IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 14 August 2024, accepted 2 September 2024, date of publication 11 September 2024,
date of current version 30 September 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3457863

RESEARCH ARTICLE

A Generalist Reinforcement Learning Agent for Compressing Multiple Convolutional Networks Using Singular Value Decomposition

GABRIEL GONZALEZ-SAHAGUN, SANTIAGO ENRIQUE CONANT-PABLOS, (Member, IEEE), JOSÉ CARLOS ORTIZ-BAYLISS, (Member, IEEE), AND JORGE M. CRUZ-DUARTE, (Senior Member, IEEE)
School of Engineering and Sciences, Tecnologico de Monterrey, Monterrey, Nuevo Leon 64849, Mexico
Corresponding author: Santiago Enrique Conant-Pablos (sconant@tec.mx)

ABSTRACT Deep learning models have gained popularity in the last decade for computer vision tasks. Although these models are widely used, they process data in cloud services due to requiring large amounts of memory unavailable on consumer devices. Multiple techniques have been proposed to reduce the memory needed for these models. Nonetheless, finding the best method to compress each model can be a time-consuming process as the parameters of these techniques significantly affect the results. We propose a methodology for training a reinforcement learning model that exploits similarities between models to select how to compress other models it has not seen before. By reusing the generalist agent and exploiting the similarities, searching for how to compress a new model can be avoided. The agent receives a set of feature maps and compresses a model by choosing the percentage of singular values to use in a low-rank factorization of the weights of each layer. We chose the feature maps by generating an embedding for all the images and selecting the most representative image of each class. Our agent trained to compress two models, the first trained using fashion MNIST, whereas the second, using Kuzushiji-MNIST, reduced a model trained on MNIST to 15% of its original size with minimal accuracy loss. Reusing the generalist agent permitted us to skip 4.6 days of searching for a solution for MNIST.

INDEX TERMS Computer vision, deep learning, model compression, model optimization, reinforcement learning, singular value decomposition, low-rank factorization.

I. INTRODUCTION

Most Convolutional Neural Networks (CNNs) have many redundant weights when performing transfer learning. When training a model for a new task, the model is typically selected from the state-of-the-art instead of being created from scratch. The reason for reusing a pre-trained CNN model is that its lower layers, which are convolutional, learn filters to extract useful features from the images. These filters can be reused as they detect shapes and colors that may be useful for other tasks. Furthermore, different datasets might have a subset of the same classes or similar images. The knowledge is transferred and tweaked for the new dataset by retraining the feature extraction layers (i.e., convolutional layers) and removing the dense layers since the weights in the first dense layers depend on the height and width of the images used throughout the training process. Although transfer learning can reduce training time using prior knowledge, the model can have more parameters than required. Alvarez and Salzmann [1] confirm that networks are "known to have many redundant parameters." Due to the size of deep learning models, real-world applications of deep learning models tend to process data using cloud services instead of running it onsite. Processing data on the cloud is not efficient due to the latency of the internet connection. Furthermore, there are locations where there is no internet connection.

Techniques to reduce the size of deep learning models can be summarized in two streams: compressing a trained model or performing a Neural Architecture Search (NAS) to

The associate editor coordinating the review of this manuscript and approving it for publication was Yongjie Li.

generate suitable network architecture for a given problem. In the first stream, the number of parameters of a layer (also referred to as weights) is reduced using one or more kinds of methods, such as pruning, quantization, Low-Rank Factorization (LRF), and Knowledge Distillation (KD). A pruning method removes the parameters that satisfy specific criteria. In Absolute Weighted Sum (AWS), the filters with the lowest magnitudes are pruned [2]. Chang et al. clustered the filters in a layerwise fashion to summarize redundant filters using the centroids of the clusters [3]. Another pruning method called Average Percentage of Zeros (APoZ) removes the neurons with the least activations for the dataset [4]. When used to compress LeNet trained on MNIST, APoZ achieved a 3.85x compression rate and an accuracy of 99.26%. As for VGG16 trained on ImageNet, the results were a 2.11x compression rate and a top-1 accuracy of 70.88%. LRF methods are data compression methods that approximate a matrix subject to constraints, like the rank of the approximating matrix. Two LRF techniques for compressing deep learning models are Principal Component Analysis (PCA) [5] and Singular Value Decomposition (SVD) [6], [7].

Some researchers have opted to use gradient descent algorithms for optimizing layers via unconstrained optimization. Liu et al. [42] decomposed the convolution into three steps to find a fast sparsified version of the convolution. The first step was a matrix multiplication between the input feature maps and the channel basis of the kernel. Then, the result was convolved by a kernel basis. Finally, the output was calculated by performing another matrix multiplication with a sparse kernel matrix $\mathcal{S}$. The compression was achieved by minimizing the logistic loss function of the output layer of the network plus the element-wise l_1 and l_2 norms of the sparse kernel matrix $\mathcal{S}$. In addition to having a speedup between 2.24x and 6.88x for each layer, the sparsity of the sparse kernel matrix was between 92.7% and 95.1%. Although not clearly stated, it can be concluded that compression was achieved due to the high sparsity of the sparse kernel matrix because its size is the biggest among the three matrices. Sparse Weight Activation Training (SWAT) [43] is a CNN training algorithm that searches for sparse topologies while training a CNN. The sparsity produced by the algorithm results in a model compression of 5.0x for ResNet-50 trained on ImageNet.

Knowledge Distillation (KD) is an approach for training a small model using data labeled by a bigger model, also known as teacher. Nonetheless, a set of layers can be trained to emulate another set of more complex layers. For instance, a shallower version of BERT, a language model, was trained using the intermediate layer information from the original large model [8]. In the case of computer vision models, the output feature map of a convolutional layer can be predicted by another kind of layer, such as an MLPConv layer [9] or the depth-wise separable convolution used in Xception [10]. In the second stream, an algorithm is used to find the optimal number of layers and their corresponding parameters to maximize the accuracy for a given dataset while minimizing the number of parameters of the model. According to Ren et al. [11], all NAS frameworks use a controller for searching the space and generating candidate architectures. Then, an evaluation strategy scores and ranks the architectures to select the optimal architecture. Evaluating candidates is the most time-consuming because it involves training the candidate architecture on a training dataset and testing the accuracy of the architecture on the test dataset. NAS is a multi-objective optimization problem that requires training and testing randomly generated neural networks. For small datasets, it is feasible to use this approach. As for larger datasets like ImageNet, it would be impractical to train several models as a single epoch can take more than an hour. Since Neural Architecture Search is more manageable for smaller datasets, Zoph et al. scaled the architecture found for a small dataset to handle a more complex dataset [12]. They used reinforcement learning to discover an architecture that maximized the accuracy for CIFAR-10. Then, they reused modules of the architecture to build a model with more layers, which they later trained on ImageNet. The modules were similar to the inception module found in GoogLeNet [13]. Another strategy for reducing the long training time is the parameter sharing of the proposed architectures. In Efficient Neural Architecture Search, candidate architectures are subgraphs of a larger graph that considers all the search space [14]. Since candidates are a selection of nodes to produce a Directed Acyclic Graph, the candidates that select the same nodes will be sharing the parameters of those nodes. Thus, those candidates update the weights of a node sequentially.

Finding which compressors to use and the parameters to get the best results can be challenging. Multiple solutions that involve Reinforcement Learning (RL) have been proposed to solve this problem [15], [16], [17], [18]. Nonetheless, RL could be more efficient when training an agent to compress only one model. Since the process involves generating samples to train the agent, training the agent, and sometimes fine-tuning, the computational cost is more than other solutions, such as genetic algorithms and particle swarm optimization, that do not require training the agent or fine-tuning depending on how the model is compressed. The only advantage of using RL over Stochastic Optimization (SO) algorithms is that RL can find patterns between states; this can help reuse a trained agent on other models, thus avoiding the time it takes to search for and train the agent. We refer to agents capable of solving multiple problems as *generalist agents* as the term has been used before in the literature. However, we use the term to refer to an agent capable of solving multiple instances of model compression. In contrast, Reed et al. [19] use the term for an agent capable of working in various environments.

Generalist agents do not bring only advantages. One major disadvantage is that the number of states an agent has to learn is much larger. Let us suppose we train a generalist agent to compress three layers for a small dataset, such as

MNIST, using ten actions; it will look at 1.8 million states (3 layers $\times$ 60, 000 images $\times$ 10 actions). If trained on three different datasets of the same size, that will represent looking at 5.4 million states. Training an agent becomes unfeasible in a reasonable amount of time for compressing more layers, datasets with more examples, and more datasets. Giving priority to the most representative states could reduce the training time to a reasonable amount. Not only is the training time a problem but the stability of training is also affected by the number of states. As there are too many states, the model can forget what it learned from a state. If a smaller subset of states is used to train the generalist agent, it is easier for the agent to learn as there is less information to generalize. A potential way to deal with this problem is using autoencoders, which have been used before in RL, to reduce the size of the observation space [20].

Our objective is to avoid searching for how to compress a model. We propose using a generalist reinforcement learning agent to learn strategies for multiple models to replicate them in new models. A generalist agent can be trained only once to produce acceptable solutions so that various people can use it. Furthermore, we use an autoencoder to reduce the possible states required to train a generalist reinforcement learning agent. More specifically, the autoencoder encodes the images into a smaller feature vector than the original image size. After getting the encoding, the most representative image per class is extracted from the dataset and used to train the agent. Instead of 60,000 images, only ten (one per class) are needed per dataset, thus dramatically reducing the state space.

The contributions of this article are as follows:

- Using autoencoders dramatically reduces the number of states required to train a model-compression reinforcement-learning agent.
- Our adaptable methodology trains an agent to compress multiple models, empowering it to compress other models without transferring learning, demonstrating its versatility in the field.
- It illustrates the impact of fine-tuning when singular value decomposition is used to compress a convolutional neural network

The remainder of this document is organized as follows. Section II presents the related work for model compression. Section III describes our methodology for solving model compression using SVD. Section IV presents the main results derived from our experiments. Section V discusses our findings and how they fare against the related work. Finally, Section VI summarizes our findings and explores some paths for future work.

II. RELATED WORK

Given the multitude of methods for compressing the models, some researchers have opted to automate selecting when, how, and by how much to compress a layer. Among the available methods, we can highlight two popular strategies for automating model compression.

The first strategy relies on an optimization algorithm, such as a genetic algorithm, to search the compressor's parameter space. Fortunately, a few compressors do not require a fine-tuning process, and this process can be skipped. Two examples of how genetic algorithms can maximize the model's accuracy are the works of Agarwal et al. [21] and Zhang et al. [22]; the last one for a multi-objective optimization scenario. Besides, Particle Swam Optimization (PSO) has also been used for automating model compression [3].

The second popular strategy is to train an agent using RL. This agent looks at the state of the environment (the current state of the CNN) and decides how to compress it. The literature is rich in information the RL agent can use to make decisions. For example, some agents use a vector that contains information regarding the configuration used to build the layer (number of units, input channels, output channels, floating-point operations, among others). Other agents use the input feature map of each layer to decide how to compress it. Finally, a few agents use different information, such as layer weights or embedding representing the neural network.

