Computers & Graphics 77 (2018) 30–49

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Survey Paper

A task-and-technique centered survey on visual analytics for deep learning model engineering[☆]

Rafael Garcia[a,b,*], Alexandru C. Telea[c], Bruno Castro da Silva[a], Jim Tørresen[b], João Luiz Dihl Comba[a]

[a] Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
[b] Universitetet i Oslo, Oslo, Norway
[c] University of Groningen, Groningen, The Netherlands

ARTICLE INFO

Article history:
Received 15 April 2018
Revised 17 September 2018
Accepted 24 September 2018
Available online 3 October 2018

Keywords:
Visual analytics
Neural network visualization
Deep learning visualization
Deep learning
Neural Networks
Visual analytics survey

ABSTRACT

Although deep neural networks have achieved state-of-the-art performance in several artificial intelligence applications in the past decade, they are still hard to understand. In particular, the features learned by deep networks when determining whether a given input belongs to a specific class are only *implicitly* described concerning a considerable number of internal model parameters. This makes it harder to construct interpretable hypotheses of *what* the network is learning and *how* it is learning—both of which are essential when designing and improving a deep model to tackle a particular learning task. This challenge can be addressed by the use of visualization tools that allow machine learning experts to explore which components of a network are learning useful features for a pattern recognition task, and also to identify characteristics of the network that can be changed to improve its performance. We present a review of modern approaches aiming to use visual analytics and information visualization techniques to understand, interpret, and fine-tune deep learning models. For this, we propose a taxonomy of such approaches based on whether they provide tools for visualizing a network's architecture, to facilitate the interpretation and analysis of the training process, or to allow for feature understanding. Next, we detail how these approaches tackle the tasks above for three common deep architectures: deep feedforward networks, convolutional neural networks, and recurrent neural networks. Additionally, we discuss the challenges faced by each network architecture and outline promising topics for future research in visualization techniques for deep learning models.

1. Introduction

One of the main goals of Artificial Intelligence (AI) is to build systems that achieve human-level efficiency in recognition tasks, such as image classification, speech recognition, and sentiment analysis. Although most of these tasks seem trivial to human beings, they are extre mely challenging for computer algorithms due to the lack of a formal description of how to solve such problems. For example, humans can, in general, easily recognize if there is a dog in a given image, but it is hard to tell how we got to this conclusion. In other words, it is not clear how to formalize which features in the image makes humans recognize the presence of dogs [1]. Is it the shape of the objects? Is it the color contrast between different regions in the image? Moreover, even harder, how do humans learn to recognize dogs in the first place? How do we learn to use such features to identify dogs? How can we teach such learning abilities to machines? One of the areas of AI that focus on finding solutions to approach these problems is called Machine Learning (ML). ML algorithms use statistical techniques to optimize functions to progressively achieve better performances in a particular task [2]. To train an ML model, the designer must provide the model with training inputs with known answers—e.g., images of dogs and images without dogs—, the model thus automatically learn to model a function that minimizes the chances of wrongly predicting such inputs.

In the area of machine learning, a particular class of techniques has proven increasingly efficient and effective in pattern recognition applications in the past years: deep learning (DL) techniques. In contrast to what may be called 'classical' ML techniques, DL techniques do not rely on the designer to provide a set of

[☆] This article was recommended for publication by Stefan Bruckner.
[*] Corresponding author at: Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil.

E-mail addresses: rgarcia@inf.ufrgs.br (R. Garcia), a.c.telea@rug.nl (A.C. Telea), bruno.silva@inf.ufrgs.br (B. Castro da Silva), jimtoer@ifi.uio.no (J. Tørresen), joao.comba@gmail.com (J.L. Dihl Comba).

https://doi.org/10.1016/j.cag.2018.09.018
0097-8493/© 2018 Elsevier Ltd. All rights reserved.

hand-engineered features to be used in the learning and decision processes. Rather, they rely on a (large) set of labeled samples to automatically extract and store such features in the so-called architecture of a deep neural network (DNN) [3]. DNNs contain multiple (up to hundreds of) layers that perform simple filtering, thresholding, and aggregation operations on subsets of the input data samples. The power of such architectures relies (1) on their ability to model very complex nonlinear decision functions and decision boundaries in the input data space by combining many such simple operations; and (2) on the fact that the set of learned parameters involved in implementing such operations (also known as the *model* learned by the network) can be automatically inferred from a (typically large) set of labeled data samples, by minimizing the error between the predicted and the ground-truth labels.

DNNs have recently shown excellent performance in pattern recognition tasks in part due to the increase in computing power available for training ever larger architectures, e.g., by using GPU computing. Early on, Krizhevsky et al. proposed the AlexNet network [4], a convolutional neural network (CNN) architecture with six hidden layers that won the 2012 ILSVRC competition on the well-known ImageNet dataset [5], with a top 5 test error rate of 15.4%; i.e., the percentage of images on the test set whose true label was not among the 5 classes considered to be more likely by the model. This architecture was later refined by Zeiler and Fergus [6], which decreased the training set size by one order of magnitude while providing an improved 11.2% error rate. In that publication, the authors have argued that often the “development of better models is reduced to trial and error”, and proposed a pioneering visualization for depicting feature maps to aid the understanding of the network training process. By increasing the network size (or *depth*, as measured in terms of number of layers), and the training set size, subsequent works managed to provide further increases in task accuracy; e.g., in the work of Simonyan and Zisserman [7], where a network with over 100 layers was used; in GoogLeNet [8] (22 layers, achieving top 5 error 6.7% for ILSVRC); and in Microsoft’s ResNet [9] (152 layers, where a 3.6% error was achieved for ILSVRC). Several other similar examples have been published in the past few years.

Even though higher accuracy may be achieved by increasing the architectural complexity of DNNs, such a design choice also brings about several important challenges in deploying such networks. First, the computational training effort required by such systems becomes quite high—DNNs can often require days or even weeks of training time even when using a system composed of many GPUs. Suboptimal configurations of the network architecture, such as its training set size and the network hyperparameters, can also require the entire learning process to be restarted or run from scratch, which is very expensive. As a consequence of these challenges, it becomes increasingly important for one to be able to *understand* the complex interaction between all of these aspects (i.e., the characteristics and size of the training and testing datasets; the network architecture; the network hyperparameters; and the test results) in order to effectively fine-tune a DNN for a specific task. This is by itself a very complex challenge as raised by Liu et al. [10], grounded chiefly in two reasons: (1) the size of the space spanned by all these design dimensions is *huge*, so an exhaustive exploration thereof is impossible; moreover, this space is highly non-linear, since small changes to the hyperparameters may cause significant changes to the performance of the corresponding trained model; (2) the *abstract* nature of this space—i.e., the fact that it is hard to intuitively understand the effect of particular changes to hyperparameters in the corresponding performance—makes it hard to understand how and why a DNN behaves in a certain way.

The above constraints determine that, in practice, most designers construct and train their DNN models virtually as ‘black box’ algorithms. More specifically, while designers do have, at least at a high conceptual level, a relatively good idea of what pooling, dropout, and convolutional layers implement, the joint effect of combining several (tens to hundreds) of such layers and varying their parameters can only, in practice, be assessed via empirical end-to-end measurements of the performance of the resulting DNN. This introduces several important and, so far, only partially answered questions:

- *What has a model learned?* Without a good understanding of *what* decision hypothesis a given trained network has acquired, users and designers of a DNN typically have no idea about what regions of the original data space (sampled by the training process) were ‘internalized’ by the resulting model. As such, it is not clear how to allocate further training efforts to improve the network. Also, a model may have poor generalization performance. Currently, this can only be verified by testing the network on novel data (validation) or by deploying it in field operation—when it may be too late to detect generalization issues. Understanding what a model has inherently learned (or not) is therefore of key importance.
- *Why has a model learned a particular decision hypothesis?* Related to the above point is the challenge of understanding why a model has learned (or not) to generalize certain aspects of the training data. Knowing this may directly provide feedback to the designer as to what needs to be changed in the network architecture, hyperparameters, or in the training data to further strengthen desired properties of the network or to address particular issues. Moreover, this will also give a better understanding of the properties of the input data space, e.g., in terms of sub-spaces that are particularly challenging for the learning process. Without knowing why a particular model resulted from the training process, DNN optimization can very much be a blind search process.
- *How has a model learned a particular decision hypothesis?* Even if a DNN performs well, one may want to understand how it has internally stored knowledge about the training data; e.g., in which specific layers (or parts thereof) or parameter value ranges were particular types of knowledge encoded. This may help in understanding the reason why specific DNN architectures are appropriate for specific tasks and thus help to extrapolate such knowledge when tackling new problems. Without this, addressing a new problem in a different data space might require starting the entire network engineering process from scratch.

