TENSOR COMPLETION METHODS
FOR COLLABORATIVE INTELLIGENCE

by

Lior Bragilevsky

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science in the School of Engineering Science Faculty of Applied Science

© Lior Bragilevsky 2020
SIMON FRASER UNIVERSITY Spring 2020

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.
Approval

Name: Lior Bragilevsky
Degree: Master of Applied Science (Engineering Science)
Title: Tensor Completion Methods for Collaborative Intelligence
Examinining Committee: Chair: Andrew Rawicz
Professor
Ivan V. Bajić
Senior Supervisor
Professor
Mirza Faisal Beg
Committee Member
Professor
Jie Liang
Internal Examiner
Professor
Date Defended/Approved: March 20, 2020
Abstract

In the race to bring Artificial Intelligence (AI) to the edge, collaborative intelligence has emerged as a promising way to lighten the computation load on edge devices that run applications based on Deep Neural Networks (DNNs). Typically, a deep model is split at a given layer into edge and cloud sub-models. The deep feature tensor produced by the edge sub-model is transmitted to the cloud, where the remaining computationally intensive workload is performed by the cloud sub-model. The communication channel between the edge and cloud is imperfect, which will result in missing data in the deep feature tensor received at the cloud side, an issue that has mostly been ignored by existing literature on the topic. In this thesis I study four methods for recovering missing data in the deep feature tensor. Three of the studied methods are existing, generic tensor completion methods, and are adapted here to recover deep feature tensor data, while the fourth method is newly developed specifically for deep feature tensor completion. Simulation studies show that the new method is $3 - 18$ times faster than the other three methods, which is an important consideration in collaborative intelligence. For VGG16’s sparse tensors, all methods produce statistically equivalent classification results across all loss levels tested. For ResNet34’s non-sparse tensors, the new method offers statistically better classification accuracy (by $0.25\% - 6.30\%$) compared to other methods for matched execution speeds, and second-best accuracy among the four methods when they are allowed to run until convergence.

Keywords: Tensor completion, missing data imputation, tensor decomposition, tensor reconstruction, collaborative intelligence, deep feature transmission, artificial intelligence, deep learning, machine learning, cloud computing, adaptive algorithms, pre-trained models.
Acknowledgements

All it took was 2 years of grueling research into the unknown, but now I can finally present my master’s thesis. Now, it is time to acknowledge those individuals that made this achievement possible.

To my senior supervisor - Dr. Ivan Bajić - thank you for guiding me through this difficult, but rewarding, process. Your insight during the critical moments of my research resulted in an academic journal publication (my first), which is a very important accomplishment and a big stepping stone in my life.

To my thesis committee members - Dr. Jie Liang and Dr. Mirza Faisal Beg - thank you for thoroughly analysing my thesis paper and listening to my defence presentation. I believe that your constructive criticism has added significant value to this document.

To my close family members - Igor, Anna, and Dan - in your own unique way each one of you has supported me throughout my undergraduate thesis journey and somehow provided even more support through this one. Words cannot describe how grateful I am for that.

Lastly, to my best friend - Bamba - thank you for helping me push through the truly tough times with your playfulness and joyful attitude.
Table of Contents

Approval ii
Abstract iii
Acknowledgements iv
Table of Contents v
List of Tables vii
List of Figures viii
List of Algorithms x
List of Symbols xi

1 Introduction and Preliminaries 1
  1.1 Motivation ........................................ 2
  1.1.1 Federated Learning ............................. 3
  1.2 Deep Learning Overview ........................... 3
  1.2.1 Loss Function, Optimizer, and Backpropagation ...... 4
  1.2.2 Overfitting ...................................... 5
  1.2.3 Convolution, Activation, and Pooling Layers ........ 6
  1.2.4 Classification Stage - Output Layer ................. 8
  1.3 Background Knowledge .............................. 8
  1.3.1 Tensor Folding and Unfolding ...................... 9
  1.3.2 Singular Value Decomposition and Shrinkage .......... 9
  1.3.3 Canonical Polyadic Decomposition ................... 11
  1.3.4 Tensor Data Packetization and Transmission ........ 12
  1.4 Thesis Preview .................................... 13

2 General Tensor Completion Algorithms 14
  2.1 Simple Low Rank Tensor Completion (SiLRTC) ............ 14
  2.2 High Accuracy Low Rank Tensor Completion (HaLRTC) ...... 15
List of Tables

Table 4.1  VGG16 classification accuracy results. Among the last four columns, the first two show the results with default settings, and the last two show the results with matched execution speed. . . . . . . . . . . . . 33
Table 4.2  VGG16 statistical significance results. The middle three columns show $p$-values of the 2-sided Welch’s t-test for pairwise comparison of accuracies with default settings, while the last three columns show $p$-values for the case when the execution speeds are matched. . . . . . . . . . . . . 34
Table 4.3  Interesting examples from VGG16 experiments . . . . . . . . . . . . . . . . . . . . . . 35
Table 4.4  $L_1$ norm difference values for channel #257 . . . . . . . . . . . . . . . . . . . . . . 36
Table 4.5  ResNet34 classification accuracy results. Among the last four columns, the first two show the results with default settings, and the last two show the results with matched execution speed. . . . . . . . . . . . . 40
Table 4.6  ResNet34 statistical significance results. The middle three columns show $p$-values of the 2-sided Welch’s t-test for pairwise comparison of accuracies with default settings, while the last three columns show $p$-values for the case when the execution speeds are matched. . . . . . . . . . . . . 41
Table 4.7  Interesting examples from ResNet34 experiments . . . . . . . . . . . . . . . . . . . . . 42
Table 4.8  $L_1$ norm difference values for channel #73 . . . . . . . . . . . . . . . . . . . . . . 43
List of Figures

Figure 1.1 Collaborative Intelligence Blueprint ........................................ 1
Figure 1.2 Deep Learning Performance .................................................. 3
Figure 1.3 Loss Function Optimizer (Stochastic Gradient Descent) ............... 5
Figure 1.4 Convolutional Layer with $3 \times 3$ Filter ................................. 6
Figure 1.5 Deep Feature Tensors from Convolutional Layer ....................... 7
Figure 1.6 Pooling Layer ......................................................................... 8

Figure 3.1 Value Distribution of Weight Matrices .................................... 21
Figure 3.2 Visualization of Weight Matrices ........................................... 23
Figure 3.3 Weight Configuration Reconstruction Analysis for Various Packet Loss Values .................................................. 25

Figure 4.1 Test conditions: NL (no loss, baseline), NC (no tensor completion), TC (tensor completion) ................................................................. 28
Figure 4.2 Average Frobenius norm of the differences between completed tensors in neighboring iterations ..................................................... 29
Figure 4.3 VGG16 model split at layer “block4_pool” ................................ 30
Figure 4.4 Percentage of zero elements in the VGG16 tensor as a function of packet loss ................................................................. 31
Figure 4.5 Execution speed of tensor completion algorithms - VGG16 ........ 32
Figure 4.6 ResNet34 model split at layer “add_7” ..................................... 37
Figure 4.7 Percentage of zero elements in the ResNet34 tensor as a function of packet loss ................................................................. 38
Figure 4.8 Execution speed of tensor completion algorithms - ResNet34 ........ 39

Figure A.1 Ground Truth Images .............................................................. 50
Figure A.2 Channel #257 of Figure A.1a (VGG16), ordered left $\rightarrow$ right: NL, NC, SiLRTC, HaLRTC, FCP, ALTeC .............................................. 51
Figure A.3 Channel #257 of Figure A.1b (VGG16), ordered left $\rightarrow$ right: NL, NC, SiLRTC, HaLRTC, FCP, ALTeC .............................................. 52
Figure A.4 Channel #73 of Figure A.1c (ResNet34), ordered left $\rightarrow$ right: NL, NC, SiLRTC (cont’d next page) .............................................. 53
## List of Algorithms

1. Simple Low Rank Tensor Completion (SiLRTC) ............................................. 15
2. High Accuracy Low Rank Tensor Completion (HaLRTC) .......................... 16
3. Fused Canonical Polyadic (FCP) ................................................................. 17
4. Adaptive Linear Tensor Completion (ALTeC) .......................................... 20
List of Symbols

\( \mathcal{X} \) Bold calligraphic letters \( \rightarrow \) tensors

\( \mathbf{X} \) Bold uppercase non-italic letters \( \rightarrow \) matrices

\( \mathbf{x} \) Bold lowercase non-italic letters \( \rightarrow \) vectors

\( x \) or \( X \) Italic letters \( \rightarrow \) scalars

\( \mu_{\mathbf{x}} \) Mean or average (Expected Value) of vector \( \mathbf{x} \)

\( \sigma_{\mathbf{x}} \) Standard deviation of vector \( \mathbf{x} \)

\( \circ \) Vector outer product

\( \odot \) Khatri-Rao product of two matrices

\( \otimes \) Kronecker product of two matrices/vectors

\( \odot \) Hadamard product of two matrices/vectors
As the Internet of Things (IoT) infrastructure gets deployed, there will be many opportunities for innovative applications that make use of the newly available sensor data. Many of these new applications will rely on DNNs to process the sensor data in order to produce useful predictions and analytics. One current research direction is towards miniaturization of DNNs, such that they can be implemented at (or even near) the edge sensors, with limited computation and energy resources. While such DNNs may be sufficient for certain applications, resources available in the cloud will always be able to support larger and more sophisticated models than those that could possibly be deployed at the edge. Hence, cloud-based analytics will remain essential even if some of the AI-based processing gets moved to the edge.

Collaborative Intelligence (CI) [1–4] is an AI deployment strategy that leverages both edge-based and cloud-based resources to make DNN computing faster and more efficient. In CI, a deep model is split into an edge sub-model and a cloud sub-model. For example, as shown in Figure 1.1 an edge sub-model may consist of the initial \( m \) layers of a DNN, while the cloud sub-model is made up of the remaining DNN layers.

Figure 1.1: Collaborative Intelligence Blueprint [1]
When an input signal is captured by an edge sensor, the edge sub-model processes the signal and produces a tensor of deep features, which is then transmitted to the cloud for subsequent processing by the cloud sub-model. Due to the imperfect channel between the edge and the cloud, deep feature tensor data may be damaged or missing, much like data transmitted over the Internet. Hence, error control schemes/methods must be deployed to achieve seamless operation of edge and cloud sub-models in AI.

In this thesis, I study four methods for recovery of missing data in a deep feature tensor. Three of these methods come from existing literature: Simple Low Rank Tensor Completion (SiLRTC) [5], High Accuracy Low Rank Tensor Completion (HaLRTC) [5], and the recent Fused Canonical Polyadic (FCP) decomposition [6]. All three are general tensor completion methods based on the low-rank tensor assumption. I explain how these methods can be adapted to reconstruct deep features in corrupt tensors using tensor completion. The fourth method is simple and tailor-made for recovery of missing data (i.e., imputation) in deep feature tensors. It is adaptive and linear in nature. Specifically, missing data in the tensor are recovered as a linear combination of other, available data, so the tensor rank is not increased. Due to these attributes, I call it Adaptive Linear Tensor Completion (ALTeC). A summary of this thesis has been published in [7].

1.1 Motivation

There are numerous applications in the real world where privacy is a concern, such as video surveillance [8, 9]. Imagine installing a security camera in your home and finding out that someone gained access to it without your knowledge. How would such a situation make you feel? A complete stranger knows exactly what you look like, what you do, and where you are throughout the day. What if your surveillance camera could extract all of its needed information for a latent space of deep features based on the input image? It is clear that such a system is much more secure due to its privacy preservation. In essence, the input images that the camera captures could be transformed into anything, as long as the decoding system can correctly predict what the input is.

This is where collaborative intelligence can play a big role. Due to its transmission of deep feature tensors from the edge to the cloud, an extra privacy layer is ensured which prevents malicious users from capitalizing on your system’s weaknesses. Even if they gain access to the deep feature tensors, they would not see anything compromising, unlike in a surveillance camera system that does not incorporate collaborative intelligence.

In addition to the privacy aspect, since the deep feature tensors are malleable, one can apply compression to them and greatly decrease the required bit-rate [3]. Deep feature compression is an active area of research [10–15]. Compressing the deep feature tensors would reduce their “volume” and thus their transmission latency, as well as the overall inference latency, can be reduced [16]. Note that compression essentially adds an extra
security layer, to the above mentioned inference process, as the deep feature tensors are further “deformed”.

1.1.1 Federated Learning

Federated Learning, where a global model is incrementally updated by many users, can be thought of as an encapsulation of collaborative intelligence [17, 18]. Here each user “downloads” the model from the cloud onto their edge device, where improvements are made from data present on the edge device, and a summary of these improvements is then sent back to the global model on the cloud. As a result, the overall (global) model is able to significantly improve with lower latency and less power consumption, while maintaining each individual’s privacy as their edge device is not seen by the rest. Additionally, each user benefits from this process as their edge device’s model is also improved. Rather than improving the model solely on the edge device, a user can implement collaborative intelligence methods to further benefit from the above-mentioned properties.

1.2 Deep Learning Overview

Deep learning architectures began to gain popularity once it was understood that they are able to outperform conventional machine learning models when trained on a sufficiently large data-set, as shown in Figure 1.2.

![Figure 1.2: Deep Learning Performance [19]](image)

In the figure it is clear that both deep and machine learning architecture types have similar performance until a certain data-set size threshold is met. Once that threshold is
passed, the prediction accuracy of deep learning models continues to increase, while that of the shallow machine learning models approaches its asymptotic maximum, for larger data-set sizes. This phenomena occurs due to the extreme learning potential of deep learning models, as a result of their many layers and parameters. Their counterparts, shallow structured machine learning models, contain only a few layers and fewer trainable parameters. As a result, machine learning models often have a hard limit on their learning capacity, regardless of the size of the training data-set.

The many layers of a deep learning network are made up of nodes, called neurons, forming what is known as a neural network that closely resembles the human brain [19]. The neurons in a given layer are able to communicate with other neurons in neighboring layers through weighted connections, forming many different input to output signal communication routes. In general, a labelled input (supervised learning) is provided to the input layer, whose neurons are then connected to the next layer’s neurons with corresponding weights. The layers between the input and output are referred to as hidden layers. The neurons in hidden layers also connect to one another until the output layer is reached. During the training stage these weighted connections between neurons are continuously updated, using a well known method called backpropagation [20], to determine the optimal weights according to some loss function criteria. These optimal weights allow various input signals to traverse through the hidden layers towards the output layer in such a way that the model can very accurately predict which class a given signal belongs to. This means that during the test stage, when the trained model is fed with a test data-set that it has never seen previously, it is able to generate predictions that closely match the desired target output.