Most of the model compression agents use layer configurations and compression statistics. AutoML for Model Compression (AMC) uses a Deep Deterministic Policy Gradient agent to compress CNNs [16]. The agent's inputs are the layer parameters and the accumulated compression statistics to select the sparsity ratio for the weights in each layer. AMC used information like the layer index, kernel shape of the current layer, number of floating-point operations (FLOPs) reduced in previous layers, number of remaining FLOPs in the following layers, and the action taken in the last layer. The approach above was combined with a quantization agent that chooses the optimal bit representation for each layer [23]. The Reinforcement Learning Pruning Method (ReLP) shares most of the features of AMC with two changes [24]. First, ReLP does not use the stride of the convolutional layer. Second, two new features are added: the importance of filters and the layer. The first feature deals with the importance of filters, which is found by ranking the filters in the network. The second feature relates to the layer importance, calculated using the importance of each filter in the current layer to be compressed. The purpose of these two features is to let the agent know if a layer has more important filters than other layers to decide the degree of pruning for that layer. The filter importance is based on APoZ [4]. Meanwhile, the layer importance is calculated using the filter importance of all filters located in the layer. Zhang et al. [24] mention that ReLP outperforms other methods in top-1 accuracy "due to passing the filter importance to the DDPG agent." In their work, Yang et al. filtered sensitive layers to decide which layers to prune using the same approach as AMC [25]. Their method deleted 39.9% of the parameters when tested on ResNet-50 on ImageNet. ShrinkML used RL to compress an End-to-End Automatic Speech Recognition model using Singular Value Decomposition [26]. In N2N [15], two policies were trained

using REINFORCE. The first was a Bidirectional LSTM policy that removed or kept layers, whereas the second shrunk layers. A bidirectional policy was used because the hidden states of each layer were passed to the adjacent layers (layer before and layer after) to decide, at the same time, which action to take in each of the layers. The features were the parameters used to create the layer (type, kernel size, stride, padding, number of filters) in addition to using the hidden states for the adjacent layers and the number of layers, before and after the current layer, to reach the start and end of the current block (where the skip connections are). The second policy was not bidirectional, as the hidden states were only passed to the next layer. In addition, the chosen action for the current layer was used to decide how to shrink the following layer.

Only two reinforcement learning agents use feature maps to decide how to act. The AdaDeep framework used a Deep Q-Network (DQN) and a DDPG to select how to compress each layer and the recommended parameter values of the chosen compressor [17]. The DQN was used to select which compressor to apply, whereas the DDPG agent chose the values of the parameters for the selected compressor. Both networks used the input feature maps of the layer to be compressed. Following AdaDeep, Gonzalez-Sahagun et al. proposed using a Region of Interest layer in the DQN to handle feature maps of different heights and widths [27].

Instead of looking at each layer separately, Multi-stage Graph Embedding and Reinforcement Learning (GNN-RL) generated an embedding to represent a neural network's computational graph [28]. After generating the embedding, the agent pruned the neural network, producing a renewed embedding for the new model. This process was performed iteratively until the compressed model met the user's demands. In contrast, DECORE pruned the channels of each layer simultaneously using a REINFORCE agent per channel [29]. Each agent looked at each channel separately and learned whether to prune or keep the channel.

Most of the reinforcement learning agents in the related work were trained to compress a single model. The N2N agent trained to compress ResNet18 was retrained for ten policy update epochs to compress ResnNet34 [15]. The agent was tested after completing the transfer learning. Their results showed that transfer learning helped to accelerate the search for better policies when the knowledge was transferred to the deeper model. In contrast, Yang et al. proved that generalization is possible by training a channel pruning agent for CIFAR-100 and later testing it, without transfer learning, on the same VGG19 architecture trained on ImageNet [18]. González et al. trained two DQNs to compress multiple Convolutional Neural Networks simultaneously [27]. One DQN learned to compress convolutional layers, whereas the other learned to compress dense layers. Their exploratory results show that two LeNet models, trained on fashion MNIST and MNIST, respectively, have the same optimal strategy for compressing them.

The described approaches have their advantages and disadvantages. Reinforcement learning algorithms use a data structure called *experience replay* to store samples generated when compressing each layer. A sample consists of the data related to one step: the current state of the environment s, the action a taken in step s, the next state s', and the reward r. The experience replay contains samples from multiple episodes (compressing a model numerous times using different actions) to generate mini-batches of independent samples to train the agents. Training an RL agent has a high memory consumption because the experience replay holds hundreds of thousands or even millions of samples. The memory consumption is proportional to the number of samples stored in the replay and the complexity of the representation for the state of the environment. Once training is over, the experience replay is no longer needed. In contrast, Stochastic Optimization (SO) algorithms do not require an experience replay. SO algorithms only store a small population of possible solutions where each solution contains a value per decision variable. For example, a decision variable can be to compress or not to compress a channel. A decision variable can also be by how much to compress a layer. Another advantage of SO over reinforcement learning is that no training is involved. SO methods perform simple operations, like crossover and mutation, that can be performed in parallel without a GPU. The only advantage of RL is that knowledge can be transferred to other instances of the same problem or even different domains. A generalist reinforcement learning agent was trained simultaneously in various tasks like playing Atari, chatting, captioning images, and stacking blocks with a robotic arm [19].

III. METHODOLOGY

In this section, we present how we set up the environment of model compression. Moreover, we explain how we use Singular Value Decomposition (SVD) to find a Low-Rank Factorization (LRF) of a layer's parameters. In addition, we describe the methodology for training a generalist reinforcement learning agent that uses singular value decomposition to compress unseen models. Furthermore, we describe the autoencoder architecture.

A. REINFORCEMENT LEARNING ENVIRONMENT

Model compression can be adapted as a Markov Decision Process where a series of decisions are made to compress a model. A RL agent can be used to look at the state s of a feature map, decide on an action a that changes the feature map into a new feature map s', and a reward r is awarded depending on the accuracy after compression and the compression rate.

We eliminate the option of using layer weights to decide how to compress a model for two reasons. The first reason is that it is impractical for large models since dense layers become hundreds of times larger than a feature map. For example, VGG16 uses a weight matrix with shape (25088, 4096) for the first dense layer. A neural network is not able

to receive an input of that size. Although the weight matrix could be partitioned to feed it to the agent, that would work better for pruning as the decision to prune each neuron can be calculated independently. In contrast, feature maps are more straightforward to process since Convolutional Neural Networks (CNN) can handle them without partitioning them. The second reason is that weights are not affected by the actions in previous layers. Independently of what actions were taken to compress previous layers, the weights of the current compression target will be the same. In comparison, the output feature map of a layer will change depending on the action taken to compress that layer. A wrong strategy will affect the output feature maps the most, whereas a good strategy will produce a feature map close to the original.

We discarded using a layer's configuration parameters because the same model trained on different datasets will have the same layer configurations. Imagine two VGG16 models trained on two different datasets. Since they have the same architecture, they will have the same number of units and kernel shapes. Although both models have different weights, there is only one model from the agent's perspective due to using features that do not represent the knowledge of the models. Despite the layers of each model being different, the action taken to compress these models will be the same as the features are the same. The only solution to this problem is adding features like filter importance and layer importance to the state representation. Nonetheless, this feature is slow to calculate as it passes the entire dataset through the network to measure the average percentage of zero activations for each neuron in the neural network. For large datasets and models, it is inconvenient.

We base the reward r of the environment on the objective function of MnasNet [30]. However, we replaced the latency with the number of parameters $\boldsymbol{\theta}$ before and after compression. Since we want to give a higher reward when achieving higher compression, we use the complement of the percentage of parameters after compression. Afterward, the complement is multiplied by the accuracy after compression A_a to keep a balance between compression and accuracy, such as

$$r = A_a \cdot \left(1 - \frac{\boldsymbol{\theta}_a}{\boldsymbol{\theta}_b}\right). \tag{1}$$

This reward is only calculated once the episode ends. Only the second compressed, dense layer has a nonzero reward due to being the last action taken before evaluating the model. As for the convolution, the reward was originally zero. Nonetheless, we assign the same reward as the final reward so that the agent learns faster.

In addition, SVD decomposes a matrix W of shape $m \times n$ into three smaller matrices U, S, and V that, if multiplied, produces the original matrix W as follows

$$W_{m\times n} = U_{m\times m} S_{m\times n} V^{\intercal}_{n\times n} = U_{m\times m} N_{m\times n}. \tag{2}$$

Since matrix S is a diagonal matrix with values from highest to lowest, keeping only the highest c values will approximate matrix W using fewer numbers, such as

$$W_{m\times n} = U_{m\times c} N_{c\times n}. \tag{3}$$

The same approach can be used to calculate an approximation of the output O of a layer using fewer parameters: a batch B is multiplied by the decomposition of the weight matrix W, as follows

$$O_{b\times n} = B_{b\times m} U_{m\times c} N_{c\times n}. \tag{4}$$

Nonetheless, the number of singular values c must be sufficiently small to reduce the number of parameters when replacing W with matrices U and N, i.e.,

$$m \times n > m \times c + c \times n. \tag{5}$$

After grouping by c and rewriting (5), the maximum number of singular values to reduce the number of parameters in a layer is given by:

$$\left\lfloor \frac{m \times n}{m + n} \right\rfloor > c. \tag{6}$$

This work uses a discrete action space, referred to as $\mathcal{A}$, with eleven possible actions. Each discrete action is mapped to an integer value that represents a percentage. When taking an action, the percentage is multiplied by the Maximum number of Singular Values (MSV), c_i, which allows the number of parameters to remain almost equal. This MSV is determined individually per layer via (6). Since multiplying the product of c_i and a_i can result in non-zero decimals, we opted to use the ceiling of the result as we require an integer k_i when specifying the shape of the tensors for layer i, such as

$$k_i = \left\lceil \frac{a_i c_i}{100} \right\rceil. \tag{7}$$

The actions for our experiments are 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100%. The action 100% is interpreted as not compressing the current layer. The actions are the same for convolutional and dense layers because the convolutional layers are replaced by a Multi-Layer Perceptron Convolution (MLPConv) when compressing them. The MLPConv uses dense layers to learn the filters of a convolutional layer [9]. Since each of the units of the output dense layer learns a filter, the kernel of the convolutional layer can be reshaped so that each column has the weights of a filter (Fig. 1). If the kernel has the same shape as the weight matrix of a dense layer, SVD can be easily applied. Thus, we can use the same actions for convolutional and dense layers. In this work, the MLPConv only has one hidden layer and the output layer. We assign the values of U and N to the hidden dense and output layers, respectively.

The first step for predicting the output feature maps of the MLPConv layer is to extract the patches of the feature map that would be used when performing the convolution. Since a dense layer will be predicting a convolution's output feature map, the patches must be flattened before being multiplied by the reshaped kernel (Fig. 2). The predictions are then reshaped to get the output feature map (Fig. 3).