Recently, these concerns have been discussed primarily by the DL community. For example, Marcus [11] argues that the DL field may be ‘approaching a wall’ and outlines ten challenges: (1) requirements for huge amounts of labeled data; (2) limited capacity for transfer between problems; (3) difficulty of dealing with hierarchical structure; (4) difficulty to deal with open-ended inference; (5) *DL is not sufficiently transparent*; (6) it is hard to integrate prior knowledge; (7) it is hard to distinguish correlation from causation; (8) assumption of a stable world; (9) *DL’s answers cannot be always trusted*; (10) *DL is hard to engineer with*. Among these, we focus here on challenges (5), (9), and (10), which directly relate to our previously-made points regarding the ‘black box,’ unpredictable, and hard to fine-tune nature of DNNs, respectively. Similar concerns regarding these issues have been expressed in the works of Samek et al. [12] and Ribeiro et al. [13].

The need to address the above challenges has been recognized by scientists at the confluence of several domains (data science, machine learning, and data visualization). One particular approach to achieving this goal, which we survey in this paper, is to use *visualization* techniques and tools. This approach is rooted in earlier works related to understanding high-dimensional data spaces

[14–18] and visualizing the operation of classical ML algorithms—in particular, visualizing feature spaces and the impact of using different feature selection processes [19–27]. However, such earlier techniques do not directly address the challenges that are particular to DNNs, such as the ones mentioned earlier in this section, and instead focus on understanding more general correlations between the network structure, the high-dimensional input-feature spaces used during training, and the resulting network performance. More recently, novel techniques in information visualization and visual analytics (VA) have aimed at supporting and improving DNN engineering by taking into account their particular challenges; however, the relatively fast growth of this field has not yet been fully covered by existing surveys.

Contributions: We aim to alleviate the problems discussed above with the following contributions:

1. we propose a task-and-architecture based taxonomy of visualization techniques that help engineering DNNs by whether they tackle one of *three possible tasks*: (1) facilitating the visualization of the network structure; (2) facilitating the interpretation and analysis of the training process; or (3) allowing for feature understanding. We explain the particularities of these different tasks, discussing their goals, their challenges and how they are applied in the context of three types of network architecture: deep feedforward networks, convolutional neural networks, and recurrent neural networks;
2. we survey a comprehensive body of over 40 papers related to visualization in DNN engineering, and explain how these fit within the proposed task taxonomy;
3. we outline important limitations of current work in the area and suggest directions for future exploration.

The structure of this survey is as follows. In Section 2, we introduce import concepts and notation regarding classical machine learning and deep learning engineering. In Section 3, we detail the above-mentioned tasks and in Section 4 how visual analytics techniques relate to the main deep learning architectures in order to complete such tasks. In Section 5, we introduce our taxonomy and discuss in details different visualization techniques used for tackling these tasks in the context of deep feedforward networks, convolutional neural networks, and recurrent neural networks. Section 6 discusses how these techniques cover the needs of different types of DNN designers and also outlines important technical limitations which can spawn future research directions. Section 7 concludes the paper.

Relation with other surveys: Several other surveys partially touch our focus of interest. The most important existing surveys related to ours include tutorials on deep learning visualization [28–31]; visualization of convolutional networks [32,33]; visualization of machine learning models [34]; predictive visual analytics [35]; interactive machine learning [36–38]; interpretable machine learning [39]; and surveys of multidimensional visualization techniques [14,16,17]. Closest to our focus, Hohman et al. [40] present a survey on visual analytics for deep learning. Their survey follows a human-centered approach to answer the following questions: (1) why to visualize different aspects of a deep model or its corresponding training process; (2) who uses deep learning visualization; (3) what to visualize in deep learning; (4) how to visualize deep learning; and (5) when in the process of designing and training a network the visualization process will take place—e.g., during the network engineering step; throughout training; or after the model parameters and corresponding features are learned. Our survey also addresses these questions but offers another angle of attack that looks at visualization techniques classified by the task and subtask it addresses and which model architecture (DFNs, CNNs, RNNs) they apply it. Furthermore, we focus specifically on how such tools support a visual analytics approach for solving the above tasks, rather than on the more general perspective of how to visualize deep learning data. As such, our survey is complementary to, and also extends, the work of Hohman et al.

2. Classical machine learning and deep learning

In this section, we introduce basic notation relevant both for classical machine learning algorithms and also for deep learning models. Later, we will clarify what are the main differences in how these techniques work.

Let $\mathbf{x} \in X$ be an input (e.g., an image or a sound) drawn from a set or distribution of possible inputs X (e.g., the set of all possible images). Let $y \in Y$ be a label or output associated with a particular input $\mathbf{x}$. If the model is tackling a *regression* task, y is a (possibly high-dimensional) continuous value. Otherwise, in *classification* tasks, y is a discrete label associated to one or more members of a finite set of classes which elements of X can belong to. Both regression and classification are considered supervised learning techniques, as they are trained with inputs which the actual labels are known. For instance, if $\mathbf{x}$ is an image, Y could be the set {dog, cat}, used to denote the possible animals that may appear in the image. In many practical applications, a particular label y in a set of d possible labels is represented by a vector $\mathbf{y} \in \{0, 1\}^d$, where the ith element of $\mathbf{y}$ is 1 only if $\mathbf{y}$ corresponds to the ith possible label in Y. Let $D = \{\mathbf{x}_i, y_i\}$, for $i \in \{1, \ldots, N\}$, be a set of N training examples associating particular inputs $\mathbf{x}$ with their corresponding labels y. The objective of a supervised learning algorithm is to analyze a training set D and construct a function f: $X \to Y$ so that when f is presented with novel inputs (e.g., unseen images) it can correctly predict their corresponding label.

Machine learning algorithms usually optimize their performances by incrementally improving the function f in order to minimize a given cost function C that measures how well f performs; i.e., how well that function correctly predicts the labels of novel inputs. In the regression setting, when f predicts labels that are continuous numbers, C can be a Mean Squared Error such as $C(f) = E_{\mathbf{x} \sim X}\left[(f(\mathbf{x}) - y)^2\right]$. In many practical applications, the function f might be easier to learn if the input information is presented to it in a pre-processed way; for instance, when training an algorithm for detecting cats in pictures, it might be easier to learn an f function that takes as inputs edge and color information instead of raw pixel values. This can be achieved by transforming the inputs $\mathbf{x} \in X$ via a so-called feature function $\Psi : X \to \mathbb{R}^m$ mapping any element of X to a point in some m-dimensional. Each of the m values in $\Psi(\mathbf{x})$ is a *feature* of $\mathbf{x}$ denoted as $\Psi^i(\mathbf{x})$, where i is the index of a given feature in $\Psi(\mathbf{x})$. In this case, $f : \mathbb{R}^m \to \mathbb{R}^d$, i.e., it takes as input some $\mathbf{x}$ and feeds $\Psi(\mathbf{x})$ to f, which produces a prediction in $\mathbb{R}^d$ associated with a given label.

When deploying an algorithm to learn a function f that minimizes the given cost function C, the designer needs to implicitly specify what is the space of possible functions that will be searched over. This is typically done by representing f via a set of so-called *model parameters* $\Theta = \{\theta_1, \ldots, \theta_m\} \in \mathbb{R}^m$. By assigning different numerical values for each of the m model parameters, we obtain a different function f; for instance, f could be in the form $f(\mathbf{x}) = \theta_0 + \theta_1\mathbf{x}$ in case of a simple linear prediction model. On the other hand, by using a Θ with a much higher number of parameters (typical case in neural networks), f may become an arbitrarily complex, nonlinear, and thus difficult to interpret function. Prior to using a learning algorithm, it is common to group the features of all training inputs into a single matrix $\mathbf{X} \in \mathbb{R}^{N \times m}$, whose i-th row is associated with input $\mathbf{x}_i$ and where columns store the m components of the feature vector $\Psi(\mathbf{x}_i)$. In what follows we denote as $\mathbf{X}_i^j$ as the jth feature value of the ith input in $\mathbf{X}$.

Classical ML algorithms: Classical ML algorithms include techniques such as *k-nearest neighbors* [41], *support vector machines*

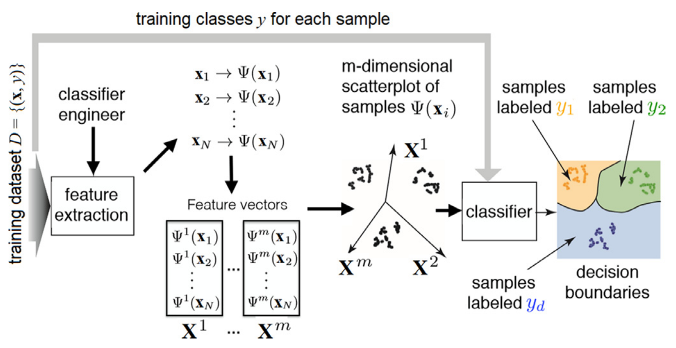

training classes y for each sample
training dataset $D = \{(\mathbf{x}, y)\}$
classifier engineer
feature extraction
$\mathbf{x}_1 \to \Psi(\mathbf{x}_1)$
$\mathbf{x}_2 \to \Psi(\mathbf{x}_2)$
$\mathbf{x}_N \to \Psi(\mathbf{x}_N)$
Feature vectors
$\mathbf{X}^1$ ... $\mathbf{X}^m$
m-dimensional scatterplot of samples $\Psi(\mathbf{x}_i)$
classifier
samples labeled y_1
samples labeled y_2
samples labeled y_d
decision boundaries

Fig. 1. Design of classical machine learning algorithms. The pipeline starts with the collection of a training set D and a test set (not shown in the picture). Secondly, the engineer selects the features that will be used for learning. Such features can be regular attributes of the original dataset or more complex features generated by some preprocessing. Often, VA techniques play a key role in helping engineers to select relevant features. Once the features are chosen, the model is trained by learning to make accurate predictions on training set samples. Once the training is finished, the engineer checks the performance of the model on the test set and then decides what to do next. If the model is not performing properly, some choices one could take are: resume training for more epochs; modify model hyperparameters; or select a different set of features for the training set.