1.2.1 Loss Function, Optimizer, and Backpropagation

To ensure that the weighted connections of the neural network are optimal, the error between its predictions and desired output is recorded and minimized. This error minimization is achieved with the help of an optimization algorithm (optimizer) that attempts to find the minimum of some loss function (or surface in the more complex cases), as shown in Figure 1.3. Stochastic Gradient Descent [21] is a common and very popular optimizer for neural networks in a wide range of applications, with many other optimizers being derived from it.

The rate at which the optimizer finds the global minimum of a loss function heavily depends on the chosen initialization state. In general, many loss functions contain local minima, which can cause an optimizer to converge even when the error is not at the lowest point. Due to this, the optimizers contain many hyper-parameters, such as learning rate (step size) and momentum, that increase the chances of finding a global minimum.

The entire training data-set is fed into the network many times to iteratively update the connection weights through a process known as backpropagation [19,22]. At each iteration, the optimizer attempts to go down the error surface. Thus, one can expect that the loss
function error starts relatively high, but decreases and eventually plateaus as the number of times (iterations) that the model sees the training data-set increase.

1.2.2 Overfitting

Due to the complexity of deep learning models, if not properly designed and trained, their brain like structure tends to memorize the training data-set rather than learn meaningful features from it [19]. This produces exceptional results (very low error) during the training stage, but fails to generalize to a test data-set of never before seen inputs. The shallow structured machine learning counterparts did not suffer from such an issue, making them more appealing to researchers.

With computational improvements, new regularization methods emerged [23]. These include dropout and data augmentation, and are used to prevent overfitting [19]. Dropout randomly zeros out neurons in a given layer, essentially preventing the weight updates for connections to that neuron from any layer. This minimizes the ability of the neural network to heavily rely on a given neuron and encourages the participation of other neurons for each data-set input - discouraging any memorization tactics. Data augmentation increases the variety of the training data-set by adding random rotations, zooming, scaling, flipping, etc. Again, this allows the model to better generalize to new inputs, as it won’t get lost if a new input matches ones that it saw previously but was slightly different.
Often times, a challenge that deep learning researchers face is determining the size of a training data-set that is required for optimal results (with each deep learning model of varying complexity), without overfitting [24].

1.2.3 Convolution, Activation, and Pooling Layers

With the increased interest in deep learning models, Convolutional Neural Networks (CNN) were developed for object classification, detection, and segmentation applications [19]. The CNN architecture consists of convolutional, activation, and pooling blocks connected in a “feedforward” configuration, forcing information to propagate in the same direction from the input layer towards the output layer.

Convolutional layers apply many filters to their inputs. These filters are often much smaller in size than the input dimensions. The convolutional layer with many filters creates an output with a corresponding number of channels, or feature maps, that contain the relevant learnt features of the input. As a result, the model is able to analyse the various features of all the input data both broadly (large filters) and in fine detail (small filters). This gives deep learning models the ability to “see” the entire input tensors by squeezing their spatial dimension while simultaneously expanding the channel dimension. An example of a convolutional layer applying a $3 \times 3$ filter to its input is shown in Figure 1.4.

![Convolutional Layer with 3 x 3 Filter](image)

**Figure 1.4: Convolutional Layer with 3 x 3 Filter [25]**

The deep feature tensors produced by the above convolutional layer are illustrated in Figure 1.5. In the figure, the convolutional layer used 5 filters of size $3 \times 3$ on the $32 \times 32$ input layer to produce a 5 channel deep feature tensor. Each channel has size $30 \times 30$ due to the trimming effect caused by the applied filters. That is, only the outputs produced by the full overlap of the filter’s kernel and the input are considered valid, thereby reducing the spatial dimension of the feature map compared to the input.
Activation layers, also known as the non-linearity layers [19], are used to transform the values of the linear portion of the layer. Generally, the Rectified Linear Units (ReLU) activation function is used in convolutional layers to clip negative values to 0 and linearly map any positive value. The ReLU activation function is given by:

$$\text{ReLU}(x) = \max(0, x). \quad (1.1)$$

Likewise, the Sigmoid activation function is often applied at the output layer to bound the output in the range $[0, 1]$. This produces probabilistic prediction results, which are very useful for classification applications. The Sigmoid activation function is given by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (1.2)$$

Thus, as the signal progresses through the model, the values in deep feature channels are adjusted by the activation functions to allow for successful training to occur and meaningful predictions to be made.

Pooling layers are used to further reduce the spatial dimension of the deep feature channels produced by convolutional layers, as shown in Figure 1.6. Different kinds of pooling layers may be used, with max pooling and average pooling being the most popular. This ultimately allows the model to be invariant to augmented input data, allowing it to better generalize to other test data-sets.
1.2.4 Classification Stage - Output Layer

Lastly, in classification models, the classification stage generates the predictions of the model. The deep feature tensors produced by the final pooling layer (after the final convolutional layer) is flattened into a $N \times 1$ feature vector. This vector of neurons is then fully connected to future layers. As the number of connections is very large, often times there are a few fully connected layers between the feature vector and the output layer that utilize dropout in an attempt to reduce overfitting.

In many image classification applications where deep learning models are deployed, the majority of the training parameters come from the fully connected layer due to the sheer number of weight connections between neurons. Often times, this creates such a computational burden on CPU based machines that researchers are forced to use GPUs instead to increase the given model’s training speed. Current edge devices, like mobile phones, do not possess the necessary hardware to make the training of large deep learning models feasible. However, inference on edge devices is already practically possible, especially smaller models. Collaborative intelligence allows a large model to be split between the edge and the cloud, so that the edge device can still utilize the inference accuracy of the large model, without the full computational cost.

1.3 Background Knowledge

This section illustrates several tensor and matrix operations that are used throughout the thesis. In terms of notation, bold calligraphic letters ($\mathbf{X}$) will denote tensors, bold uppercase non-italic letters ($\mathbf{x}$) will denote matrices, bold lowercase non-italic letters ($\mathbf{x}$) will denote vectors, and italic letters ($x$ or $X$) will denote scalars.
1.3.1 Tensor Folding and Unfolding

Tensor unfolding is a structured mapping from a tensor (3D) to a matrix (2D). A tensor can be unfolded along any of its dimensions. For example, consider a 3D tensor $\mathbf{X}$ with two channels:

$$
\mathbf{X} = \left(\begin{array}{ccc}
0 & 2 & 4 \\
6 & 8 & 10 \\
12 & 14 & 16
\end{array}\right) \begin{bmatrix}
1 & 3 & 5 \\
7 & 9 & 11 \\
13 & 15 & 17
\end{bmatrix},
$$

(1.3)

where the left matrix (even integers) is the first channel and the right matrix (odd integers) is the second channel. Then unfolding along the x-, y-, and z-axis (axis 0, 1, and 2) produces matrices $\mathbf{X}_0$, $\mathbf{X}_1$, and $\mathbf{X}_2$, respectively, as shown in Equation 1.4.

$$
\mathbf{X}_0 = \text{unfold}(\mathbf{X}, 0) = \begin{bmatrix}
0 & 2 & 4 & 1 & 3 & 5 \\
6 & 8 & 10 & 7 & 9 & 11 \\
12 & 14 & 16 & 13 & 15 & 17
\end{bmatrix}
$$

$$
\mathbf{X}_1 = \text{unfold}(\mathbf{X}, 1) = \begin{bmatrix}
0 & 6 & 12 & 1 & 7 & 13 \\
2 & 8 & 14 & 3 & 9 & 15 \\
4 & 10 & 16 & 5 & 11 & 17
\end{bmatrix}
$$

(1.4)

$$
\mathbf{X}_2 = \text{unfold}(\mathbf{X}, 2) = \begin{bmatrix}
0 & 6 & 12 & 2 & 8 & 14 & 4 & 10 & 16 \\
1 & 7 & 13 & 3 & 9 & 15 & 5 & 11 & 17
\end{bmatrix}
$$

Once a 3D tensor is unfolded into a 2D matrix, then matrix operations such as Singular Value Decomposition (SVD) can be performed. When matrix processing is done, a folding operation converts the 2D matrix back into a 3D tensor. Folding along a given axis is the inverse of unfolding along the same axis:

$$
\text{fold}(\text{unfold}(\mathbf{X}, i), i) = \mathbf{X}.
$$

1.3.2 Singular Value Decomposition and Shrinkage

Singular Value Decomposition (SVD) of a given $m \times n$ matrix $\mathbf{A}$ is given by [26]:

$$
\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^T
$$

(1.5)

where $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices ($\mathbf{U}\mathbf{U}^T = \mathbf{I}, \mathbf{V}\mathbf{V}^T = \mathbf{I}$) whose dimensions are $m \times m$ and $n \times n$, respectively. The matrix $\Sigma$ is $m \times n$ and contains the singular values of $\mathbf{A}$ along its main diagonal, $\Sigma = \text{diag}(\sigma_i)$, with zero-padding to obtain the desired dimensions.
In some applications, soft-thresholding (also known as soft shrinkage) [27] is applied to singular values of $\Sigma$ in order to arrive at a lower-rank matrix that is still a good approximation to the original matrix $A$. Specifically, if $\sigma_i$ are the singular values of $A$ and $\tau$ is a given threshold, then the corresponding soft shrinkage operation is defined as

$$\text{shrink}(A, \tau) = U \Sigma_r V^T,$$

(1.6)

where $\Sigma_r = \text{diag}(\max(\sigma_i - \tau, 0))$. In words, shrinkage reduces all singular values by $\tau$, clips the negative results to 0, and then re-synthesizes the matrix with the new singular values.

**SVD and Shrinkage Toy Example**

The following $4 \times 5$ matrix:

$$A = \begin{bmatrix}
1 & 0 & 2 & 0 & 3 \\
0 & 4 & 0 & 5 & 0 \\
6 & 0 & 7 & 0 & 8 \\
0 & 9 & 0 & 10 & 0 \\
\end{bmatrix},$$

can be exactly represented using the matrix multiplication (Equation 1.5) of the following $U$ ($4 \times 4$), $\Sigma$ ($4 \times 5$), and $V^T$ ($5 \times 5$) matrices:

$$U = \begin{bmatrix}
0 & -0.28 & -0.96 & 0 \\
-0.43 & 0 & 0 & 0.90 \\
0 & -0.96 & -0.28 & 0 \\
-0.90 & 0 & 0 & -0.43 \\
14.90 & 0 & 0 & 0 & 0 \\
\end{bmatrix},$$

$$\Sigma = \begin{bmatrix}
0 & 0 & 0.34 & 0 \\
0 & 12.73 & 0 & 0 & 0 \\
0 & 0 & 0.96 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.34 \\
\end{bmatrix},$$

$$V^T = \begin{bmatrix}
0 & -0.66 & 0 & -0.75 & 0 \\
-0.47 & 0 & -0.57 & 0 & -0.67 \\
-0.78 & 0 & -0.08 & 0 & -0.62 \\
0 & -0.75 & 0 & 0.66 & 0 \\
0.41 & 0 & -0.82 & 0 & 0.41 \\
\end{bmatrix}.$$
Applying soft shrinkage to the above SVD with a threshold of $\tau = 1$ would change $\Sigma$ to produce the following matrix:

$$\Sigma_\tau = \begin{bmatrix}
13.90 & 0 & 0 & 0 \\
0 & 11.73 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}$$

Using the new thresholded singular value matrix $\Sigma_\tau$, one can get an approximation of $A$ by applying the matrix multiplication in Equation 1.6:

$$\tilde{A} = U\Sigma_\tau V^T = \begin{bmatrix}
1.58 & 0 & 1.91 & 0 & 2.24 \\
0 & 3.94 & 0 & 4.48 & 0 \\
5.33 & 0 & 6.43 & 0 & 7.53 \\
0 & 8.29 & 0 & 9.42 & 0
\end{bmatrix}$$

In this toy example, the approximation matrix $\tilde{A}$ produces non-zero values that are fairly close to those of the original matrix $A$ (at the correct locations). On average, these values are only 0.50 away from the true values in $A$. Also, the location of the zero values in $\tilde{A}$ exactly matches those in $A$. However, $\text{rank}(\Sigma) = 4$, while $\text{rank}(\Sigma_\tau) = 2$, allowing for data compression if necessary - a useful property (computation wise) when dealing with large matrices.

### 1.3.3 Canonical Polyadic Decomposition

A $m \times n \times p$ tensor $\mathcal{X}$ can be approximated with a low-rank tensor $\tilde{\mathcal{X}}$ through Canonical Polyadic Decomposition (CPD) by solving [28]:

$$\min_{\tilde{\mathcal{X}}} \| \mathcal{X} - \tilde{\mathcal{X}} \|_F,$$

where $\| \cdot \|_F$ is the Frobenius norm of a tensor, and the low-rank approximation tensor, $\tilde{\mathcal{X}}$, is given by:

$$\tilde{\mathcal{X}} = \sum_{r=1}^{\text{rank}(\mathcal{X})} u_r \circ v_r \circ w_r = \begin{bmatrix} U, V, W \end{bmatrix}$$

The dimensions of $u_r, v_r$, and $w_r$ are $m \times 1$, $n \times 1$, and $p \times 1$, respectively. The symbol $\circ$ represents the vector outer product, such that $u_r \circ v_r \circ w_r$ is a $m \times n \times p$ tensor for any $r$. The matrices $U, V,$, and $W$ represent the CP factor matrices whose columns are the corresponding vectors $u_r, v_r$, and $w_r$, namely, $U = [u_1, ..., u_{\text{rank}(\mathcal{X})}], V = [v_1, ..., v_{\text{rank}(\mathcal{X})}],$ and $W = [w_1, ..., w_{\text{rank}(\mathcal{X})}]$. The above minimization problem (Equation 1.7) can also be
rephrased and solved in terms of the unfolded tensors \( \hat{\mathbf{X}}(i) \) for \( i = 1, 2, 3 \):

\[
\hat{\mathbf{X}}(1) = (\mathbf{W} \odot \mathbf{V})\mathbf{U}^T \\
\hat{\mathbf{X}}(2) = (\mathbf{W} \odot \mathbf{U})\mathbf{V}^T \\
\hat{\mathbf{X}}(3) = (\mathbf{V} \odot \mathbf{U})\mathbf{W}^T, 
\]

(1.9)

where the symbol \( \odot \) represents the Khatri-Rao product defined as

\[
\mathbf{U} \odot \mathbf{V} = [u_1 \otimes v_1, \ldots, u_{\text{rank}(\mathbf{X})} \otimes v_{\text{rank}(\mathbf{X})}],
\]

and \( \otimes \) is the Kronecker product of the corresponding column-vectors. Note that the Kronecker product creates a block matrix out of the two operands. More specifically, \( u_i \otimes v_i \), \( u_i \otimes w_i \), and \( v_i \otimes w_i \) have dimensions \((m \cdot n) \times 1\), \((m \cdot p) \times 1\), and \((n \cdot p) \times 1\) for \( i = 1, 2, 3 \), respectively.