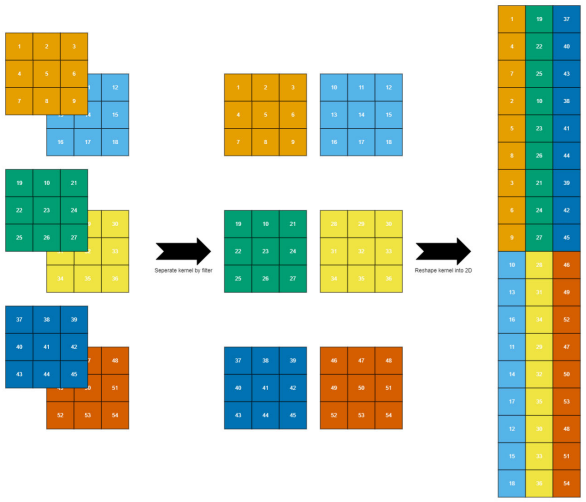


FIGURE 1. The weights of the kernel are reshaped into the shape of a dense layer to replace the convolutional layer with an MLPConv layer. The weights are reshaped into three columns for a kernel with three filters, each associated with a filter output.

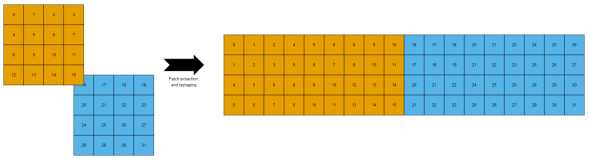


FIGURE 2. The MLPConv layer extracts patches of the feature maps and flattens them before predicting the output. When applying a 3 × 3 filter with one stride to a feature map with shape (4, 4, 2), four patches of size (3, 3, 2) are flattened.

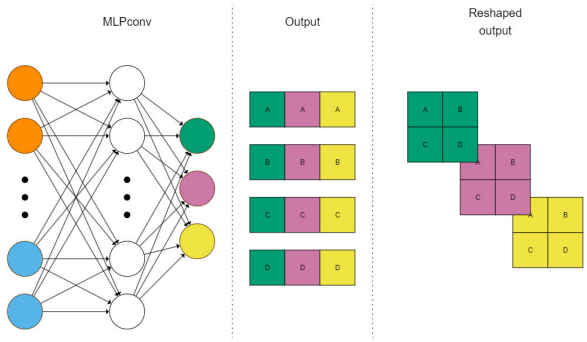


FIGURE 3. Each neuron of the MLPConv predicts a channel's value for each of the extracted patches. The predictions are reshaped to get the output feature map.

The current state s is the input feature map of the target layer of compression l. The input for convolutions is a 4D tensor if we consider the batch dimension. Meanwhile, the input of dense layers is a 2D tensor. Two DQNs estimate the action-value function for the proposed environment. The action-value function Q gives the value of an action a being taken in the state s while following the politic π on the following states until the end of compression. Because the estimation depends on the state, action, and parameters of the DQN $(\boldsymbol{\theta}, \boldsymbol{\alpha}, \boldsymbol{\beta})$, the state value function is given by $Q(s, a; \boldsymbol{\theta}, \boldsymbol{\alpha}, \boldsymbol{\beta})$. The value cannot be calculated when there is a transition between layer types because the update rule of the DQN requires the estimation of the Q-value for the next state s'. Thus, we used $Q(s, a; \boldsymbol{\theta}_c, \boldsymbol{\alpha}_c, \boldsymbol{\beta}_c)$ for estimating the Q-value of the current state and $Q(s', a'; \boldsymbol{\theta}_d, \boldsymbol{\alpha}_d, \boldsymbol{\beta}_d)$ for estimating the value of the following state when there was a transition between a convolutional layer and a dense layer. The c and d stand for convolutional and dense, respectively. The next state s' is the input of the next layer to compress instead of the output of the current layer since there might be noncompressible layers between two compressible, such as pooling layers. If we use the input of the next layer as s' instead of the next compressible layer, s' will never be used to train the agent as there is no action to take there. Thus, $Q(s', a; \boldsymbol{\theta}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ will return inaccurate estimations.

The agent has two Deep Q-Networks. Although the feature maps can be reshaped into the shape of a dense layer's input so that the same DQN can process both types of inputs, we use a different DQN for convolutional and dense layers. The reason for having two DQNs is that there is a spatial relationship between pixels that could not be exploited if the images were flattened. In dense layers, it only matters whether the features are present or not and to what degree. By having a DQN for convolutional layers, we are able to use an agent with convolutional layers to exploit the spatial relationship.

Since feature maps have different shapes at each layer, we padded the feature maps with zeros so that all feature maps have the same shape before being passed to the agent. The shape was obtained using the maximum height, width, and depth between all feature maps. Depending on the layer type that will be compressed, the agent chooses which network to use to select an action. A DQN compresses convolutional layers (Table 2) and another compresses dense layers (Table 1). We opted to predict the Q-value of each feature map independently to use two-dimensional convolutions. Since each feature map can lead to different actions, we use the mode of the actions, which is the action that most frequently has the highest Q-value. For the DQNs, we use a dueling network architecture [31]. The networks can be divided into two parts. The first part generates features, whereas the second part has two branches to calculate the action-value function using the features generated in the first part. The first of the branches calculates the state-value estimation$V(s; \boldsymbol{\theta}, \boldsymbol{\alpha})$. Meanwhile, the second branch calculates the advantage of each action $A(s, a; \boldsymbol{\theta}, \boldsymbol{\beta})$. The parameters $\boldsymbol{\theta}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ are the parameters of the feature extractors, the first branch and the second branch, respectively. The DQNs were trained using Double Q-learning [32]. The Q-values were calculated using the equation proposed in [31]:

$$Q(s, a; \boldsymbol{\theta}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = V(s; \boldsymbol{\theta}, \boldsymbol{\alpha}) + \left[A(s, a; \boldsymbol{\theta}, \boldsymbol{\beta}) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \boldsymbol{\theta}, \boldsymbol{\beta}) \right]. \tag{8}$$

TABLE 1. Dueling DQN for compressing dense layers. The first branch calculates the state-value estimation, and the second calculates the advantage of each action.

Layer (type)	Output Shape	Previous Layer
Input	(None, 400)	-
Dense 1	(None, 1024)	Input
Dense 2	(None, 1024)	Dense 1
Dense 3	(None, 1024)	Dense 2
Dense 4	(None, 1024)	Dense 2
Dense 5	(None, 1024)	Dense 3
Dense 6	(None, 1024)	Dense 4
Dense 7	(None, 1)	Dense 5
Dense 8	(None, 11)	Dense 6
Lambda	(None, 11)	Dense 5 and Dense 6

TABLE 2. Dueling DQN for compressing convolutional layers. Convolutional layers generate the features before being flattened and passed to two branches. The first branch calculates the state-value estimation, and the second the advantage of each action.

Layer (type)	Output Shape	Previous Layer
Input	(None, 14, 14, 16)	-
Conv 1	(None, 12, 12, 256)	Input
Conv 2	(None, 1024)	Conv 1
Conv 3	(None, 1024)	Conv 2
Flatten	(None, 36864)	Conv 3
Dense 1	(None, 1024)	Flatten
Dense 2	(None, 1024)	Flatten
Dense 3	(None, 1)	Dense 1
Dense 4	(None, 11)	Dense 2
Dense 5	(None, 11)	Dense 3
Dense 6	(None, 11)	Dense 4
Lambda	(None, 11)	Dense 5 and Dense 6

B. EXPERIENCE REPLAY

We use a Prioritized Experience Replay (PER) to reduce the agent's training by being more sample-efficient than regular experience replay. PER assigns a sample probability depending on the temporal difference error of the sample. We use the equation proposed in [33] to assign the probability of choosing sample i:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{9}$$

where $p_i = 1/\text{rank}(i)$ assigns a probability depending on the rank of the sample, and α is the importance of the prioritization. We use $\alpha = 1$ for all of our experiments. The rank(i) represents the ranking of each sample after ordering them from the highest temporal difference to the lowest. The temporal difference of a sample is calculated, using a $\gamma = 0.99$, as:

$$\delta = r + \gamma Q_{target}(s', \text{argmax}_{a'} Q(s', a')) - Q(s, a). \tag{10}$$

After generating the first samples using Algorithm 1, Equation 10 dictates which samples have a higher priority for being selected for training. The samples are generated using $\epsilon-$greedy exploration with an ϵ starting at 1.0 and an ϵ decay of 0.9987 for 2000 training epochs. The minimum ϵ is 0.1. Every 50 epochs, the parameters of the DQNs are copied to the target models. When updating the parameters according to [33], we employ the weighted importance sampling w_i to assign importance depending on the temporal difference error δ_i. The weights are calculated as follows

$$w_i = (N \cdot P(i))^{-\beta}, \tag{11}$$

where β has a starting value of 0.5 and steadily increases with a step size of $(1.0 - 0.5)/2000$ for 2000 epochs. The idea behind weighted importance sampling is that the optimizer gives higher importance to reducing the temporal difference error of less frequent states as the probability of sampling those states again is low given the number of samples in the experience replay. In contrast, more frequently sampled states have less importance as they will be sampled more often. The complete algorithm is shown in Algorithm 2.

When inserting a new sample in the experience replay buffer, the replay divides the samples by dataset origin. The samples were stored separately to train the generalist agents with an equal number of samples from each dataset. Because there are two kinds of features (conv and dense), two replay buffers store them separately. The convolution replay buffer has a size of 20000. In contrast, the dense replay buffer has a size of 10000.

Algorithm 1 Generate sample by compressing a model.

Require: Model M, Layers to compress $\mathcal{L}$, Agent($\boldsymbol{\theta}_c, \boldsymbol{\alpha}_c, \boldsymbol{\beta}_c, \boldsymbol{\theta}_d, \boldsymbol{\alpha}_d, \boldsymbol{\beta}_d$), Selected images $\mathcal{D}$, Convolution replay R_c, Dense replay R_d, Epsilon-greedy value ϵ

```
T ← ∅
for l in L do
    s ← Feature_Maps(l, D)
    if type(l) is conv then
        a_t ← argmax_a Q(s, a; θ_c, α_c, β_c, ε)
    end if
    if type(l) is dense then
        a_t ← argmax_a Q(s, a; θ_d, α_d, β_d, ε)
    end if
    s', r_t, done ← Compress(M, l, a_t, D)
    T ← T ∪ (s, a_t, r_t, s')
    if done then
        Stop compression
    end if
end for
Store (s, a_t, r_t, s') for last convolution.
Store T in replay buffers R_c and R_d
```

C. AUTOENCODER FOR STATE SELECTION

A subset of images $\mathcal{D}$ is selected from each dataset to train the agents. The datasets are Fashion MNIST [34], Kuzushiji-MNIST [35], and MNIST [36]. We select the most representative images of each class to reduce the number of feature maps used to train the agents. An autoencoder is trained for each training dataset to generate an image embedding for selecting the images. As all datasets are of low resolution, we use a simple autoencoder with dense layers (Table 3). The autoencoder generates an embedding of size

Algorithm 2 Reinforcement Learning strategy for selecting the compressors for N layers

Require: Set of datasets $\mathcal{D}$, Batch size n_b, Num. of weight updates n_u, Num. of tests T, Layers to compress $\mathcal{L}$, Epochs n_e, LeNet Model M, Priorization importance α, Weighted importance sampling β.

