A Methodology for Neural Network Architectural Tuning Using Activation Occurrence Maps

Rafael Garcia
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
rgarcia@inf.ufrgs.br

Alexandre Xavier Falcão
Instituto de Computação
Universidade de Campinas
Campinas, Brazil
afalcao@ic.unicamp.br

Alexandru C. Telea
Department of Computer Science
University of Groningen
Groningen, The Netherlands
a.c.telea@rug.nl

Bruno Castro da Silva
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
bruno.silva@inf.ufrgs.br

Jim Tørresen
Department of Informatics
University of Oslo
Oslo, Norway
jimtoer@ifi.uio.no

João Luiz Dihl Comba
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
comba@inf.ufrgs.br

***Abstract*—Finding the ideal number of layers and size for each layer is a key challenge in deep neural network design. Two approaches for such networks exist: *filter learning* and *architecture learning*. While the first one starts with a given architecture and optimizes model weights, the second one aims to find the best architecture. Recently, several visual analytics (VA) techniques have been proposed to understand the behavior of a network, but few VA techniques support designers in architectural decisions. We propose a hybrid methodology based on VA to improve the architecture of a pre-trained network by reducing/increasing the size and number of layers. We introduce Activation Occurrence Maps that show how likely each image position of a convolutional kernel's output activates for a given class, and Class Selectivity Maps, that show the selectiveness of different positions in a kernel's output for a given label. Both maps help in the decision to drop kernels that do not significantly add to the network's performance, increase the size of a layer having too few kernels, and add extra layers to the model. The user interacts from the first to the last layer, and the network is retrained after each layer modification. We validate our approach with experiments in models trained with two widely-known image classification datasets and show how our method helps to make design decisions to improve or to simplify the architectures of such models.**

***Index Terms*—Deep Learning, CNNs, Visual Analytics, Model Understanding, Architecture Tuning**

I. Introduction

Designing the appropriate neural network for a learning task requires deciding over several factors such as the optimizer algorithm, loss function, regularization parameters, activation functions, number of layers, and type and size of each layer [1]. Most such decisions are made empirically, using experience from previous similar problems and general 'good practice' guidelines, and often use a trial-and-error approach to search for the best architecture. This task is time-consuming and may not lead to models with expected performance.

Visual Analytics (VA) techniques have recently been increasingly used to help designers with architecture decisions [2]. Most such approaches focus on feature understanding, *i.e.*, explain which type of features a neuron learned to recognize, and support interpretability [3]–[8], by helping to understand how the model process the input features to predict output labels. However, there is still a gap between architectural and interpretability tasks. While many VA tools tackle the task of finding the particular features a neuron learned [9], [10], they do not address the *end-to-end* problem of deciding if these features are indeed enough or useful for the prediction task, and, implicitly, the question whether a given (set of) neuron(s) helps the network's overall task.

To close this gap, we propose a VA tool to help designers make architectural decisions on the number and size of layers in a model. Our method uses three visualizations: Activation Occurrence Maps (AOMs), Occurrence Difference Matrix (ODMs), and Class Selectivity Maps (CSMs); and a novel metric to evaluate the overall selectivity of a neuron. These tools help the designer make decisions such as: (1) remove neurons performing redundant roles, *i.e.*, recognizing the same features, or neurons that do not contribute to the prediction process; (2) increase the size of a layer if no neurons learned useful features for one or more classes; and (3) add more layers to the model if sets of classes still do not present very selective features. By following our methodology, practitioners can guide the design of novel models from scratch or improve pre-trained models by identifying neurons that can be dropped — reducing overfitting — or the need for more neurons or layers — improving performance. Additionally, our method can help the task of *transfer learning* [11], as our tools can guide the selection of the most useful features to transfer to the new model. While the focus of our method is to address convolutional networks (CNNs), whose convolutional neurons

This work is partially supported by FAPESP, grant no. 2014/12236-1; CNPq, grant no. 303808/2018-7 and 308851/2015-3; FAPERGS, grant no. 17/2551-000; RCN, grant no. 240862; RCN and SIU, grant no. 261645; SIU, grant no. UTF-2016-short-term/10128.

from now on we call *kernels*, our approach can easily be adapted to fully-connected and recurrent models.

Our paper is organized as follows. Section II discusses related work in both architecture modeling and VA for deep learning (DL). Section III details the DL engineering tasks that our approach assists. Section IV explains in detail our approach and the involved techniques and metrics. Section V presents a series of experiments where we validate our approach on two image classification problems. Finally, Section VI concludes the paper, outlining future work directions.

II. Related Work

Architecture tuning is one of the main challenges for DL engineering [12]. The simplest way to tune such networks is to grid search for the best architecture, testing many combinations of number/type of layers and layer sizes and other hyperparameters, and choose the set-up maximizing performance. This strategy is unpractical when working with deep models because training a single model is computationally expensive. Recent approaches involve learning the architecture adaptively during model training [13], training a reinforcement learning model to create architectures for a given input learning problem [14], or to automatically adapt the network's topology to the input sample, so that the network can have a faster and simpler prediction process if the sample is easy to predict or a more complex topology otherwise [15]. Still, no such methods allow designers to use their experience to modify the model.

Several works tackle the subgoal of reducing redundant or unimportant model components. Cogswell *et al.* [16] use a regularization technique to reduce redundancy by minimizing the cross-covariance of activations in the model's layers. Pruning methods can reduce the number of weights in a fixed architecture without significant accuracy loss [17]–[20]. A model can learn the pruned architecture alongside weights during training [21]–[23]. Recent studies show that such methods often achieve results similar to the same reduced architecture learned from scratch, turning over-parameterized training unnecessary [24]. Our approach, in contrast, provides designer-reasoning to the network reduction process without the need for pre-training a much larger model than necessary.

Visual Analytics (VA) has provided significant support for deep learning [2]. Until now VA mostly focused on feature understanding and model interpretability: Salience maps [9] highlight the image pixels contributing most to a given neuron's activation, thereby showing to designers which image features contribute to the prediction process. Activation Maximization [10] creates an image that maximizes the activation of a given neuron, giving insights on which features the neuron recognizes. While such methods help *interpreting* the learned features, they do not tell if these features are selective towards the possible output labels. Our approach tackles this task, as it does not focus on *which* features a neuron learned, but how *useful* these features are to the prediction process.

Closer to our goal, some VA methods address the goal of evaluating the quality of a model's components. Rauber *et al.* [7] project activation vectors of hidden layers to help deciding whether a given layer distinguishes the classes or not. While adequate for evaluating the quality of a layer, this approach does not give any information about the quality of individual neurons. Zhong *et al.* [25] propose two metrics to gauge neuron's quality, but their metrics are not class-specific like ours. Arguably closest to our work, DeepEyes [26] uses heatmap matrices to let the user search for kernels that do not activate for any image or activate for all images, which in either case are ineffective kernels. Yet, they do not allow exploring activations in different regions of the kernel's output, which is vital in datasets where a feature's position may add value to the prediction process.