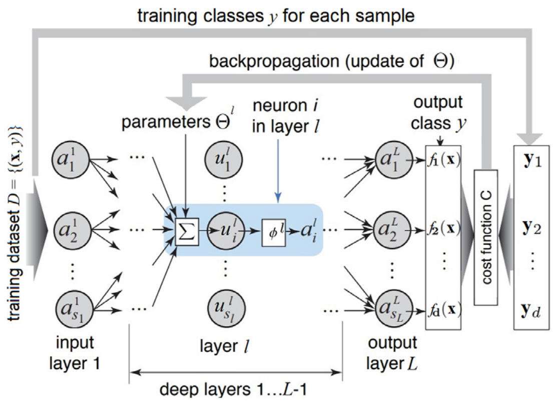


Fig. 2. Architecture of typical DNN (training phase). Raw inputs are received by the first layer, which produces an activation a^1 by multiplying the input vector by a matrix of parameters followed by a (possibly non-linear) activation function. This activation works as input to the next layer that computes its own activation a^2. The process continues until a prediction is outputted on the last layer of the model. As in classical machine learning algorithms, the predicted output is compared to the actual one and a cost error is calculated measuring how good the prediction was. Thus, parameters in the model are updated accordingly, in order to minimize the prediction error. As neural networks have parameters distributed among multiple layers, a backpropagation algorithm is needed to update all layers properly. In this algorithm, the error in deeper layers is transmitted back to shallower layers, updating the parameters of every layer in the model.

[42], *decision trees* [43], *logistic regression* [41], and *random forest classifiers* [44]. These methods, when computing an f function that minimizes the cost C, rely on *manual* specification of a feature function Ψ for transforming arbitrary inputs $\mathbf{x}$ into a new representation $\Psi(\mathbf{x}) \in \mathbb{R}^m$ (see Fig. 1). Classical ML algorithms require manual work by the designer in order to construct or identify meaningful features that allow the algorithm to efficiently discriminate between the classes. For instance, Ψ could take as input an image and return another image where the magnitude of each pixel indicates whether an edge exists in that region. Alternatively, Ψ could be the identity function, in which case the learning algorithm operates not over a new feature-based representation of the input $\mathbf{x}$, but on the values of $\mathbf{x}$ directly. When this is done, the step of computing features is replaced by a step of *selecting* features—essentially, selecting a subset of the columns of $\mathbf{X}$ that the designer identifies as sufficient for the learning algorithm (see Fig. 1). For instance, when classifying whether a person is a male or female based on features corresponding to age, height, weight and eye color, the column corresponding to eye color may be removed by the feature selection process since it is not correlated with gender labels. Manually constructing Ψ and/or manually performing feature selection has two advantages: (1) it may be possible to manually design features that are informative enough so that it becomes easy to identify a clear discrimination rule to determine whether an input belongs to a given label; and (2) features that have an intuitive, application-domain related meaning allow for an effective way to *understand* the entire process of engineering a learning algorithm and analyzing its training process.

Deep learning: Deep learning (DL) techniques target the cases when assumption (1) above does not hold. In such cases, both the extraction of the features $\mathbf{X}^j$ and the construction of f are automatically performed by a Deep Neural Network (DNN). A neural network is one particular form of representing a prediction function f. It is a graph of connected *neurons* typically organized in $L > 2$ layers. The purpose of each layer is to further transform the input data given to the network, automatically building new and more abstract feature representations of it which allow for more efficient classification of that input. Each layer l is composed by a set of s_l neurons or units u_i^l (Fig. 2). Neurons in the first layer take as input all attributes $\Psi(\mathbf{x})$ of a given input data $\mathbf{x}$; neurons in subsequent layers ($l > 1$) take as input the output of all neurons of the previous layer. Based on these inputs, each neuron computes its output by linearly combining them with their set of learnable parameters and applying some non-linear function to the result, thereby producing an activation or output a_i^l that is passed as input to all neurons in the next layer. The output (or *activation* value) of a neuron in layer l is given as input to neurons in layer $l + 1$ and is associated with a parameter θ_{ji}; in particular, θ_{ji} is a parameter indicating the importance of the output or activation of the jth neuron in the previous layer l to the activation of the ith neuron in layer $l + 1$. More formally, each neuron j in layer l computes its output/activation a_j^l as $\phi^l(\sum_i a_i^{l-1}\theta_{ji})$, where θ_{ji} is a parameter indicating the importance of the activation of the jth neuron in the previous layer $l - 1$ to the activation of the ith neuron in layer l; and ϕ^l is a so-called *activation function* performed by the layer l, usually a non-linear transformation such as tanh, a sigmoid function, or a rectified linear function [1]. In case the function f modeled as a DNN outputs $\mathbf{y} \in \mathbb{R}^d$, the network's final layer L is composed of d neurons; when presented with some input $\mathbf{x}$, the network's label prediction is given by a vector $\mathbf{a}^L \in \mathbb{R}^d$ of activations of each of the d neurons in layer L.

In a DNN, besides layers of neurons as described above, it is also possible to create so-called convolutional layers, which are composed of neurons that essentially implement filters that compute features based on their inputs—e.g., the set of neurons of a particular convolutional layer could implement filters for performing edge detection when given an input image $\mathbf{x}$. The advantage of this in comparison to the approach taken by classical machine learning algorithm is that the filters/features computed by each layer are automatically learned by the algorithm, thereby eliminating the need for manually designing a feature function Ψ. When deploying a DNN, one needs to make a set of design choices: how many layers the network will contain; how many of those will be convolutional layers; which particular activation function ϕ to use in each layer; and how many neurons u_l will compose each layer l. These choices are often referred to as the *architecture* of the DNN and are encoded as a set of hyperparameters P. The choice of hyperparameters is typically manually decided by the DNN engineer based on earlier experience with similar problems. The parameters Θ of the network, on the other hand, are learned by a training algorithm in order to minimize some cost function C. Training algorithms of neural networks are usually based on performing gradient descent over the cost C with respect to parameters Θ on the final layer—i.e., by updating $\Theta \leftarrow \Theta - \alpha\nabla_{\Theta}C$—and using the *backpropagation algorithm* to propagate this gradient update to the previous layer, recursively modifying the parameters of each neuron according to the results computed in subsequent layers [1]. However, computing the gradient $\nabla_{\Theta}C$ requires a linear pass over the entire training set D, which in typical DNN applications may be very large. For this reason, one can alternatively use a so-called mini-batch training process, which splits the dataset D into B subsets of D—each one a smaller data batch compared to the entire set D—and then computes an estimate of the true gradient based on that reduced number of training examples. Updating the network by processing all B batches once is referred to as an *epoch*. This process is repeated for E epochs. Note that this training methodology introduces additional hyperparameters to the problem: the batch size B, a learning rate α, and number of training epochs E. These are determined by the DNN designer before training based on their earlier experiences or heuristics.

Deep neural networks may have several kinds of architectures, each with their own type of neurons performing a different set of operations. In this paper, we focus on three architectures that have been very popular in deep learning applications in recent years: *Deep Feedforward Networks* (DFNs); *Convolutional Neural Networks* (CNNs); and *Recurrent Neural Networks* (RNNs). For the purpose of this paper, it is important to differentiate between these three architectures because the distinctiveness of their neurons brings different challenges when performing the tasks we propose in our taxonomy. As it follows, we introduce in more details the definition of these three architectures.

Deep Feedforward Networks: The most traditional deep learning models are deep feedforward networks (DFNs), also called *multilayer perceptrons* (MLPs). As in other supervised learning approaches, the objective of a DFN is to approximate an unknown function f that can efficiently reproduce the relationship between inputs and outputs of a training set. What distinguishes DFNs from other machine learning techniques is that they are composed of multiple layers, each with multiple neurons. This hugely increases the number of learnable parameters the network has (parameters Θ, Fig. 2) and allows it to model functions that are much more complex than those encoded with only a few parameters. In a DFN, layers are fully connected, which means that the ith neuron in layer l, u_i^l, receives as input a vector $\mathbf{a}^{l-1}$ containing the activations of all neurons in the previous layer $l-1$. The final layer L can have a single neuron—thus, the network produces a single output, used e.g., for one-class classification—or multiple neurons, used for discriminative classification goals or multidimensional outputs.