for d **in** D **do**
  $m_d \leftarrow$ Train(M) with d
end for
$r_{\text{highest}} \leftarrow 0$
$R_c \leftarrow$ Convolution replay
$R_d \leftarrow$ Dense replay
$R \leftarrow (R_c, R_d)$
$\boldsymbol{\theta}_c, \boldsymbol{\alpha}_c, \boldsymbol{\beta}_c \leftarrow$ new DDQN
$\boldsymbol{\theta}_d, \boldsymbol{\alpha}_d, \boldsymbol{\beta}_d \leftarrow$ new DDQN
$A \leftarrow (\boldsymbol{\theta}_c, \boldsymbol{\alpha}_c, \boldsymbol{\beta}_c, \boldsymbol{\theta}_d, \boldsymbol{\alpha}_d, \boldsymbol{\beta}_d)$
$\epsilon \leftarrow 1.0$
for $i = 1, \ldots, n_e$ **do**
  for d **in** D **do**
    Generate_Sample(m_d, $\mathcal{L}$, A, R, d_{train}, ϵ)
  end for
  for $i = 1, \ldots, n_u$ **do**
    $b_c, b_d \leftarrow$ sample(R_c, n_b, α), sample(R_d, n_b, α)
    $\boldsymbol{\theta}_c, \boldsymbol{\alpha}_c, \boldsymbol{\beta}_c \leftarrow$ train($\boldsymbol{\theta}_c, \boldsymbol{\alpha}_c, \boldsymbol{\beta}_c, b_c, \beta$)
    $\boldsymbol{\theta}_d, \boldsymbol{\alpha}_d, \boldsymbol{\beta}_d \leftarrow$ train($\boldsymbol{\theta}_d, \boldsymbol{\alpha}_d, \boldsymbol{\beta}_d, b_d, \beta$)
  end for
  $\epsilon \leftarrow \max\{0.1, 0.9987 \times \epsilon\}$
  if i mod 50 **equals** 0 **then**
    Copy weights from agent to target agent
    $r \leftarrow 0$
    for d **in** D **do**
      $r \leftarrow r +$ test(d_{val})
    end for
    $r \leftarrow r/\text{len}(\mathcal{D})$
    if $r >= r_{\text{highest}}$ **then**
      $r_{\text{highest}} \leftarrow r$
      Save weights of the DQNs
    end if
  end if
end for

64. The embedding is used to calculate the centroid per class using K-means. After obtaining the centroids, the image closest to each centroid is stored in $\mathcal{D}$ to generate the feature maps used to train the agents. The autoencoder is then reused to select the most representative validation and test sets of images.

TABLE 3. Architecture of the autoencoder used to generate the image embeddings. The activation function of the first dense layer was the rectifier linear unit, whereas the second dense layer used a sigmoid. The autoencoder generates an embedding of length 64.

Layer (type)	Output Shape	Previous Layer
Input	(None, 28, 28, 1)	-
Flatten	(None, 784)	Input
Dense 1 (Embedding)	(None, 64)	Flatten
Dense 2	(None, 784)	Dense 1
Reshape	(None, 28,28, 1)	Dense 2

D. REINFORCEMENT LEARNING EXPERIMENTS

Our objective is to reduce the time needed to compress a model. This time can be reduced in two ways. The first is to avoid time-consuming processes, such as training an agent or using a stochastic search algorithm to find an acceptable compression scheme. The second is to prevent fine-tuning as much as possible.

Each dataset is divided into training, validation, and test sets. We use the training set to generate samples and fill the experience replay. The validation set is used for saving the model with the highest mean reward on all validation datasets. Thus, it is used for model selection. Finally, the test set is used to evaluate how good the agents are.

We performed a set of experiments to check the viability of time-saving. First, the agents were trained using one or more datasets. Training with only one dataset was meant to be used as a reference point to measure how good generalist agents are compared to reusing agents trained to compress a single model. If an agent trained on a single dataset performed equally or better than a generalist agent, then there is no reason to train a generalist agent. Second, metrics were stored before and after each fine-tuning step to gauge the impact of fine-tuning in improving accuracy.

The metrics we used were the accuracy loss (Δ Acc) and the remaining parameters/weights percentage. The accuracy loss was calculated as the difference between the accuracy before and after compression. The percentage of remaining parameters was calculated as the number of parameters after compression divided by the number of parameters before compression. Finally, the result was multiplied by one hundred to make it a percentage.

We tested our methodology on LeNet-5, which has two convolutional layers and two dense hidden layers. The number of units per layer was 6, 16, 120, and 84, respectively. The agents compressed three LeNet layers: the last convolutional layer and the two hidden dense layers. The MSV for these three layers were 14, 92, and 49, respectively. We trained the agents using the selected images from the training set. The selected images from the validation set were used for model selection. The chosen images from the test set were used to decide how to compress the models and evaluate the agents' performance. In addition, we evaluated the performance using the whole test set to verify if the results would be the same as using the most representative images.

The experiments were run on a server with 128GB of RAM and a V100 GPU with 32 GB. The time it took to train the agents for compressing the models for FMNIST, KMNIST, and MNIST was 4.5 days, 4.5 days, and 4.6 days, respectively. The training time for the agents that were trained to compress two datasets was 9.5 days (FMNIST and KMNIST), 9.5 days (FMNIST and MNIST), and 9 days (KMNIST and MNIST).

E. VGG16 AND IMAGENET EXPERIMENTS

The complexity of training a generalist agent for a larger and more complex dataset, such as ImageNet, dramatically

increases compared to grayscale datasets due to the images' size and quantity. There are three inconveniences for more complex datasets:

- Memory consumption dramatically increases. It increases due to loading batches of the dataset into RAM and storing the samples in the experience replay.
- Fine-tuning can take hours or even days. Thus, avoiding fine-tuning is highly recommended.
- More layers are to be compressed, resulting in a bigger search space.

Given the inconveniences we just pointed out, we focused our experiment on ImageNet to verify the existence of compression solutions that do not require fine-tuning to prevent high accuracy loss.

We used a Genetic Algorithm (GA) to perform a stochastic search. We decided to handle model compression as a multiobjective optimization problem with two conflicting objectives. The first objective function is the accuracy, whereas the second is the number of parameters. The former is maximized, while the latter is minimized.

We initialized the GA with a population P_t of 50 randomly created solutions. We use the subscript t to indicate that the population will change during each generation. A solution is represented as a list of integer values. The list has a length of l, where l is the number of layers to be compressed. Each value in the list corresponds to the percentage of MSV to use in each layer. The minimum is 1, whereas the maximum is 101. The value 101 is interpreted as skipping compression of that layer.

In each generation, we generate $\lambda = 26$ descendants using crossover or mutation and store them in Q_t (Algorithm 3). After generating λ children, the children are evaluated using Algorithm 4. Finally, P_{t+1} is filled with 26 solutions selected from P_t and Q_t. The process is repeated 100 generations.

- **Selection**: We use the same approach as the NSGA-II to find a spread of solutions in the Pareto Front that can meet user demands [44].
- **Crossover**: We perform a two-point crossover using two random individuals of the population P to disrupt solutions as much as possible for better exploration. Only the first offspring is stored in Q.
- **Mutation**: We mutate the chromosome by adding a value v to each position. The value v is generated from a uniform distribution in the interval of $[-10, 10]$. After adding the values to the chromosome, each position in the chromosome is clipped to keep it in the interval $[1, 101]$.

We store in F the non-dominated solutions in the Pareto Front (PF) due to the trade-off between accuracy and compression. Non-dominated solutions are those that are not worse than other solutions in all objective functions. The PF contains the best solutions that can be of interest to meet user demands. During each generation, solutions in F and Q_t are compared to determine if one or more solutions in F should be replaced by a solution in Q_t.

Algorithm 3 Genetic Algorithm

```
P_t ← random population of 50 solutions
F ← Pareto optimal solutions from P_t
for t ← 0 to N do
    Q_t ← ∅
    for i ← 0 to λ do
        r ← random(0,1)
        if r < 0.5 then
            Q_t ← Q_t ∪ crossover(P_t)
        else
            Q_t ← Q_t ∪ mutation(P_t)
        end if
    end for
    Evaluate fitness Q_t
    Update Pareto front F using Q_t
    P_t ← NSGA2(P_t, Q_t, μ)
end for
```

Algorithm 4 Fitness Function

```
Require: solution s, list of layers to compress l, model θ,
dataset D
θ_c ← Copy θ
for layer i in l do
    w, b ← θ^i        ▷ Extract weights and bias from layer i
    if i is Conv2D then
        w ← Reshape(w)
        U, N ← SVD(w, s_i)          ▷ s_i singular values
        θ_c^i ← MLPConv(U, N, b)
    else
        U, N ← SVD(w, s_i)          ▷ s_i singular values
        t_u, t_n ← Dense(U), Dense(N, b)
        θ_c^i ← Sequential(t_u, t_N)
    end if
end for
a ← Accuracy(θ_c, D)
p ← count parameters in θ_c
Return a, p
```

IV. RESULTS

In this section, we present the results of our two experiments. First, we present the results for the generalist agent using reinforcement learning. Then, we show the results of the stochastic search using Genetic Algorithms to find the parameters to compress VGG16 using Singular Value Decomposition (SVD).

A. REINFORCEMENT LEARNING

The results are divided into two parts. The first part shows the fine-tuning results to check if it is indispensable or can be skipped when using Singular Value Decomposition to compress models. The second part presents the performance of generalist agents when compressing unseen models. Furthermore, the effect of using a subset of images to select actions versus the whole test set is also presented.

The agents are identified by the first letters of the datasets on which they were trained. The letters were concatenated alphabetically (F, K, and M). An agent trained using the feature maps of FMNIST is referred to as agent F. Agent FK is the agent trained using the feature maps of FMNIST and KMNIST. Agent KM is the agent trained using the feature maps of KMNIST and MNIST.

Fine-tuning is a process that can enhance the accuracy. Nonetheless, it can be too time-consuming for some datasets. When fine-tuning is avoided, the results are deterministic, as we used Singular Value Decomposition (SVD) to compress the models. Since results are non-deterministic when fine-tuning is involved, the mean accuracy loss (MeanΔ Acc) is only reported for the experiments that apply fine-tuning as the same sequence of actions might lead to slightly different results.

The compressed LeNet model trained on MNIST reached a peak reward of 0.7608 when avoiding fine-tuning; this was the highest reward, considering all the datasets. The reward was reached by compressing the model to 12.1965% of its original number of parameters and losing 11.02% of accuracy (Table 4). The best solution accuracy-wise lost 6.09% of accuracy and kept 16.9481% of the parameters. The accuracy dropped drastically for FMNIST and KMNIST when compressing without fine-tuning. The smallest accuracy loss for FMNIST was 34.77%, whereas it was 45.06% for KMNIST.

TABLE 4. Top 10 highest rewards found during exploration for each dataset. Only results without fine-tuning are shown. The highlighted rows show the results with the highest accuracy. The accuracy, number of parameters, and reward were deterministic due to compressing with Singular Value Decomposition.