III. Modeling Tasks

A DL engineer must make several hard architectural decisions when using neural networks to solve a learning problem. First, one must choose the right number and size of model layers [27]. However, there is no analytic way to find the best architecture a model should have to solve a problem. Designers often end up choosing more or fewer layers/neurons than they should, leading to the issues listed below. These issues appear on the design of most types of model architectures — DFNs, CNNs, RNNs, etc. — and most learning tasks — classification, regression, generative models, etc. Our model can be adapted to address all of those.

Too few layers: A model with fewer layers than ideal may fail to distinguish classes separated by highly abstract features which are created from simple features found in early layers.

Too many layers: Conversely, the use of too many layers in a model also creates problems. Deep networks have, by construction, a high number of parameters, which significantly increases when adding new layers. As other ML methods, neural models are prone to overfitting, which mainly happens when the number of parameters in the model is too high. A high number of layers or neurons also severely increases the training time and the memory space required to store the model, turning their use prohibitive in systems such as embedded devices, which are increasingly necessary for applications in fields like robotics and embedded computing [17].

Too small layers: For a neural network to operate well, it must iteratively, layer-by-layer, change the representation of the input to distinguish between the different classes in the final layer [7]. It does it by recognizing low-level features that are more likely to appear when the input belongs to a particular class. If a layer does not have enough neurons to learn all such features, that layer will likely harm the overall prediction capability. When this happens, increasing the number of layers does not help, as the next layers do not have enough 'simple' features to build meaningful higher-level features.

Too large layers: An overestimated number of neurons per layer can also harm the model, increasing the chance of overfitting, training time, and the space required for the model, just as for a model with too many layers.

Ineffective neurons: Even if not overfitting, when training a model having a high amount of neurons, it is unlikely that all neurons become equally important for the task at hand after training. While some neurons learn features that are crucial to the prediction task, others may learn features that do not contribute much to it or even harm the performance [23].

Redundant neurons: Multiple neurons may learn to identify closely-related features, leading the model to have redundant information across its components [28]. If one can find such groups of neurons and keep from them only a subset, one can reduce the size of trained models and maintain (nearly) the same performance as before reduction.

We identify three main tasks where our approach addresses a subset of the above issues:

1. Find redundant or ineffective kernels: Ineffective kernels (that do not learn to recognize features useful for recognizing any class) are strong removal candidates without decreasing performance. Conversely, groups of kernels (in the same layer) that learned to recognize very similar input features can be reinitialized to see if they next can learn more diverse features. Alternatively, the designer can simplify such a group by reducing it to one or a few kernels.

2. Find too small layers: When the number of kernels in a layer is not enough to learn all the required features, two or more classes may always activate in the same set of kernels in that layer. This behavior makes it hard for the next layers to build more complex features capable of recognizing each class and can cause the model to underperform. In this case, the designer may want to increase the layer's size by adding more kernels.

3. Find too shallow models: Ideally, the activations of the model's last hidden layer should be very discriminative, with each class activating in a different set of neurons. If this does not happen, *e.g.*, if many neurons activate for multiple classes, the designer may want to add extra layers so that the model can build higher-level features.

Automating these tasks is challenging as it is hard or impractical to define accurate quantitative tests to 'query' for the presence of too small layers, too shallow models, kernel redundancy, and kernel (in)effectiveness. Consider kernel redundancy: Comparing all activations of just two kernels is too expensive, as it needs $|K| \cdot |K| \cdot |\mathbf{D}|$ comparisons for $|K|$ kernels in a layer and $|\mathbf{D}|$ elements in the training set. Moreover, convolutional kernels usually output high-dimensional activations, further increasing the complexity of such a comparison. Considering groups of more than two kernels makes the problem quickly intractable. Even if such computational costs would be acceptable, there is not an exact threshold for defining the degree of activation similarity that makes two kernels redundant. Comparing the learnable parameters of each kernel is also unreliable, as different parameter sets can learn to recognize (nearly) the same features and the function of weight vectors is hard to interpret, especially in deeper layers.

IV. Visual Analytics Approach

Most previous VA solutions for DL engineering used *activation values* to analyze neurons and answer questions such as what features a kernel is learning [3], [9], [29] or for which class it is specializing itself [8], [30]. The activations of a neuron contain the values that will be used as input by the next layer. We can see them as a new representation of features from the input sample. If a neuron activates a high value at some position because it is looking for features an input sample has, this becomes an indication of what the neuron is doing. Activations are easier to understand than weight values, which are hard to interpret, especially in deeper layers where they interact with activations produced by the composition of many neurons with equally hard to interpret weights.

To better understand the role of a neuron, we must look at activations produced by *several* inputs. Otherwise, we do not know how useful is the feature producing high activations in the neuron. Indeed, such a feature may appear in every class or be an input-specific feature that does not appear in other elements of the same class. In both cases, learning such a feature does not help the prediction process. A typical way to overcome this issue is to look at the *mean activation* produced by a kernel for all elements of a given class [8], [30]. If a neuron has a high mean activation, we can assume more confidently that it indeed learned a feature useful for that class. Yet, high mean activations can be misleading. The activation produced by a neuron in a non-final layer serves as input to the next layer. If the weights from the next layer that interact with this activation value are small enough, having a high activation may not be that important. Moreover, finding the neuron(s) with high(est) activations does not directly help the tasks defined in Sec. III. For instance, two kernels can recognize essentially the same feature, but have different activation values, due to the way their weights evolved during training. Thus, analyzing only their mean activation value does not tell us that there is redundancy between two kernels.

Given the above, we propose to look at the *proportion* of positive activations instead of their absolute values. When a kernel outputs a positive value, it tells that it found to some extent the learned feature. A non-positive activation value (clamped to zero when using the common *ReLU* activation function), tells that the kernel did not find such a feature. Considering this, we can assume that a kernel that learned a feature capable of distinguishing one class (or a set of classes) from the rest will produce positive activations for most elements from this class (or set of classes). Hence, if we find kernels with a high proportion of positive activations for elements of some class c_i and a low proportion for elements of some other class c_j, we can say that this kernel is *selective* toward this pair of classes.