Convolutional Neural Networks: In recent years, many applications in image classification and pattern recognition achieved state-of-the-art performance through the usage of Convolutional Neural Networks (CNNs) [45]. CNNs specialize standard DFNs as they use convolution operations in at least one of their layers. The objective of these operations is to find (small-scale) patterns in the input and send this information to subsequent layers codified via their output activations. The following layers, in turn, look for more complex patterns, thereby creating a chain of pattern detections that, when trained well, can achieve close-to-human performance in image-related applications. Convolutional layers are usually composed of three stages: multiple convolutional operations; a nonlinear function; and a pooling function that changes the current output value of a layer by aggregating some statistics computed on neighboring outputs [1]. However, the convolutional operation performed by CNNs also bring some challenges to their analysis. A single parameter in a convolutional unit acts over every region of the input domain, meaning that small modifications of the parameters of a convolutional unit may affect all the domain of the output activation. Additionally, convolutional units usually produce multidimensional activation outputs, differently from DFN units, that usually produce a single scalar as activations [10].

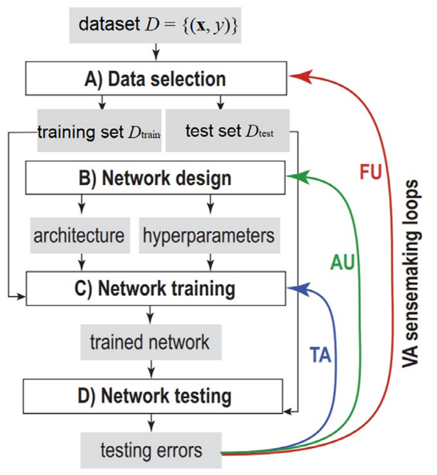


Fig. 3. Visual analytics workflow for DNN workflow engineering support showing the tasks of training analysis (TA), architecture understanding (AU), and feature understanding (FU). The workflow starts by collecting a dataset D with known labels and splitting it into training and test set (A). The second step is to design the network architecture (B), i.e., to decide how many layers the model should have, the type and the order of such layers plus the tuning of hyperparameters. Next, the model is trained by minimizing the prediction error on training set instances (C) and tested by comparing predictions for test set instances with their actual labels (D). The three tasks proposed on this survey aim to help designers in several steps of the workflow. Architecture understanding provide more information to design better model architectures, training analysis helps to explain what went wrong in the training and how to fix it, and finally, feature understanding provides experts with ways to interpret models and explain how input features were used in order to build the prediction assigned by the model.

Recurrent neural networks: Although DFNs and CNNs have achieved impressive results in classification and recognition tasks, they are not suitable for applications where inputs have a temporal or sequential relationship, such as word prediction or machine translation. Recurrent neural networks (RNNs), a different deep learning model, were proposed to handle this type of problem [46]. RNNs are intrinsically different from DFNs and CNNs because their neurons store a different kind of information called *hidden states*. Hidden states have internal values that are combined with the traditional learnable parameters Θ of neural networks when computing output activations. In contrast to the parameters Θ, which are frozen after training, the hidden states modify their values each time a new input is processed. This way, the same input instance can, and likely will, generate a different output if the previous inputs in the temporal sequence were different.

3. Deep learning engineering: workflow and tasks

The process of developing a deep learning model comprises multiple phases. A typical workflow proceeds as follows (Fig. 3). First, given a dataset D consisting of labeled samples $(\mathbf{x}, \mathbf{y})$, the designer splits D into two disjoint subsets: the training set D_{train} and the test set D_{test} (Fig. 3A). Such approach is necessary because ML algorithms—particularly those containing a very high number of parameters Θ, as is the case of DNNs—can easily overfit the training data—i.e., they learn to make good predictions for the training set but fail to generalize well to novel inputs, such as the ones in D_{test}. By testing the performance of the network on a set of completely new inputs, one can check whether the model can generalize its predictions to novel data samples that were not in the training set. This selection should be done carefully. Otherwise, even powerful learning algorithms may not perform as desired [1].

Selecting a good training set is not trivial, even if one takes care of issues such as class-label balancing. For instance, it is not clear how well the samples in D_{train} capture the variability of the entire data domain, i.e., how well D_{train} helps to learn all information needed to generalize the task at hand to unseen inputs. The second step in the development workflow is to design a DNN (Fig. 3B). As outlined in Section 2, this means choosing an architecture and suitable hyperparameters. Both operations are done largely based on similar designs from the past or heuristics. Yet, it is far from clear how suitable such design choices will be in practice for a given problem. The third step (Fig. 3C) consists of training the designed DNN, following the process detailed in Fig. 2. Finally, the performance of the trained model is measured on the test set D_{test} (Fig. 3D). This is done using aggregated error metrics such as accuracy, precision, recall, or area under the received operator characteristic curve (AUROC) [47,48]. Alternative schemes involve computing the *confusion matrix* [49] (for classification tasks), which calculates the number of correct predictions for each class and how many times a particular label was mistaken by each one of the alternative labels. This approach is useful for simple models with only a few class choices, but confusion matrices become hard to inspect and visualize for tens or thousands of classes.

If testing delivers satisfactory performance, the workflow in Fig. 3 can be finalized. When this is not the case, the key question is: what can the designer do to improve performance? This involves feedback loops at several workflow levels, each one involving a specific *task* as defined in the Contributions section. Additionally, even models performing successfully bring important questions. Often experts want to understand what kind of features the model learned to recognize or understand how the model operates over an input in order to predict its labels. These are additional tasks that we address in this paper. Note that the tasks that we propose to tackle follow the terminology proposed by Brehmer and Munzner [50], in which they define an arbitrary task as a set of both high-level and low-level, domain-specific (but also data-specific) activities that may be involved in answering the aforementioned question at a specific workflow level. The particular tasks that we consider in this paper are the following:

1. *Architecture Understanding* (AU): it is important to be able to analyze how the network architecture affects its performance to determine *how* a given model (which may be performing poorly) might be updated. To do so, one needs to understand how the network works so they can determine which aspects of the network to modify and when;
2. *Training Analysis* (TA): one needs to understand *why* training did not perform as expected; otherwise one does not know what to change next when trying to improve network performance. Based on insights from TA, one can modify the design of the network, e.g., change its number of layers, neurons per layer, activation functions, inter-layer connections, or hyperparameters;
3. *Feature Understanding* (FU): at the highest level, one needs to understand *which* aspects of the input data (samples and/or features) affect the quality of the learning process. By doing this, a designer may choose to, e.g., increase the number of convolutional layers in a DNN so that more powerful feature sets can be discovered. Additionally, before applying a deep learning solution in a practical application, it is desirable to understand exactly what the model is doing, i.e., which features in the input the model takes into account when deciding the output label and how the model operates on these features in order to calculate such result. Without this understanding, it is difficult to ensure the model is working as desired and users may hesitate in apply it in practice.

Note that unless suitable support is provided for the TA, AU, and FU tasks, designing an effective DNN is very much a ‘blind search’ process, which requires many costly iterations, either in an automated form (e.g., hyperparameter grid search) or done manually [10], by empirically choosing the model architecture and hyperparameter based on the developer expertise. These tasks can profit from visualization and visual analytics methods in several aspects [32,34]. For instance, visual tools prove to be an efficient way to understand which features a deep model has learned [51] and which particular neurons are responsible for computing those features [52]. In Section 4, we introduce visual analytics and how it can help deep learning engineering, while in Section 5, we further explain the visual analytics role in terms of the taxonomy we propose and how recent techniques have been tackling variations of the tasks above.

4. Visual analytics of deep learning networks

Visual analytics (VA) has emerged as an extension of information visualization (infovis) having as aim the analytical reasoning about problems described by large, complex, and abstract datasets using interactive visual interfaces [53–55]. While infovis (usually) aims at visually depicting a dataset with the aims of potentially gaining some insights, VA covers the more involved tasks of formulating, refining, and (in)validating hypotheses about the phenomena that lay behind the data at hand. As such, VA techniques and tools propose a so-called ‘sensemaking loop’ in which designers explore the data from multiple perspectives, posing increasingly more targeted questions [56]. Hence, the ability to interactively change visualizations and pose complex on-the-fly defined queries is key to VA. Important VA techniques target problems related to understanding high-dimensional datasets represented by features like $\mathbf{X}^j$ introduced in Section 2. VA has proven effective in many fields such as software maintenance, health science, e-government, homeland security, and social sciences [57,58].

In the last years, VA has increasingly focused on supporting machine learning applications [59]. VA integrates with ML and DL in terms of proposing specific types of sensemaking loops for specific DL tasks. Note that the deep learning tasks (AU, TA, and FU) are typically executed several times during the iteration of the sensemaking loop (Fig. 3), as typical in VA workflows. At each iteration, one obtains additional insights and either change the DNN settings to improve it, or digs deeper into querying the available data to make a decision. Separately, visual tools and techniques that support the three tasks are not disjoint. For instance, to understand training results (TA), one can use a network visualization (AU) that shows the roles of neurons in different layers in computing specific class labels.

