversarial Networks (GAN) starts to bound the Lipschitz constant of its discriminator neural network to be smaller than 1, *i.e.*, $L \leq 1$, which allows using Earth-Mover distance as the GAN training loss [5]. We adopt one method that is computationally light and easy to be incorporated into existing implementations, called spectral normalization [36]. Here, we briefly introduce the intuition behind spectral normalization design and its implementation.

Neural networks are layered structures. According to the definition (8), if we can bound the Lipschitz constant of each layer to be less than 1, the whole neural network becomes a 1-Lipschitz function. Without loss of generality, we regard the operations in each layer to be an affine transformation, $\mathbf{W}\mathbf{x} + \mathbf{b}$, followed by an activation function. On the one hand, the Lipschitz constants of most activation functions are less than 1, including ReLU and sigmoid function. On the other hand, Lipschitz constant of an affine transformation is controlled by the largest singular value of weight matrix $\mathbf{W}$, denoted by $\sigma(\mathbf{W})$. To keep the Lipschitz constant of each layer less than 1, we normalize the largest singular value of weight matrix:

$$SN(\mathbf{W}) = \mathbf{W}/\sigma(\mathbf{W}). \tag{9}$$

And we can use power iteration method to estimate $\sigma(\mathbf{W})$ with very small computational overhead [19].

With the orthogonal regularization on encoder (7) and spectral normalization on decoder (9), we can ensure that the encoder meets S-REC with $\gamma \rightarrow 1$ and the neural network decoder is a 1-Lipschitz function. These two properties provide deep compressive offloading a theoretical guarantee to reduce network latency without hurting the inference accuracy. Therefore, we update our offline training objective function

$$\arg\min_{\theta,\phi} \left\|\mathbf{x} - G_{SN(\theta)}(\mathbf{E}_\phi \circledast \mathbf{x})\right\|^2 + \alpha \left\|\mathbf{E'}_\phi^\intercal \mathbf{E'}_\phi - \mathbf{I}\right\|, \tag{10}$$

where α is a hyperparameter to control the orthogonal regularization, and $SN(\theta) = \{SN(\mathbf{W}) : \forall \mathbf{W} \in \theta\}$.

3.2.3 Compressive Encoder & Decoder Structures.

Although deep compressive offloading can apply almost all types of encoder-decoder designs with the objective function (10), we provide our designs here for illustration. Our encoder and decoder are empirically proved to be effective in computer vision and speech recognition applications. However, users are welcome to design their own encoder or decoder for better speed-up or accuracy.

We illustrate the default encoder and decoder structures and configurations we used in all our experiments in Figure 3. On the encoder side, we design a single convolution $\mathbf{E}_\phi$ to compress the data. The data to be transferred contains two types of dimensions, spatial-temporal dimensions, and a feature dimension. Spatial-temporal dimensions, including height and width of images as well as the time of voice, share the convolution kernel. The feature dimension is the left flattened dimension that serves as the input channel of convolution. To have a lightweight encoder on the mobile and embedded devices, we set the convolution strides to be equal to the

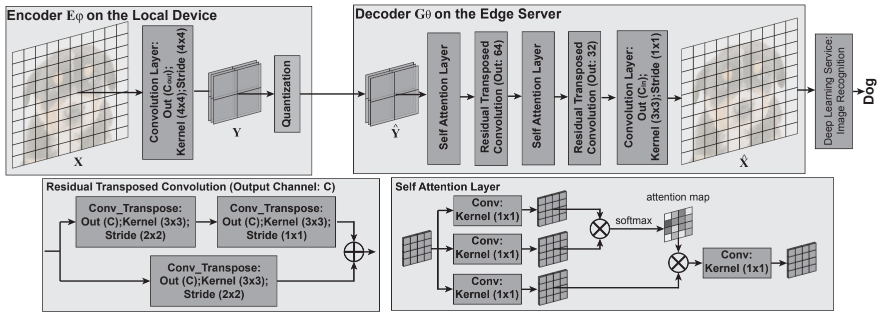


Figure 3: The default designs and configurations of decoder and decoder structures that used in all our experiments.

kernel sizes (equals to 4×4 by default). For other data types, such as speech and time-series sensing data, we can have a similar design using a 1D convolution. We will introduce the details of the quantization module in Section 3.4.1. On the decoder side, we adopt the ideas from recent generative models, using residual transposed convolution to expand the size of spatial-temporal dimensions [12, 24] and using self-attention layer to enhance the spatial-temporal dependency during the generation [51]. The detailed structures of residual transposed convolution and self-attention layer are illustrated in Figure 3.

Notice that the default encoder and decoder structures, illustrated in Figure 3, can be readily applied to the cases of offloading intermediate features, and other types of deep learning services that take image-based and time-series data as input.

3.3 Distilling Knowledge from Deep Learning Service

The design and analysis motioned in Section 3.2 focus on proposing an imbalanced encoder-decoder pipeline that minimizes the data reconstruction loss. However, the main purpose of deep compressive offloading is not to precisely reconstruct encoded data, but rather to *reconstruct the encoded data in a manner that achieves the best inference result*. Given the constraint of reducing the size of offloading data (*i.e.*, limiting the amount of information we can transmit), reducing data reconstruction loss and improving the inference performance of the deep learning service can conflict with each other. Therefore, in order to achieve a better accuracy-efficiency tradeoff, deep compressive offloading needs to distill knowledge from the original deep learning service, and learn to encode and decode data in a way that not only reduces reconstruction loss but also improves the inference accuracy.