To allow designers to find and analyze such kernels, and also find cases, such as groups of classes for which such patterns do not occur, we propose three visualization techniques:

The *Activation Occurrence Maps* (AOMs) display the proportion of samples from a given class c_i activating positive values at each position of the output activation of a given kernel k. Similar AOMs for a kernel for all classes c_i indicate kernels which react similarly over all classes, thus ineffective for the overall network goal (Task 1). AOMs also show kernels that strongly activate for similar input subsets, thus

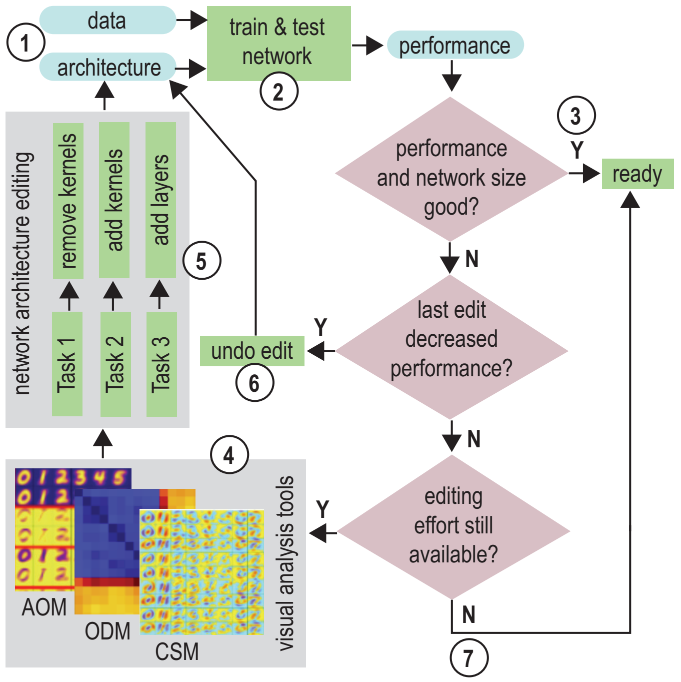

Fig. 1. Workflow that describes all the steps followed by a ML designer when applying visual analytics in the design of neural networks.

highlighting potentially redundant kernel groups that can be further simplified (Task 1). Set of classes producing similar AOMs in every kernel indicate a small layer size (Task 2). Finally, kernels producing AOMs with strong occurrence for multiple classes is a signal of a need for more layers to discriminate between these classes (Task 3).

The *Occurrence Difference Matrix* (ODM) visually summarizes the difference between occurrence values in the AOMs of two kernels k_i and k_j.

Finally, the *Class Selectivity Maps* (CSMs), which displays how selective each image position of a given kernel k is for distinguishing elements of class c from elements of other classes, thus helping in the insights provided by the AOMs (Tasks 1..3). Additionally, we propose a metric to assign an overall selectivity value for a kernel, which allows designers to decide which kernel should be kept in the network when simplifying a group of redundant kernels (Task 1).

The AOM, ODM, and CSM visualizations are used to support tasks 1..3 for DL engineering via the following VA workflow (Fig. 1). The workflow starts with the engineer selecting some initial network architecture (number and size of layers), based on heuristics or good practices in the domain (1). The network is next trained and tested, as usual (2). If the result is deemed satisfactory, *e.g.*, a good accuracy is obtained, and the network's overall size and learned features are acceptable, the process stops (3). If not, the three views are used to detect cases where the network's architecture likely can be improved (4). Rather than using automatically computed thresholds, the engineer examines such cases visually and decides which of tasks 1..3 (s)he wants to execute next and where, *e.g.*, which group of neurons is redundant and which can be eliminated from it (5). Upon finding such a case, the engineer next edits the network and repeats from step 2. If the accuracy drops significantly after such an edit, the engineer undoes the edit (6). The process stops when no additional edits can be done without losing too much accuracy, when the engineer decides that the network has been re-architected satisfactorily, or when the available time for editing has finished (7).

We next describe the AOM, ODM, and CSM visualizations for convolutional kernels. These visualizations can be easily adapted to handle fully-connected neurons, by interpreting the neuron's output activation as having one single position.

A. *Activation Occurrence Maps*

Deep neural networks — particularly those employed in image-based problems — often have a sequence of convolutional layers with *ReLU* activation functions as the first layers, typically followed by a *pooling* and some regularization layers such as *dropout* [31]. The role of convolutional layers is to find particular features that help in the prediction task regardless of where these appear in the image. For example, a neuron from the first convolutional layer takes the original image as input and produces another image as activation. This activation image has positive values in places where the original image contains the feature(s) this neuron is looking for, and zero activations (after *ReLU*) elsewhere [1].

To let the designer identify how a kernel behaves for different classes, and how different kernels act for the same class, we must understand how the occurrence of positive activations changes in different regions of the kernel's image output. This knowledge is needed since different classes may present very similar features — particularly in the first layers — but, for a given class, these features may appear more often in a different model-layer than for other classes [30]. To allow the designer to analyze such differences, we construct an Activation Occurrence Map (AOM) for each (kernel, class) pair in the layer of interest.

For a giving kernel k and a giving class c in the training set, we compute the corresponding AOM $M^{k,c} \subset \mathbb{R}^{WxH}$, where W and H are the dimensions of the activation produced by the kernel k when the network process any given input image. Each cell $M^{k,c}_{i,j}$ in the AOM shows the proportion of training images from class c activating positive values at position (i,j) of the activation produced by k, and is computed as

$$M^{k,c}_{i,j} = \frac{1}{|\mathbf{D}^c|} \sum_{x \in \mathbf{D}^c} U(a^k(x)_{i,j}) \tag{1}$$

where $\mathbf{D}^c$ is the set of images in the training set $\mathbf{D}$ belonging to class c; $a^k(x)_{i,j}$ is the value at position (i,j) in the activation matrix produced by kernel k for the input sample x; and U is the Heaviside step function.

Figure 2 displays the AOMs corresponding to the kernels in the first (left) and second (right) layer of a model trained with the MNIST dataset [32] (see Sec. V), with the values $M^{k,c}_{i,j}$ color-coded via an ordinal colormap. Rows in this image

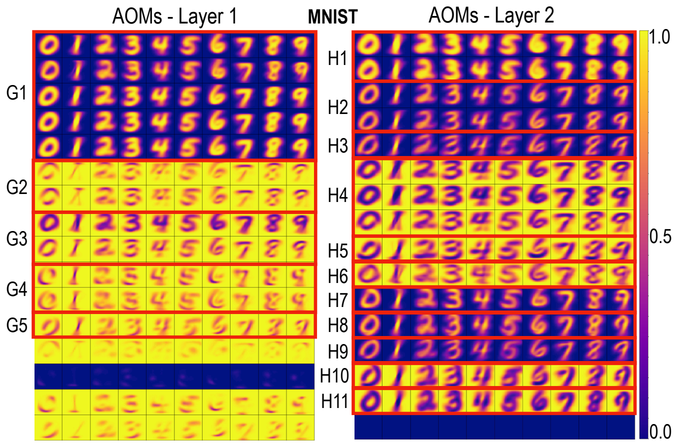

Fig. 2. AOMs produced by the first two layers in a model trained with the MNIST dataset. Each row represents one of the 16 kernels in the layer, while each column is a class. The highlighted groups (red border rectangles) contains kernels recognizing similar features (See Sec. V). We notice that some kernels activate more often to specific features in the images, such as the digit structure (G1, H1), particular border orientations (G2, G4, G5), background (G3, H4), or different handwriting styles (H2, H3, H7, H9, H9).