Although a couple of different algorithms exist for computing a tensor’s CPD, generally (due to its simplicity) the well known Alternating Least Squares (ALS) [29] algorithm can be applied on Equation 1.9 to solve Equation 1.7.

1.3.4 Tensor Data Packetization and Transmission

In collaborative intelligence (CI), tensor data needs to be transferred from the edge to the cloud. This process involves writing tensor values into data packets (which I refer to as packetization) and sending these data packets over the network to the cloud. It is too early to say what kind of tensor packetization schemes will be adopted for CI applications in the future. However, transmission of another kind of tensor data, namely video, has been around for a while. Various video packetization schemes are well-established and tested in practice. I believe that similar schemes will be strong contenders for deep feature tensor data packetization as well. Therefore, for the purposes of this thesis, I adopt a packetization method that is popular in video streaming [30], where in each video frame, rows of macroblocks are mapped to packets. In a similar manner, tensor data is written into packets row-by-row and then channel-by-channel. For example, the data in tensor \( \mathbf{X} \) in Equation 1.3 would generate six packets, each composed of one row of tensor data:

\[ [0, 2, 4], [6, 8, 10], \ldots, [13, 15, 17]. \]

Errors in the communication channel may cause data packets not to arrive at the cloud sub-model. Such packets are referred to as “lost” and the probability of their loss is \( P_{\text{loss}} \).

The result of packet loss at the cloud side is that the corresponding tensor rows are not

---

1 This probability depends on many factors, including physical layer modulation and error control, noise and interference in the channel, network congestion, etc.
available. As an example, the tensor $\mathbf{X}$ from Equation 1.3, with two rows of missing data, is shown below:

$$
\hat{\mathbf{X}} = \begin{pmatrix}
0 & 2 & 4 & \text{?} & \text{?} & \text{?} \\
\text{?} & \text{?} & \text{?} & 1 & 3 & 5 \\
12 & 14 & 16 & \text{?} & \text{?} & \text{?}
\end{pmatrix},
$$

where the missing values are indicated by question marks (‘?’). The goal of a given tensor completion algorithm is to recover these missing values so that the cloud sub-model can perform successful inference.

### 1.4 Thesis Preview

The thesis is organized as follows. As seen above, Chapter 1 provides the motivation and background that is necessary to understand the tensor completion algorithms under study throughout the thesis. In Chapter 2, I review the SiLRTC, HaLRTC, and FCP algorithms and explain how they are applied to deep feature tensor completion. Chapter 3 introduces ALTeC and describes how its parameters are computed. Chapter 4 discusses the simulation environment & experiments performed, and then provides statistical analysis of the results. Lastly, Chapter 5 concludes the thesis and indicates potential avenues for further work.
Chapter 2

General Tensor Completion Algorithms

Tensor completion has found applications in a number of research areas, including computer vision, data analytics, etc. From my independent literature survey, however, it has not been studied in the context of recovering missing feature tensor values in collaborative intelligence. Often, the underlying assumption is that tensor data is “low-rank”, or, more generally, “smooth” in some sense. A number of methods [5, 31, 32] were developed based on the assumption that tensors lie in a low-rank manifold, which leads to iterative procedures for approximating the original tensor by a low-rank tensor. In these cases, it is not crucial to know where the data comes from, so long as the low-rank assumption holds. I refer to these methods as general, meaning that they could be applied to any kind of tensor. This chapter will review three such methods, namely Simple Low Rank Tensor Completion (SiLRTC) [5], High Accuracy Low Rank Tensor Completion (HaLRTC) [5], and Fused Canonical Polyadic (FCP) [6], and show how to adapt them to the problem of recovering missing feature tensor values produced by packet loss in collaborative intelligence.

Note that it is not clear whether in fact low-rank assumption holds for deep feature tensors. The existence of adversarial examples [33] shows that small perturbations in the input of a deep model may cause large changes downstream. This might indicate that the notions of smoothness and low-rank are less applicable to the deep feature tensors than they could be to the kind of data for which tensor completion has mostly been used so far, such as color images. Nonetheless, it is still important to establish what level of performance existing tensor completion methods can achieve on this new problem.

2.1 Simple Low Rank Tensor Completion (SiLRTC)

This section briefly reviews SiLRTC [5] and shows how it can be applied to completion of feature tensors in collaborative intelligence. A summary of SiLRTC is provided in Algorithm 1. The inputs are the corrupt tensor $\tilde{X}$ (with some of its rows missing), the number
of iterations $K$, and non-negative scaling factors $\alpha_i, \beta_i, i \in \{1, 2, 3\}$ where $\alpha_i$’s add up to 1. The scaling factors are chosen randomly [5], and each pair of $(\alpha_i, \beta_i)$ is for one dimension of the tensor. The output is the “completed” tensor $\hat{\mathbf{X}}$.

**Algorithm 1 SiLRTC [5]**

**Input:** $\mathbf{X}$, $K$, $\alpha_i, \beta_i \geq 0, \sum_{i=1}^{3} \alpha_i = 1$

**Output:** $\hat{\mathbf{X}}$

1: $\hat{\mathbf{X}} \leftarrow \mathbf{X}$
2: for iter = 1, 2, ..., $K$, do
3:     $\mathbf{M} \leftarrow \text{zeros(} \mathbf{X}.\text{shape})$
4:     for $i = 1, 2, 3$, do
5:         $\mathbf{X} \leftarrow \text{unfold}(\mathbf{X}, i)$
6:         $\mathbf{X}_\tau \leftarrow \text{shrink}(\mathbf{X}, \tau = \alpha_i/\beta_i)$
7:     $\mathbf{X}_\tau \leftarrow \text{fold}(\beta_i \cdot \mathbf{X}_\tau, i)$
8:     $\mathbf{M} \leftarrow \mathbf{M} + \mathbf{X}_\tau$
9:     end for
10: $\mathbf{M} \leftarrow \mathbf{M}/\sum_{i=1}^{3} \beta_i$
11: Replace values in $\mathbf{M}$ by known rows in $\hat{\mathbf{X}}$
12: $\hat{\mathbf{X}} \leftarrow \mathbf{M}$
13: end for

To reconstruct the missing values in $\hat{\mathbf{X}}$, SiLRTC makes a copy of $\hat{\mathbf{X}}$ in $\mathbf{X}$ (step 1) and then loops over $K$ iterations. In each iteration, $\hat{\mathbf{X}}$ is unfolded along each dimension (step 5), shrinking with threshold $\tau = \alpha_i/\beta_i$ is applied (step 6), and the resulting 2D matrix is folded back into a 3D tensor (step 7). The three folded tensors are added (step 8) and the result is re-scaled (step 10). The above operations change all values in the tensor, however, some of the rows in $\hat{\mathbf{X}}$ are known (i.e., not missing). Therefore, known rows are replaced in the resulting tensor (step 11) and the result is passed to the next iteration (step 12).

In essence, SiLRTC attempts to iteratively reduce the rank of the corrupt tensor $\hat{\mathbf{X}}$ by performing shrinkage on the unfolded versions of the tensor and averaging the folded results. The next method I review, HaLRTC, is based on a similar idea, but is more accurate.

## 2.2 High Accuracy Low Rank Tensor Completion (HaLRTC)

HaLRTC [5] follows the same reasoning as SiLRTC, but uses an alternating direction method of multipliers (ADMM) [34] to find the solution. It is more sophisticated than SiLRTC and thus is expected to produce better tensor computation results. The steps are presented in Algorithm 2. The inputs are the corrupt tensor $\hat{\mathbf{X}}$ (with some of its rows missing), the number of iterations $K$, and non-negative scaling factors $\rho, \alpha_i, i \in \{1, 2, 3\}$ where $\alpha_i$’s add up to 1. The scaling factors are chosen randomly [5], and each $\alpha_i$ is for one dimension of the tensor. The output is the “completed” tensor $\hat{\mathbf{X}}$. 
Algorithm 2 HaLRTC \[5\]

Input: $\tilde{X}$, $K$, $\alpha_i$, $\rho \geq 0$, $\sum_{i=1}^{3} \alpha_i = 1$

Output: $\hat{X}$

1: $\hat{X} \leftarrow \tilde{X}$
2: for $i = 1, 2, 3$, do
3: $\mathcal{M}_i, \mathcal{Y}_i \leftarrow \text{zeros}(\hat{X}.\text{shape})$
4: end for
5: for $\text{iter} = 1, 2, ..., K$, do
6: for $i = 1, 2, 3$, do
7: $X \leftarrow \text{unfold}(\hat{X}, i)$
8: $Y \leftarrow \text{unfold}(\mathcal{Y}_i, i)/\rho$
9: $Z_{\tau} \leftarrow \text{shrink}(X + Y, \tau = \alpha_i/\rho)$
10: $\mathcal{M}_i \leftarrow \text{fold}(Z_{\tau}, i)$
11: end for
12: $\hat{X} \leftarrow \frac{1}{3} \cdot \sum_{i=1}^{3} (\mathcal{M}_i - \frac{1}{\rho} \cdot \mathcal{Y}_i)$
13: Replace values in $\hat{X}$ by known rows in $\tilde{X}$
14: for $i = 1, 2, 3$, do
15: $\mathcal{Y}_i \leftarrow \mathcal{Y}_i - \rho \cdot (\mathcal{M}_i - \hat{X})$
16: end for
17: $\rho \leftarrow 1.2 \cdot \rho$
18: end for

The algorithm starts by initializing the output tensor $\hat{X}$ (step 1) and auxiliary tensors $\mathcal{M}_i, \mathcal{Y}_i$ (step 3), both containing three tensors (one for each dimension). Then, in each of the $K$ iterations, for each tensor dimension, $\hat{X}$ and $\mathcal{Y}_i$ are unfolded along the corresponding dimension (steps 7-8), their sum is shrunk (step 9) and the resulting matrix is folded into $\mathcal{M}_i$ (step 10). Finally, output tensor $\hat{X}$, auxiliary tensors $\mathcal{Y}_i$, and scaling parameter $\rho$ are updated (steps 12-13, 15, and 17, respectively).

2.3 Fused Canonical Polyadic (FCP) Decomposition

The FCP algorithm \[6\] is somewhat more involved than SiLRTC and HaLRTC. The main steps are presented in Algorithm 3, but the reader is referred to \[6\] for full details. The inputs to the algorithm include the corrupt tensor $\tilde{X}$ (with some of its rows missing), the CP factor matrices $A_i$, $i \in \{1, 2, 3\}$ (which represent $U, V, W$ in Equation 1.8), regularization matrices $L_s$ and $L_p$, the number of iterations $K$, non-negative scaling factors $\mu, \alpha_i, \beta_i, \delta_i, \zeta_i, i \in \{1, 2, 3\}$, tensor rank $R$ along with its corresponding increment ($R_u$) and maximum value ($R_m$), and stopping criteria variables $\eta$ and $\epsilon$. The output is the “completed” tensor $\hat{X}$.
Algorithm 3 FCP [6]

Input: \( \tilde{\mathbf{X}}, \mathbf{A}_i, \mathbf{L}_p, \mathbf{L}_s, K, \mu, \alpha_i, \beta_i, \delta_i, \zeta_i, R, R_u, R_m, \eta, \epsilon \)

Output: \( \hat{\mathbf{X}} \)

1: \( \hat{\mathbf{X}} \leftarrow \tilde{\mathbf{X}} \)
2: for \( \text{iter} = 1, 2, \ldots, K \), do
3: \[ \text{for } i = 1, 2, 3, \text{ do} \]
4: \[ \mathbf{S}, \mathbf{V} \leftarrow \text{Compute } \mathbf{S} \text{ as in Equation 2.1, unfold}(\hat{\mathbf{X}}, i) \]
5: \[ L \leftarrow \| \mathbf{S}^T \mathbf{S} + \delta_i \mathbf{I} \|_F^2 + \beta_i \| \mathbf{L}_p \|_F^2 + \frac{\alpha_i}{\mu} + \frac{\zeta_i}{\mu} \| \mathbf{L}_s \|_F^2 \]
6: \[ \nabla h(\mathbf{A}_i) \leftarrow \mathbf{S}^T \mathbf{A}_i - \mathbf{S}^T \mathbf{V} + \frac{\alpha_i}{\mu} \mathbf{A}_i + \beta_i \mathbf{A}_i \mathbf{L}_p^T + \delta_i \mathbf{A}_i + \frac{\zeta_i}{\mu} \mathbf{Q}(\frac{1}{\mu} \mathbf{A}_i \mathbf{L}_s^T) \mathbf{L}_s^i \]
7: \( \mathbf{Y}_0, d_0, k \leftarrow \mathbf{A}_i, 1, 0 \)
8: while stopping_criterion \((\nabla h(\mathbf{Y}_k)) > \epsilon \) do
9: \[ \mathbf{A}^k \leftarrow \mathbf{Y}_k - \frac{1}{L} \nabla h(\mathbf{Y}_k) \]
10: \[ d_{k+1} \leftarrow \frac{1}{2}(1 + \sqrt{4d_k^2 + 1}) \]
11: \[ \mathbf{Y}_{k+1} \leftarrow \mathbf{A}^k + \frac{d_{k+1}}{d_k} (\mathbf{A}^k - \mathbf{A}^{k-1}) \]
12: \( k \leftarrow k + 1 \)
13: end while
14: \end for
15: if \( \sum_{n=1}^3 \| \mathbf{A}_n^{\text{iter}} - \mathbf{A}_n^{\text{iter}-1} \|_F < \eta \) and \( R < R_m \) then
16: \[ R \leftarrow R + R_u \]
17: end if
18: \( \hat{\mathbf{X}}^{\text{iter}-1} \leftarrow \text{fold}(\mathbf{A}_1(\mathbf{A}_3 \odot \mathbf{A}_2), 1) \)
19: \( \hat{\mathbf{X}}^{\text{iter}} \leftarrow \hat{\mathbf{X}}^{\text{iter}-1} \)
20: Replace values in \( \hat{\mathbf{X}}^{\text{iter}} \) by known rows in \( \tilde{\mathbf{X}} \)
21: if \( \| \hat{\mathbf{X}}^{\text{iter}} - \hat{\mathbf{X}}^{\text{iter}-1} \|_F < \eta \) then
22: break
23: end if
24: end for