Visualization of DNNs has caught the attention of the research community for many years. Pioneering work in this field dates back to the beginning of the century. Streeter et al. proposed a technique called NVIS [60], where an artificial neural network is represented as a matrix heatmap that encodes the parameters of all neurons u_i^l over all layers $1 \leq l \leq L$. Another early work, Tzeng and Ma [61] propose a node-link graph visualization to show a network’s architecture, coloring each neuron according to the strength of its activation a_i^l for a given selected input. While effective for depicting the structure and operation of small networks, such methods do not effectively scale to treat current-day DNNs that have millions of parameters and connections. Following these early developments, several new visualization techniques have been proposed to tackle open challenges in DL. By analysing such techniques and relating them to the workflow tasks described in Section 3, we built a taxonomy classifying existing VA approaches for DL in three tasks–AU, TA, FU—, explaining how they tackle such tasks and their respective goals.

We also discuss how those techniques tackle particular problems for the three more common deep learning architectures—DFNs, CNNs, RNNs. As follows, we present this taxonomy in Section 5

5. A Taxonomy on visual analytics for deep learning

In this section, we present a taxonomy on how VA techniques have been applied to support the proposed tasks (AU, TA, FU) for the three main DL architectures. We first discuss the methodology used to create this taxonomy, followed by sections where we detail the goals and sub-tasks on each task. We also describe how recent publications have been using VA to tackle them and how such techniques are applied to different network types.

5.1. Methodology

Deep learning visualization is a relatively new topic that has caught the attention of researchers from both machine learning and visual analytics communities over the past few years. Therefore, publications in this area are widely spread over proceedings and journals of different domains, imposing a need for a strong methodology when selecting papers for a survey like this one. To ensure that we were able to find all of the most relevant publications in the area, we searched for contributions in proceedings and journals of several areas, such as machine learning, visual analytics and computer vision. Particularly, we focused on well-regarded proceedings in the mentioned areas such as IEEE VAST, IEEE InfoVis, EuroVis, IEEE Transactions on Visualization and Computer Graphics, ICML, NIPS, ACM SIGKDD, ICCV, and CVPR. In particular, about a quarter of the papers presented at IEEE VAST 2017 focused on visual analytics for deep learning. As the interest in deep learning visualization is recent, we decided to focus on publications released from 2010 onwards, although we mention some earlier works that have historical importance in the field [51,60]. From all the set of publications retrieved on the mentioned proceedings, we filtered the ones mentioning, in their title or abstract, the deployment of visualization methods to understand or analyze neural network models or features. We also searched for papers with keywords such as *model* and *neural network visualization* on online platforms like *arXiv* and Google Scholar. Finally, we searched through the references of all the filtered publications to find other relevant papers.

From the set of collected papers, we identified how VA techniques have been applied to support the AU, TA, and FU tasks for three of the most popular deep learning architectures in the literature: *deep feedforward networks* (DFNs), *convolutional neural networks* (CNNs), and *recurrent neural networks* (RNNs). Table 1 provides an overview of the papers we surveyed and their respectively addressed tasks and network architectures. For completeness, we mention that other DL architectures exist, e.g., *Autoencoders* [62], *Generative Adversarial Networks* (GANs) [63], *Deep Belief Networks* (DBNs) [64], and *Deep Q-Networks* (DQNs) [65] architectures. For a recent overview of such architectures, we refer to Gibson and Patterson [66]. We did not include such architectures in our survey as we did not find enough papers proposing visualization techniques addressing their particular problems [67,68]. That said, it is worth to note that these architectures face many challenges similar to the ones addressed in this survey, making the techniques reviewed here also relevant for the analysis of them. It is also important to note that complex applications may require the use of networks combining elements of two or more type of models—e.g., a network containing both convolutional and recurrent layers. In such cases, VA tools must be adapted to tackle the needs of all the network's components.

Furthermore, we note that these tasks do not exhaustively cover the applications of VA in deep learning engineering. Visual analytics can (or could) be employed in other tasks, such as training data analysis, performance analysis, and model comparison. For *training data analysis*, visualizations can help identifying the lack of necessary features or bias in a training set. However, the literature still shows a lack of approaches addressing this problem, which makes it an interesting topic for future research. For *performance analysis*, VA can provide powerful tools to analyze the performance of deep models by providing ways to compare the confusion between multiple classes [69] or to depict the distribution of an output value over the set of input features [70]. For *model comparison*, VA can answer why a model performs better than another [71]. This also brings difficult challenges, mainly because it is hard to compare models that do not share the same structure (for instance, number of layers and neurons per layer) since such models end up having completely distinct parameters, which make them recognize features in different ways. However, comparing networks with similar structure but different hyperparameters [71–73] can be an effective way to understand how the hyperparameters affect the final performance of the model.

5.2. Architecture understanding

As explained in Section 2, modern DNNs may have hundreds of layers and hundreds of thousands of neurons. When using such wide and deep models, it is easy to lose track of all the aspects of their architectures or which computations they do at each unit of the model. Thus, visual analytics can play a key role in helping designers have a better insight into the characteristics and behavior of their models during the development pipeline. The main goal of visualizing information related to the architecture of a DNN is to give a good understanding of both high and low-level aspects of the model [75]. At a very high level, performing the ***architecture visualization***, i.e., showing the network topology (as a graph) quickly tells designers the overall structure of the network, e.g., how many layers L it has, how their sizes s_i vary, and which kind of operation they perform. This helps to understand a DNN much in the same way that architectural diagrams reverse-engineered from source code help software maintenance [103].

At finer levels, visualizing the connections between neurons on consecutive layers—encoded as the parameters Θ_l of a given layer l—may help understanding how simple features get merged into more abstract ones in the classification process [10]. Additionally, one can visualize the combination of DNN structure-and-data, by annotating the DNN graph with parameter vectors, activations, and training statistics. This helps in understanding how structure correlates with behavior [10,71]. Showing this helps to find possible inefficiencies of the model, such as neurons who are activating for too many classes and thus are not relevantly contributing to the final prediction [80], inert units, or redundant components that recognize the same features [52]. In other words, such techniques aim to provide an ***architecture validation*** of the model.

Another way to understand the impact of architecture choices such as hyperparameter tuning is to perform ***model comparison*** of two or more networks. Achieving a clear insight on how hyperparameters affect statistical models is not trivial and the VA community has devoted a whole field of study, called *Visual Parameter Space Analysis* (VPSA), for this topic [73]. However, this analysis is particularly difficult for deep neural networks, as they have an insane number of parameters that significantly grows with each added layer. Researchers in the VA community have been tackling this problem by comparing similar models with different architectural choices, in order to achieve better insight in what differences in the final performance these choices had [72].

Table 1
Taxonomy of VA-related publications related to different DNN architectures and engineering tasks. Tasks are further refined as follows: Architecture Understanding: architecture visualization (AVis), architecture validation (AVal) and model comparison (MC); Training Analysis: real-time analysis (RTA) and evolution of model metrics (EMM); and finally Feature Understanding: model interpretability (MI), feature explainability (FE) and performance validation (PV).

Technique	Taxonomy					
	Tasks			Networks		
	Architecture understanding	Training analysis	Feature understanding	DFN	CNN	RNN
Zeiler and Fergus, 2014 [6]			FE		•	
CNNVis, 2017 [10]	AVis and AVal		FE	•	•	
Samek et al., 2017 [12]			FE		•	•
Montavon et al., 2018 [28]			FE		•	•
FeatureVis, 2016 [33]			FE		•	
Erhan et al., 2009 [51]			MI		•	
Rauber et al., 2017 [52]		EMM	MI	•		
Zahavy et al., 2016 [67]			MI	•	•	
DGMTracker, 2018 [68]	AVis and AVal	EMM	FE	•	•	•
RNNVis, 2017 [71]			MI and FE			•
CNNComparator, 2017 [72]	AVal and MC			•	•	•
Activis, 2018 [74]	AVis		FE	•	•	
TensorFlow GraphVis., 2018 [75]	AVis			•	•	
TensorFlow Playground, 2017 [76]	AVis and AVal	RTA	MI	•		
ReVACNN, 2016 [77]	AVis	RTA	FE	•	•	
Harley, 2015 [78]	AVis		FE	•	•	
BIDViz, 2017 [79]		RTA and EMM		•	•	•
DeepEyes, 2018 [80]	AVal		MI and FE	•	•	
Deep View, 2017 [81]	AVal	EMM	FE	•	•	
Grad-CAM, 2016 [82]			FE		•	
RNNbow, 2017 [83]		EMM				•
Yosinski et al., 2015 [84]			MI and FE		•	
Alsallakh et al., 2018 [85]			MI	•	•	•
Nguyen et al., 2016–1 [86]			MI		•	
Nguyen et al., 2016–2 [87]			MI		•	
Aubry and Russell, 2015 [88]			MI		•	
Simonyan et al., 2013 [89]			MI and FE		•	
Wei et al., 2015 [90]			MI and FE		•	
Mahendran and Vedaldi, 2015 [91]			FE		•	
Mahendran and Vedaldi, 2016 [92]			FE		•	
Zintgraf et al., 2016 [93]			FE		•	
Dosovitskiy and Brox, 2016 [94]			FE		•	
Zintgraf et al., 2017 [95]			FE		•	
Heyi Li et al., 2017 [96]			FE		•	
VisualBackProp, 2016 [97]			FE		•	
LSTMVis, 2018 [98]			MI and FE			•
Jiwei Li et al., 2015 [99]			MI and FE			•
LAMVI, 2016 [100]			FE			•
Ding et al., 2017 [101]			FE			•
Karpathy et al., 2015 [102]			FE			•

5.2.1. *Architecture visualization*

Deep learning architectures are essentially directed acyclic graphs (DAGs) where nodes represent neurons and edges represent connections between subsequent layers [10]. For this reason, *graph visualization* has been a straightforward way to visualize the architecture of deep models [10,74–77]. Such visualizations help deep learning engineers in multiple ways. First, they provide an overview of the operations that the model is performing when an input flows through the network [74]. As a refinement to this, showing what the network does in different layers and parts thereof helps the engineer understand whether the chosen architecture is appropriate and, if not, where it should be adapted or modified [75].