The idea of knowledge distillation was first proposed as a model compression technique [25] in which a small model is trained to mimic a pre-trained, larger model (or ensemble of models). In this paper, the compressive encoder and decoder are trained to mimic the feature map/pattern that can most significantly trigger the deep learning service to generate the right inference result. However, as mentioned before, there is a conflict between reducing data reconstruction loss and inference loss. If we add additional supervision for reducing the discrepancy between the inference results from data before encoding and data after decoding, there will be a conflict with the original supervision for perfect reconstruction (10), which can hurt the final inference accuracy. The intuition is that, among all possible data that can achieve the same inference result, many can be different from or even partly conflict with the perfectly reconstructed data.

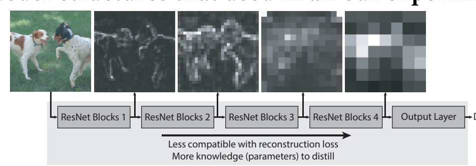


Figure 4: An illustration of intermediate representations in ResNet-50 image recognition service.

Therefore, we have to carefully design the way of distilling knowledge from deep learning services. Instead of utilizing the final inference result, we can utilize intermediate features as additional supervision. We denote the neural network in deep learning service as $C_{\psi}^{(j)}(\mathbf{x}_i)$, taking input feature from i-th layer and generating output feature at j-th layer. The knowledge distillation can be formulated as

$$\arg\min_{\theta,\phi} \left\| C_{\psi}^{(j)}\left(G_{SN(\theta)}(\mathbf{E}_{\phi} \circledast \mathbf{x}_i)\right) - C_{\psi}^{(j)}(\mathbf{x}_i) \right\|^2 \tag{11}$$

where $0 \leq i < j \leq L$; i-th layer is the place where the offloading takes place, and j-th layer is the intermediate feature chosen for knowledge distillation. We omit the orthogonal regularization (7) for simplicity.

When choosing the layer of representation in knowledge distillation, there exists a tradeoff between the amount of knowledge to distill and the intensity of conflict with reconstruction loss. A simple example is illustrated in Figure 4, where we separate the ResNet-50 image recognition model [24] into four blocks and illustrate the heat maps of their intermediate representations. Assume that we are offloading the input image. On the one hand, lower-level intermediate representations, providing image edge detection information, are more compatible with the perfect reconstruction but contain less knowledge, *i.e.*, parameters in the neural network, for distillation. On the other hand, high-level representations contain more knowledge by backpropagating through a large proportion of the neural network. However, many images are likely to be mapped to the same representation, which clearly interferes with the original reconstruction-based training (10).

In this paper, we make a tradeoff by selecting all the intermediate features between the offloading and final prediction layers. Summing up the knowledge distillation losses (11) for all intermediate layers naturally balances the tradeoff between the amount of knowledge to distill and the intensity of conflict with perfect reconstruction, which works well in our evaluations. Moreover, we add knowledge distillation loss (11) as an additional loss after the convergence of training with reconstruction loss (10) solely. It helps us reduce training time because knowledge distillation requires additional computation to backpropagate loss signals from intermediate representations to compressive encoder and decoder. Note that, with our knowledge distillation design, training deep compressive offloading does not require any label information.

3.4 Offloading-Supporting Components

We introduce additional offloading-supporting components, including quantization and entropy encoding, for further data compression and dynamic offloading partitioning concerning varying wireless conditions.

3.4.1 Quantization & Entropy Encoding. We can further reduce the size of data to transfer from the information theory perspective through quantization and entropy encoding, which is a standard pipeline in data compression [44]. In deep compressive offloading, we employ a learning-based quantization technique [2, 35, 43] and the Huffman coding [27] to quantize and encode the result, $\mathbf{y}$, produced by the compressive encoder as shown in Figure 3.

Given a set of centers $C = \{c_1, \cdots, c_L\} \subset \mathbb{R}$, we assign every scalar in $\mathbf{y}$ to a center in C based on the nearest neighbor principle

$$\hat{y}_i = \arg\min_j \left\| y_i - c_j \right\|. \tag{12}$$

However, in order to learn the optimal set of quantization centers, we also rely on the differentiable soft quantization

$$\tilde{y}_i = \sum_{j=1}^{L} \frac{\exp\left(-v\left\|y_i - c_j\right\|\right)}{\sum_{l=1}^{L}\exp\left(-v\left\|y_i - c_l\right\|\right)} c_j \tag{13}$$

to compute the gradient during backward propagation together with the a straight-through estimator (STE) [9]. In all our experiments, we set the annealing factor v to be 1 for all the time. After the quantization step, we further encode the result, $\bar{\mathbf{y}}$, with Huffman coding to reduce the number of bits to transfer.

3.4.2 Dynamic Offloading Partitioning. In order to deal with the dynamic wireless link condition, we introduce a dynamic offloading partitioning algorithm to actively select the best possible offloading point. Since offloading partitioning has been widely recognized and investigated in previous literature [30], we do not consider this part to be the technical contribution of our paper. However, we introduce it for completeness. Assume that there are P possible offloading partitions in the neural network. For each partition p, we denote the execution time on the edge server as $t_p^{(edge)}$, the time on the local device as $t_p^{(local)}$, and the size of offloaded data as d_p. The wireless link bandwidth is denoted as B. Due to the dynamic wireless link, we constantly update our estimate of wireless bandwidth, B. For example, we can measure the harmonic mean of data transfer speeds over recent offloading operations, which is robust to the outliers [29]. (We will discuss the details of estimating these network performance statistics in Section 4.2.) With this information, we dynamically select the offloading partition p that minimizes the end-to-end latency as follows,

$$\arg\min_{p\in\{1,\cdots,P\}} t_p^{(edge)} + t_p^{(local)} + d_p/B. \tag{14}$$

4 DEEPCOD DESIGN

In this section, we introduce our offloading system, DeepCOD, bringing the deep compressive offloading technique proposed in Section 3 as a flexible service to intelligent IoT applications. DeepCOD can dynamically provide the near-optimal offloading decision to deep learning services based on the current local-edge hardware, software, and network configurations. Intelligent applications using DeepCOD enjoy substantial speed-up and almost no loss in inference accuracy. DeepCOD consists of an offline training & deployment phase and a runtime phase. The system overview is illustrated in Figure 5.