Dataset	Actions (%)	Δ Acc	Parameters (%)	Reward
FMNIST	20,5,30	-0.4512	11.9664	0.3740
FMNIST	20,5,40	-0.4442	13.6194	0.3730
FMNIST	20,5,20	-0.4616	10.3134	0.3717
FMNIST	20,5,50	-0.4520	15.2724	0.3592
FMNIST	20,5,60	-0.4559	16.9254	0.3490
FMNIST	20,10,100	-0.3980	27.5565	0.3463
FMNIST	20,5,70	-0.4546	18.5784	0.3431
FMNIST	10,5,20	-0.4950	10.0444	0.3427
FMNIST	20,20,100	-0.3477	35.1408	0.3427
FMNIST	10,5,30	-0.4889	11.6974	0.3418
KMNIST	40,20,20	-0.4556	22.9184	0.3664
KMNIST	40,20,30	-0.4513	24.5714	0.3618
KMNIST	40,20,40	-0.4506	26.2244	0.3544
KMNIST	40,10,20	-0.5154	15.3340	0.3519
KMNIST	50,20,20	-0.4800	23.1874	0.3464
KMNIST	100,20,20	-0.4695	25.1937	0.3452
KMNIST	90,20,20	-0.4729	24.8015	0.3445
KMNIST	40,20,50	-0.4536	27.8774	0.3443
KMNIST	50,20,30	-0.4759	24.8404	0.3421
KMNIST	100,20,30	-0.4674	26.8467	0.3391
MNIST	70,5,20	-0.1102	12.1965	0.7608
MNIST	80,10,20	-0.0609	16.9481	0.7606
MNIST	60,5,20	-0.1134	11.9275	0.7603
MNIST	50,5,20	-0.1207	11.3895	0.7585
MNIST	80,5,20	-0.1076	12.7346	0.7584
MNIST	90,10,20	-0.0618	17.2171	0.7574
MNIST	70,10,20	-0.0709	16.4101	0.7572
MNIST	60,10,20	-0.0755	16.1411	0.7557
MNIST	50,10,20	-0.0815	15.6030	0.7555
MNIST	90,5,20	-0.1089	13.0036	0.7550

Fine-tuning increased the accuracy considerably after compressing with SVD. The accuracy loss for FMNIST and KMNIST was reduced to less than 10% (Table 5). For FMNIST, all of the top 10 highest combinations by reward had an accuracy loss of less than 10%. The model for KMNIST was more challenging to compress as four of the top 10 had an accuracy loss greater than 10%. Finally, all the solutions for MNIST in the top 10 had less than 7% accuracy loss.

MNIST had the lowest mean accuracy loss when fine-tuning. The accuracy decreased by only 1.54% while using only 11.7428% of the parameters. In contrast, the lowest accuracy loss was 5.37% and 8.72% for FMNIST and KMNIST, respectively.

TABLE 5. Top 10 highest rewards found during exploration for each dataset. Accuracy was evaluated after fine-tuning. The highlighted rows show the results with the highest accuracy among the solutions with the highest reward.

Dataset	Actions (%)	Mean Δ Acc	Parameters (%)	Mean Reward
FMNIST	20,5,10	-0.0644	8.6604	0.7413
FMNIST	30,5,10	-0.0597	9.1985	0.7412
FMNIST	40,5,10	-0.0576	9.4675	0.7409
FMNIST	50,5,10	-0.0568	9.7365	0.7395
FMNIST	20,5,5	-0.0752	7.9992	0.7367
FMNIST	60,5,10	-0.0566	10.2745	0.7352
FMNIST	70,5,10	-0.0551	10.5435	0.7343
FMNIST	5,5,10	-0.0799	8.1224	0.7314
FMNIST	10,5,10	-0.0776	8.3914	0.7314
FMNIST	80,5,10	-0.0537	11.0816	0.7312
KMNIST	100,10,20	-0.0891	17.6093	0.6936
KMNIST	40,10,30	-0.0965	16.9870	0.6928
KMNIST	80,10,30	-0.0872	18.6011	0.6868
KMNIST	70,10,30	-0.0945	18.0631	0.6854
KMNIST	30,10,30	-0.1119	16.7180	0.6822
KMNIST	100,5,20	-0.1445	13.3958	0.6811
KMNIST	50,10,30	-0.1083	17.2560	0.6808
KMNIST	100,10,30	-0.0892	19.2623	0.6797
KMNIST	40,10,40	-0.0997	18.6400	0.6763
KMNIST	90,10,30	-0.1013	18.8701	0.6731
MNIST	20,5,10	-0.0435	8.6604	0.8524
MNIST	40,5,10	-0.0375	9.4675	0.8503
MNIST	30,5,10	-0.0405	9.1985	0.8500
MNIST	100,5,10	-0.0154	11.7428	0.8484
MNIST	10,5,10	-0.0507	8.3914	0.8483
MNIST	50,5,10	-0.0371	9.7365	0.8481
MNIST	60,5,10	-0.0356	10.2745	0.8444
MNIST	70,5,10	-0.0353	10.5435	0.8421
MNIST	20,5,5	-0.0626	7.9992	0.8410
MNIST	40,5,5	-0.0568	8.8063	0.8389

The Pareto front shows the effect of fine-tuning on accuracy loss. The Pareto front is a set that contains all solutions that are not better in every dimension than any other solution. In our case, the dimensions are the accuracy loss and the percentage of parameters remaining after compression. Multiple solutions do not require fine-tuning while decreasing accuracy by less than 10%. However, those solutions use more than 80% of the original weights before compression (Fig. 4). In contrast, solutions that involve fine-tuning can compress the models to less than 20% of their original size while losing less than 10% of accuracy.

The top 10 solutions per dataset reach an accuracy loss of less than 10% within less than 80 fine-tuning epochs. For example, the LeNet model with 5344 parameters trained on FMNIST requires almost 40 epochs for a 10% accuracy loss (Fig. 5). In contrast, the same model with 5344 parameters requires less than 20 epochs to reach a higher accuracy on

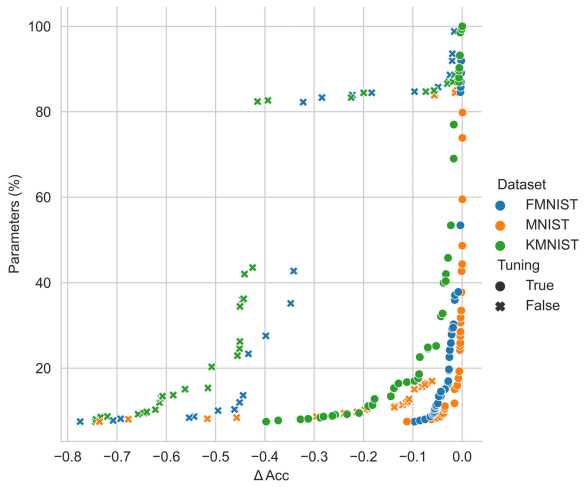


FIGURE 4. Pareto front per dataset for non-dominated solutions of model compression. The front is shown for solutions without fine-tuning and solutions with fine-tuning. The elements of both fronts are not the same since fine-tuning allows more aggressive compression schemes.

MNIST. Finally, KMNIST requires almost double the number of parameters to reduce the accuracy loss to less than 10%.

The combination of actions with the highest reward was the same for FMNIST and MNIST. The sequence 20, 5, and 10 had the highest reward for both datasets. When fine-tuning, all top 10 solutions for FMNIST and MNIST used 5% of the MSV for the first dense layer. For KMNIST, only one sequence used 5% of the MSV. As for the second dense layer, most sequences for FMNIST and MNIST used 10% of the MSV. There was only one exception for FMNIST that did not use 10% of the MSV. Meanwhile, there were two exceptions for MNIST. As for KMNIST, a wide variety of actions that ranged from 20 to 40% were used to compress the second dense layer.

The compression of the convolutional layer has a high impact on accuracy. Table 5 shows a correlation between the percentage of MSV and the accuracy. For FMNIST and KMNIST, slightly compressing the convolutional layer achieved the highest accuracy among solutions in the top 10. In the case of MNIST, not compressing the convolutional layer was the best choice for maximizing the accuracy.

Most of the time, the agents chose the same actions, whether or not the whole test set was used for selecting actions. There were only three exceptions, all for KMNIST (Table 6). The differences were always only one action in the sequence. In two cases, the difference was in the second dense layer, whereas the last case was in the convolutional layer. The differences in the dense layers were one action away in the discrete action space. For agent F, the actions used were 5% and 10%. The same happened for agent KM, with actions being 20% and 10%. Agent K was the only agent that chose different actions for the convolutional layer. The actions were not subsequent in the action space as there was one action between them (action 50%).

Only two agents chose a solution within the top 10 for KMNIST. The first agent was Agent K, whereas the second was Agent KM. Agent K selected the second top suggestion, while Agent KM selected the sixth. These two agents produced solutions in the top 10 only using the subset of images instead of the entire dataset.

Agent FM was the only one who learned the best policy for the datasets it was trained on. We use the term best instead of optimal as it is unknown if it is the optimal policy because not all solutions were explored. Furthermore, the action space was discretized with actions that may not encompass the optimal number of singular values. The best policy for both models (FMNIST and MNIST) was using actions 20, 5, and 10 (Fig. 6). The other agents were not far off from using the best policy since they had part of the sequence right. For example, Agent KM suggested the sequence (100,5,20) for the subset of KMNIST. It was only one action away from the best policy (100, 10,20). Agents F and FK chose (100,5,10) and (60,5,10) for FMNIST, which missed the recommended number of singular values to use in the convolutional layer. Despite not choosing the best policy, the distances in mean reward were only 0.0116 and 0.0061, respectively.

Even the agents trained on a single dataset achieved decent results when tested on models they were not trained to compress. Agent K compressed the models for FMNIST and MNIST to less than 15% while keeping the accuracy loss to less than 5%. Both agents F and M performed poorly when tested on KMNIST as they compressed the second dense layer too much. Top solutions for KMNIST used between 20 and 40% of the MSV. In contrast, agents F and M used less than or equal to 10% of the MSV.

The agents generated compression solutions mainly in the Pareto front of each dataset. Although there are some exceptions, there is little distance between these solutions and the Pareto front. The suggested solutions for FMNIST were all in the Pareto front except for the solutions of agent K and KM (Fig. 7). It is barely visible that the solutions of agent FM and the best-known policy are overlapping. Only two agents generated solutions in the Pareto front for KMNIST (Fig. 8). One was agent K, whereas the other was agent KM. Finally, only agent FK did not propose a solution in the Pareto front for MNIST (Fig. 9).

B. GENETIC ALGORITHM FOR COMPRESSING VGG16

In this subsection, we present our findings in attempting to verify if it is possible to skip fine-tuning when compressing a large model. We used a Genetic Algorithm (GA) to compress the VGG16 model using a chromosome of 13 integer values. Each value corresponds to a percentage of the number of singular values to use for compressing a particular layer.

We obtained 67 solutions in the Pareto front. Nonetheless, most were located in the lower spectrum of the accuracy domain (Fig. 10). The point farthest to the right represents the uncompressed model. There are only 3 solutions that are close to the same level of accuracy.