For such a graph visualization to be effective, it is not enough to depict only the connections between neurons, as this information is typically already known by the architect and is the same in all layers (i.e., drawing the individual neuron connections with no associated value does not bring additional insights). Graph visualizations become effective when they show additional data on the *activity* of the neurons. One way to do this is to show the neuron parameters, e.g., by color coding. However, parameters are hard to interpret, particularly in deeper layers. A more insightful design is to show neuron activations for specific inputs [10]. For this reason, many existing works show the activation vectors $\mathbf{a}^l$ produced for one or more inputs via color-coded matrices or vectors [10,68].

As DNNs become deeper and wider to tackle more complex applications, scalability becomes an important issue for visualizing the network's graph structure. One solution for this is to visually cluster neurons having similar activations [10] and next use *edge bundling* to visually group connections linking neurons in the same clusters [10,75]. By clustering groups of neurons with similar activations, architecture visualization can be made clearer. Also, this approach can highlight large groups of neurons that may be involved in correctly predicting a particular label, as well as to highlight classes that are not being sufficiently learned by the model (e.g., few neurons respond to inputs of that class) [10]. Edge bundling has proven very effective to trade clutter for overdraw when creating simplified views of graphs of millions of edges [104,105] and hence it has the required scalability for handling very large DNNs. Another approach proposed to improve the scalability of graph visualizations is the omission of non-critical operations (e.g., *pooling* layers, which implement a type of pre-processing on the outputs of a given layer using fixed operations that are not updated by the training process, and thus sometimes

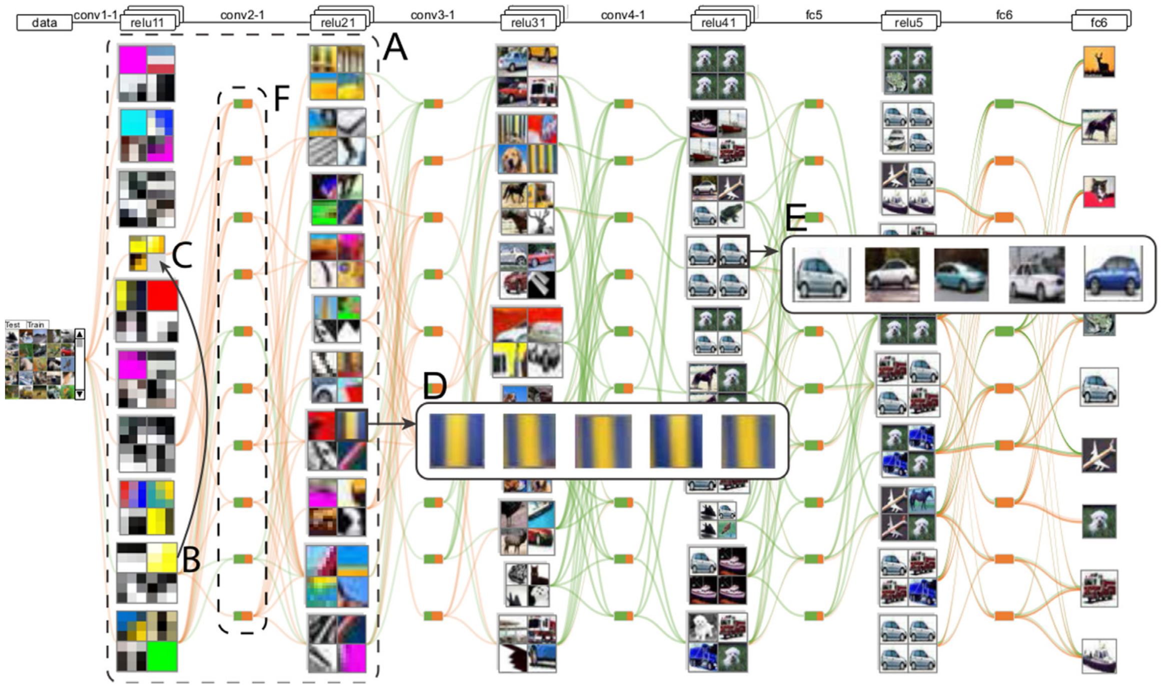


Fig. 4. CNNVis Tool [10]: the structure of the network is shown as a directed acyclic graph (DAG), where adjacent layers are grouped (A) and neurons within a layer are clustered by the activation vectors they produce (B), allowing the identification of groups of neurons that learned to recognize similar features (E).

may be omitted in a visual analysis). Finally, to improve scalability, it is also possible to highlight particular network regions with similar properties—e.g., parts of a DNN with similar parameters and activations [75]. Graph visualizations are strongly aided by interactivity which can help designers focus on and explore in more detail particular parts of the model they deem more interesting [75].

5.2.2. Architecture validation

Architecture validation is a sub-task of AU where the focus is to validate if the chosen architecture (i.e., number, order, type, and size of layers) is the correct one. This sub-task is tightly correlated to the previous one, as visualizing the network graph with the display of information such as activation output for each neuron can be effective in helping to find underperforming components in the model [10]. Visualizing the graph structure is not the only way to analyze the architecture of a DNN though. Other kinds of visualizations can also be helpful in giving insights if the chosen architecture is the right one, without explicitly visualizing the model's graph workflow. For instance, several authors used heatmaps to identify layers and neurons that are not being used for the model, either because they are not producing significant high activations for any kind of input—and thus can be dropped from the architecture—or because they are activating too often for too different inputs—what may indicate the need for more units [10,68,74,80].

5.2.3. Model comparison

Additionally, another sub-task worth mentioning is *model comparison*. In many contexts, it would be useful for experts to compare different models to understand, for instance, why one performs better than the other in a given application or dataset. This is particularly useful when comparing two models with same architecture but different hyperparameters—thus getting more insight in the role the hyperparameter played in the learning process—or in two models with same architecture and hyperparameters but trained until different epochs—enabling to analyze how much the model learned between both epochs. In CNNComparator [72], the authors do that by comparing the difference in learnable parameters after training using heatmaps and histograms displaying the difference between learned parameters.

5.2.4. Architecture understanding on different models

Deep feedforward networks: DFNs are composed only by fully-connected layers where each neuron in a particular layer l receives as input the output activation of all neurons in the previous layer $l-1$ and the output activation of a neuron is a scalar value. Most VA techniques aiming to analyze the architecture of DFNs use graph visualization to display the architecture structure combined with heatmaps and color encodings to represent associated activation values, either on the edges [76,77] or on the nodes [68,78]. In most heatmaps, rows represent different input instances or classes while columns represent the many neurons in a layer [10,74]. This way, the analyst can have a good understanding of how the architecture is working, i.e., how it is performing the class prediction when an input is processed.

Convolutional Neural Networks: Just as DFNs, CNNs can also be seen as directed acyclic graphs. However, CNNs have what it is called *convolutional layers*. In such layers, neurons perform convolutional operations in the input data received from the previous layer. This brings some differences for the analysis of the architecture of a CNN if compared to DFNs. First of all, the analysis must take into account that a single parameter in a convolutional neuron is applied to not only one but several input values. For instance, if an input image is sent to a convolutional layer, a given parameter in the layer will operate all over the image values, and not in just a single pixel as it happens in DFNs. Additionally, visualizing the activation of CNNs layers is more challenging as each unit produces a multidimensional output activation, and not a single value as DFNs do—e.g., each unit in the convolutional layer will produce an *activation map* with similar dimensions as the input, while DFN units produce a single unidimensional value. Such particularities must be taken into account when analyzing the architecture of CNN models. This is particularly true when visualizing activation heatmaps. As now neuron activations are not scalar values but multidimensional vectors, such heatmaps