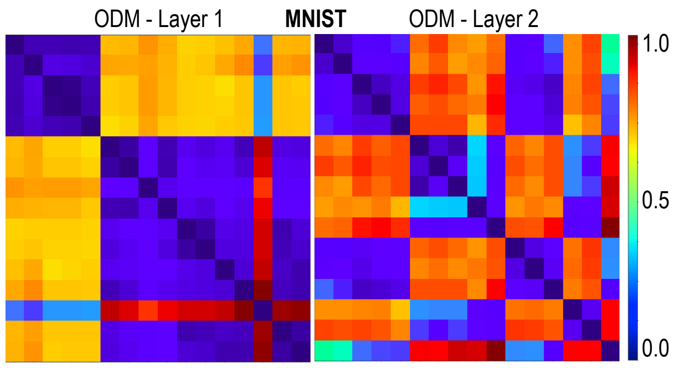

Fig. 3. ODMs for the two first layers in the MNIST model (see Sec. V). The ODM is a symmetric matrix displaying the average difference between the AOMs produced by each pair of kernels. It provides a concise overview of the similarities than the AOMs, with the trade-off of also providing fewer details. Nonetheless, it can be used as a first step to search for groups of similar kernels in large layers.

correspond to kernels, and columns to classes, respectively. Rows are ordered using groups of kernels with similar AOMs.

When one kernel produces similar AOMs for all classes, it is very likely that this kernel is not effective for the prediction process. For example, if a position in the output of a kernel produces positive activations — or instead, activates very rarely — to all or almost all inputs from every class in the training set, this position is unlikely to give the next layer useful information about which label the model should assign. We see such a pattern for the kernels shown in row 13 and 14 from the top in Fig. 2 (left). These kernels produce very similar AOMs for all classes in the training set, with all positions activating very rarely, or never at all. Hence, they are good candidates for removal or reinitialization. We show later in Sec. V that we can remove this type of kernel with little or no performance reduction.

AOMs are helpful to find redundant kernels as well. When multiple kernels display too similar maps for every class in, this tells us that these kernels recognize very similar features, and thus may be redundant. The kernels in groups denoted by the red rectangles in Fig. 2 shows an example of this issue. These kernels recognize almost identical features across the classes, which may make unnecessary to keep both.

B. Occurrence Difference Matrix

To analyze a single layer using AOMs, we need to display $|K| \cdot |C| \cdot W \cdot H$ squares, where $|K|$ is the number of kernels in the layer, $|C|$ is the number of classes in the training set, and W and H are the kernel's output dimensions. DL networks used in complex prediction tasks often have large layers with hundreds or even thousands of kernels. Also, training sets for such tasks can contain as many classes [33]. Constructing and using AOM matrices as shown in Fig. 2 has thus limited scalability. An alternative to finding kernels producing similar AOMs (that may thus be redundant) is to build the *Occurrence Difference Matrix* (ODM) of the layer.

The ODM is a symmetric matrix D where each cell $D_{k_i,k_j} \in \mathbb{R}^+$ measures the average difference between the AOMs corresponding to kernels k_i and k_j for every corresponding position in the AOMs and every class in the training set, computed as

$$D_{k_I,k_J} = \frac{1}{|C| \cdot W \cdot H} \sum_{c \in C} \sum_{i=0}^{i<W} \sum_{j=0}^{j<H} \left| M_{i,j}^{k_I,c} - M_{i,j}^{k_J,c} \right|, \quad (2)$$

where C is the set of classes in the training set; W and H are the dimensions of the kernels' output; and $M_{i,j}^{k,c}$ is the position (i,j) in the AOM of kernel k and class c (Eqn. 1).

Figure 3 shows the ODMs calculated for the AOMs of both layers displayed earlier in Fig. 2. Each value D_{k_I,k_J} is encoded using an ordinal blue-to-red colormap. Rows and columns in this matrix-like image follow the kernel order in Fig. 2. In practical settings, the designer can sort rows with a matrix-reordering algorithm [34] to easily identify similar-value cells. We can see in Fig. 3 that kernels whose AOMs display strong similarity in Fig. 2 also display strong similarity in the ODM. However, ODMs are more compact than AOM matrices: As each of its cells encodes just a single value by color, we can easily visualize ODMs for layers up to thousand kernels on a single screen. In contrast, the AOM matrix can visually scale only to a few tens of kernels, given that each of its cells requires a resolution equal to that of the activation output. The ODM is particularly helpful for **Task 1**, as it can concisely inform the designer about redundant kernels that can be removed or reinitialized to improve the model's accuracy.

C. Class Selectivity Maps

Our third visualization, *Class Selectivity Maps* (CSMs), helps the DL engineer find how selective each position of a kernel k's output is towards a given class c. If a position often activates for items of a given class c but does not often activate for items of any other class, this position is *selective*