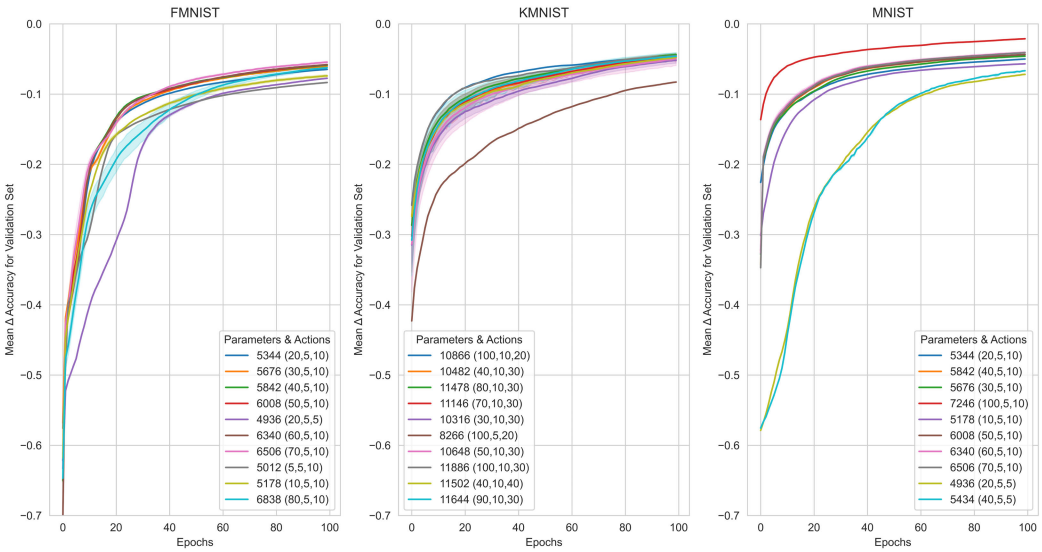


FIGURE 5. Mean accuracy loss for the validation set when fine-tuning the top 10 solutions per dataset. Labels are in descending order by reward.

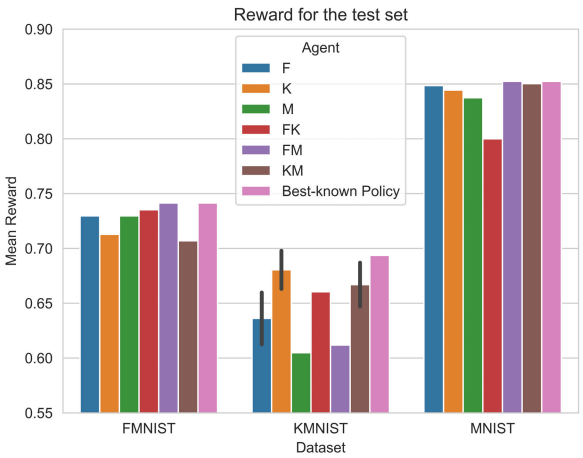


FIGURE 6. Mean reward of the agents for different datasets. The mean considers the subset of the test set and the full set. The error bar shows the standard deviation of the mean reward due to the agent choosing different actions for the full test set and a subset of it.

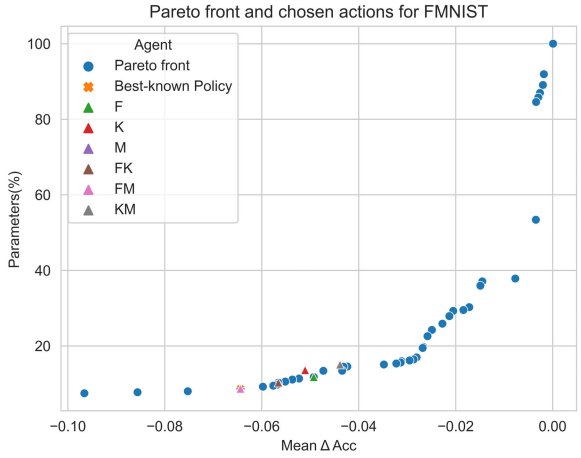


FIGURE 7. Pareto front of compression solutions found during exploration for FMNIST. In addition, the solutions proposed by agents are also included.

The solutions with the highest accuracy in the Pareto front barely compressed the convolutional layers. In contrast, the most compressed layer was the first hidden dense layer. The second row in Table 7 only shows a solution that involves only compressing the first hidden dense layer. Meanwhile, the third solution aggressively compressed both dense layers, whereas barely compressing two convolutional layers. The aforementioned solution produced an accuracy loss of 0.58% and kept less than half of the original amount of parameters. The fourth solution is apparently the best solution due to using 26.87% of the parameters and losing only 1.14% of accuracy. As for the fifth and sixth, both compressed the first dense layer using 2% of the MSV and avoided compressing the second dense layer. Although both used close to 25% of the parameters, the fifth obtained a higher accuracy than the sixth due to not compressing significantly the convolutional layers. In comparison, the sixth solution used only 8% to compress one of the convolutional layers, which resulted in a drop of 9.36% accuracy with respect to the previous solution.

V. DISCUSSION

This research aims to reduce the time it takes to compress a model by reusing a reinforcement learning agent instead of searching from scratch for the optimal parameters to compress that model. In addition, we analyzed the impact of fine-tuning after compression to verify if it can be avoided.

Fine-tuning can be avoided depending on the desired amount of compression when using an action space of eleven

TABLE 6. Result of each agent for each dataset when using a subset of images or the whole test set. The reward column shows the reward of a single test. Meanwhile, the mean reward considers all the samples generated during exploration independently of the agent trained. Highlighted rows show the agents that suggested different sequences of actions for the same dataset when using the whole test set and a subset of the test set. *Delta* Acc refers to the difference in accuracy for one particular test. The same applies to the reward.

Dataset	Test	Agent	Actions	Δ Acc	Parameters (%)	Reward	Mean Reward	Mean Δ Acc
FMNIST	all	F	100,5,10	-0.0490	11.7428	0.7299	0.7297	-0.0493
FMNIST	best	F	100,5,10	-0.0492	11.7428	0.7297	0.7297	-0.0493
FMNIST	all	FK	60,5,10	-0.0574	10.2745	0.7345	0.7352	-0.0566
FMNIST	best	FK	60,5,10	-0.0563	10.2745	0.7355	0.7352	-0.0566
FMNIST	all	FM	20,5,10	-0.0645	8.6604	0.7412	0.7413	-0.0644
FMNIST	best	FM	20,5,10	-0.0648	8.6604	0.7409	0.7413	-0.0644
FMNIST	all	K	60,5,30	-0.0408	13.5805	0.7218	0.7129	-0.0510
FMNIST	best	K	60,5,30	-0.0416	13.5805	0.7211	0.7129	-0.0510
FMNIST	all	KM	100,5,30	-0.0431	15.0488	0.7076	0.7069	-0.0438
FMNIST	best	KM	100,5,30	-0.0438	15.0488	0.7070	0.7069	-0.0438
FMNIST	all	M	100,5,10	-0.0482	11.7428	0.7306	0.7297	-0.0493
FMNIST	best	M	100,5,10	-0.0480	11.7428	0.7308	0.7297	-0.0493
KMNIST	all	F	100,5,10	-0.1936	11.7428	0.6508	0.6529	-0.1913
KMNIST	best	F	100,5,5	-0.2345	11.0816	0.6193	0.6194	-0.2344
KMNIST	all	FK	60,10,20	-0.1440	16.1411	0.6600	0.6604	-0.1435
KMNIST	best	FK	60,10,20	-0.1430	16.1411	0.6608	0.6604	-0.1435
KMNIST	all	FM	20,5,20	-0.2604	10.3134	0.6014	0.6118	-0.2488
KMNIST	best	FM	20,5,20	-0.2607	10.3134	0.6012	0.6118	-0.2488
KMNIST	all	K	60,10,30	-0.0898	17.7941	0.6915	0.6682	-0.1182
KMNIST	best	K	40,10,30	-0.0969	16.9870	0.6924	0.6928	-0.0965
KMNIST	all	KM	100,5,10	-0.1914	11.7428	0.6528	0.6529	-0.1913
KMNIST	best	KM	100,5,20	-0.1443	13.3958	0.6813	0.6811	-0.1445
KMNIST	all	M	100,10,5	-0.2160	15.2951	0.6056	0.6048	-0.2170
KMNIST	best	M	100,10,5	-0.2160	15.2951	0.6056	0.6048	-0.2170
MNIST	all	F	100,5,10	-0.0155	11.7428	0.8483	0.8484	-0.0154
MNIST	best	F	100,5,10	-0.0155	11.7428	0.8483	0.8484	-0.0154
MNIST	all	FK	60,5,40	-0.0336	15.2335	0.7994	0.7999	-0.0330
MNIST	best	FK	60,5,40	-0.0342	15.2335	0.7989	0.7999	-0.0330
MNIST	all	FM	20,5,10	-0.0428	8.6604	0.8530	0.8524	-0.0435
MNIST	best	FM	20,5,10	-0.0443	8.6604	0.8517	0.8524	-0.0435
MNIST	all	K	60,5,10	-0.0175	10.2745	0.8606	0.8444	-0.0356
MNIST	best	K	60,5,10	-0.0173	10.2745	0.8608	0.8444	-0.0356
MNIST	all	KM	40,5,10	-0.0365	9.4675	0.8512	0.8503	-0.0375
MNIST	best	KM	40,5,10	-0.0369	9.4675	0.8508	0.8503	-0.0375
MNIST	all	M	100,5,5	-0.0346	11.0816	0.8377	0.8374	-0.0349
MNIST	best	M	100,5,5	-0.0350	11.0816	0.8373	0.8374	-0.0349

TABLE 7. Solutions in the Pareto front for VGG16 trained on ImageNet. Only solutions with an accuracy loss of less than 20% are presented. Only 13 layers were compressed. The column B2C1 represents the layer *block2_conv1*. The same nomenclature applies to the other convolutional layers. The value for each layer corresponds to the percentage of the maximum number of singular values (MSV) used to compress that layer. The percentage 101 corresponds to not compressing the layer. The first row corresponds to not compressing the model.

B2C1	B2C2	B3C1	B3C2	B3C3	B4C1	B4C2	B4C3	B5C1	B5C2	B5C3	FC1	FC2	Δ Acc	Parameters (%)
101	101	101	101	101	101	101	101	101	101	101	101	101	0.0000	100.0000
101	101	101	101	101	101	101	101	101	101	101	81	101	-0.0001	85.9072
101	101	101	101	101	101	101	86	101	93	101	38	8	-0.0058	42.4349
101	101	101	101	101	101	101	101	77	101	47	16	22	-0.0114	26.8755
101	101	101	62	101	101	101	101	101	56	44	2	101	-0.0578	25.3598
101	101	101	101	80	101	101	101	71	8	101	2	101	-0.1514	25.0767

actions. The Pareto front before fine-tuning showed that multiple solutions for each dataset have minimal accuracy loss. For FMNIST and KMNIST, LeNet can be reduced by less than 20% at most. In comparison, MNIST can be reduced by more than 80% without fine-tuning. Knowing that fine-tuning can be avoided in some cases, it can be a

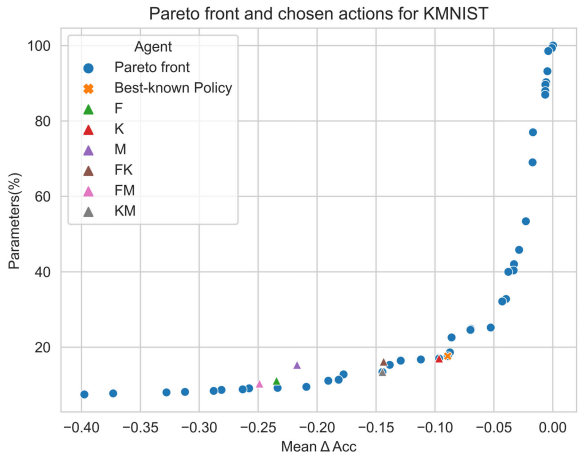


FIGURE 8. Pareto front of compression solutions found during exploration for KMNIST. In addition, the solutions proposed by agents are also included.