The algorithm starts by initializing the output tensor \( \hat{\mathbf{X}} \) (step 1). Then it loops \((\text{iter})\) through each of the \( K \) iterations and for each tensor dimension, \( \hat{\mathbf{X}} \) is unfolded along the corresponding dimension \( i \) and matrix \( \mathbf{S} \) is computed (step 4) from the Khatri-Rao product between a pair of CP factor matrices (depending on \( i \)) as follows:

\[
\begin{align*}
\mathbf{S} &= \begin{cases} 
\mathbf{A}_3 \odot \mathbf{A}_2 & \text{for } i = 1 \\
\mathbf{A}_1 \odot \mathbf{A}_2 & \text{for } i = 2 \\
\mathbf{A}_2 \odot \mathbf{A}_1 & \text{for } i = 3 
\end{cases} 
\end{align*}
\]
Then the reciprocal of the step size $L$ and the gradient of regularized error $\nabla h$ are computed (steps 5-6). The gradient computation involves the rectified linear function $Q(\cdot)$ defined as

$$Q(x) = \begin{cases} 
-1 & \text{for } x < -1 \\
x & \text{for } -1 \leq x \leq 1 \\
1 & \text{for } x > 1 
\end{cases}$$

(2.2)

After that, the CP factor matrices $A_i$ are updated (steps 9-12) until the stopping criteria (computed by the function `stopping_criterion(\cdot)` in step 8) is met. It should be noted that stopping criteria, as well as several other parameters and scaling factors, are different for sparse and non-sparse tensors [6]. Next, the tensor rank is updated if needed (steps 15-17), an operation is performed on the CP factor matrices and the result is folded back into $\tilde{\mathcal{X}}^{\mathrm{iter}-1}$ along the first tensor dimension (step 18). Lastly, $\tilde{\mathcal{X}}^{\mathrm{iter}-1}$ is assigned to $\tilde{\mathcal{X}}^{\mathrm{iter}}$ (step 19), $\tilde{\mathcal{X}}^{\mathrm{iter}}$ is updated with known rows in the corrupt tensor $\tilde{\mathcal{X}}$ (step 20), and the Forbenious norm difference between successive iterations of the completed tensor $\tilde{\mathcal{X}}$ is compared to a tolerance $\eta$ to determine if the algorithm converged (steps 21-23).
Chapter 3

Adaptive Linear Tensor Completion (ALTeC)

The algorithms presented in Chapter 2 manipulate the singular values (SiLRTC and HaLRTC) and CP factors (FCP) of an unfolded tensor to reconstruct the missing elements of the corrupt deep feature tensor. While each singular value or CP factor highlights some relevant feature(s) in the tensor, it does not capture/take into account the tensor in its entirety. In addition, SVD and CPD computation is expensive, and in the above-mentioned algorithms, these need to be performed at each reconstructive iteration.

This chapter presents a simple tensor completion method specifically designed to recover missing rows of a deep feature tensor. I refer to it as Adaptive Linear Tensor Completion (ALTeC). The proposed method assumes an approximate linear relationship among the rows of a deep feature tensor and its neighbors. Let \( x_i^{(c)} \) be the \( i \)-th row in channel \( c \) of tensor \( \mathbf{X} \). The focus on rows comes from the specific row-by-row packetization scheme described in Section 1.3.4. If a different packetization scheme is adopted, \( x_i^{(c)} \) and its neighborhood would need to be redefined, but the methodology below would still be applicable. I assume that \( x_i^{(c)} \) can be approximated by a linear combination of its neighbors – co-located rows in other channels and two spatial neighbors in the same channel, one above and one below:

\[
x_i^{(c)} \approx \sum_{j=1}^{n} w_j^{(c)} x_j^{(c)} + w_{n+1}^{(c)} x_{i-1}^{(c)} + w_{n+2}^{(c)} x_{i+1}^{(c)},
\]

(3.1)

where \( w_j^{(c)} \)'s are the weights and \( w_i^{(c)} \) (the weight for the row itself in the sum on the right-hand side) is set to zero. When \( x_i^{(c)} \) is at the top (bottom) of channel \( c \), it’s top (bottom) neighbor \( x_{i-1}^{(c)} \) (\( x_{i+1}^{(c)} \)) is not available, so it is assumed to be all-zero. The above equation can be written in a matrix-vector form as

\[
x_i^{(c)} \approx \mathbf{X}_i^{(c)} \mathbf{w}_i^{(c)},
\]

(3.2)
where the neighbor rows of \( \mathbf{x}_i^{(c)} \) have been stacked into matrix \( \mathbf{X}_i^{(c)} \) as columns, and the corresponding weights have been placed into the column vector \( \mathbf{w}_i^{(c)} \). Finding the optimal weights amounts to solving the following problem:

\[
\min_{\mathbf{w}_i^{(c)}} \| \mathbf{x}_i^{(c)} - \mathbf{X}_i^{(c)} \mathbf{w}_i^{(c)} \|_2^2, \tag{3.3}
\]

which has a well-known solution [35]:

\[
\mathbf{w}_i^{(c)} = \left( \left( \mathbf{X}_i^{(c)} \right)^T \mathbf{X}_i^{(c)} \right)^{-1} \left( \mathbf{X}_i^{(c)} \right)^T \mathbf{x}_i^{(c)}. \tag{3.4}
\]

To obtain the weights \( \mathbf{w}_i^{(c)} \), I used 5,000 randomly selected images from the validation set of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [36]. This training set was separate from the test set used to compare the tensor completion methods in Chapter 4. Every input image generates a deep feature tensor at a chosen intermediate layer of a given deep model, so that \( \mathbf{w}_i^{(c)} \) can be computed from Equation 3.4 for every row \( i \) and every channel \( c \). These \( \mathbf{w}_i^{(c)} \)'s are averaged across all training images. Furthermore, in order to reduce weight storage requirements, I averaged all the weights for different rows \( i \) in each channel, to eventually obtain weights \( \mathbf{w}^{(c)} \). The resulting weights can now be stored in a \( (n+2) \times n \) matrix

\[
\mathbf{W} = \begin{bmatrix} \mathbf{w}^{(1)} | \mathbf{w}^{(2)} | \cdots | \mathbf{w}^{(n)} \end{bmatrix}, \tag{3.5}
\]

where the \( c \)-th column represents the weights for channel \( c \) of the deep feature tensor.

The process of tensor completion is summarized in Algorithm 4. The inputs are the corrupt tensor \( \mathbf{\hat{X}} \) and the weight matrix \( \mathbf{W} \). The output is the “completed” tensor \( \mathbf{\hat{X}} \).

**Algorithm 4 ALTeC**

**Input:** \( \mathbf{\hat{X}}, \mathbf{W} \)

**Output:** \( \mathbf{\hat{X}} \)

1. \( \mathbf{\hat{X}} \leftarrow \mathbf{\hat{X}} \)
2. for channel \( c = 1, 2, \ldots, n \) do
3. for each row \( i \) in channel \( c \) of \( \mathbf{\hat{X}} \) do
4. if row \( i \) is missing then
5. Collect neighbors of the \( i \)-th row into \( \mathbf{\hat{X}}_i^{(c)} \)
6. Extract the \( c \)-th column of \( \mathbf{W} \) into \( \mathbf{w}^{(c)} \)
7. \( \mathbf{\hat{x}}_i^{(c)} = \mathbf{\hat{X}}_i^{(c)} \mathbf{w}^{(c)} \)
8. Put \( \mathbf{\hat{x}}_i^{(c)} \) into the \( i \)-th row of channel \( c \) of \( \mathbf{\hat{X}} \)
9. end if
10. end for
11. end for
First, the output tensor \( \hat{\mathbf{X}} \) is initialized (step 1). Then the algorithm loops through the tensor channels and each missing row is reconstructed as a linear combination of its neighbors using the corresponding weights (step 7). Note that if some of the neighbors of the missing rows are also missing, the corresponding column in \( \hat{\mathbf{X}}^{(c)}_i \) is all-zero. This ensures that the corresponding term is effectively eliminated from the linear combination in Equation 3.1.

### 3.1 Weight Matrix Analysis

The deep feature tensors produced by the edge sub-model contain channels consisting of the relevant features for a given input. That is, each channel is different, but there is some underlying correlation between them. After all, this is precisely why \( n \) co-located neighbors were included in the linear combination that reconstructs the missing rows (Equation 3.1).

To analyze this correlation further, the value distribution of the weight matrices obtained for both classification models are given in Figure 3.1. Note that in these distributions the y-axis has logarithmic scale.

![Weight Value Distribution](image)

(a) VGG16

Figure 3.1: Value Distribution of Weight Matrices
In Figure 3.1a (VGG16), the mean value of weights is 8.09E-4 with a standard deviation of 7.09E-4, indicating that there is little correlation between many of the channels - although they are not completely uncorrelated. However, considering that the maximum weight value is 1.77E-2, there are also many instances (>9.50E-3) where the weight values are significantly higher and thus must be taken seriously. Likewise, from the distribution in Figure 3.1b, again the mean and standard deviation are quiet low at 5.83E-3 and 9.19E-3, respectively. Nonetheless, there is a cluster of weight values (>5.0E-2) that are significant when the maximum value of 2.26E-1 is taken into consideration.

To provide further evidence that the channels in deep feature tensors have some correlation that must be taken into consideration, the classification model weight matrices are illustrated in Figure 3.2. In the figure, the individual weight values represent the pixel values in the corresponding location. For visualization purposes, the weights were shifted to the positive range, normalized to values in [0, 255], and the resulting pixel values were then inverted such that brighter values corresponds to lower correlation and darker values correspond to higher correlation. This operation is summarized as follows:

\[
\begin{align*}
W &= W + \min(W) \cdot I \rightarrow \text{Shift}, \\
W &= W \cdot 255.0 / \max(W) \rightarrow \text{Normalize}, \\
W &= \|W - 255.0\| \rightarrow \text{Invert}
\end{align*}
\]

where I is the identity matrix whose dimensions match those of W.
Each column corresponds to the weights in a given channel of the deep feature tensor, as indicated in Equation 3.5. That is the weights for channel 1 are represented with the first column, and likewise the weights for channel $n$ are represented with the $n$-th (last) column. Each column’s pixel values indicate the weight of the neighbors for that channel. Darker rows indicate larger weights and higher correlation.

As expected, the spatial neighbors (bottom two rows) yield high correlation for a given missing row on average for both classification models. Additionally, as mentioned above the co-located neighbors in other channels exhibit some correlation, although much lower than the spatial neighbors, with a given missing row. This is clearly seen in both weight matrices by the grey-ish rows and/or darker elements corresponding to the $n$ co-located neighbors ($n \times n$ sub-matrix that excludes the last two rows). It is also noticeable that the ResNet34 model’s weight matrix has a darker shade, which can be explained by the many non-zero rows providing more information for reconstruction - unlike in the VGG16 case.

Thus, from the above analysis of the weight matrices produced by both classification models one can conclude that the effect of co-located neighbors in deep feature tensors, received at the cloud sub-model side, should be taken into account when reconstructing a given row.

3.2 Weight Matrix Pre-processing (Re-distribution)

Two different possibilities were considered when the weight matrix $W$ was supplied as input. Namely, all the weights were either pre-processed or directly passed to the algorithm. For
pre-processing, the weights in the channel were re-distributed away from any lost packets in a manner that is proportional to the existing weight magnitudes. Thus, terms that had high weights to start with would get a higher proportion of the weight from missing packets, and vice-versa.

Let \( r_i^{(c)} \) be the vector of indicators, showing which neighbors of row \( i \) in channel \( c \) are available: \( r_i^{(c)} = [r_{i,1}^{(c)}, r_{i,2}^{(c)}, \ldots, r_{i,n+2}^{(c)}]^T \), where \( r_{i,j}^{(c)} = 1 \) if the \( j \)-th neighbor is available, and \( r_{i,j}^{(c)} = 0 \) otherwise. The weight adjustment can be summarized in Equation 3.7:

\[
\tilde{w}_i^{(c)} = \frac{w_i^{(c)} \odot r_i^{(c)}}{(w_i^{(c)})^T \cdot r_i^{(c)}},
\]

where \( \odot \) is the Hadamard product of two matrices (or vectors) and essentially computes the element-wise product of the operands, \( i \) is the given row in a channel, \( c \) is the corresponding channel, and \( n + 2 \) is the total number of neighbours (spatial and co-located) to row \( i \).

### 3.2.1 Weight Re-distribution Toy Example

Given the following \( 1 \times 3 \) weight \((w)\) and binary missing neighbor indication \((r)\) vectors:

\[
w = \begin{bmatrix} 0.5 & 0.3 & 0.2 \end{bmatrix}^T, \quad r = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T
\]

The re-distribution algorithm would know that the middle row is assumed to be missing due to the zero in \( r \). Thus, using Equation 3.7, the re-distributed weights will be calculated as follows:

\[
w \odot r = \begin{bmatrix} 0.5 \cdot 1 & 0.3 \cdot 0 & 0.2 \cdot 1 \end{bmatrix}^T = \begin{bmatrix} 0.5 & 0 & 0.2 \end{bmatrix}^T
\]

\[
w^T \cdot r = (0.5 \cdot 1) + (0.3 \cdot 0) + (0.2 \cdot 1) = 0.7
\]

\[
\therefore \tilde{w} = \begin{bmatrix} 0.714 & 0.000 & 0.286 \end{bmatrix}^T
\]

From the above toy example, one can see that the weight corresponding to the missing packet was re-distributed across the other weights. This re-distribution was proportional to the respective initial weight magnitude, as \( \tilde{w}_1 = w_1 + 0.214 \), while \( \tilde{w}_3 = w_3 + 0.086 \).