4.1 Offline Training & Deployment

The offline phase contains two main modules: deep compressive offloading training, and latency profiling and modeling.

Deep compressive offloading training: In order to provide an adaptive compressive offloading strategy to an intelligent application under different devices and network configurations, DeepCOD trains compressive encoders and decoders to all potential offloading partitioning points according to the design in Section 3. Since most of the state-of-the-art neural networks have a block-based design [16, 24, 42], the starting points of neural network blocks are good candidates for potential offloading points, which are also used in our evaluation. We will discuss and evaluate the training overhead of adding a new offloading point in Section 6.7. Once we decide the potential offloading points, deep compressive offloading training is agnostic to hardware, software, and network. Therefore, it only needs to be done once for each application.

Latency profiling and modelling: In order to have a holistic understanding of latency, DeepCOD profiles the mobile/embeded device and edge server with deep learning inference engine to generate the execution-time models for a wide range of neural network operations. Recently, great efforts have been made to predict deep learning execution time accurately through profiling and modelling neural network operations [17, 30, 48]. The design of effective deep learning execution time predictors is an interesting research direction in its own right. DeepCOD employs a state-of-the-art execution time modelling technique, called FastDeepIoT [48], that models the execution time of neural networks on a platform with the corresponding operation types and configurations as inputs. Neural network profiling and modelling is application agnostic and only needs to be done once for the given local and edge devices. In addition, for cold-start throughput prediction, DeepCOD profiles offloading data transfer delay between the local device and edge server with different wireless connections for all partition points.

4.2 DeepCOD Runtime

During the execution of intelligent applications with neural networks on local devices, DeepCOD can decide the best compressive offloading point with the least latency from the partition candidates based on performance predictors.

Performance predictors: DeepCOD includes neural network execution-time predictors for the local device and edge server, as well as a network bandwidth predictor, as mentioned in Section 3.4.2. With the help of profiling and modeling results from the offline phase, DeepCOD uses a simple but effective predictor [48] to estimate the execution time of neural network operations on the local device and edge server. Given an offloading point, we can estimate the computation latency by analyzing the network configurations of neural network partitions and compressive encoders & decoders.

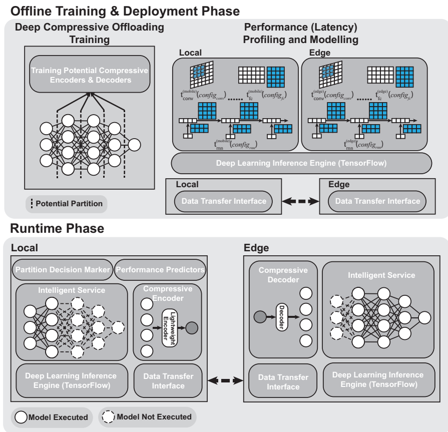


Figure 5: System overview of DeepCOD.

Building on insights from prior work, we estimate the current wireless bandwidth based on the harmonic mean of the observed throughput of the last ten offloading transmissions [29]. In practice, we can only obtain the round-trip latency, including the execution time of the compressive decoder and the second partition of the application inference model on edge. Therefore, we estimate the network latency by deducting the executing time on the edge (calculated by neural network execution-time predictor) from the round-trip latency. Then, we can estimate throughput according to the size of transferred data. In addition, we use the mean value of network profiling results from the offline phase as a rough estimation during the cold-start period.

Partition decision maker: DeepCOD dynamically selects the optimal offloading point by leveraging the latency estimates from neural network execution-time and network throughput predictors. The partition decision is made based on Equation (14).

Compressive offloading: According to the partition decision maker, the deep learning engine on the local device executes the assigned proportion of the neural network and the compressive encoder, and transfers compressed data together with partition decision to the edge. The system on edge takes the transferred data to execute the corresponding compressive decoder as well as the remaining neural network. The inference result is transferred back to the local device.

5 IMPLEMENTATION

In this section, we briefly introduce the hardware and software implementation of DeepCOD.

5.1 Hardware

The mobile client is implemented on Android OS and tested on two different Android phones. Google Pixel is equipped with a Quad-core (2x2.15 GHz & 2x1.6 GHz) Kryo CPU and Adreno 530 GPU; Nexus 6 is equipped with a 2.7 GHz quad-core Krait 450 CPU and Adreno 420 GPU. Mobile devices are connected to the edge server through WiFi connection with a TP-Link AC1200 router or connected through LTE connection. The edge server is a Linux desktop equipped with an Intel Core i7-5820K CPU and two types of GPUs, including Nvidia Titan V and Nvidia GeForce GTX Titan X. We place the edge server inside a campus office building, and the server is linked to the router through a 1Gbps cable.

5.2 Software

We assume that IoT deep learning services have TensorFlow SavedModels (or checkpoints) for their original deployments. The compressive encoders and decoders are built upon the original model and trained accordingly. We export the resulting model as a TensorFlow SavedModel for deployment with an additional placeholder to control the offloading partition during runtime. SavedModel is further converted to a TensorFlow Lite model for the deployment on the mobile side with Android OS.

We utilize TensorFlow Model Benchmark Tool [26] to profile the execution time of deep learning components on both the mobile device and the edge server. The benchmark tool has one warm-up run to initialize the model and then profiles all component execution times for 20 runs without internal delay. We then compute mean values as the profiled execution time.