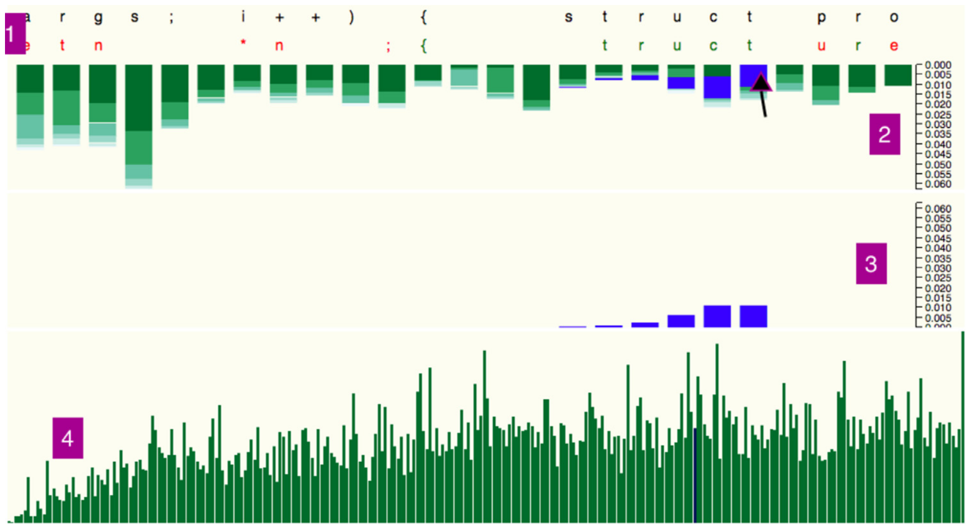


Fig. 5. RNNbow Tool [83] uses stacked bar charts to visualize how the gradient loss progresses through the hidden states of a recurrent neural network. On the top (1), the designer can compare predicted labels with actual ones. The top bars measure the gradient magnitude used to update parameters at each timestep (2). The bars are decomposed according to the source of each fraction of the whole gradient magnitude (3). On the bottom, the designer can interactively choose which training batch should be displayed in the visualization (4).

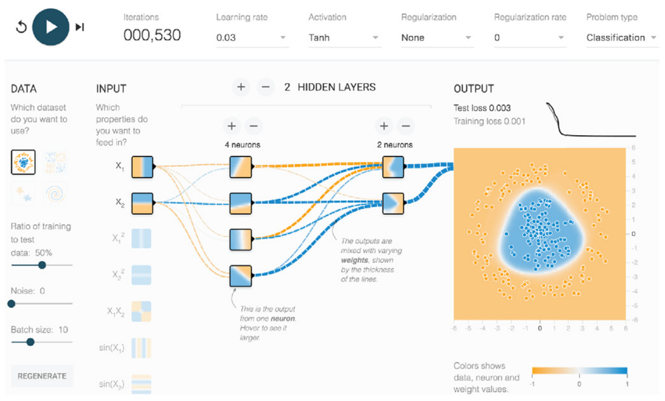
are not so straightforward to be produced. Nonetheless, graph visualization still is a natural way to visualize CNNs [10,68,77,78]. A typical approach when visualising convolutional layers is to use the nodes of the graph to display either the convolutional filter of the unit (i.e., the unit parameters in the exact order they are applied on the input [72], the activation map produced by a particular input [68,77,78], or inputs that produce strong activations on that neuron [10]. Fig. 4 shows an example of CNN architecture visualization [10]. Here, neurons with similar activations are clustered together, and only the features that produce the strongest activations on them are displayed, to limit visual clutter.

Recurrent Neural Networks: The visualization of the structure of a RNN is a difficult and, to the best of our knowledge, still unexplored topic. One of the issues to overcome when building architectural visualizations for RNNs is the large number of possible output formats they can have. In some applications, such as sentiment analysis, the RNN processes the whole sequence of inputs and then returns a single output [106]. However, in other applications such as machine translation, the network must output a new value at each element of the input stream that is processed [107]. Other variations, usually used for sequence prediction tasks, aim to predict an output equal to the following elements of the input stream every time an input unit is processed [1]. Additionally, RNNs have a recursive structure in which input elements are processed in a sequential manner. When one element is processed, the *hidden state* of the unit is modified and this can, and probably will modify the behavior of the unit for future input elements. When visualizing the architecture of RNN models, analysts should be aware of this particularities, as understanding the how the hidden state is being affected by input and how it is affecting the outputs is as important as understanding the structure of parameters applied on inputs and output activations. All in all, visualizations supporting architecture understanding for RNNs are weakly developed and, given the complexity of these architectures, we consider this a promising future research topic.

5.3. Training analysis

The training of a deep model is a hidden process that gives little to no insight to the designer about what is happening. If a model is not performing well, it is very hard for experts to identify what should be changed in the hyperparameters or even whether there is a problem in the training process at all. As understanding the training process is still an open challenge in deep learning, visual analytics can play a powerful role in addressing it. By visualizing the *evolution of model metrics* during the training process, VA techniques can help to understand how the model achieved some performance and to identify undesirable behaviors. For instance, visualizing metrics about gradient values can help to understand how the updating process is changing the parameters of neurons and hidden units [80,83], allowing the designer to find out network parts that are not stabilizing or that are not being sufficiently changed by backpropagation. Additionally, analyzing how the neurons' parameters and activations change through the training epochs are key to comprehend how the DNN evolved and how it learned to recognize the relevant features for the relevant task at hand [52]. Training metrics are not restricted to gradient and activation evolution, though. Recent works have shown that several user-defined metrics can be effective in giving insight about what and how the network is learning [79,81]. Visual analytics also uncovers new possibilities for training DNNs, as it can help designers to analyze the training process in real-time, allowing the designer to make assumptions about the model, and take corrective decisions, without having to wait for the entire training to finish [10,79].

As with most machine learning techniques, neural networks are trained via gradient-based methods, such as gradient descent, that minimize a cost function C (see Sec. 2). Since DFNs have multiple layers, the traditional gradient-based method used (as described in earlier sections) is the backpropagation algorithm, which updates all parameters in the network, starting from the ones in final layer and moving towards shallower layers of the network, in order to minimize the prediction error for inputs in the training set [1]. Two common problems that can occur during backpropagation process are the vanishing gradient and the exploding gradient [108] problem. In the former, the gradient becomes insignificantly small very quickly, making the training process unable to relevantly change the parameters of layers far away from the output layer. In the latter case, the gradient keeps an exaggeratedly large value for many layers, changing the parameters Θ so drastically that the model never stabilizes. To understand this type of phenomena related to the gradient flow, Cashman et al. propose a stacked bar chart visualization (Fig. 5). Each stacked bar represents the magnitude of the gradient that produced the model's parame-


Iterations 000,530
Learning rate 0.03
Activation Tanh
Regularization None
Regularization rate 0
Problem type Classification

Ratio of training to test data: 50%
Noise: 0
Batch size: 10

Test loss 0.003
Training loss 0.001

DATA
Which dataset do you want to use?
REGENERATE
INPUT
Which properties do you want to feed in?
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
2 HIDDEN LAYERS
4 neurons
2 neurons
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Colors shows data, neuron and weight values.
-1
0
1

Fig. 6. TensorFlow Playground tool [76] showing jointly the structure and activations of a simple DFN, and allowing interactive control and monitoring of the training.

ter update at a single time step. Each partition of a bar represents how far in time each part of the gradient came from. Using this visualization, the authors were able to effectively identify cases of vanishing gradients, where large gradient partitions in one stack bar quickly become very narrow partitions in the next time steps.

5.3.1. *Visualization of model metric evolution*

In this sub-task, the expert analyses how a particular metric evolves over the training of the network. Such metrics are user-defined and can contain information helping to understand, for instance, if the gradient updating is being performed as expected [83], if the model is improving its performance [81] or if particular units or layers are indeed converging to some learning [80]. The most traditional metric to visualize when analyzing the training process of ML models is to the accuracy of the model predictions for the training or test set through the training epochs. While this gives a good intuition on how accuracy changes with training, it does not provide insights on how to improve it apart from more training. For instance, information about how the parameters are being updated on individual layers or neurons along the training is not provided. Qi et al. [79] alleviate this by allowing designers to define and plot their own metrics in real-time, allowing a more effective guiding of the training process. As the learning is not uniform in all layers, developers may want to identify layers or neurons that are not being well trained. To support this, Pezzotti et al. [80] propose so-called perplexity histograms, a visual technique to find stable layers, *i.e.,* the ones that already stopped receiving relevant parameter updates. By visualizing the progress of individual layers during training, this helps to identify if a given layer's parameters converge to a good solution or if the layer is not learning to recognize any useful patterns in the input data. An alternative approach, proposed by Zhong et al. [81], uses heatmaps to depict how neurons parameters and activations change over the training process. For this, they propose two metrics: discriminability—measuring how different is the average activation produced in a layer by elements of a particular class from elements of any other class—; and density— which evaluates the quantity of higher activations a particular neuron produces.

5.3.2. *Real-time analysis*

Training of complex deep learning models can take up to days or weeks to be completed. For this reason, designers can not always afford to wait until it is done to analyze the behavior and features learned of the model. The ability to perform real-time analysis while training deep models is essential and VA can play a key role in providing it to experts. Indeed, many of the surveyed VA solutions have as a goal to provide model information in real-time, even in an interactive way [77]. For instance, Qi et al. [79] propose a system where model designers can codify and plot model information in real-time and thus make modifications in the model as needed. TensorFlow playground [76] (Fig. 6), a very popular online solution aimed to teach DL concepts, uses a real-time and interactive visualization to let designers analyze how the many components of the network learn to divide the input space in a way the model performance is optimized. Is important to note that real-time analysis can be related to any information regarding the model—e.g., architecture, performance metrics, learned features. Thus, basically, any other task or sub-task could be combined with a real-time analysis in order to make the engineering pipeline of DL models faster and most effective.