### 3.2.2 Weight Matrix Configuration Reconstruction Analysis

Using the procedures presented in Section 4.1, I was able to compare the two weight matrix configurations described above to decide which one would perform best against the other tensor completion algorithms. Figure 3.3 illustrates this analysis using solid curves and shaded bands around those solid curves, representing the average tensor completion reconstruction accuracy a given test set and the corresponding standard deviation, respectively.
As can be seen in the figure, using weight re-distribution causes the reconstruction results to be poorer, across all packet loss values for both models, than those when no tensor completion algorithm is applied (corrupt). Although it is not clear why this is the case for both VGG16’s sparse and ResNet34’s non-sparse input corrupt deep feature tensors, one explanation can be that the weights computed in the original weight matrix were perfectly tuned to every row of the corresponding deep feature tensor. Thus, any disturbance, such as magnitude scaling, caused by the re-distribution process will also scale the reconstructed row’s pixel values (Equation 3.2) accordingly. On the contrary, without weight re-distribution, ALTeC’s reconstruction results perform as expected. That is, they always improve a given model’s ability to accurately predict the reconstructed tensor. For this rea-
son, in the rest of this thesis, I will be discussing ALTeC under the assumption that there is no re-distribution applied to the input weight matrix $W$.

### 3.3 ALTeC’s Key Points

From this chapter, the main takeaway is that tensor reconstruction in ALTeC is linear in nature, and the linear combination for row reconstruction changes from channel to channel (hence “adaptive”). The key feature that makes ALTeC attractive compared to SiLRTC, HaLRTC, and FCP for collaborative intelligence applications, where latency is important, is its speed. ALTeC does not use computationally expensive operations such as shrinkage-based SVD or CPD. Moreover, ALTeC only reconstructs missing rows, whereas SiLRTC, HaLRTC, and FCP update the entire tensor in each iteration. Despite its relative simplicity, ALTeC achieves similar reconstruction accuracy results to the other three methods, as will be seen in the next chapter.
Chapter 4

Experiments and Results

4.1 Experimental setup

This chapter outlines the experiments that were carried out to compare the reconstruction performance of the four tensor completion algorithms described earlier – SiLRTC, HaLRTC, FCP, and ALTeC – on two pre-trained deep models for image classification, VGG16 [37] and ResNet34 [38], implemented in Keras.¹ Even though they are no longer state-of-the-art models for image classification, I selected these two models since they are well known in the research community and well-studied in the literature under a variety of application scenarios, including collaborative intelligence [1, 2]. The data used in the experiments is a randomly-selected subset of 1,000 images from the ILSVRC [36] validation set. This subset differed from the 5,000 images with which ALTeC was trained in order to obtain the necessary weight matrix (Chapter 3).

Following a common practice in video streaming simulations [30], I consider an independent random packet loss channel with packet loss probability:

\[ p_{\text{loss}} \in \{5\%, 10\%, 15\%, 20\%, 25\%, 30\%\}. \]

It is assumed that at the receiver (cloud sub-model), missing packets are identified via packet sequence numbers provided by a transport-layer protocol such as the Real-time Transport Protocol (RTP) [39].

For each value of \( p_{\text{loss}} \), each image in the test set is input to the edge sub-model and the resulting deep feature tensor \( \mathcal{X} \) is transmitted over the packet loss channel \( N = 100 \) times, to obtain statistically meaningful results (independent trials). For each of these \( N = 100 \) channel realizations, on the receiver side, a corrupt tensor \( \tilde{\mathcal{X}} \) is received, a specific tensor completion method is executed to obtain the completed tensor \( \hat{\mathcal{X}} \), and this tensor is input to the cloud sub-model to complete the inference task.

¹https://keras.io/applications/
Due to the use of image classification models, I measure classification accuracy under the three conditions illustrated in Figure 4.1:

- no loss (NL), to establish baseline performance;
- no tensor completion (NC), where all missing data is assumed to be zero;
- tensor completion (TC), where a specific tensor completion algorithm is applied to the corrupt deep feature tensor.

The average Top-1 classification accuracy for the three cases ($\mu_{NL}$, $\mu_{NC}$, and $\mu_{TC}$) and the standard deviation of Top-1 classification accuracy under NC and TC conditions ($\sigma_{NC}$ and $\sigma_{TC}$) are measured. It is important to note that for each combination of packet loss level and tensor completion algorithm, random packet loss is simulated using a random seed corresponding to the independent trial index (1, 2, ..., 100), so that each completion algorithm sees the same set of packet loss realizations. This ensures that there is fairness when the four algorithms are compared, that is, no bias towards a given algorithm.

Before moving on to the reconstruction performance comparison, I examine the convergence of SiLRTC, HaLRTC, and FCP. Recall from Chapter 2 that these algorithms iteratively update the to-be-completed tensor via unfolding and shrinking (or CPD) for a given number of iterations $K$. Figure 4.2 shows the Frobenius norm ($\|\cdot\|_F$) of the difference between tensors in two successive iterations ($K$ and $K + 1$):

$$
\left\| \hat{X}^{K+1} - \hat{X}^K \right\|_F = \left\| \hat{A} \right\|_F = \left[ \sum_{i,j,l} |\hat{a}_{i,j,l}|^2 \right]^{\frac{1}{2}}
$$

The curves in Figure 4.2 represent the average Frobenius norm of the difference between tensors in two consecutive iterations across the six packet loss probabilities ($P_{\text{loss}}$). As seen in the figure, all three algorithms have essentially converged and tensor updates have stopped (are negligible) at around $K = 50$. Hence, I use $K = 50$ iterations in our experiments for these three algorithms.
All four tensor completion algorithms were implemented and tested on a Linux based machine with the following specifications:

- Ubuntu 16.04.5 LTS (Xenial Xerus)
- 128GB (12GB) of CPU RAM (GPU RAM)
- Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz
- 12 processor cores
- Titan X (Pascal) GPU
- Tensorflow 1.9.0, Keras 2.2.4, Python 3.6.7

4.2 Results on VGG16

Figure 4.3 shows the block diagram of the VGG16 network, and indicates the point where the network is split into the edge and cloud sub-models. The split is at the output of the “block4_pool” layer. At this point, the deep feature tensor is of size $14 \times 14 \times 512$, and its total number of elements (100,352) is less than the number of pixels in the input image (which is $224 \times 224 \times 3 = 150,528$). The volume of data to be transferred from the edge device to the cloud is an important consideration in collaborative intelligence [1, 40] and one wants to choose a split point where the data volume in the feature tensor is less than the data volume of the input. With such a split, the edge sub-model contains 7,635,264 (5.52%) of the total (trainable) parameters of the VGG16 network, and the cloud sub-model contains...
the remaining 130,722,280 (94.48%) of the total (trainable) parameters. Again, this is a reasonable workload distribution considering the computational resources available at the edge and in the cloud.

Note that in the VGG16 network, each convolutional layer applies the Rectified Linear Unit (ReLU) activation to its output. Hence, the resulting feature tensors already contain
many zero elements. Since the missing rows in the received tensor $\tilde{X}$ are initially filled in with zeros (the NC case), one can expect that the non-completed tensors $\tilde{X}$ will be relatively similar to the completed tensors $\hat{X}$. To illustrate this point, Figure 4.4 shows the percentage of zero and non-zero elements in the feature tensors for each loss value. As seen in the figure, the percentage of zero elements in the deep feature tensors under no loss ($p_{\text{loss}} = 0\%$) is already more than 75% and does not increase proportionally to packet loss. Hence, I expect that in this case, the difference in classification between NC and TC cases will be relatively small, as will be confirmed by quantitative results later in this chapter.

![Figure 4.4: Percentage of zero elements in the VGG16 tensor as a function of packet loss](image)

4.2.1 Execution Speed

First I compare the execution speed of the four tensor completion algorithms and present the results in Figure 4.5. The solid curves in the figure represent the average tensor completion speed over the test set of 1,000 images, across various packet loss levels. The shaded band around solid curves indicates one standard deviation of the execution speed. As seen in the figure, ALTeC is significantly faster than the other methods, with FCP being the second-fastest (due to its built-in convergence metric), followed by SiLRTC and HaLRTC, as expected.

Part of the reason for ALTeC’s speed advantage over the other three algorithms comes from avoiding the shrinkage-based SVD (or CPD) at each iteration. Another reason is that, in each iteration, SiLRTC, HaLRTC, and FCP update all elements of the tensor only to replace the non-missing (known) values once the update is done. Meanwhile, ALTeC only spends computation on the tensor elements that are actually missing. ALTeC does require off-line training to compute the weights matrix, however, this can be done in parallel with
the training of the main model. Thus, in terms of the overall collaborative system design, the extra training stage does not add any extra overhead, and yet results in run-time savings upon system deployment.

### 4.2.2 Classification Accuracy

Table 4.1 shows the average Top-1 classification accuracy and its standard deviation for the various loss levels and tensor completion algorithms. The first three numerical columns show the average accuracy under no packet loss ($\mu_{NL}$), the average accuracy with no tensor completion ($\mu_{NC}$) and its standard deviation ($\sigma_{NC}$). Note that due to the large number of zeros produced by the ReLU activation functions in certain VGG16 layers (Figure 4.4), $\mu_{NC}$ is fairly close to $\mu_{NL}$, as predicted earlier. Even under 30% loss, the classification accuracy drops by less than 4%.

The next two columns show the accuracy ($\mu_{TC}$) and standard deviation ($\sigma_{TC}$) for the four tensor completion algorithms with default settings. By this I mean that SiLRTC, HaLRTC, and FCP are run for $K = 50$ iterations, as explained earlier. However, this means that their execution speed is significantly higher than that of ALTeC (Figure 4.5). Therefore, in the last two columns, I report the results for the matched execution speed, where SiLRTC, HaLRTC, and FCP only run until their execution speed matches that of ALTeC. As a result, they were only able to run 1-5 (SiLRTC & HaLRTC) or 5-20 (FCP) iterations, depending on the case.

In the default-settings case, the accuracies of all four methods were similar, with the maximum difference of around 0.25% (between SiLRTC and HaLRTC) under 30% loss. To better understand the underlying distributions, I show the various loss level prediction

![Average Tensor Completion Speed vs. Loss](image)
Table 4.1: VGG16 classification accuracy results. Among the last four columns, the first two show the results with default settings, and the last two show the results with matched execution speed.

| \(p_{\text{loss}}\) | Algorithm | \(\mu_{\text{NL}}\) | \(\mu_{\text{NC}}\) | \(\sigma_{\text{NC}}\) | \(\mu_{\text{TC}}\) | \(\sigma_{\text{TC}}\) | \(\mu_{\text{TC}}\) | \(\sigma_{\text{TC}}\) |
|---|---|---|---|---|---|---|---|---|
| 5% | SiLRTC | 56.20% | 55.96% | 0.41% | 56.09% | 0.39% | 55.96% | 0.41% |
|   | HaLRTC | 56.20% | 55.96% | 0.41% | 56.06% | 0.36% | 55.96% | 0.41% |
|   | FCP | 56.20% | 55.96% | 0.41% | 56.09% | 0.41% | 55.99% | 0.44% |
|   | ALTeC | 56.20% | 55.96% | 0.41% | 56.07% | 0.40% | 56.07% | 0.40% |
| 10% | SiLRTC | 56.20% | 55.50% | 0.55% | 55.75% | 0.37% | 55.51% | 0.54% |
|   | HaLRTC | 56.20% | 55.50% | 0.55% | 55.78% | 0.51% | 55.78% | 0.53% |
|   | FCP | 56.20% | 55.50% | 0.55% | 55.70% | 0.48% | 55.70% | 0.48% |
|   | ALTeC | 56.20% | 55.50% | 0.55% | 55.70% | 0.48% | 55.70% | 0.48% |
| 15% | SiLRTC | 56.20% | 54.76% | 0.60% | 54.99% | 0.58% | 54.79% | 0.62% |
|   | HaLRTC | 56.20% | 54.76% | 0.60% | 55.14% | 0.44% | 54.75% | 0.59% |
|   | FCP | 56.20% | 54.76% | 0.60% | 55.17% | 0.55% | 55.15% | 0.59% |
|   | ALTeC | 56.20% | 54.76% | 0.60% | 55.11% | 0.56% | 55.11% | 0.56% |
| 20% | SiLRTC | 56.20% | 54.18% | 0.63% | 54.51% | 0.61% | 54.24% | 0.64% |
|   | HaLRTC | 56.20% | 54.18% | 0.63% | 54.67% | 0.55% | 54.21% | 0.64% |
|   | FCP | 56.20% | 54.18% | 0.63% | 54.74% | 0.65% | 54.72% | 0.67% |
|   | ALTeC | 56.20% | 54.18% | 0.63% | 54.64% | 0.63% | 54.64% | 0.63% |
| 25% | SiLRTC | 56.20% | 53.45% | 0.79% | 53.95% | 0.69% | 53.51% | 0.76% |
|   | HaLRTC | 56.20% | 53.45% | 0.79% | 54.19% | 0.67% | 53.48% | 0.80% |
|   | FCP | 56.20% | 53.45% | 0.79% | 54.16% | 0.71% | 54.16% | 0.72% |
|   | ALTeC | 56.20% | 53.45% | 0.79% | 54.03% | 0.75% | 54.03% | 0.75% |
| 30% | SiLRTC | 56.20% | 52.57% | 0.77% | 53.13% | 0.73% | 52.69% | 0.78% |
|   | HaLRTC | 56.20% | 52.57% | 0.77% | 53.39% | 0.67% | 52.65% | 0.78% |
|   | FCP | 56.20% | 52.57% | 0.77% | 53.31% | 0.78% | 53.26% | 0.74% |
|   | ALTeC | 56.20% | 52.57% | 0.77% | 53.25% | 0.81% | 53.25% | 0.81% |

accuracy histograms over 100 trials in Figure B.1. As can be seen in the figure, at each loss level all algorithms have similar distribution shapes that are centered around roughly the same mean, with only minor differences in a few “group-bin” heights. To test for statistical significance of these differences, I applied Welch’s t-test for samples with unequal variance [41]. The resulting \(p\)-values are shown in the middle three columns of Table 4.2. In experimental sciences, a \(p\)-value of less than 0.05 is usually taken as a sign of statistically significant difference. As seen in Table 4.2, all pairwise differences between average accuracies in the default-settings case were insignificant, except for the difference between SiLRTC & HaLRTC for 25% & 30% loss and SiLRTC & FCP for 15%, 20%, and 25% loss. To aid
Table 4.2: VGG16 statistical significance results. The middle three columns show p-values of the 2-sided Welch’s t-test for pairwise comparison of accuracies with default settings, while the last three columns show p-values for the case when the execution speeds are matched.

| $p_{loss}$ | Algorithm | Default | Speed-matched |
|----------|-----------|---------|--------------|
|          | ALTeC    | SiLRTC | HaLRTC       | ALTeC | SiLRTC | HaLRTC |
| 5%       | SiLRTC   | 0.669  | -            | -     | 0.054  | -     |
|          | HaLRTC   | 0.897  | 0.563        | 0.986 | -     | -     |
|          | FCP      | 0.662  | 0.986        | 0.558 | 0.199  | 0.567 |
| 10%      | SiLRTC   | 0.681  | -            | -     | 0.268  | 0.001 |
|          | HaLRTC   | 0.435  | 0.199        | -     | 0.148  | 5.519E-07 |
|          | FCP      | 0.245  | 0.108        | 0.592 | 0.009  | 0.721 |
| 15%      | SiLRTC   | 0.149  | -            | -     | 1.423E-04 | - |
|          | HaLRTC   | 0.666  | 0.059        | -     | 1.563E-05 | 0.658 |
|          | FCP      | 0.419  | 0.025        | 0.700 | 0.188  | 5.19E-07 |
| 20%      | SiLRTC   | 0.132  | -            | -     | 1.083E-05 | - |
|          | HaLRTC   | 0.740  | 0.052        | -     | 2.675E-06 | 0.733 |
|          | FCP      | 0.270  | 0.009        | 0.400 | 0.421  | 5.59E-07 |
| 25%      | SiLRTC   | 0.461  | -            | -     | 3.272E-06 | - |
|          | HaLRTC   | 0.110  | 0.015        | -     | 1.454E-06 | 0.773 |
|          | FCP      | 0.201  | 0.037        | 0.769 | 0.201  | 3.336E-09 |
| 30%      | SiLRTC   | 0.264  | -            | -     | 1.242E-06 | - |
|          | HaLRTC   | 0.183  | 0.009        | -     | 2.640E-07 | 0.731 |
|          | FCP      | 0.626  | 0.099        | 0.404 | 0.942  | 2.968E-07 |

readers, these cases are indicated with green shading in the table. Since no algorithm came out as the clear winner (i.e., provided significantly better results than all alternatives), no accuracy in the corresponding column of Table 4.1 is indicated in bold.