At runtime, on the mobile side, we use TensorFlow Lite AAR hosted at JCenter as the mobile inference engine. The saved TensorFlow Lite model is preloaded to GPU or CPU. TCP link to the edge server is also built beforehand with general socket API. Before each inference, the partition decision maker with performance predictor generates the offloading partition, which is fed into the corresponding placeholder in the model. In order to reduce the TensorFlow Lite Engine overhead, ByteBuffers are used to feed input data and fetch compressed data to transfer. For offloading, data are transferred to the edge through a TCP link over Android APIs. With the runtime on edge, the TensorFlow serving [37] also preloads the model into corresponding GPU memory to reduce the initialization overhead. Once the serving receives the data with the selected partition point, it will feed data into the corresponding compressive decoder and the remaining partition of the neural network is executed to finish the inference. The inference result is transferred back to the mobile device via the same link.

6 EVALUATION

In this section, we conduct two sets of experiments. The first set evaluates the performance of deep compressive offloading as a general and (nearly) "lossless" offloading technique for deep learning services. We investigate the trade-off between the final inference accuracy and the compression ratio of offloaded data with different offloading points compared to state-of-the-art offloading techniques. The second set evaluates the DeepCOD system and verifies the efficacy of deep compressive offloading in real-world settings, when collaborating with other modules introduced in Section 4. Compared with state-of-the-art neural network offloading systems and on-device processing with model compression [30–32, 35, 48], DeepCOD demonstrates consistent and substantial reductions on end-to-end service latency under various real-world mobile, edge, and network configurations.

6.1 Applications & Datasets

All experiments are conducted with three of the most widely deployed applications, image recognition, speech recognition, and object detection in the video stream.

The image recognition service classifies an image into one of 1000 object categories. We take the widely-deployed deep learning model, ResNet-50 [24], as the image recognition service. The ResNet-50 model is pre-trained on the ImageNet ILSVRC2012 dataset [15].

The ImageNet dataset is a large scale hand-labeled image recognition dataset, containing 1.2 million samples in the training set and 50000 samples in the testing set. For all experiments related to the image recognition service, We will train all our learning-based offloading/compression models on the training set of ImageNet and test all the models and systems on the testing set of ImageNet.

The speech recognition service converts recorded utterances into English text transcriptions. We take the widely-deployed deep learning model, DeepSpeech [23], as the speech recognition service. The DeepSpeech model is pre-trained on the LibriSpeech dataset [38]. The LibriSpeech dataset is a large-scale corpus of reading English speech, containing 100 hours speech in the training set and 5 hours speech in the testing set. For all experiments related to the speech recognition service, We will train all our learning-based offloading/compression models on the training set of LibriSpeech and test all the models and systems on the testing set of LibriSpeech.

The object detection service identifies and locates objects within images and videos. We take YOLOv3 as the object detection module [41], pre-trained on the COCO dataset [33]. Due to space limitations, we illustrate the object detection service with a video demo [2]. In the demo, we deploy a YOLOv3-based object detection application on a Raspberry Pi with edge offloading. Compared to the traditional offloading with optimal partition, DeepCOD enjoys ×8 speed up in practice.

6.2 Baseline Systems

We compare the proposed deep compressive offloading (***DeepCOD***) with the other five baseline systems, including both offloading and on-device processing based approaches.

Offload-Intp: subsamples the resolution of offloading data before transmission, *e.g.*, spatial resolution for images and temporal resolution for speech, and interpolates with bilinear or bicubic method on the edge. Such design works for intermediate representation as well by subsampling and interpolating the non-feature dimensions. We then apply the same quantization and encoding step as DeepCOD, which improves the performance.

Offload-CS: compresses and reconstructs offloaded data based on compressive sensing. For the basis that ensures sparsity, we choose the best performing one from DCT and wavelet basis accordingly. We use iterative shrinkage-thresholding algorithm (ISTA) for reconstruction [8].

Offload-Lossy: integrates the state-of-the-art lossy offloading designs for deep learning services [31, 32]. One uses JPEG based compression, and the other uses quantization with Huffman Coding. We cheat in their favor by choosing a better-performing design from these two.

Offload-AE+: leverages the state-of-the-art deep learning data compression technique [35] to compress (including quantization and coding) and reconstruct the offloaded data based on the autoencoder structure, called Offload-AE. As mentioned in Section 2, the encoder of Offload-AE has a very complicated neural network structure, so we cheat in their favor by using the state-of-the-art model compression method [48] to compress the encoder with almost no performance loss, which significantly reduces the end-to-end latency. The resulted "cheated" model is called Offload-AE+.

[2]https://youtu.be/Acqm0iDnBWw

On-Device: is an on-device processing system without any offloading component. We compress the deep learning services using the state-of-the-art model compression method [48], and deploy the compressed models on the local device. We compare it with offloading systems when having limited network access.

We also include lossless offloading with no additional processing on offloaded data [30], denoted as ***Offload***, which helps us better to understand the speed-up and accuracy loss of these techniques. We do not illustrate the detailed results of previous works using neural networks as compressive sensing priors [11] due to their inefficiency. With a Titan V GPU, [11] takes more than one second to recover a 224×224 image, while the DeepCOD decoder takes around 2 ms. These works employ online iterative optimization, taking at least hundreds of gradient descend steps to obtain a reasonably good recovery, which is not suitable for practical offloading usage.

6.3 Network Latency vs. Accuracy Loss

In this subsection, we evaluate the tradeoff between the model inference accuracy and the averaged compression ratio of offloaded data. The On-Device baseline is not included here because it does not contain the offloading module.

We denote by t_{net} the time used for transferring data during offloading through a WiFi connection of 450Mbps bandwidth, calculated by deducting computation time on the mobile device and the edge server from the end-to-end offloading latency. Therefore, t_{net} is a round-trip time, including transferring offloaded data from the mobile device to the edge server and receiving results from the edge server to the mobile device. Since ResNet-50 adopts the block-based design, the sizes of intermediate representations only change after each block's first layer. Therefore, selecting other layers as offloading points in each block can only increase computation time with no data transfer time reduction with a high probability. The first layer in each block is thus chosen in Table 1. Since there are only five layers in the DeepSpeech model, the tradeoffs of all layers in DeepSpeech are shown in Table 2.