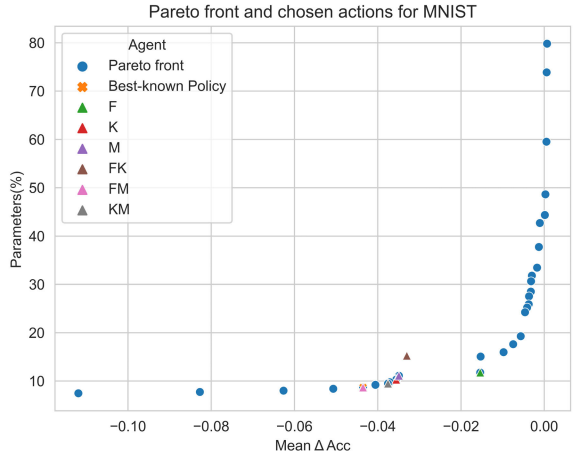


FIGURE 9. Pareto front of compression solutions found during exploration for MNIST. In addition, the solutions proposed by agents are also included.

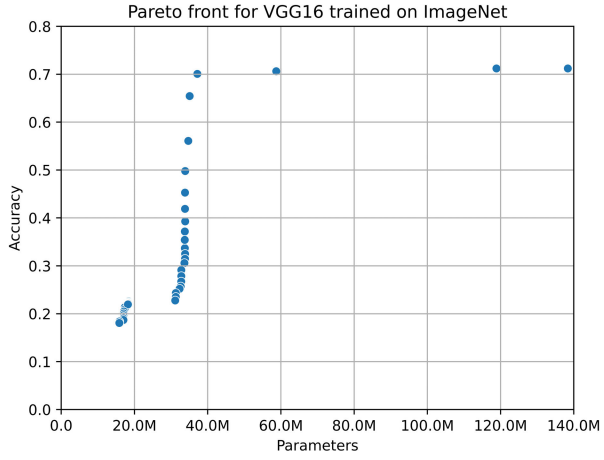


FIGURE 10. Pareto front found by the genetic algorithm for VGG16. First point on the right represents the uncompressed model.

considerable advantage to avoid fine-tuning larger datasets that require hundreds of hours.

VGG16 can be compressed without fine-tuning. We found four solutions in the Pareto front that compress the VGG16 model between 85.90% and 25.35%. Furthermore, the accuracy loss for those solutions is within an acceptable range as the highest accuracy loss is only 5.78%. The above proves that it is possible to compress larger models using SVD without fine-tuning if there is an adequate resolution space.

Increasing the resolution of the action space can lead to better results. Currently, the agents choose between eleven values that might not be close to the optimal values required by the models. The maximum number of singular values (MSV) for the convolutional and two hidden dense layers we compressed was 14, 92, and 49, respectively. The action of avoiding compressing a layer adds one to each of the numbers, which results in 68,355 possible solutions $((14+1)\times(92+1)\times(49+1))$. Our action space only considers 1,331 solutions (11^3 since we have eleven actions for three layers). Increasing the action space has the disadvantage that more exploration is needed. Following the same logic, there are 8.5929147×10^{33} solutions for 13 layers of VGG16 (105×116 x 210×231 x 231×419 x $461 \times 461 \times 461 \times 461 \times 461 \times 3522$ x 2049).

There are multiple ways to increase the resolution of the action space. The first option is to use more percentages for the actions, which can be achieved by reducing the step size of the discrete action space. Our experiment on VGG16 represents this option. An alternative uses a continuous action space with a lower bound close to zero and an upper limit of one. This option can enhance the results because values between two neighboring actions might still increase the compression rate without a significant accuracy loss. The second option is to use the number of singular values rather than the percentages. However, this would require more exploration as the number of solutions would go from 1,331 to 68,355. In addition to the larger search space, another disadvantage is that there would be actions that cannot be taken in layers with few neurons. If a layer can use over one thousand singular values at most and another a few hundred, many actions that could be taken in the first layer would not be usable for the second layer. Thus, a mask would be needed so that those actions are ignored. Given these inconveniences, using percentages is a better alternative for larger models.

A subset of images is enough to train the agents instead of using the whole dataset. Since the subsets were put together using the most representative image of each class, the suggested actions did not change in most cases when using a subset rather than the complete test set. The agents obtained a higher reward when using subsets rather than the entire dataset. There was only one exception where using the entire dataset achieved a higher reward. Another autoencoder architecture is needed to choose the subset for more complex datasets. MnasNet, an optimized architecture found in [30] using reinforcement learning, can encode the datasets' images. MnasNet was used to generate an embedding per image [37]. The embeddings

were then aggregated, using K-means, to create a dataset embedding that was used to train a model for selecting which network architecture was the most appropriate for each dataset.

There is no need to train an agent to compress each model since reusing a model can achieve results similar to those of the agents trained in that model. Our generalist agent trained on FMNIST and KMNIST beat other strategies for compressing models trained on MNIST (Table 8). The only exception was the FK agent based on AdaDeep [27], which had a higher compression rate and less accuracy loss. Nonetheless, the FK agent based on AdaDeep had fixed parameters for the compressors and could suggest the optimal policy for its action space. In contrast, our FK agent based on SVD did not apply the optimal strategy, meaning there is room to grow if the agent can learn the optimal policy for FMNIST since FMNIST has the same optimal policy as MNIST. Although the best-known policy has 1.94% more accuracy loss than the FK agent based on AdaDeep, it has 63% of the parameters of their compressed architecture. This shows that the time it takes to compress a model can be drastically reduced by reusing a generalist agent instead of training an agent from scratch without transfer learning. Furthermore, other researchers have opted for a similar approach. Yang et al. [18] successfully reused an agent trained using CIFAR-10 to avoid training an agent from zero.

TABLE 8. Results for MNIST found in the related work.

Strategy	Model	Δ Acc	W. Before	W. After	Compress.
AdaDeep [17]	LeNet-5	-0.025	61.7k	-	1.8x‡
APoZ [4]	LeNet-20-50-500-10	-0.0005	656.1k	170.4k	3.85x
Deep Compression* [40]	LeNet-5	-0.0010	431.0k	35.9k	12x
Deep Compression* [40]	LeNet-300-100	-0.0010	266.0k	22.2k	12x
N2N [15]	VGG13	0.0001	9.4M	73k	127x
PCA* [5]	MLP	0.0011	407.1k	11.5k	35.4x
Pruning* [41]	LeNet-5	≈-0.0000	431.0k	29.0k	15.0x
Pruning* [41]	LeNet-300-100	≈0.0010	266.0k	14.0k	19.0x
FK agent AdaDeep [27]	LeNet-5	-0.0241	61.7k	8.4k	7.3x
FK agent SVD†	LeNet-5	-0.0342	61.7k	9.4k	6.5x
Best-known SVD policy	LeNet-5	-0.0435	61.7k	5.3k	11.6x

†Our proposal
*Non-RL compression approaches
‡Their reported value was calculated with respect to using only W_3

Our methodology has multiple advantages. The first advantage is that our approach can compress various kinds of deep learning architectures as long as the architectures use dense layers. We focused on compressing models for computer vision tasks to exploit the similarity between datasets. However, similarities can also be exploited in other domains. The second advantage is that different layers can be compressed as long as the layers can be rewritten using matrix multiplications of two-dimensional tensors. For example, we used dense layers to generate the convolution output by reshaping the kernel and the patches extracted from the feature maps, which was proposed in [9]. Extracting the patches and flattening them for the matrix multiplication is easily achieved using TensorFlow's *extract_patches* function. Similarly, researchers in Natural Language Processing (NLP) have transitioned from Long-Short Term Memory (LSTM) layers to transformers since transformers allow more parallelization [38]. The third advantage is that our methodology drastically reduces the number of states used for training the agent, which has multiple benefits. One benefit is that the size of the experience replay buffer is smaller, easing the memory requirements of the computer used to train the generalist agent. Another benefit is that fewer iterations are needed to train the agent as there is less information to learn. In contrast, other approaches use thousands of images to decide how to compress, and the agent must learn how the feature maps are modified by each action [27]. Finally, training time is also reduced due to making predictions for only a subset of images rather than the entire dataset.

There are multiple disadvantages to the proposed methodology. One disadvantage is that it increases the number of layers in a model, which can be undesirable for huge models when fine-tuning due to the vanishing gradient problem. Another disadvantage is that compressing non-dense layers requires reshaping the batch of features, which can significantly increase the execution time, depending on the implementation. If it can be implemented using TensorFlow's *reshape* function, it is a fast implementation because it is reusing the data buffer instead of rearranging the data in memory. In contrast, the *transpose* function is less efficient because it rearranges the data into a new tensor. Another disadvantage is that we train two DQNs instead of a single model. The ultimate drawback of this solution is that it fails to cater to user demands. Since demands are not part of the training loop, the user might not favor the solution provided by the agent.

We plan to apply reinforcement learning algorithms, such as Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO), to handle compression as a continuous action space since more refined precision is needed to avoid fine-tuning. Furthermore, reinforcement learning with human feedback might prove helpful in meeting user demands. Christiano et al. showed in [39] that they can "successfully train complex novel behaviors with about an hour of human time" by making a human choose the better option between two samples. This strategy can drastically reduce the training time of the agent if, as they mention, "it is possible to reduce the interaction complexity by roughly three orders of magnitude." Finally, we also intend to analyze the effect of recursively applying Singular Value Decomposition. Singular Value Decomposition has the potential to be used recursively for compressing models. Since compressing the model using SVD replaces dense layers with two smaller dense layers, each of the two smaller dense layers can also be replaced with another two dense layers. Nonetheless, this approach increases the depth of the model, which can be an inconvenience for deeper models if they are fine-tuned. The

inconvenience comes from the vanishing gradient that will cause the parameters of the first layers of a deep model to have a gradient close to zero, resulting in parameters that might not be updated due to having a low impact on the loss function.

VI. CONCLUSION

In this work, we proposed a methodology for training a generalist agent that uses Singular Value Decomposition (SVD) to compress convolutional neural networks that the agent was not trained to compress. The agent used a discrete action space mapped to compression percentages to propose compression solutions in the Pareto front of explored solutions.

Singular Value Decomposition is a powerful tool for model compression, offering the potential to approximate the parameters of layers with minimal accuracy loss. While aggressive compression of multiple layers can lead to increased accuracy loss, this can be mitigated by increasing the resolution of the action space or fine-tuning over a relatively small number of epochs. In addition, SVD can be recursively applied to further compress layers already undergoing SVD compression.

In the future, we plan to measure the impact of recursively applying SVD in model compression. As user demands are essential, we also intend to research reinforcement learning with human feedback for model compression.