Likewise, in the matched-speed case, the accuracies of all four methods were again very similar, with a maximum difference of about 0.70% (between HaLRTC and FCP) under 25% loss. As in the default case, it is important to see the prediction accuracy distributions over the 100 trials to get the full picture, thus the corresponding histograms are shown in Figure B.2. Once again the figure does not provide any strong (visual) evidence that the distributions of all four methods differ at a given loss level. To verify whether this is in fact the case, I used the above-mentioned Welch’s t-test and found that there was no clear winner that statistically outperformed all of its rivals, as indicated by the p-values in Table 4.2. Thus, again, no accuracy in the corresponding column of Table 4.1 is indicated in bold.

Overall, results obtained from the VGG16 model suggest that ALTeC offers equivalent performance to SiLRTC, HaLRTC, and FCP, both when complexity is not constrained and when the algorithms are constrained to be equally fast.
4.2.3 Interesting Examples

Finally, I highlight several interesting examples that were observed during the testing of tensor completion algorithms on VGG16. Here, all tensor completion algorithms run in the default (i.e., not speed-matched) configuration. Table 4.3 shows classification predictions made on two images: #102 (‘Sleeping Bag’ or ‘SB’) and #3 (‘Bulbul’), whose ground truth images are shown in Figure A.1a and Figure A.1b, respectively. Ground truth labels are listed in the row that starts with “GT Label”. The next two rows show the results under no loss (NL). In other words, these are the outputs obtained from the pre-trained VGG16 model. For image #102, the Top-1 label is wrong (‘Cloak’) and the model is fairly confident about it (74.83%). After 30% packet loss and without tensor completion (NC), the model is still wrong (‘Cloak’), but it is less confident than before (49.73%). Finally, after tensor completion (TC), ALTeC produces the correct result (‘SB’), while SiLRTC, HaLRTC, and FCP are wrong, but not as confident about it as the original model (around 52-56%).

Table 4.3: Interesting examples from VGG16 experiments

| Image | Trial # | Algorithm | GT Label | NL Label | NL Score | NC Label | NC Score | TC Label | TC Score |
|-------|---------|-----------|----------|----------|----------|----------|----------|----------|----------|
|       |         | ALTeC | FCP | HaLRTC | SiLRTC | ALTeC | FCP | HaLRTC | SiLRTC |
|       |         | SB | SB | SB | SB | Bulbul | Bulbul | Bulbul | Bulbul |
|       | 102     | Cloak | Cloak | Cloak | Cloak | Bulbul | Bulbul | Bulbul | Bulbul |
|       | 3       | Bulbul | Bulbul | Bulbul | Bulbul | Kite | Kite | Kite | Kite |
|       | 3       | Cloak | Cloak | Cloak | Cloak | Kite | Kite | Kite | Kite |
|       |         | SB | Cloak | Cloak | Cloak | SB | Cloak | Cloak | SB |
|       |         | 51.85% | 55.24% | 56.37% | 56.37% | 51.85% | 55.24% | 56.37% | 56.37% |
|       |         | 74.83% | 74.83% | 74.83% | 74.83% | 74.83% | 74.83% | 74.83% | 74.83% |
|       |         | 95.80% | 95.80% | 95.80% | 95.80% | 95.80% | 95.80% | 95.80% | 95.80% |
|       |         | 49.73% | 49.73% | 49.73% | 49.73% | 49.73% | 49.73% | 49.73% | 49.73% |
|       |         | 42.24% | 43.13% | 45.45% | 46.02% | 42.24% | 43.13% | 45.45% | 46.02% |
|       |         | 36.93% | 43.13% | 45.45% | 46.02% | 36.93% | 43.13% | 45.45% | 46.02% |

SB = Sleeping Bag

For image #3, the pre-trained model is correct (‘Bulbul’), and also confident about it (95.80%). After 30% packet loss and without tensor completion (NC), the model produces an incorrect prediction (‘Kite’) with confidence of 42.24%. Here, all tensor completion algorithms lead to the wrong prediction result (‘Kite’), but ALTeC leads to least confidence about this prediction (36.93%) while SiLRTC, HaLRTC, and FCP increase their confidence about the wrong decision (to 43-46%) compared to the NC case. Although all tensor completion methods lead to the wrong decision in this case, one could argue that ALTeC is still better than the other three methods because with ALTeC-completed tensor, the VGG16 model is least confident about its wrong answer.

The fact that ALTeC is superior to the other methods for these specific images can be further indicated by an analysis of the completion abilities of all the algorithms on the corresponding channels. To make this analysis feasible, I chose a single channel which best illustrates this point. In this case, channel #257 was chosen from the deep feature tensors.
produced by both images and the completion results are shown in Figures A.2 & A.3, respectively. In these figures, the corrupt row(s) are highlighted in red.

Note that in Figure A.2, only one row becomes corrupt and is considered missing as indicated with the red highlighting. That is, even though row 1 is originally an all zeros row, it does not get corrupted, and hence is not completed in any of the algorithms. Whereas, in Figure A.3, in addition to three other rows, row 1 is again originally an all zeros row, but it gets corrupted, and thus is completed by each of the algorithms.

To get a good understanding of which algorithm better completes the corrupt row(s), I computed the $L_1$ norm ($\| \cdot \|_1$) of the channel difference between the original (ground truth – NL) channel and the corresponding tensor completed channel. It is clear, this result only depends on the differences caused by corrupt row(s):

$$|NL_{row} - TC_{row}|_1 = \sum_{col=1}^{14} |NL_{row,col} - TC_{row,col}|,$$

where $row \in \{12\}$ (Figure A.2) or $row \in \{1, 7, 10, 11\}$ (Figure A.3), $NL_{row}$ and $TC_{row}$ are the corresponding corrupt row in the original channel and TC channel, respectively, and a sub-script of $row, col$ indicates the element value in the corresponding row and column of the channel. The resulting errors are provided in Table 4.4.

Table 4.4: $L_1$ norm difference values for channel #257

| Algorithm | SiLRTC | HaLRTC | FCP | ALTeC | Row |
|-----------|--------|--------|-----|-------|-----|
| Image #102 (Figure A.2) | 2942   | 3036   | 1743| 1254  | 12  |
| Average   | 2942   | 3036   | 1743| 1254  |     |
| Image #3 (Figure A.3)  | 830    | 459    | 177 | 1045  | 1   |
|               | 698    | 703    | 687 | 755   | 7   |
|               | 1333   | 1209   | 1244| 1094  | 10  |
|               | 1888   | 1953   | 2020| 1176  | 11  |
| Average     | 1187.25| 1081   | 1032| **1017.5** |     |

From Table 4.4, it can be seen that ALTeC has the lowest average $L_1$ norm values out of all the algorithms for both images. This indicates that in both cases, ALTeC completed channels were the closest to the ground truth channels, with FCP being second-best. This is expected as unlike SiLRTC and HaLRTC, these algorithms use the locality of neighboring rows to impute the missing row(s).

### 4.3 Results on ResNet34

Figure 4.6 shows the architecture of the ResNet34 model. By similar reasoning as in Section 4.2, I decided to split the model at layer “add_7”. With this split, the edge sub-model
produces tensors of size $28 \times 28 \times 128$, which contain the same number of elements as in the VGG16 case. Also, the edge sub-model contains 1,351,619 (6.19%) trainable parameters, while the cloud sub-model contains the remaining 20,488,488 (93.81%) trainable parameters. Again, this is a reasonable workload distribution considering the computational resources available at the edge and in the cloud.

Figure 4.6: ResNet34 model split at layer “add_7” [38]
ResNet34, like VGG16, uses ReLU activation functions. However, it also makes use of batch normalization [42], which centers the data distribution before adding it to the data passed down the residual (skip) connection [38]. Hence, the deep feature tensor produced by the edge sub-model in this case does not contain as many zeros as the one produced by the VGG16 edge sub-model. Indeed, Figure 4.7 shows that without loss, there are virtually no zeros in the tensor $\mathbf{X}$ produced by the ResNet34 edge sub-model. As the loss increases, the percentage of zero-elements in the received tensor $\tilde{\mathbf{X}}$ increases proportionally, and one can expect a larger difference between non-completed tensors $\tilde{\mathbf{X}}$ and completed tensors $\mathbf{X}$ than in the VGG16 case.

![Tensor Element Values vs. Loss](image)

Figure 4.7: Percentage of zero elements in the ResNet34 tensor as a function of packet loss

### 4.3.1 Execution Speed

Similarly to the VGG16 case, I tested the execution speed of the four tensor completion algorithms and report the results in Figure 4.8. The solid curves represent the average tensor completion speed over the test set of 1,000 images, across various packet loss levels. The shaded band around solid curves indicates one standard deviation of the execution speed. As in the VGG16 case, ALTeC is significantly faster than the other three methods, with FCP being the second-fastest, followed by SiLRTC and HaLRTC.

When comparing these results to those in Figure 4.5, the execution speeds of SiLRTC and HaLRTC are similar, about 1.2–1.5 seconds/tensor. Likewise, the execution speeds of FCP are similar at about 0.25–0.30 seconds/tensor. This is not surprising considering that in both VGG16 and ResNet34 cases, the tensors (and their unfolded versions) have the same number of elements. However, ALTeC is noticeably faster on ResNet34 tensors (Figure 4.8) than on VGG16 tensors (Figure 4.5). This can be explained by the fact that ResNet34
tensors have fewer channels than VGG16 tensors, so matrix $\tilde{X}_i^{(c)}$ in step 7 of ALTeC is smaller in the ResNet34 case, which leads to faster matrix-vector multiplication.

4.3.2 Classification Accuracy

As in the VGG16 case, I compute the classification accuracy offered by the four tensor completion methods in the default case (where SiLRTC, HaLRTC, and FCP run $K = 50$ iterations) and in the speed-matched case (where they are only allowed to run for the same duration of time as ALTeC). The results are shown in Table 4.5. First, note that for ResNet34, there is a large difference between the no-loss case (NL) and no tensor completion case (NC). Specifically, for 30% loss, the difference in Top-1 classification accuracy is now over 40%, whereas in the case of VGG16 it was less than 4%. As mentioned earlier, this is due to the fact that tensors $\mathbf{X}$ produced by the ResNet34 edge sub-model (without loss) contain virtually no zeros, so they are quite different compared to the corrupt $\tilde{\mathbf{X}}$ which haven’t been completed yet. This also means that tensor completion (TC) has the potential to bring much higher gain over no-completion (NC), compared to the VGG16 case.
Table 4.5: ResNet34 classification accuracy results. Among the last four columns, the first two show the results with default settings, and the last two show the results with matched execution speed.