For both vision and speech tasks under all the offloading points, DeepCOD performs significantly better than all baseline algorithms. Compared with the lossless offloading, DeepCOD can reduce the size of offloaded data by a factor of 50 to 1000, and reduce the data transmission time by a factor of 10 to 100. At the same time, DeepCOD only suffers at most 1% accuracy loss.

As a comparison, all other non-learning-based baseline algorithms (including Offload-CS and Offload-Lossy) are at least twice slower than DeepCOD, while suffering more accuracy loss. The other deep-learning-based offloading system, Offload-AE+, is a competitive design in terms of the quality of compression of offloaded data. This is because the autoencoder neural network used [35] in that system is designed for data compression. Since Offload-AE+ has a more complicated encoder network, it should achieve the best tradeoff between the data compression ratio and the inference quality. However, DeepCOD still manages to beat Offload-AE+ due to the knowledge distillation component designed in Section 3.3. DeepCOD is designed not only for reducing the data reconstruction loss but also for improving the inference performance. In addition, we will show that the encoder of Offload-AE+ imposes a significant overhead on mobile devices, resulting in a large end-to-end latency.

Table 1: Tradeoff between model inference accuracy (Top-5 classification accuracy) and compression ratio of offloaded data for image recognition service with ResNet-50 model through WiFi connection with 450Mbps bandwidth.

	Input			Block1			Block2		
	Size	t_{net}	Acc	Size	t_{net}	Acc	Size	t_{net}	Acc
DeepCOD	**5.7KB (0.97%)**	**4.3ms (7.1%)**	**92.0% (-1.1%)**	**980B (0.12%)**	**2.8ms (3.5%)**	**92.0% (-1.1%)**	**245B (0.02%)**	**2.4ms (1.5%)**	**92.1% (-1.0%)**
Offload-CS	18.5KB (3.1%)	10.9ms (18.0%)	91.7% (-1.4%)	94.6KB (12.1%)	16.0ms (20.1%)	80.1% (-13.1%)	177KB (11.3%)	26.8ms (16.8%)	79.0% (-14.1%)
Offload-Intp	24.8KB (4.2%)	12.8ms (21.2%)	91.8% (-1.3%)	95.2KB (12.1%)	16.1ms (20.2%)	75.7% (-17.4%)	178KB (11.4%)	26.8ms (16.8%)	78.8% (-14.3%)
Offload-Lossy	19.5KB (3.3%)	12.3ms (20.3%)	91.7% (-1.4%)	94.7KB (12.1%)	16.0ms (20.1%)	77.1% (-16.0%)	178KB (11.4%)	26.8ms (16.8%)	79.0% (-14.1%)
Offload-AE+	12.2KB (2.1%)	7.8ms (12.9%)	92.0% (-1.1%)	14.7KB (1.9%)	8.5ms (10.7%)	92.0% (-1.1%)	17.2KB (1.1%)	8.6ms (5.4%)	92.1% (-1.0%)
Offload	588KB	60.5ms	93.1%	784KB	79.6ms	93.1%	1568KB	159.2ms	93.1%

	Block3			Block4		
	Size	t_{net}	Acc	Size	t_{net}	Acc
DeepCOD	**184B (0.02%)**	**2.3ms (2.9%)**	**92.0% (-1.1%)**	**123B (0.03%)**	**2.2ms (5.1%)**	**92.1% (-1.0%)**
Offload-CS	87.4KB (11.1%)	15.3ms (19.2%)	86.5% (-6.6%)	80.4KB (20.5%)	15.2ms (35.1%)	89.0% (-4.1%)
Offload-Intp	87.5KB (11.2%)	15.3ms (19.2%)	86.9% (-6.2%)	80.6KB (20.6%)	15.2ms (35.1%)	89.5% (-3.6%)
Offload-Lossy	87.4KB (11.1%)	15.3ms (19.2%)	86.6% (-6.5%)	80.8KB (20.6%)	15.2ms (35.1%)	89.1% (-4.0%)
Offload-AE+	12.3KB (1.6%)	7.7ms (9.7%)	92.0% (-1.1%)	6.7KB (1.7%)	5.3ms (12.2%)	92.1% (-1.0%)
Offload	784KB	79.6ms	93.1%	392KB	43.3ms	93.1%

Table 2: Tradeoff between Word Error Rate (WER) and compression ratio of offloaded data for speech recognition service with DeepSpeech model through WiFi connection with 450Mbps bandwidth.

	Input			Layer1			Layer2		
	Size	t_{net}	WER	Size	t_{net}	WER	Size	t_{net}	WER
DeepCOD	**17.9KB (1.5%)**	**8.8ms (8.2%)**	**0.085 (+0.003)**	**7.3KB (0.2%)**	**6.9ms (1.8%)**	**0.087 (+0.005)**	**5.5KB (0.1%)**	**4.3ms (1.1%)**	**0.085 (+0.003)**
Offload-CS	140KB (12.1%)	25.8ms (24.1%)	0.231 (+0.149)	551KB (11.5%)	50.6ms (13.4%)	0.144 (+0.062)	550KB (11.5%)	50.5ms (13.4%)	0.128 (+0.046)
Offload-Intp	142KB (12.3%)	25.9ms (24.2%)	0.262 (+0.18)	550KB (11.5%)	50.6ms (13.4%)	0.148 (+0.066)	550KB (11.5%)	50.6ms (13.4%)	0.313 (+0.231)
Offload-Lossy	144KB (12.4%)	25.9ms (24.2%)	0.264 (+0.182)	551KB (11.5%)	50.6ms (13.4%)	0.145 (+0.063)	551KB (22.4%)	50.6ms (13.4%)	0.135 (+0.053)
Offload-AE+	21.7KB (1.9%)	8.9ms (8.3%)	0.088 (+0.006)	45KB (0.9%)	23.3ms (6.2%)	0.09 (+0.008)	30KB (0.6%)	20.3ms (5.4%)	0.087 (+0.005)
Offload	1158KB	107.2ms	0.082	4800KB	377.9ms	0.082	4800KB	377.9ms	0.082