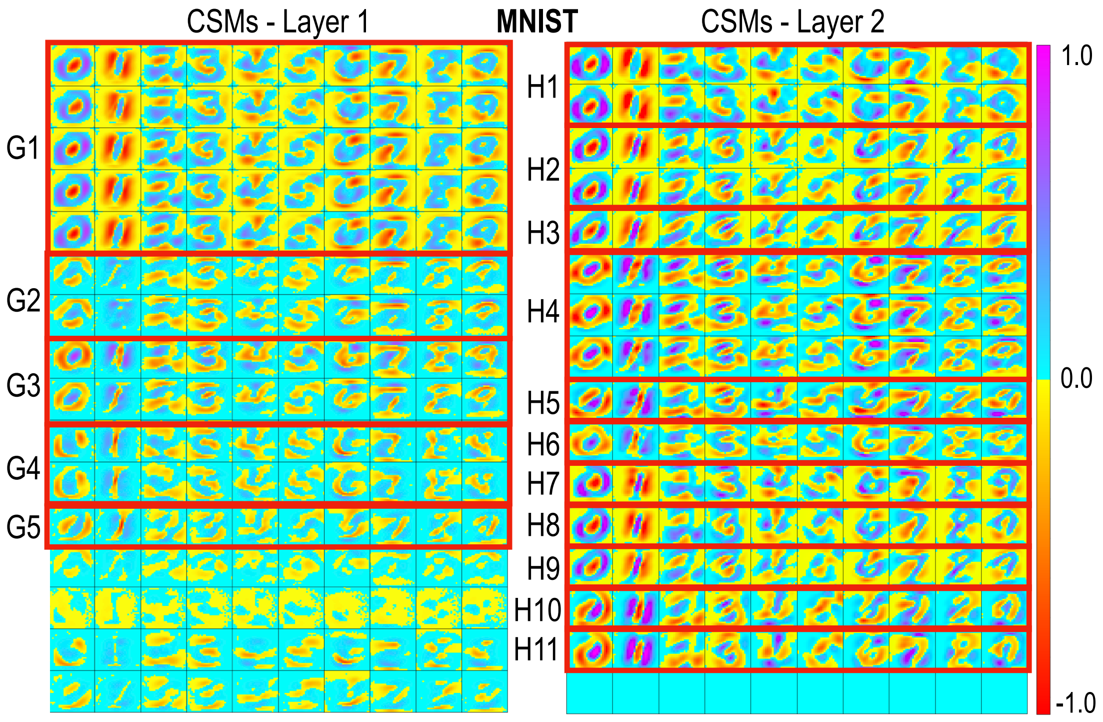

Fig. 4. CSMs produced by the first two layers in the MNIST model (see Sec. V). Each row represents one of the 16 kernels in each layer, while each column is a class. Kernels in each layer respect the same order of Fig. 2. With this view, we can identify regions of the kernel's output image where some classes are more selective, i.e., they activate more often than other classes. For instance, the kernels in group H1 display a strong selectivity towards round-shaped inner structures in the digits, while group H2 displays stronger selectivity towards flat shapes, such as digit one. Identifying such regions is important because they are the most likely to help the prediction process — e.g., discriminating between class 0 and 1.

towards class c, *i.e.*, it learned how to (partially) distinguish items of class c from items of other classes.

A CSM $S^{k,c}$ is a matrix where each element $S^{k,c}_{i,j}$ tells how selective position (i,j) of kernel k is for class c, computed as

$$S^{k,c}_{i,j} = \frac{1}{|C|-1} \sum_{d \in C}^{d \neq c} M^{k,c}_{i,j} - M^{k,d}_{i,j}, \tag{3}$$

where C is the set of classes in the training set; and $M^{k,c}_{i,j}$ is the position (i,j) in the AOM of kernel k and class c (Eqn. 1).

Figure 4 shows the CSMs produced by the first and second layer of our MNIST model (see Sec. V), *i.e.*, the same layers analyzed in Figs. 2 and 3. Rows indicate kernels and columns indicate classes, like in the AOM matrix. The values $S^{k,c}_{i,j} \in [-1,1]$ are color-coded using a two-segment colormap ranging from cyan (0) to purple (1) and yellow (0) to red (-1), respectively. Hence, purple regions indicate where the analyzed kernel k is very selective towards the selected class c; red regions indicate positions where the kernel k is very 'dismissive' of class c, that is, the position outputs positive values for c far less often than for any other class. For example, one can notice that several kernels in the 2nd layer in Fig. 4 (right) are selective in most of the digit area in images from class 'zero'. However, some kernels, such as the one in group H11, are more selective in the 'inner circle' of the digit zero. This behavior indicates how different kernels may learn the different features of the class. We give more details about the insights the user can take from this visualization in Section V.

D. *Kernel Selectivity*

While CSMs depict well the selectivity of different *regions* in each kernel of a layer, they do not assign an *overall* selectivity value for the whole kernel. Having such a value would help when deciding which kernel to keep in the model from a group of redundant kernels or evaluate if a kernel is a good candidate for removal or reinitialization due to its poor contribution to the model (Task 1). This editing of the network can have three added values. First, one can keep the most selective kernel and thereby minimize the potential performance loss when simplifying the network. Secondly, the network size is decreased leading to the already mentioned speed and space benefits. Finally, the overall performance can be increased by reinitializing unsatisfactory kernels that do not aggregate useful features or introduce overfitting to the model.

Given the matrix S^k containing all the CSMs of kernel k for every position and class, we compute the overall selectivity $s(k)$ of kernel k as

$$s(k) = \sum_{c_0,c_1 \in C | c_0 \neq c_1} |avg(S^{k,c_0}) - avg(S^{k,c_1})|, \tag{4}$$

where C is the set of classes in training set, and $avg(S^{k,c})$ is the average value of all positions in the CSM for kernel k

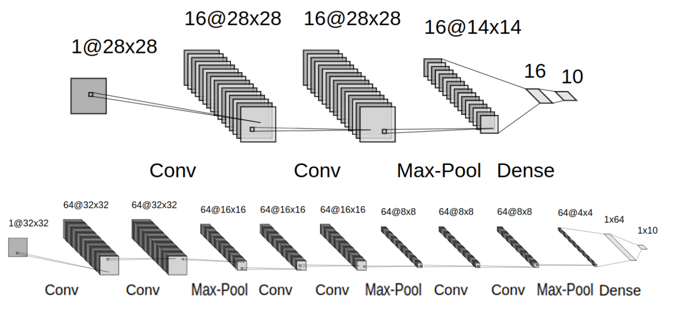

Fig. 5. The MNIST model (top) contains two convolutional layers followed by a max pooling layer, a hidden fully-connected layer, and the output decision layer. After training, this model achieves 96.96% accuracy on the test set. The CIFAR10 model (bottom) contains a more complex architecture with three sequential groups of layers formed by two convolutions, one max pooling and one dropout layer followed by a hidden dense layer and the output decision layer. After training, the model achieves 80.54% accuracy on the test set.

and class c. In Sec. V, we demonstrate the usefulness of our metric with an experiment.

V. Experiments

We ran a series of experiments to show how designers can use our techniques to perform the tasks described in Sec. III. For this, we use two image classification models trained with two widely-known datasets: (1) *MNIST*, containing images of handwritten digits from 0 to 9; and (2) *CIFAR10*, containing images displaying an object belonging to one of ten different classes — airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. For both datasets, we aim to design a deep neural network that can classify images into the respective ten classes. Figure 5 displays the architecture for both models. In this experiment, we apply our method to the activation of each convolutional layer before pooling. However, the method can easily be applied after the subsequent pooling layer — regardless of the pooling technique —, as these layers contain a more concise representation of the features identified in the previous layer.

A. Analysis and layer size reduction of the MNIST model

We next use our VA tools to simplify this model but keep a high accuracy. After training, we create the visualizations shown in Figs. 2, 3, and 4 using all training-set images. Following the insights from our VA approach, we remove kernels that do not help the prediction task, either because they do not recognize features useful to distinguish any class or because they are redundant *vs* other same-layer kernels (Task 1, Sec. III). A more complex alternative to kernel removal is to reinitialize these kernels, so we prefer removal for simplicity.