| $p_{\text{loss}}$ | Algorithm | $\mu_{\text{NL}}$ | $\mu_{\text{NC}}$ | $\sigma_{\text{NC}}$ | Default | Speed-matched |
|-------------------|-----------|-------------------|-------------------|-------------------|---------|--------------|
|                   |           | $\mu_{\text{TC}}$ | $\sigma_{\text{TC}}$ | $\mu_{\text{TC}}$ | $\sigma_{\text{TC}}$ | $\mu_{\text{TC}}$ | $\sigma_{\text{TC}}$ |
| 5%                | SiLRTC    | 58.10%            | 57.57%            | 0.61%             | 57.77%  | 0.49%        | 57.75%  | 0.53%        |
|                   | HaLRTC    | 58.10%            | 57.57%            | 0.61%             | 57.94%  | 0.37%        | 57.75%  | 0.61%        |
|                   | FCP       | 58.10%            | 57.57%            | 0.61%             | 57.92%  | 0.43%        | 57.59%  | 0.60%        |
|                   | ALTeC     | 58.10%            | 57.57%            | 0.61%             | 58.04%  | 0.44%        | 58.04%  | 0.44%        |
| 10%               | SiLRTC    | 58.10%            | 54.57%            | 0.68%             | 56.47%  | 0.60%        | 56.12%  | 0.66%        |
|                   | HaLRTC    | 58.10%            | 54.57%            | 0.68%             | 57.65%  | 0.46%        | 54.57%  | 0.68%        |
|                   | FCP       | 58.10%            | 54.57%            | 0.68%             | 56.56%  | 0.66%        | 55.98%  | 0.69%        |
|                   | ALTeC     | 58.10%            | 54.57%            | 0.68%             | 57.18%  | 0.61%        | 57.18%  | 0.61%        |
| 15%               | SiLRTC    | 58.10%            | 49.30%            | 0.78%             | 53.89%  | 0.64%        | 52.84%  | 0.71%        |
|                   | HaLRTC    | 58.10%            | 49.30%            | 0.78%             | 57.02%  | 0.51%        | 49.31%  | 0.78%        |
|                   | FCP       | 58.10%            | 49.30%            | 0.78%             | 53.96%  | 0.75%        | 53.20%  | 0.78%        |
|                   | ALTeC     | 58.10%            | 49.30%            | 0.78%             | 55.09%  | 0.71%        | 55.09%  | 0.71%        |
| 20%               | SiLRTC    | 58.10%            | 40.87%            | 0.86%             | 49.61%  | 0.77%        | 48.64%  | 0.80%        |
|                   | HaLRTC    | 58.10%            | 40.87%            | 0.86%             | 56.26%  | 0.60%        | 40.87%  | 0.86%        |
|                   | FCP       | 58.10%            | 40.87%            | 0.86%             | 49.76%  | 0.76%        | 49.10%  | 0.87%        |
|                   | ALTeC     | 58.10%            | 40.87%            | 0.86%             | 51.99%  | 0.72%        | 51.99%  | 0.72%        |
| 25%               | SiLRTC    | 58.10%            | 29.11%            | 0.86%             | 43.56%  | 0.87%        | 41.40%  | 0.99%        |
|                   | HaLRTC    | 58.10%            | 29.11%            | 0.86%             | 55.09%  | 0.65%        | 29.11%  | 0.87%        |
|                   | FCP       | 58.10%            | 29.11%            | 0.86%             | 44.10%  | 0.81%        | 43.07%  | 0.82%        |
|                   | ALTeC     | 58.10%            | 29.11%            | 0.86%             | 47.52%  | 0.67%        | 47.52%  | 0.67%        |
| 30%               | SiLRTC    | 58.10%            | 15.72%            | 0.77%             | 34.56%  | 0.89%        | 31.85%  | 0.83%        |
|                   | HaLRTC    | 58.10%            | 15.72%            | 0.77%             | 53.63%  | 0.68%        | 15.73%  | 0.77%        |
|                   | FCP       | 58.10%            | 15.72%            | 0.77%             | 36.06%  | 0.76%        | 34.93%  | 0.80%        |
|                   | ALTeC     | 58.10%            | 15.72%            | 0.77%             | 41.23%  | 0.80%        | 41.23%  | 0.80%        |

In Table 4.5, the middle two columns labeled $\mu_{\text{TC}}$ and $\sigma_{\text{TC}}$ correspond to the case with default settings, where SiLRTC, HaLRTC, and FCP are able to execute all $K = 50$ iterations until convergence. Again, I provide the prediction accuracy histograms in Figure B.3 to uncover any differences in the distributions at each loss level. From the figure, one can see that distributions are not centered around a similar mean. This strongly indicates that there must be at least one loss level where the distributions statistically differ. To test this assumption, the corresponding statistical significance results using Welch’s t-test are shown in the middle three columns of Table 4.6. As seen in the table, there are a lot more significant differences in prediction accuracy now, compared to the VGG16 case. At the 5% loss level, ALTeC, HaLRTC, and FCP are statistically tied, and all give a higher classification accuracy than SiLRTC. At higher loss levels, HaLRTC statistically outperforms ALTeC, SiLRTC, and FCP, with ALTeC being the next best performing algorithm, followed by FCP and
SiLRTC. For this reason, the corresponding accuracies are made bold in the table. Note that these statistical significance results are also easily observable in Figure B.3, as HaLRTC’s distributions always appear superior (visually, centered around a higher mean) to other methods, except for at the 5% loss level.

Table 4.6: ResNet34 statistical significance results. The middle three columns show $p$-values of the 2-sided Welch’s t-test for pairwise comparison of accuracies with default settings, while the last three columns show $p$-values for the case when the execution speeds are matched.

| $p_{loss}$ | Algorithm | Default | Speed-matched |
|-----------|-----------|---------|---------------|
|          |           | ALTeC  | SiLRTC | HaLRTC | ALTeC | SiLRTC | HaLRTC |
| 5%       | SiLRTC    | 7.35E-06 | - | - | 4.42E-05 | - | - |
|          | HaLRTC    | 0.105  | 0.022 | 0.687 | 3.93E-09 | 0.031 | - |
|          | FCP       |         |         |         | 1.19E-08 | 0.055 | 0.806 |
| 10%      | SiLRTC    | 3.79E-15 | - | - | 2.2E-24 | - | - |
|          | HaLRTC    | 2.84E-09 | 2.64E-35 | - | 3.3E-71 | 1.98E-38 | - |
|          | FCP       | 1.49E-10 | 0.317 | 7.69E-29 | 6.21E-28 | 0.148 | 6.40E-33 |
| 15%      | SiLRTC    | 1.15E-26 | - | - | 6.3E-56 | - | - |
|          | HaLRTC    | 9.90E-53 | 2.81E-90 | - | 1.67E-120 | 8.25E-83 | - |
|          | FCP       | 7.90E-28 | 0.178 | 3.50E-82 | 8.24E-26 | 1.28E-16 | 6.62E-98 |
| 20%      | SiLRTC    | 3.23E-56 | - | - | 3.13E-77 | - | - |
|          | HaLRTC    | 2.19E-104 | 2.87E-133 | - | 1.05E-166 | 3.43E-136 | - |
|          | FCP       | 1.40E-52 | 0.165 | 2.22E-132 | 5.99E-47 | 1.65E-148 | 6.80E-145 |
| 25%      | SiLRTC    | 7.16E-86 | - | - | 3.54E-106 | - | - |
|          | HaLRTC    | 5.46E-153 | 8.97E-167 | - | 5.33E-205 | 7.42E-163 | - |
|          | FCP       | 8.25E-80 | 8.59E-06 | 2.02E-170 | 9.17E-65 | 6.25E-64 | 2.85E-193 |
| 30%      | SiLRTC    | 4.28E-122 | - | - | 2.05E-153 | - | - |
|          | HaLRTC    | 4.93E-182 | 1.62E-205 | - | 7.43E-242 | 9.33E-200 | - |
|          | FCP       | 7.09E-109 | 1.14E-27 | 2.08E-215 | 7.78E-88 | 8.29E-108 | 4.57E-227 |

In Table 4.5 and Table 4.6 the columns labelled “Speed-matched” represent a configuration where the execution speeds of the four tensor completion algorithms are matched. In this case, SiLRTC and HaLRTC are only able to run 1-3 iterations, while FCP can run 3-8 iterations. The corresponding prediction accuracy histograms over the 100 trials are provided in Figure B.4. From the figure, one can be confident that a given algorithm will be superior to the rest in at least one loss level. This is due to the fact that the distributions are centered around varying means for loss levels above 5%. From the $p$-values presented in the speed-matched analysis of Table 4.6, it is clear that ALTeC statistically outperforms the other three methods at all loss levels.

In summary, ResNet34 results show that when complexity is not constrained and execution speed is of no concern, HaLRTC is the best of the four methods, followed by ALTeC in the second place. However, when the execution speeds are matched, ALTeC is superior to the other three methods.
4.3.3 Interesting Examples

Again, I highlight several interesting examples that were observed during the experiments on ResNet34 with default configurations of the tensor completion algorithms. Table 4.7 shows the classification predictions made on two images: #4 (‘Toyshop’) and #7 (‘Drake’). Ground truth labels are listed in the row that starts with “GT Label”. The next two rows respectively show the prediction label and confidence results under no loss (NL), obtained from the pre-trained ResNet34 model. In both cases, Top-1 labels are correct, but the model is relatively less confident about image #4 (16.66%). After packet loss of 30%, but without tensor completion (NC), the model makes wrong predictions in both cases - image #4 is classified as ‘Abacus’ and #7 is classified as ‘Ptarmigan’ - though both predictions are made with low confidence (7.89% and 26.57%, respectively).

| Image  | Trial # | Algorithm | GT Label | NL Label | NL Score | NC Label | NC Score | TC Label | TC Score |
|--------|---------|-----------|----------|----------|----------|----------|----------|----------|----------|
| 4      | 52      | ALTeC     | Toyshop  | Toyshop  | 16.66%   | Abacus   | 7.89%    | Toyshop  | 6.32%    |
|        |         | FCP       | Toyshop  | Toyshop  | 16.66%   | Abacus   | 7.89%    | Toyshop  | 6.47%    |
|        |         | HaLRTC    | Toyshop  | Toyshop  | 16.66%   | Abacus   | 7.89%    | Toyshop  | 20.06%   |
|        |         | SiLRTC    | Toyshop  | Toyshop  | 16.66%   | Abacus   | 7.89%    | Toyshop  | 6.89%    |
| 7      | 27      | ALTeC     | Drake    | Drake    | 78.19%   | Ptarmigan| 26.57%   | AC       | 57.92%   |
|        |         | FCP       | Drake    | Drake    | 78.19%   | Ptarmigan| 26.57%   | AC       | 59.83%   |
|        |         | HaLRTC    | Drake    | Drake    | 78.19%   | Ptarmigan| 26.57%   | AC       | 75.67%   |
|        |         | SiLRTC    | Drake    | Drake    | 78.19%   | Ptarmigan| 26.57%   | AC       | 62.62%   |

WS = Window Screen, CR = Coral Reef, SC = Shower Curtain, AC = American Coot

Finally, after tensor completion (TC), ALTeC produces correct prediction result in the case of image #4 (‘Toyshop’), but with low confidence (6.32%), while SiLRTC, HaLRTC, and FCP are wrong. In the case of image #7, all four methods produce wrong results, but ALTeC is the least confident about it (57.92% vs. 60-75% for other methods). Again, among the four wrong decisions, the best one is where the model has the least confidence. Note that in the case of image #7, although final TC labels are wrong for all the methods, they are relatively similar to the ground truth, as both ‘Drake’ and ‘American Coot’ are birds that resemble each other.

As in the VGG16 case, it is possible to further understand ALTeC’s deep feature tensor reconstruction superiority for these specific images by computing the $L_1$ norm for a given channel, using Equation 4.2. In this case, channel #73 was chosen from the deep feature tensors produced by both images and the completion results are illustrated in Figures A.4 & A.5, respectively. Once again, the corrupt row(s) are highlighted red for ease of reference. Here these include the following: row $\in \{2, 5, 8, 10, 16, 18, 20, 21, 26, 28\}$ (Figure A.4) or row $\in \{4, 5, 6, 11, 13, 16, 21, 22, 23, 24, 25\}$ (Figure A.5) and the corresponding $L_1$ norm errors are provided in Table 4.8.
Table 4.8: $L_1$ norm difference values for channel #73

| L$_1$ Norm Case | Algorithm   |   |   |   |   |   |
|-----------------|-------------|---|---|---|---|---|
|                 | SiLRTC      | HaLRTC | FCP | ALTeC | Row |
| Image #4 (Figure A.4) | 4954        | 3520  | 2586 | 2685 | 2  |
|                  | 5587        | 3426  | 2171 | 2051 | 5  |
|                  | 4198        | 2609  | 2105 | 2015 | 8  |
|                  | 5302        | 2885  | 1852 | 1228 | 10 |
|                  | 4885        | 2971  | 2327 | 2094 | 16 |
|                  | 4406        | 2779  | 2267 | 1985 | 18 |
|                  | 4089        | 2605  | 2195 | 2811 | 20 |
|                  | 3972        | 2739  | 1903 | 1307 | 21 |
|                  | 4056        | 2866  | 2177 | 1502 | 26 |
|                  | 4470        | 3787  | 3655 | 2876 | 28 |
| Average          | 4591.9      | 3018.7| 2323.8| 2055.4 |   |

| Image #7 (Figure A.5) | 6334 | 2388 | 2410 | 815 | 4 |
|                       | 6480 | 1754 | 2379 | 986 | 5 |
|                       | 6159 | 2032 | 2080 | 1368 | 6 |
|                       | 5668 | 2384 | 2131 | 2241 | 11 |
|                       | 5505 | 2341 | 2609 | 2534 | 13 |
|                       | 5043 | 2533 | 2062 | 2360 | 16 |
|                       | 4802 | 2413 | 2450 | 2903 | 21 |
|                       | 4141 | 2822 | 2452 | 2847 | 22 |
|                       | 3886 | 2560 | 2671 | 2814 | 23 |
|                       | 4723 | 2430 | 2203 | 1645 | 24 |
|                       | 4828 | 2721 | 2437 | 2154 | 25 |
| Average              | 5233.5| 2398 | 2353.1| 2060.6 |   |

From Table 4.8, ALTeC once again has the lowest average $L_1$ norm values out of all the algorithms. This indicates that in both cases, ALTeC-completed channels were the closest to ground truth (NL) channels, with FCP being second-best, as in the VGG16 case. However, note that here SiLRTC performs relatively poorly compared to the other algorithms for both images, which indicates that this specific channel was especially problematic for such a “simple” algorithm. This can be seen by the near zero values in its completed rows in both Figure A.4 and Figure A.5.
Chapter 5
Conclusions and Future Work

In this thesis, I studied several methods for tensor completion in collaborative intelligence applications. Specifically, I focused on three representative methods from the literature - Simple Low Rank Tensor Completion (SiLRTC), High-accuracy Low Rank Tensor Completion (HaLRTC), and Fused Canonical Polyadic (FCP) - and a simple newly-developed Adaptive Linear Tensor Completion (ALTeC). These algorithms were compared on their ability to recover the missing data caused by packet loss in the deep feature tensors produced by image classification models, namely VGG16 and ResNet34. Among the four studied methods, ALTeC was the fastest, which is well-suited for collaborative intelligence applications where inference latency is one of the important issues. Regarding reconstruction accuracy, on VGG16 tensors (which tend to be sparse), all four methods were in a statistical tie, both when SiLRTC, HaLRTC, and FCP were allowed to run sufficiently many iterations to converge and when their execution speeds were matched with that of ALTeC by restricting the number of iterations. On ResNet34 tensors (which are generally non-sparse), HaLRTC statistically showed the best prediction accuracy when it was allowed to converge, followed by ALTeC as the second-best. However, when the execution speeds were matched, ALTeC emerged as the winner.