	Layer3			Layer4			Layer5		
	Size	t_{net}	WER	Size	t_{net}	WER	Size	t_{net}	WER
DeepCOD	**4.4KB (0.1%)**	**4.0ms (1.1%)**	**0.085 (+0.003)**	**3.7KB (0.08%)**	**3.8ms (1.0%)**	**0.084 (+0.002)**	**2.9KB (0.06%)**	**3.7ms (1.0%)**	**0.084 (+0.002)**
Offload-CS	552KB (11.5%)	50.7ms (13.4%)	0.145 (+0.063)	550KB (11.5%)	50.5ms (13.4%)	0.126 (+0.044)	550KB (11.5%)	50.5ms (13.4%)	0.131 (+0.049)
Offload-Intp	551KB (11.5%)	50.6ms (13.4%)	0.099 (+0.017)	550KB (11.5%)	50.5ms (13.4%)	0.098 (+0.016)	550KB (11.5%)	50.5ms (13.4%)	0.191 (+0.109)
Offload-Lossy	551KB (11.5%)	50.6ms (13.4%)	0.159 (+0.077)	551KB (11.5%)	50.6ms (13.4%)	0.119 (+0.037)	551KB (11.5%)	50.6ms (13.4%)	0.133 (+0.051)
Offload-AE+	25.5KB (0.5%)	16.3ms (4.3%)	0.087 (+0.005)	21KB (0.4%)	15.4ms (4.1%)	0.087 (+0.005)	16.5KB (0.3%)	8.4ms (2.2%)	0.086 (+0.004)
Offload	4800KB	377.9ms	0.082	4800KB	377.9ms	0.082	4800KB	377.9ms	0.082

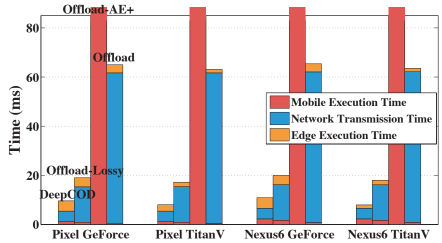


Figure 6: End-to-end offloading latency of image recognition through WiFi with 450Mbps bandwidth.

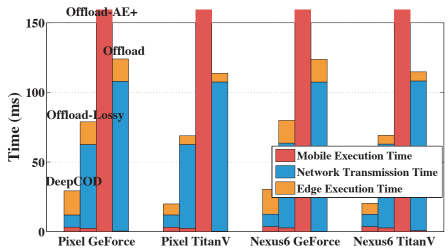


Figure 7: End-to-end offloading latency of speech recognition through WiFi with 450Mbps bandwidth.

Moreover, only learning-based offloading techniques, *i.e.*, DeepCOD and Offload-AE+, work well for compressing intermediate representations. It is not the main compression algorithms in the baselines but rather the entropy encoding components that contribute to the offloaded data compression if we want to have a similar prediction performance as DeepCOD. The reason is that compressive sensing, lossy encoding, and interpolation have assumptions on offloaded data, *i.e.*, sparsity on a particular domain, or spatial/temporal continuity. All these assumptions fail to work properly on intermediate representations in neural networks. Since developers cannot investigate the best data assumptions for all possible applications, the evaluation suggests that DeepCOD is a better solution that provides a flexible, time-efficient, and almost lossless offloading design for deep learning services.

6.4 DeepCOD: End-to-End Latency

Previous experiments have shown that deep compressive offloading is a flexible and effective solution for reducing the network transmission time with almost no accuracy loss. In this subsection, we test our offloading system, DeepCOD. We compare DeepCOD to the state-of-the-art lossless and lossy neural network offloading systems, Offload, Offload-Lossy, and Offload-AE+ [30–32, 35] with the same data compression pipelines as described in Section 6.3. We do not show the evaluation results of Offload-CS, because the reconstruction in compressive sensing uses slow iterative algorithms, which greatly increases the end-to-end offloading latency. We apply the same dynamic offloading partitioning design, as described in Section 3.4.2 and 4.2, to all baseline systems.

We measure end-to-end offloading latency (including mobile-side execution time, network transmission time, and edge-side execution time) of the offloading system for the image and speech recognition services with the same set of potential offloading points, as shown in Table 1 and 2. We conduct experiments with various mobile, edge, and network configurations to verify whether DeepCOD can enjoy consistent and significant speed-ups under various mobile-edge-network settings. In the evaluation, we have two mobile devices: Google Pixel (Pixel) and Nexus 6 (Nexus6); two GPUs on edge: Nvidia Titan V (TitanV) and Nvidia GeForce GTX Titan X (GeForce); two wireless connections: WiFi with 450Mbps bandwidth and LTE. In all experiments here, we make sure that *all offloading systems are not allowed to reduce the accuracy or to increase the error by more than 5% . DeepCOD achieves the best inference performance (at 1% loss) except for the lossless system, Offload.*

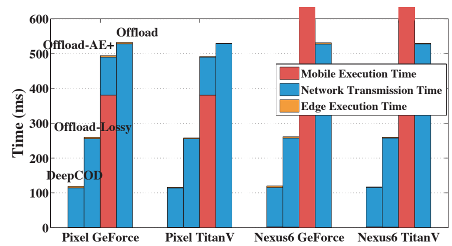


Figure 8: End-to-end offloading latency of image recognition through LTE.