We start our analysis by looking at the AOMs of the first convolutional layer kernels (Fig. 2 left). In row 14, we easily spot a kernel whose most positions rarely activate for any class. So, this is a kernel that most likely does not contribute to the prediction task. Also, we noticed groups of kernels with very similar patterns of activation occurrence. For example, kernels in the G1 group often activate at pixels inside the digit area, and rarely at pixels outside it. These kernels learn a similar feature: to recognize the digit structure.

The other kernels in the visualized layer show the opposite pattern: They activate more often in pixels outside the digit area than inside. These kernels recognize features such as the different border orientations that appear in each class. However, they do not necessarily have the same role: Some of them activate more often around the top of the digit, *e.g.*, group G4, while others activate more often around the bottom of the digit, *e.g.*, group G2. Still, some of these kernels look to be quite redundant *vs* each other, and thus may not add useful information to the prediction. To check this, we look at the occurrence difference matrix (ODM) for this layer (Fig. 3 left). Here, we can spot several groups of kernels with strong similarity (cells with dark blue shades).

Next, we analyze the CSMs produced by these potentially redundant kernels (Fig. 4 left). While the AOMs show us which kernel regions activate for different classes, the CSMs show whether these activations are selective or not, *i.e.*, if high-occurrence activations provide enough information to help the network decide if the input belongs to a given class or not. The CSMs show us that kernels in the last three rows are not selective towards any class, which makes them unlikely to be relevant to the model's decisions. In contrast, other kernels, *e.g.*, group G1, show regions of strong selectivity for some classes, telling that their features are meaningful enough to help the model's decision. We proceed by removing kernels we found as not useful for the reasons stated above. These correspond to the last three rows in Figs. 2 and 4 (left). Also, we group kernels with strong similarity in the AOMs (kernel groups correspond to red triangles in Fig. 2) and keep a single representative of each group in the model (we delete the others). The kept kernel is the one with the highest kernel selectivity (see Sec. IVD) for a given group.

After keeping only the chosen five kernels in the 1st layer of the model, we freeze the weights of this layer and retrain the model from the second layer onward. This way, we ensure that our model cannot modify the features learned by those five kernels and thus has to use only these features (and whatever more abstract features it builds in deeper layers) to perform the prediction. With just one retraining epoch, our model achieves 96.93% accuracy, virtually the same it had with all 16 kernels in the 1st layer. This shows that the five chosen kernels cover enough features to achieve the same classification capability we had with 16 kernels.

We repeat the process in the second convolutional layer of our MNIST model. The AOMs for this layer (Fig. 2 right), tell us that its kernels are much more diverse concerning the features they recognize than kernels in layer 1. Often, they learn particular features that appear in some writing styles. For instance, the kernel on groups H8 recognizes digits written in a 'rounded' way, while kernel on row H7 recognizes digits from a 'flatter' writing. Also, Figs. 2 and 4 (right) help us recognize kernels that are redundant or not selective enough to help our model. Red triangles in Fig. 2 (right) shows the found kernel groups. We perform a similar edit to layer 1, freeze weights

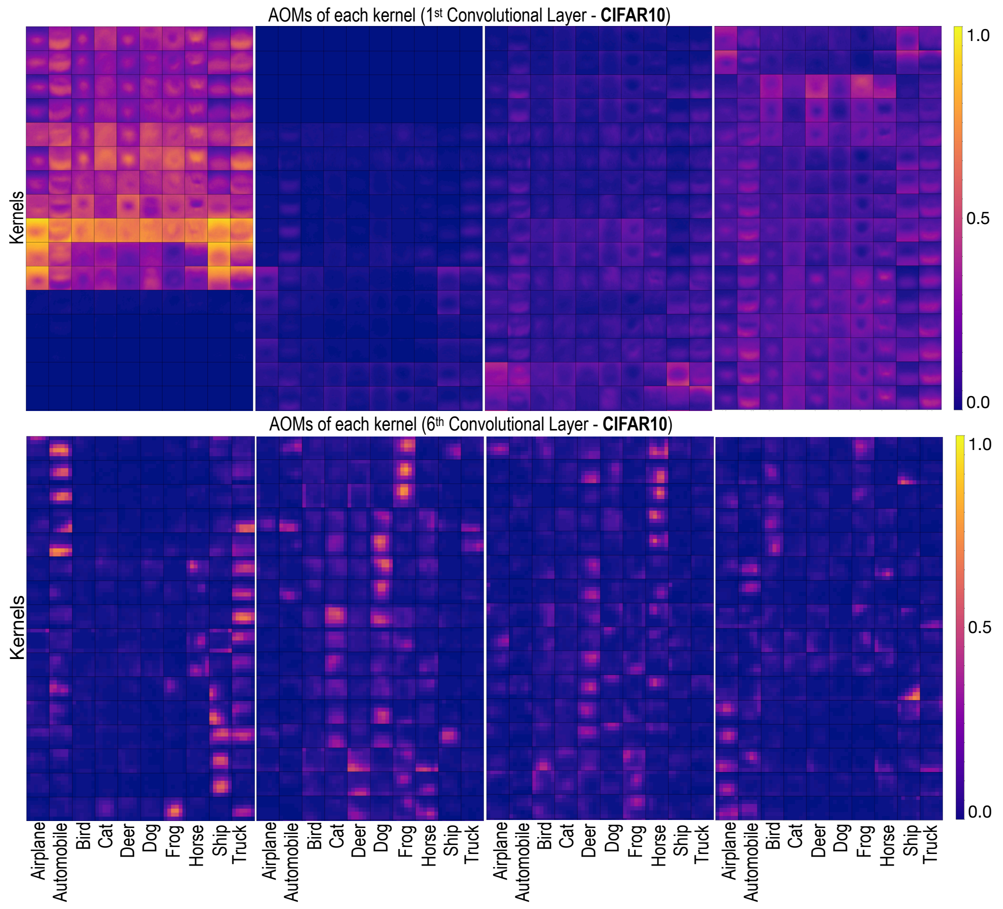

Fig. 6. AOMs for the 64 kernels in the 1st and 6th layers of the CIFAR10 model. Kernels are sorted by the similarity tree computed by *aggregative clustering* the AOM rows. The 1st layer cannot learn features discriminative enough to each class, which denotes the need for more layers in the model. Note that while kernels in the 1st layer may produce positive activations for several classes, such activations tend to appear in different regions of the output image for different pairs kernel x class. If this behavior is not present, the next layers cannot use these features to build more discriminative ones, which indicates a need for more kernels in the layer. Finally, the 6th layer provides much more discriminative kernels, often activating positive values for just one class, indicating that the model is unlikely to improve performance if more layers are added.

in layer 2, retrain, and obtain an accuracy of 97.69%. So, our approach allowed us to not only simplify the model but also *improve* its accuracy. Note that this accuracy was obtained only by retraining the weights in the hidden fully-connected layer and the output layer of the model, so this accuracy relies solely on the features previously learned — before removal — by the convolutional kernels we selected to keep.