In essence, SiLRTC, HaLRTC, and FCP pay the price for their generality. By not embedding the specifics of tensors they are supposed to complete into their procedures, they need to re-discover low-rank tensor structures anew every time they start the completion procedure. By contrast, ALTeC learns simple linear relations among the rows of tensors it is supposed to complete off-line, so it is able to execute quickly at run-time. While ALTeC requires off-line training, this is quite feasible in collaborative intelligence because the main model (VGG16 or ResNet34 in this case) also requires off-line training, and ALTeC could be trained in parallel with the main model on the same data. Further, note that ALTeC does not require labeled data for training, only input data. Each input sample produces the “ground truth” deep feature tensor at a given (split) layer of the model, which is then used to fit ALTeC’s parameters.
Note that both fast methods, as well as slower but more accurate methods, may have their place in CI tensor completion, depending on the application. For example, in a video surveillance application where a subway station is being monitored by several cameras to detect abandoned luggage, speed is of the essence, since the luggage may pose a security threat. Meanwhile, the accuracy of luggage classification (suitcase, backpack, purse) is less relevant. Hence, for such an application, inference latency (i.e., speed) would be more important than classification accuracy. On the other hand, for applications such as satellite-based surveillance of crops, speed is not as important since changes on crop fields are relatively slow. Hence, in this application, accuracy could be preferred over speed.

Further improvements to the work presented in this thesis would incorporate other tensor completion methods [43] in the context of collaborative intelligence, such as Geometric Conjugate Gradients (GeomCG) [44], Tensor SVD (t-SVD) [45], and Tensor Robust Principal Component Analysis (TRPCA) [46]. Like SiLRTC, HaLRTC, and FCP, most of these completion algorithms rely on computation-heavy procedures such as eigen-decomposition or SVD, so they are unlikely to be faster than ALTeC, but they may offer better prediction accuracy in cases where complexity is less important. The inclusion of specific constrains found in image or video completion algorithms [47,48] could also be explored. Moreover, similar studies could be performed on models trained for other collaborative intelligence applications, such as object detection, segmentation, action recognition, etc. Finally, the completion algorithms should also be evaluated on a burst-loss channel model such as the Gilbert-Elliott model [49], which offers a more realistic representation of packet loss in real networks.
References

[1] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, “Neurosurgeon: Collaborative intelligence between the cloud and edge,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615-629, Apr. 2017.

[2] A. E. Eshratifar, M. S. Abrishami and M. Pedram, “JointDNN: An Efficient Training and Inference Engine for Intelligent Mobile Cloud Computing Services,” *IEEE Trans. Mobile Computing*, Early Access, 2019.

[3] H. Choi and I. V. Bajić, “Deep feature compression for collaborative object detection,” *Proceedings of the 25th IEEE International Conference on Image Processing*, pp. 3743-3747, Oct. 2018.

[4] S. R. Alvar and I. V. Bajić, “Multi-task learning with compressible features for collaborative intelligence,” *Proceedings of the 26th IEEE International Conference on Image Processing*, pp. 1705-1709, Sep. 2019.

[5] J. Liu, P. Musialski, P. Wonka, and J. Ye, “Tensor completion for estimating missing values in visual data,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 1, pp. 208-220, Jan. 2012.

[6] Y. Wu, H. Tan, Y. Li, J. Zhang, X. Chen, “A fused CP factorization method for incomplete tensors,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 3, pp. 751-764, Mar. 2019.

[7] L. Bragilevsky and I. V. Bajić, “Tensor Completion Methods for Collaborative Intelligence,” *IEEE Access*, vol. 8, no. 1, pp. 41162-41174, Dec. 2020.

[8] S. R. Alvar, H. Choi, and I. V. Bajić, “Can you tell a face from a HEVC bitstream?,” *IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, pp. 257-261, Apr. 2018.

[9] S. R. Alvar, H. Choi, and I. V. Bajić, “Can you find a face in a HEVC bitstream?,” *Proc. IEEE ICASSP’18*, pp. 1288-1292, Calgary, AB, Apr. 2018.

[10] A. E. Eshratifar, A. Esmaili and M. Pedram, “BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services,” *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1-6, Lausanne, Switzerland, Jul. 2019.

[11] Z. Chen, K. Fan, S. Wang, L. Duan, W. Lin, and A. Kot, “Lossy Intermediate Deep Learning Feature Compression and Evaluation,” *Proc. 27th ACM International Conference on Multimedia (MM’19)*, pp. 2414–2422, Nice, France, Oct. 2019.
[12] Z. Chen, K. Fan, S. Wang, L. Duan, W. Lin, and A. C. Kot, “Toward Intelligent Sensing: Intermediate Deep Feature Compression,” IEEE Transactions on Image Processing, vol. 29, pp. 2230-2243, Sep. 2020.

[13] H. Choi, R. A. Cohen, and I. V. Bajić, “Back-and-forth prediction for deep tensor compression,” Proc. IEEE ICASSP’20, Barcelona, Spain, May 2020. (to appear)

[14] S. R. Alvar and I. V. Bajić, “Bit allocation for multi-task collaborative intelligence,” Proc. IEEE ICASSP’20, Barcelona, Spain, May 2020. (to appear)

[15] R. A. Cohen, H. Choi, and I. V. Bajić, “Lightweight compression of neural network feature tensors for collaborative intelligence,” Proc. IEEE ICME’20, London, UK, Jul. 2020. (to appear)

[16] M. Ulhaq and I. V. Bajić, “Shared mobile-cloud inference for collaborative intelligence,” demo at NeurIPS’19, Vancouver, BC, Dec. 2019. https://arxiv.org/abs/2002.00157

[17] V. Smith, C. K. Chiang, M. Sanjabi, and A. S. Talwalkar, “Federated multi-task learning,” Advances in Neural Information Processing Systems, pp. 4424-4434, 2017.

[18] Q. Yang, Y. Liu, T. Chen, and Y. Tong, “Federated machine learning: Concept and applications,” ACM Transactions on Intelligent Systems and Technology (TIST), vol. 10, no. 2, pp. 1-19, Jan. 2019.

[19] X. Wang, “Deep learning in object recognition, detection, and segmentation,” Foundations and Trends in Signal Processing, vol. 8, no. 4, pp. 217–382, 2016.

[20] D. Rumelhart, G. Hinton, and R. Williams, “Learning representations by back-propagating errors,” Nature, vol. 323, pp. 533–536, Oct. 1986.

[21] X. Chen, J. D. Lee, X. T. Tong, and Y. Zhang, “Statistical inference for model parameters in stochastic gradient descent,” The Annals of Statistics, vol. 48, no. 1, pp. 251-273, 2020.

[22] R. Hecht-Nielsen, “Theory of the backpropagation neural network,” Neural Networks, vol. 1, pp. 445–448, 1988.

[23] X. Li, Y. Grandvalet, and F. Davoine, “A baseline regularization scheme for transfer learning with convolutional neural networks,” Pattern Recognition, pp. 107049, Feb. 2020.

[24] J. G. A. Barbedo, “Impact of dataset size and variety on the effectiveness of deep learning and transfer learning for plant disease classification,” Computers and Electronics in Agriculture, vol. 153, pp. 46-53, Oct. 2018.

[25] D. X. Zhou, “Universality of deep convolutional neural networks,” Applied and Computational Harmonic Analysis, vol. 48, no. 2, pp.787-794, 2020.

[26] G. H. Golub and C. Reinsch, “Singular value decomposition and least squares solutions,” Linear Algebra, pp. 134-151, 1971.

[27] J.-F. Cai and E. Candés, “A Singular Value Thresholding Algorithm for Matrix Completion,” SIAM J. Optimization, vol. 20, no. 4, pp. 1956-1982, 2010.
[28] A. H. Phan, P. Tichavský, and A. Cichocki, “Error preserving correction: A method for CP decomposition at a target error bound,” *IEEE Transactions on Signal Processing*, vol. 67, no. 5, pp. 1175-1190, Dec. 2018.

[29] A. H. Phan, P. Tichavský, and A. Cichocki, “Partitioned hierarchical alternating least squares algorithm for CP tensor decomposition,” *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 2542-2546, Mar. 2017.

[30] Y. Wang, J. Ostermann, and Y.-Q. Zhang, *Video Processing and Communications*, Prentice-Hall, 2002.

[31] Q. Song, H. Ge, J. Caverlee, and X. Hu, “Tensor completion algorithms in big data analytics,” *ACM Transactions on Knowledge Discovery from Data*, vol. 13, no. 1, pp. 6, Jan. 2019.

[32] D. Kressner, M. Steinlechner, and B. Vandereycken, “Low-rank tensor completion by Riemannian optimization,” *BIT Numerical Mathematics*, vol. 54, no. 2, pp. 447-468, Jun. 2014.

[33] X. Yuan, P. He, Q. Zhu, and X. Li, “Adversarial examples: Attacks and defenses for deep learning,” *IEEE Trans. Neural Networks and Learning Systems*, vol. 30, no. 9, pp. 2805-2824, Sept. 2019.

[34] J. Eckstein and D. P. Bertsekas, “On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators,” *Mathematical Programming*, vol. 55, pp. 293-318, 1992.

[35] H. Stark and J. W. Woods, *Probability, Statistics, and Random Processes for Engineers*, 4th Edition, Prentice-Hall, 2012.

[36] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, and A. C. Berg, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211-252, Dec. 2015.

[37] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *Proc. ICLR’15*, San Diego, CA, May 2015.

[38] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition,” *Proc. IEEE CVPR’16*, pp. 770-778, Las Vegas, NV, Jun. 2016.

[39] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” RFC 3550, Jul. 2003.

[40] H. Choi and I. V. Bajić, “Near-lossless deep feature compression for collaborative intelligence,” *IEEE 20th International Workshop on Multimedia Signal Processing*, pp. 1-6, Aug. 2018.

[41] D. J. Sheskin, “Handbook of Parametric and Nonparametric Statistical Procedures”, 2nd Edition, Chapman and Hall, 2000.

[42] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *Proc. ICML’15*, 2015.
[43] L. Grasedyck, D. Kressner, and C. Tobler, “A literature survey of low-rank tensor approximation techniques,” *GAMM-Mitteilungen*, vol. 36, no. 1, pp. 53-78, Aug. 2013.

[44] H. Kasai and B. Mishra, “Low-rank tensor completion: a Riemannian manifold preconditioning approach,” *International Conference on Machine Learning*, pp. 1012-1021, Jun. 2016.

[45] Z. Zhang, G. Ely, S. Aeron, N. Hao, and M. Kilmer, “Novel methods for multilinear data completion and de-noising based on tensor-SVD,” *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3842-3849, 2014.

[46] C. Lu, J. Feng, W. Liu, Z. Lin, and S. Yan, “Tensor robust principal component analysis with a new tensor nuclear norm,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1-14, Jan. 2019.

[47] J. A. Bengua, H. N. Phien, H. D. Tuan, and M. N. Do, “Efficient tensor completion for color image and video recovery: Low-rank tensor train,” *IEEE Transactions on Image Processing*, vol. 26, no. 5, pp. 2466-2479, Feb. 2017.

[48] W. Wang, V. Aggarwal, and S. Aeron, “Efficient low rank tensor ring completion,” *Proceedings of the IEEE International Conference on Computer Vision*, pp. 5697-5705, Oct. 2017.

[49] G. Haßlinger and O. Hohlfeld, “The Gilbert-Elliott model for packet loss in real time services on the Internet,” *14th GI/ITG Conference-Measurement, Modelling, and Evaluation of Computer and Communication Systems*, pp. 1-15, Mar. 2008.
Appendix A

Interesting Examples - Reconstruction Analysis

This chapter analyzes the completion performance of each tensor completion algorithm at a loss level of 30% for both the VGG16 (Section 4.2) and ResNet34 (Section 4.3) models. The images that are used in each case are presented in Figure A.1, and the individual cases follow afterwards in Figures A.2-A.5.

(a) Image 102, Trial 85 (VGG16)  (b) Image 3, Trial 35 (VGG16)

(c) Image 4, Trial 52 (ResNet34)  (d) Image 7, Trial 27 (ResNet34)

Figure A.1: Ground Truth Images
Figure A.2: Channel #257 of Figure A.1a (VGG16), ordered left → right: NL, NC, SiLRTC, HaLRTC, FCP, ALTeC
Figure A.3: Channel #257 of Figure A.1b (VGG16), ordered left → right: NL, NC, SiLRTC, HaLRTC, FCP, ALTeC
Figure A.4: Channel #73 of Figure A.1c (ResNet34), ordered left → right: NL, NC, SiLRTC (cont’d next page)
Figure A.4: Channel #73 of Figure A.1c (ResNet34), ordered left → right: HaLRTC, FCP, ALTeC
Figure A.5: Channel #73 of Figure A.1d (ResNet34), ordered left → right: NL, NC, SiLRTC (cont’d next page)
Figure A.5: Channel #73 of Figure A.1d (ResNet34), ordered left → right: HaLRT, FCP, ALTeC
Appendix B

Tensor Reconstruction Accuracy Histograms

The following pages illustrate the tensor reconstruction accuracies for each algorithm in the form of a histogram to highlight the statistical difference between the algorithms over the 100 trials. The order of the next few pages is:

1. VGG16 Default Case → Figure B.1
2. VGG16 Speed-matched Case → Figure B.2
3. ResNet34 Default Case → Figure B.3
4. ResNet34 Speed-matched Case → Figure B.4

Note that as indicated throughout the thesis, the default case is when the algorithms are allowed to converge, while the speed-matched case is when all algorithms have the same execution speed as ALTeC.
Figure B.1: VGG16 Tensor Reconstruction Accuracy Histograms \( p_{loss} \in \{5\%, 10\%, 15\%\} \), default case
Figure B.1: VGG16 Tensor Reconstruction Accuracy Histograms ($p_{\text{loss}} \in \{20\%, 25\%, 30\%\}$, default case)
Figure B.2: VGG16 Tensor Reconstruction Accuracy Histograms ($p_{loss} \in \{5\%, 10\%, 15\%\}$, speed-matched case)
Figure B.2: VGG16 Tensor Reconstruction Accuracy Histograms ($p_{loss} \in \{20\%, 25\%, 30\%\}$, speed-matched case)
Figure B.3: ResNet34 Tensor Reconstruction Accuracy Histograms ($p_{\text{loss}} \in \{5\%, 10\%, 15\\%\}$, default case)
Figure B.3: ResNet34 Tensor Reconstruction Accuracy Histograms ($p_{\text{loss}} \in \{20\%, 25\%, 30\%\}$, default case)
Figure B.4: ResNet34 Tensor Reconstruction Accuracy Histograms ($p_{\text{loss}} \in \{5\%, 10\%, 15\%\}$, speed-matched case)
Figure B.4: ResNet34 Tensor Reconstruction Accuracy Histograms ($p_{\text{loss}} \in \{20\%, 25\%, 30\%\}$, speed-matched case)