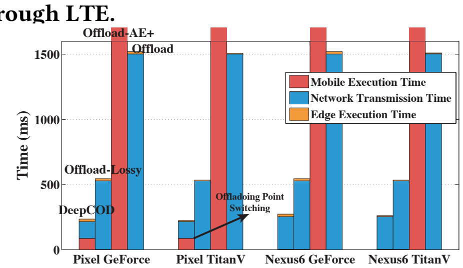


Figure 9: End-to-end offloading latency of speech recognition through LTE.

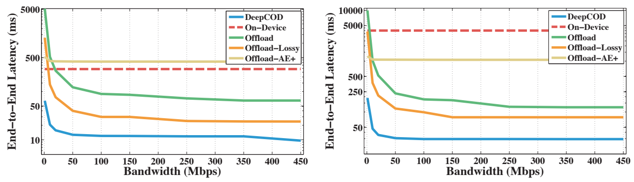
The evaluation results are illustrated in Figure 6 to 9. DeepCOD can consistently reduce end-to-end offloading latency of Offload and Offload-Lossy by a factor from 5 to 7.5 and a factor from 2 to 3.5 respectively. One possible concern about DeepCOD is that reducing network transmission time will cause computational overhead on the mobile device or edge server. The overheads of compressive encoder and decoder are limited, even though we have not made special attempts to improve their time efficiency. The compressive encoder is computationally efficient by nature, imposing only a small overhead on mobile phones. By contrast, Offload-AE+ has a relatively high end-to-end latency. Even when we already compressed the encoder of Offload-AE+,it still takes more than 300ms and 500ms to encode the image, and takes more than 1s and 2s to encode the voice features on Pixel and Nexus6. In addition, the dynamic offloading partitioning estimates the computation and network conditions, and selects the best offloading point for achieving shorter end-to-end latency as shown in Figure 9.

6.5 Energy Consumption

This subsection measures our local device's energy consumption and ensures that the encoding part of DeepCOD does not impose a large energy overhead locally. Without loss of generality, Nexus 6 is the local device. Since Offload-AE+, Offload, and On-Device have a large computational overhead on local devices, we do not include these baselines in this experiment. The partition decision maker, following the procedure described in Section 3.4, is operated under a standard 450Mbps WiFi connection. We set up the image recognition service (as mentioned in Section 6.1) and estimate the average encoding energy consumption on Nexus 6 by PowerTutor [52] with 1,000 offloading trials. As shown in Table 3, DeepCOD has little energy overhead compared to other baselines.

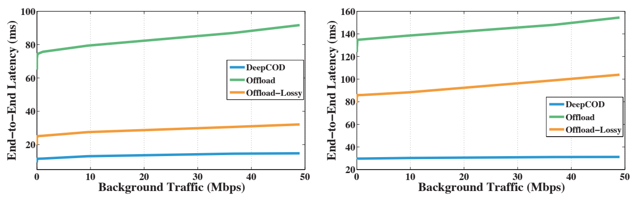
Table 3: Energy overhead (mJ) of encoding.

	DeepCOD	Offload-Intp	Offload-Lossy
Image	28	27	41
Speech	38	35	25



(a) Image recognition **(b) Speech recognition**

Figure 10: End-to-end offloading latency under various bandwidth conditions (y-axis log scale)



(a) Image recognition **(b) Speech recognition**

Figure 11: End-to-end offloading latency under various background network traffic.

6.6 Impact of Bandwidth & Background Traffic

In this subsection, we evaluate offloading systems under different network conditions by either limiting the bandwidth for the extreme conditions of the wireless link or generating background traffic for the normal usage. We choose the Google Pixel and edge server with GeForce GTX Titan X GPU.

We uniformly select ten WiFi bandwidth settings between 1 Mbps to 450 Mbps. The measured end-to-end latencies of image and speech recognition services are illustrated in Figure 10, where DeepCOD significantly mitigates the impact of low bandwidth, reducing overall latency by 6× to 35×. The reasons are twofold. On the one hand, DeepCOD transfers data of a much smaller size, thereby withstanding lower network bandwidth. On the other hand, as shown in Table 1 and 2, deep compressive offloading works for intermediate representations as well. With the help of dynamic offloading partitioning, DeepCOD can decide to do a little more work locally, which significantly reduces network latency when network bandwidth is scarce. For example, DeepCOD can maintain a 62ms and 188ms end-to-end latency for image and speech recognition services, respectively, with only 1Mbps bandwidth, which is faster than all the baseline systems. Besides, DeepCOD is the only offloading system that can beat On-Device, the model compression technique, all the time.

We gradually increase the background network traffic with a network traffic generator from 0 to 50Mbps. We exclude On-Device and Offload-AE+ from this experiment because the end-to-end latencies of these two systems are significantly larger than others under normal traffic conditions. As shown in Figure 11, the performance (end-to-end latency) of DeepCOD remains almost the same when we increase the background network traffic.

Also, as the background network traffic increases, DeepCOD has a lower growth rate of latency than the other two baseline systems.

Table 4: Training overhead of DeepCOD.

	DeepCOD	Original
ImageNet (1.3M Pictures)	4.8 ± 0.8h	134h
LibriSpeech (300h Speech)	1.6 ± 0.3h	23h

6.7 Training Overhead

Another possible concern about DeepCOD is the training overhead of the compressive encoder and decoder because adding a new offloading point needs us to train a new set of encoder and decoder. In this subsection, we demonstrate the training overhead of deep compressive encoder and decoder compared to the original neural network's training time. All models are trained with a Nvidia Titan V GPU. The training overhead is shown in Table 4. When compared with the training time of the original deep learning models, DeepCOD has a small training overhead. In addition, deep compressive offloading is agnostic to hardware and software. Therefore, the compressive offloading encoder and encoder can be trained only once on the cloud with distributed (and multi-GPU) training for further reducing the training overhead [20].