B. Analysis and layer size reduction of CIFAR10 model

We repeat the previous experiment for the model trained with the CIFAR10 dataset. Fig. 6 shows the AOMs for the 1st and the 6th convolutional layers in the model. Both views give us interesting insights into the layers' behaviors. First, many kernels in both layers do not often activate for any class, suggesting kernels that did not learn to recognize features useful for classification. Secondly, kernels in the 1st layer that activates often, usually do so to multiple classes. This tells that this layer cannot learn features complex enough to distinguish *individual* classes, suggesting that the model needs more layers to perform the prediction task (Task 3, Sec. III).

Conversely, in the 6th layer, often only one class produces positive activations in a given kernel. This tells that, at this point, the model already separates classes as best as it can, and more layers are unlikely to improve performance. Some classes, *e.g.*, cat and deer, do not achieve an occurrence close to 100% in *any* kernel. This suggests that the features the layer learned for these classes are not enough to cover all their samples, indicating the need to learn more features about them. Hence, we found that this or previous layers need more kernels to discover more features (Task 2, Sec. III).

Figure 7 shows the CSMs of each of the 64 kernels in the 1st and 6th layers of the CIFAR10 model. We see that some kernels (or regions) with high occurrence values in the corresponding AOMs (Fig. 6) are not very selective among different classes, making them a poor choice to keep in the network. In contrast, we see in layer kernels that are very selective for subsets of classes (*e.g.*, ship and truck). This tells that, while this layer is not enough to find class-specific features, it can find features that only appear in a small subset of classes, thus easing the prediction task of the next layers.

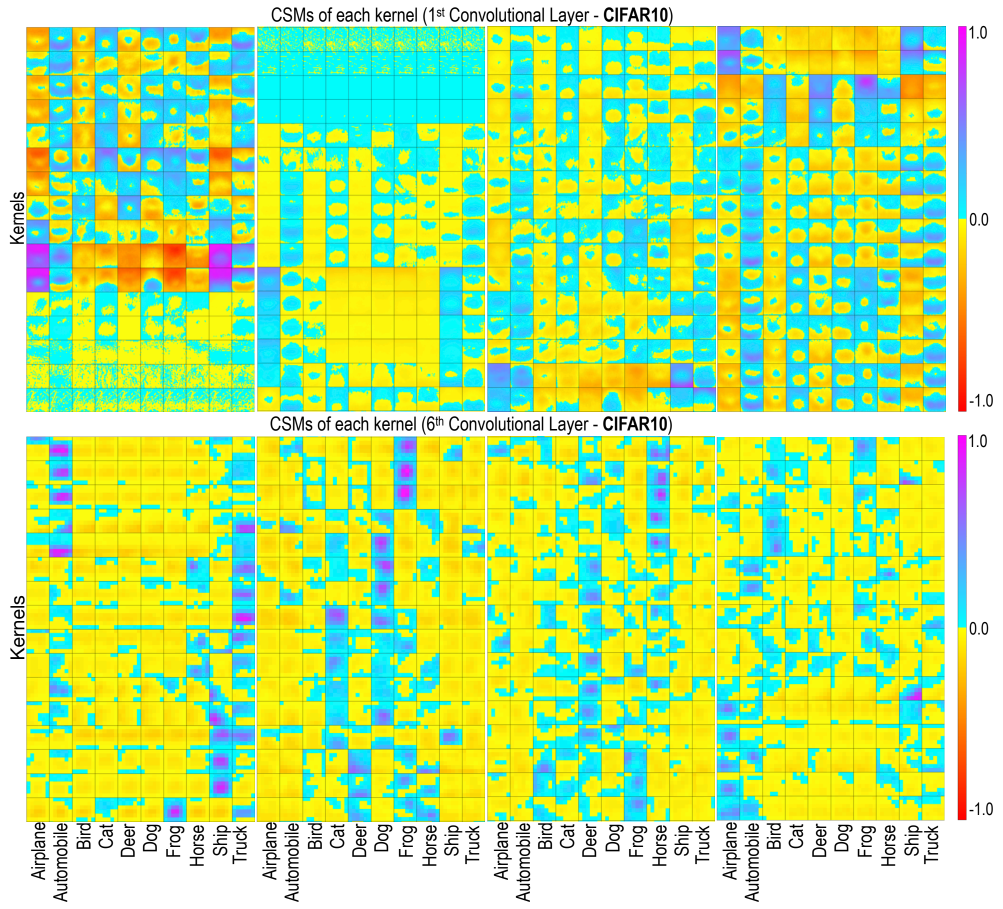

Fig. 7. CSMs for the 64 kernels in the 1st and 6th layers of the CIFAR10 model. Kernels follow the same order of Fig. 6. Notice that while several kernels in the 1st layer show a high activation occurrence for all classes (see Fig. 6), they are usually much more selective to only a couple of classes. The CSMs give more confidence to designer decisions, as it clearly states if a high occurrence pattern in AOMs indeed indicates selectivity.

Due to the width and depth of this model, we only reduce the size of the 1st and 6th convolutional layers. As for the MNIST model, we spot and remove kernels that are not selective for any class, and simplify groups of redundant kernels (details omitted for brevity). With our VA tools, we reduced the 1st layer' size to 20 kernels. After retraining the rest of the model for one epoch, and freezing the weights in the 1st layer —, our reduced model achieved 81.20% test-set accuracy, even *higher* than the initial 80.54% accuracy.

Following our VA approach, we reduced the size of the 6th layer from 64 to 35 kernels achieving a test set accuracy of 80.43% after retraining only the weights in the fully-connected layers for one epoch. This accuracy is marginally below the original one. We see here a trade-off between network size and performance: At some point, the network simplification has to stop as the accuracy will inherently drop. As stated earlier, an alternative is to reinitialize the kernels selected for removal (instead of removing them) to achieve higher accuracy.

C. *Kernel Selectivity Experiment*

To show how our kernel selectivity method indeed captures the overall selectivity of a kernel, we run the following experiment: For each pair of kernel k and class c in the network, we compute the average of the pair's class selectivity map $S^{k,c}$, denoted $s^{k,c}$. Then, for each class c_i, we remove the kernels with $s^{k_j,c_i} \leq 0$, and compute the accuracy of the resulting network only for test-set elements of class c_i. Next, we do the opposite: For each class c_i, we remove the kernels with $s^{k_j,c_i} > 0$ and compute the accuracy of the network without such kernels for test-set elements of class c_i.