We exhort others to enhance the ideas presented in this paper to achieve better results. Additionally, we encourage the development of a system that facilitates compression for non-experts. A system that can compress models for non-expert users can accelerate the adoption of deep learning models in other fields by meeting hardware requirements in an affordable time.

REFERENCES

[1] J. M. Alvarez and M. Salzmann, "Learning the number of neurons in deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 2270–2278.

[2] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*.

[3] J. Chang, Y. Lu, P. Xue, Y. Xu, and Z. Wei, "Automatic channel pruning via clustering and swarm intelligence optimization for CNN," *Appl. Intell.*, vol. 52, no. 15, pp. 17751–17771, 2022, doi: 10.1007/s10489-022-03508-1.

[4] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," 2016, *arXiv:1607.03250*.

[5] M. S. Hasan, R. Alam, and M. A. Adnan, "Compressed neural architecture utilizing dimensionality reduction and quantization," *Appl. Intell.*, vol. 53, no. 2, pp. 1271–1286, 2023, doi: 10.1007/s10489-022-03221-z.

[6] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. 15th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2016, pp. 1–12.

[7] M. Zhang, F. Liu, and D. Weng, "Speeding-up and compression convolutional neural networks by low-rank decomposition without fine-tuning," *J. Real-Time Image Process.*, vol. 20, no. 4, p. 64, 2023, doi: 10.1007/s11554-023-01274-y.

[8] S. Sun, Y. Cheng, Z. Gan, and J. Liu, "Patient knowledge distillation for BERT model compression," 2019, *arXiv:1908.09355*.

[9] M. Lin, Q. Chen, and S. Yan, "Network in network," 2013, *arXiv:1312.4400*.

[10] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 1251–1258, doi: 10.1109/CVPR.2017.195.

[11] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Comput. Surv.*, vol. 54, no. 4, p. 134, 2022, doi: 10.1145/3447582.

[12] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2018, pp. 8697–8710, doi: 10.1109/CVPR.2018.00907.

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9, doi: 10.1109/CVPR.2015.7298594.

[14] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 4095–4104.

[15] A. Ashok, N. Rhinehart, F. Beainy, and K. M. Kitani, "N2N learning: Network to network compression via policy gradient reinforcement learning," 2017, *arXiv:1709.06030*.

[16] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 815–832, doi: 10.1007/978-3-030-01234-2.

[17] S. Liu, J. Du, K. Nan, Z. Zhou, H. Liu, Z. Wang, and Y. Lin, "AdaDeep: A usage-driven, automated deep model compression framework for enabling ubiquitous intelligent mobiles," *IEEE Trans. Mobile Comput.*, vol. 20, no. 12, pp. 3282–3297, Dec. 2021, doi: 10.1109/TMC.2020.2999956.

[18] Z. Yang, Y. Zhai, Y. Xiang, J. Wu, J. Shi, and Y. Wu, "Data-aware adaptive pruning model compression algorithm based on a group attention mechanism and reinforcement learning," *IEEE Access*, vol. 10, pp. 82396–82406, 2022, doi: 10.1109/ACCESS.2022.3188119.

[19] S. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, T. Eccles, J. Bruce, A. Razavi, A. Edwards, N. Heess, Y. Chen, R. Hadsell, O. Vinyals, M. Bordbar, and N. de Freitas, "A generalist agent," 2022, *arXiv:2205.06175*.

[20] S. Lange and M. Riedmiller, "Deep auto-encoder neural networks in reinforcement learning," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2010, pp. 1–8, doi: 10.1109/IJCNN.2010.5596468.

[21] M. Agarwal, S. K. Gupta, M. Biswas, and D. Garg, "Compression and acceleration of convolution neural network: A genetic algorithm based approach," *J. Ambient Intell. Humanized Comput.*, vol. 14, no. 10, pp. 13387–13397, 2023, doi: 10.1007/s12652-022-03793-1.

[22] Y. Zhang, G. Wang, T. Yang, T. Pang, Z. He, and J. Lv, "Compression of deep neural networks: Bridging the gap between conventional-based pruning and evolutionary approach," *Neural Comput. Appl.*, vol. 34, no. 19, pp. 16493–16514, 2022, doi: 10.1007/s00521-022-07161-0.

[23] H. Zhan, W. M. Lin, and Y. Cao, "Deep model compression via two-stage deep reinforcement learning," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*. Cham, Switzerland: Springer, Sep. 2021, pp. 238–254.

[24] W. Zhang, M. Ji, H. Yu, and C. Zhen, "ReLP: Reinforcement learning pruning method based on prior knowledge," *Neural Process. Lett.*, vol. 55, no. 4, pp. 4661–4678, 2023, doi: 10.1007/s11063-022-11058-3.

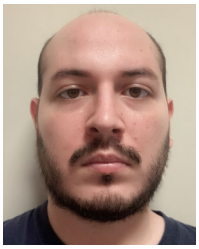
[25] W. Yang, H. Yu, B. Cui, R. Sui, and T. Gu, "Deep neural network pruning method based on sensitive layers and reinforcement learning," *Artif. Intell. Rev.*, vol. 56, no. 2, pp. 1897–1917, 2023, doi: 10.1007/s10462-023-10566-5.

[26] Ł. Dudziak, M. S. Abdelfattah, R. Vipperla, S. Laskaridis, and N. D. Lane, "ShrinkML: End-to-end ASR model compression using reinforcement learning," in *Proc. Interspeech*, Sep. 2019, pp. 1–5, doi: 10.21437/INTERSPEECH.2019-2811.

[27] G. Gonzalez-Sahagun, S. E. Conant-Pablos, J. C. Ortiz-Bayliss, and J. M. Cruz-Duarte, "A generalist reinforcement learning agent for compressing convolutional neural networks," *IEEE Access*, vol. 12, pp. 51100–51114, 2024, doi: 10.1109/ACCESS.2024.3385857.

[28] S. Yu, A. Mazaheri, and A. Jannesari, "Topology-aware network pruning using multi-stage graph embedding and reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 25656–25667.

[29] M. Alwani, Y. Wang, and V. Madhavan, "DECORE: Deep compression with reinforcement learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2022, pp. 12339–12349.
[30] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 2815–2823, doi: 10.1109/CVPR.2019.00293.
[31] Z. Wang, N. D. Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," in *Proc. Int. Conf. Int. Conf. Mach. Learn.*, vol. 48, Jun. 2016, pp. 1995–2003.
[32] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI Conf. Artif. Intell.*, vol. 30, 2016, pp. 1–7.
[33] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015, *arXiv:1511.05952*.
[34] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms," 2017, *arXiv:1708.07747*.
[35] T. Clanuwat, M. Bober-Irizar, A. Kitamoto, A. Lamb, K. Yamamoto, and D. Ha, "Deep learning for classical Japanese literature," *arXiv:1812.01718*, 2018.
[36] Y. LeCun, C. Cortes, and C. Burges. (2010). *MNIST Handwritten Digit Database*. [Online]. Available: https://yann.lecun.com/exdb/mnist
[37] L. V. Dias, P. B. Miranda, A. C. Nascimento, F. R. Cordeiro, R. F. Mello, and R. B. Prudncio, "ImageDataset2Vec: An image dataset embedding for algorithm selection," *Expert Syst. Appl.*, vol. 180, Oct. 2021, Art. no. 115053, doi: 10.1016/j.eswa.2021.115053.
[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 6000–6010.
[39] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 4302–4310.
[40] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*.
[41] F. Manessi, A. Rozza, S. Bianco, P. Napoletano, and R. Schettini, "Automated pruning for deep neural network compression," in *Proc. 24th Int. Conf. Pattern Recognit. (ICPR)*, 2018, pp. 657–664.
[42] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 806–814.
[43] A. Raihan and T. Aamodt, "Sparse weight activation training," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 15625–15638.
[44] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002, doi: 10.1109/4235.996017.



GABRIEL GONZALEZ-SAHAGUN received the B.S. degree in digital systems and robotics and the M.Sc. degree in intelligent systems from the Tecnolgico de Monterrey, in 2014 and 2017, respectively, where he is currently pursuing the Ph.D. degree in computer science with the Department of Computer Science. His research interests include computer vision, robotics, and evolutionary algorithms. Afterward, he was with Ocean Freight Exchange, where he was the Lead Engineer in developing a scheduling optimization system prototype for fuel delivery in the port of Singapore. The project was awarded first place in Singapore's Smart Port Challenge 2018. Later, he became a Research Associate with the Tecnologico de Monterrey, where he has been taught computational intelligence and computer vision courses, since 2019.

SANTIAGO ENRIQUE CONANT-PABLOS (Member, IEEE) received the B.Sc. degree in industrial engineering from the Instituto Tecnologico de Sonora and the M.Sc. degree in computer science and the Ph.D. degree in artificial intelligence from the Tecnologico de Monterrey, in 2004. He is currently an Associate Research Professor with the School of Engineering and Sciences, Tecnologico de Monterrey, a Researcher of the Group with a Strategic Focus on Advanced Artificial Intelligence, and an Adjoint Member of the SOI-STEM Interdisciplinary Group, Institute for the Future of Education. His research interests include machine learning, computer vision, evolutionary and bio-inspired computation, hyper-heuristics design, and natural language processing. He is a member of Mexican National System of Researchers and Mexican Academy of Computing.

JOSÉ CARLOS ORTIZ-BAYLISS (Member, IEEE) was born in Culiacan, Sinaloa, Mexico, in 1981. He received the B.Sc. degree in computer engineering from the Universidad Tecnologica de la Mixteca, in 2005, the B.Sc. degree in project management from the Universidad Virtual del Estado de Guanajuato, in 2019, the M.Sc. degree in computer sciences from the Tecnologico de Monterrey, in 2008, the M.Ed. degree from the Universidad del Valle de Mexico, in 2017, the M.Ed.A. degree from the Instituto de Estudios Universitarios, in 2019, and the Ph.D. degree from the Tecnologico de Monterrey, in 2011. He is currently an Assistant Research Professor with the School of Engineering and Sciences, Tecnologico de Monterrey. His research interests include computational intelligence, machine learning, heuristics, metaheuristics, and hyper-heuristics for solving combinatorial optimization problems. He is a member of Mexican National System of Researchers, Mexican Academy of Computing, and the Association for Computing Machinery.

JORGE M. CRUZ-DUARTE (Senior Member, IEEE) was born in Ocaa, Norte de Santander, Colombia, in 1990. He received the B.Sc. and M.Sc. degrees in electronics engineering from the Universidad Industrial de Santander (UIS), Bucaramanga, Santander, Colombia, in 2012 and 2015, respectively, and the Ph.D. degree in electrical engineering from the Universidad de Guanajuato (UGTO), Guanajuato, Mexico, in 2018.

From 2019 to 2021, he was on a postdoctoral stay with the Tecnolgico de Monterrey (TEC) in collaboration with Chinese Academy of Sciences (CAS). Since 2021, he has been a Research Professor with the Research Group on Advanced Artificial Intelligence, TEC, and a member of Mexican National System of Researchers (SNI-CONAHCyT), ACM, and AMEXCOMP. His research interests include automatic design, heuristics, fractional calculus, applied thermodynamics, data science, and artificial intelligence.