7 CONCLUSION

In this paper, we proposed deep compressive offloading, a general-purpose offloading framework to reduce end-to-end offloading latency with almost no accuracy loss. By taking the computational capabilities of local and edge devices into consideration, we design an asymmetric encoder-decoder structure that integrates the compressive sensing theory with deep neural networks. Therefore, deep compressive sensing can be trained based on theoretical guidelines to ensure a recovery guarantee. A real-world system, DeepCOD, is designed and implemented to provide the deep compressive offloading function to intelligent sensing and recognition services. Compared with state of the art, DeepCOD can consistently reduce offloading latency by a factor of 2 to 35 with at most 1% accuracy loss under various mobile-edge-network configurations.

DeepCOD is designed to be an application-agnostic offloading system. Given a set of hyper-parameters (potential offloading points and compression ratios), we can construct a DeepCOD offloading system for a wide range of deep vision, speech, and sensing applications based on Sections 3 and 4. We can further extend DeepCOD to non-deep-learning applications if we are able to represent the offloading data into a set of tensors. However, much more future research is needed. Automatically selecting the optimal offloading points and compression ratios are not easy tasks by themselves. In addition, we need to generalize the theory and design of DeepCOD to arbitrary offloading points and compression ratios with low cost, enabling quality-aware offloading scheduling in the future compressive offloading system. DeepCOD is also agnostic to the domain knowledge of sensing signals. More work is needed to leverage those domain-specific spatial-temporal dependencies for a better accuracy-efficiency tradeoff.

ACKNOWLEDGMENTS

We sincerely thank for the invaluable comments from anonymous shepherding and reviewing. Research reported in this paper was sponsored in part by DARPA award W911NF-17-C-0099, DTRA award HDTRA1-18-1-0026, and the Army Research Laboratory under Cooperative Agreements W911NF-17-2-0196.

REFERENCES

[1] K. M. Abadir and J. R. Magnus. *Matrix algebra*, volume 1. Cambridge University Press, 2005.

[2] E. Agustsson, F. Mentzer, M. Tschannen, L. Cavigelli, R. Timofte, L. Benini, and L. V. Gool. Soft-to-hard vector quantization for end-to-end learning compressible representations. In *Advances in Neural Information Processing Systems*, pages 1141–1151, 2017.

[3] E. Agustsson, M. Tschannen, F. Mentzer, R. Timofte, and L. V. Gool. Generative adversarial networks for extreme learned image compression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 221–231, 2019.

[4] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[5] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] R. G. Baraniuk. Compressive sensing. *IEEE signal processing magazine*, 24(4), 2007.

[8] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

[9] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[10] S. Bhattacharya and N. D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 176–189. ACM, 2016.

[11] A. Bora, A. Jalal, E. Price, and A. G. Dimakis. Compressed sensing using generative models. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 537–546. JMLR. org, 2017.

[12] A. Brock, J. Donahue, and K. Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

[13] E. Candes, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *arXiv preprint math/0409186*, 2004.

[14] I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, 57(11):1413–1457, 2004.

[15] 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*, pages 248–255. Ieee, 2009.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[17] A. E. Eshratifar and M. Pedram. Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 111–116. ACM, 2018.

[18] T. Goldstein and S. Osher. The split bregman method for l1-regularized problems. *SIAM journal on imaging sciences*, 2(2):323–343, 2009.

[19] G. H. Golub and H. A. Van der Vorst. Eigenvalue computation in the 20th century. In *Numerical analysis: historical developments in the 20th century*, pages 209–239. Elsevier, 2001.

[20] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[21] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[22] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[23] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[24] 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*, pages 770–778, 2016.

[25] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[26] https://github.com/tensorflow/tensorflow/. Tensorflow benchmark tool. tree/master/tensorflow/tools/benchmark.

[27] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[28] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[29] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (ToN)*, 22(1):326–340, 2014.

[30] 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 *ACM SIGARCH Computer Architecture News*, volume 45, pages 615–629. ACM, 2017.

[31] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. In *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2018.

[32] 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. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 671–678. IEEE, 2018.

[33] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[34] L. Liu, H. Li, and M. Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *MobiCom*. ACM, 2019.

[35] F. Mentzer, E. Agustsson, M. Tschannen, R. Timofte, and L. Van Gool. Conditional probability models for deep image compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4394–4402, 2018.

[36] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.

[37] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[38] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[39] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1421–1429. IEEE, 2018.

[40] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[41] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[42] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[43] L. Theis, W. Shi, A. Cunningham, and F. Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.

[44] G. Wade. *Signal coding and processing*. Cambridge university press, 1994.

[45] Y. Weiss, H. S. Chang, and W. T. Freeman. Learning compressed sensing. In *Snowbird Learning Workshop, Allerton, CA*. Citeseer, 2007.

[46] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web*, pages 351–360, 2017.

[47] S. Yao, A. Piao, W. Jiang, Y. Zhao, H. Shao, S. Liu, D. Liu, J. Li, T. Wang, S. Hu, et al. Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks. In *The World Wide Web Conference*, pages 2192–2202, 2019.

[48] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 278–291. ACM, 2018.

[49] S. Yao, Y. Zhao, A. Zhang, S. Hu, H. Shao, C. Zhang, L. Su, and T. Abdelzaher. Deep learning for the internet of things. *Computer*, 51(5):32–41, 2018.

[50] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher. Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, page 4. ACM, 2017.

[51] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena. Self-attention generative adversarial networks. *arXiv preprint arXiv:1805.08318*, 2018.

[52] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114, 2010.