We did these experiments for the 6th CIFAR10 model layer (CSMs shown in Fig. 7 right). Fig. 8 (top) shows the number of kernels kept in each case. Fig. 8 (bottom) compares the test-set accuracy for elements of each class c_i and considering all kernels, kernels not selective for c_i, and kernels selective for c_i. In all cases, the number of selective kernels for a given class is much smaller than the number of non-selective kernels. In all cases, the test-set accuracy drops significantly for the chosen class when we remove highly selective kernels for that class and increases when we only keep selective kernels. Hence, our selectivity metric indeed captures well how much a kernel contributes to the separation of a given class.

VI. Conclusion

In this work, we present a visual analytics set of tools, and associated workflow, that helps machine learning designers in

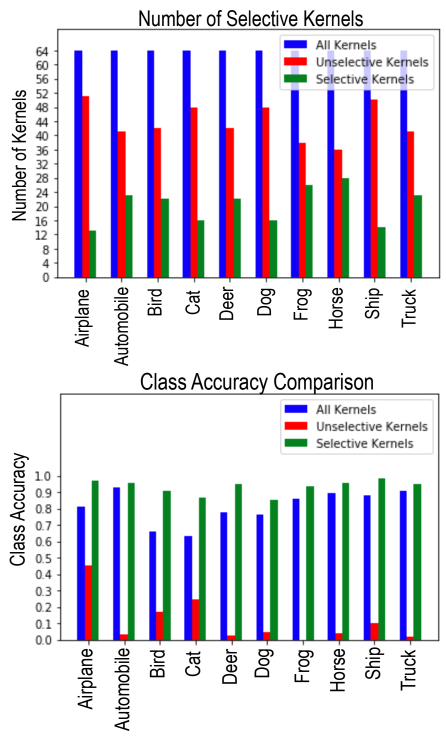

Fig. 8. Selective Experiment on 6th layer CIFAR model. Top figure displays the number of kernels considered to be selective (green) and not-selective (red) for each class. Bottom figure displays how much the class accuracy changes when we keep only the kernels from each one of these groups.

their deep learning architectural decisions. We define three tasks related to such decisions and show how our techniques support them. We show how our toolset and workflow can be used in practice to considerably reduce the sizes of two trained deep-learning models for non-trivial classification tasks while keeping, or even increasing, classification performance. Our work opens multiple future work possibilities. While our method is readily adaptable for fully-connected neurons, recurrent layers such as LSTMs are much more challenging. Such layers contain hidden internal states that modify their values at every input timestep. Analyzing how the AOMs of such hidden states change over time is an interesting direction. Separately, our visualizations do not scale well for large networks of tens of layers and thousands of neurons. We plan next to study how high-dimensional data visualization, well studied in the visual analytics community [35], could be applied to improve the scalability of our method.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[2] R. Garcia, A. C. Telea, B. C. da Silva, J. Torresen, and J. L. D. Comba, "A task-and-technique centered survey on visual analytics for deep learning model engineering," *Computers & Graphics*, vol. 77, 2018.

[3] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu, "Towards better analysis of deep convolutional neural networks," *IEEE TVCG*, vol. 23, 2017.

[4] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proc. ECCV*. Springer, 2014, pp. 818–833.

[5] J. Yosinski, J. Clune, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," in *Proc. ICML – Workshop on Deep Learning*, 2015.

[6] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush, "LSTMVis: A tool for visual analysis of hidden state dynamics in recurrent neural networks," *IEEE TVCG*, vol. 24, no. 1, pp. 667–676, Jan 2018.

[7] P. Rauber, S. G. Fadel, A. Falcão, and A. Telea, "Visualizing the hidden activity of artificial neural networks," *IEEE TVCG*, vol. 23, no. 1, 2017.

[8] M. Kahng, P. Andrews, A. Kalro, and D. Chau, "Activis: Visual exploration of industry-scale deep neural network models," *IEEE TVCG*, vol. 24, no. 1, 2018.

[9] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *CoRR*, vol. abs/1312.6034, 2013.

[10] D. Erhan, Y. Bengio, A. Courville, and P. Vincent, "Visualizing higher-layer features of a deep network," *University of Montreal*, 2009.

[11] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big Data*, vol. 3, no. 1, 2016.

[12] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *CoRR*, 2018.

[13] C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang, "Adanet: Adaptive structural learning of artificial neural networks," *CoRR*, 2016.

[14] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *CoRR*, 2016.

[15] A. Veit and S. J. Belongie, "Convolutional networks with adaptive computation graphs," *CoRR*, vol. abs/1711.11503, 2017.

[16] M. Cogswell, F. Ahmed, R. B. Girshick, L. Zitnick, and D. Batra, "Reducing overfitting in deep networks by decorrelating representations," *CoRR*, vol. abs/1511.06068, 2015.

[17] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[18] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. ICCV*, 2017.

[19] J. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *CoRR*, vol. abs/1707.06342, 2017.

[20] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *CoRR*, vol. abs/1608.08710, 2016.

[21] Z. Huang and N. Wang, "Data-driven sparse structure selection for deep neural networks," in *Proc. ECCV*, 2018.

[22] H. Liu, K. Simonyan, and Y. Yang, "DARTS: differentiable architecture search," *CoRR*, vol. abs/1806.09055, 2018.

[23] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., 2015, pp. 1135–1143.

[24] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *CoRR*, vol. abs/1810.05270, 2018.

[25] W. Zhong, C. Xie, Y. Zhong, Y. Wang, W. Xu, S. Cheng, and K. Mueller, "Evolutionary visual analysis of deep neural networks," in *Proc. ICML – Workshop on Visualization for Deep Learning*, 2017.

[26] N. Pezzotti, T. Hollt, J. V. Gemert, B. P. F. Lelieveldt, E. Eisemann, and A. Vilanova, "DeepEyes: Progressive visual analytics for designing deep neural networks," *IEEE TVCG*, vol. 24, no. 1, pp. 98–108, 2018.

[27] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.

[28] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc., 2013, pp. 2148–2156.

[29] A. Nguyen, A. Dosovitskiy, J. Yosinski, T. Brox, and J. Clune, "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks," in *Proc. NIPS*, 2016, pp. 3395–3403.

[30] B. Alsallakh, A. Jourabloo, M. Ye, X. Liu, and L. Ren, "Do convolutional neural networks learn class hierarchy?" *IEEE TVCG*, 2018.

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J Mach Learn Res*, vol. 15, no. 1, pp. 1929–1958, 2014.

[32] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, 1998.

[33] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE CVPR*, 2009.

[34] M. Behrisch, B. Bach, N. Henry Riche, T. Schreck, and J.-D. Fekete, "Matrix reordering methods for table and network visualization," *Computer Graphics Forum*, vol. 35, no. 3, pp. 693–716, 2016.

[35] S. Liu, D. Maljovec, B. Wang, P. Bremer, and V. Pascucci, "Visualizing high-dimensional data: Advances in the past decade," *IEEE TVCG*, 2017.