5.3.3. *Training analysis on different models*

Deep feedforward networks: Most methods to visualize the training process of DFNs focus on displaying how a particular metric evolves over time by a given layer or neuron. Focusing on a single element of the model rather than in the whole network is effective because then the designer can get more information about where the model is underperforming and how to modify it. One recent approach aiming at this is proposed by Rauber et al. [52]. For each sample $\mathbf{x}_i \in D_{\text{train}}$ in the training set, they consider the activation vector $\mathbf{a}(\mathbf{x}_i)^{L-1}$ of the last hidden layer. Next, these vectors are projected from $\mathbb{R}^{s_{L-1}}$ (where s_{L-1} is the number of neurons in layer $L-1$) to $\mathbb{R}^2$ using standard dimensionality reduction (DR) methods such as t-SNE [109], yielding a 2D scatterplot of points $\mathbf{p}_i$, one per sample $\mathbf{x}_i$. Key to DR methods is their ability to place points with similar high-dimensional vectors close to each other in 2D, and points with dissimilar vectors far apart in 2D, respectively—thus showing how similar are the high-dimensional vectors. This process is repeated for all training epochs, yielding a 2D trajectory of points per sample $\mathbf{x}_i$. These trajectories are then colored according to the classes y_i of the corresponding training samples, and bundled to yield a simplified, though suggestive, view of how the neurons of the last hidden layer get increasingly more class-specialized (farther apart in the 2D projection) as training progresses. Fig. 7 (middle) shows an example hereof. The same type of visualization can be used to show how the network layers learn to discriminate between the different classes (Fig. 7 (right)). Here, each trail represents the projection of activations of all hidden layers $2, \ldots, L-1$ of a test sample $\mathbf{x}_i$, after training. As in the previous image, one can see how same-class images yield increasingly more similar activations as the data flows deeper through the network.

Convolutional neural networks: The training process of a CNN is typically done via backpropagation, in a very similar way to the training of DFNs. As such, most of the visual approaches used to analyze the training of DFNs can also be used on CNNs. However, the differences between convolutional neurons and fully-connected ones have an impact on the training process analysis. For instance, convolutional neurons usually output high-dimensional activations, which forbids temporal visualizations displaying how the activation of multiple inputs evolve over time [81]. Visualizing high-dimensional datasets that evolve through time is a topic that has been intensively researched by the VA community and certainly can bring improvements to training process analysis [110]. Liu et al. [68] propose a specific visualization for CNN training—particularly for generative models—that displays how features are learned through the training process by plotting line charts showing various designer-selected statistics of interest (activations, gradients or parameter updates) over time. If the designer spots a layer with an interesting or abnormal behavior—such as an abrupt change of many activations in a single epoch—they can explore the activations of that layer's neurons in more details and visualize the subsets of input data that lead to such activations.

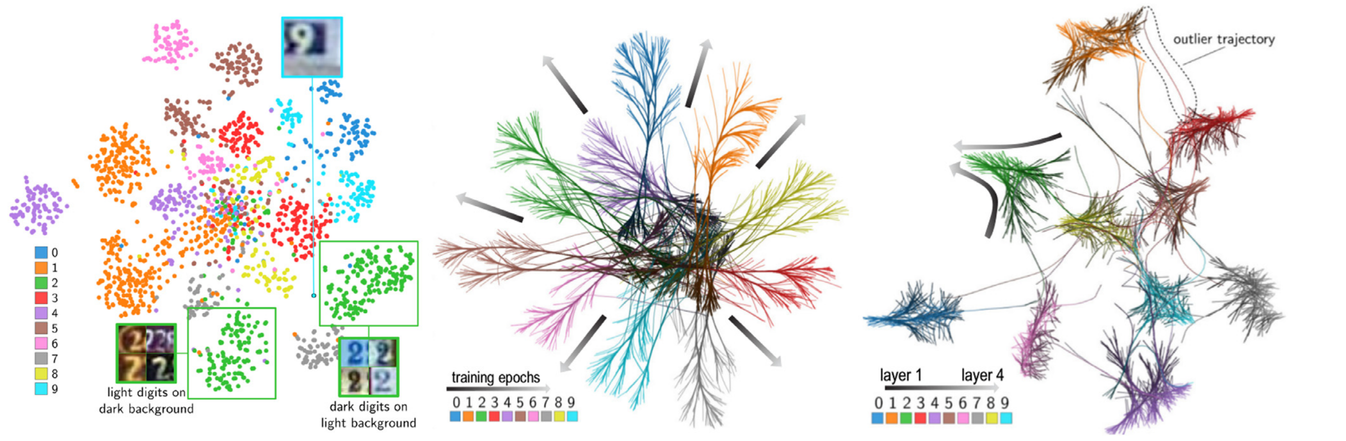


Fig. 7. By projecting activation vectors onto a bidimensional space, Rauber et al. [52] are able to explore the learned features (left), how they evolve over the training epochs (middle), and how they are identified by different layers (right).

Recurrent neural networks: The training process of RNNs is particularly delicate if compared to the two previous architectures. Although such networks are also trained via backpropagation, their recursive structure makes them much more prone to suffer from the vanishing gradient problem. This happens because, in the backpropagation algorithm, parameters are updated every time a new input element is processed by the network. However, RNNs must learn to recognize patterns in long sequences of inputs, and the error propagated by the backpropagation algorithm might not be strong enough to update the parameters in a proper manner. Cashman et al. [83] tackled this problem with a stacked bar chart visualization aiming at understanding the so-called gradient flow during the training process. Specifically, they address the problem of how gradient values change during the backpropagation phase of training. Additionally, understanding how these parameters change throughout the process helps by providing insights to the designer regarding how new training examples modify the learned model; this, in turn, may help in determining how to improve the model hyperparameters, if necessary.

5.4. Feature understanding

Although a neural network may be performing well on unseen data, it is not always clear when and why this happens. Hence, the machine learning community has put much effort in approaches to figure out which features the input data must have in order to produce the desired output [6,111] and how the model uses these features to compute its label prediction [10,52]. However, learned features in DNNs are only *implicitly* described in terms of the huge number of parameters Θ of the model, in contrast to the *explicit*, hand-engineered, features used by classical ML techniques (see Section 2). As such, VA aims to explain in an interpretable and intuitive way how the information spread over all neurons in all layers of a DNN captures these features [71]. One sub-task here is to *interpret the model*, i.e., to show how features learned in earlier (closer to input) layers get merged in subsequent layers to identify more complex patterns [10], how intermediate layers transform the input they receive [52] and what features each component learn to recognize [84]. Another goal of the FU task is to *explain learned features*, i.e., to identify in a particular input or set of inputs, which of their features were taken into account in order to decide the output label. By understanding what the network is recognizing, experts can give more reliability to their models, as they can know what they are recognizing or predicting with more certainty [52]. This also allows identifying patterns and features the model has not learned to recognize, but which a human may consider important [80].

Feature understanding techniques can be classified into *instance-based* and *feature-based* visualizations. The former ones depict the behavior of the model for specific input instances, be it a single one or a subset of many. The main goal of such visualizations is to find which features of the input produce high activations in the network and in which neurons or layers that occur [74]. Feature-based techniques, on the other hand, aim to explain which features an input must have to produce a particular output in the final layer or, more generally, in any layer [52]. Such techniques are well suited when instances are not easily interpretable or in applications with many possible outputs, where analyzing individual inputs can become tedious [74].

In contrast to feature engineering methods, DL methods do not have an explicit representation of the data features they use (Section 2). The representation of such features is actually 'scattered' in the parameter parameters Θ of all neurons over all layers. Unfortunately, because DNNs are a composition of nonlinear functions, it is difficult to retrieve any interpretable information from the parameters values, particularly the ones in deeper layers. An alternative way to understand the learned features is to analyze the *activations* produced by these parameters when a given input flows through the network. This can be done at several levels, as follows. By visualizing the activations vectors produced by a *single* input over the *entire* model, using e.g., heatmaps or matrix plots (an instance-based approach), one can understand how the sample information flows through the network until we obtain an output [74]. Alternatively, by visualizing the activation vectors of multiple input samples for a single layer or even single neuron (feature-based approach), one can find which patterns the layer, or neuron, is learning to recognize [52]. Another feature-based technique is to visualize how the network divides the input data space X at each neuron [76] (see also Fig. 6). This approach is very intuitive to use, as it effectively shows how each neuron u_i classifies every possible sample $\mathbf{x} \in X$, by color-coding a 2D plot of X with the respective activations $a_i(\mathbf{x})$, and also allows real-time changes of the architecture and hyperparameters and training monitoring. However, it only works for data spaces $X \subset \mathbb{R}^2$ and relatively small networks (a couple of layers).

As well as for the TA and AU tasks, scalability is also a problem for FU for more complex models, especially when visualizing the activations of many neurons at once [10]. Additionally, some applications may have inputs composed of different data formats. For instance, to classify a social media post, the model could have as input an image, a text, and information on the user who posted

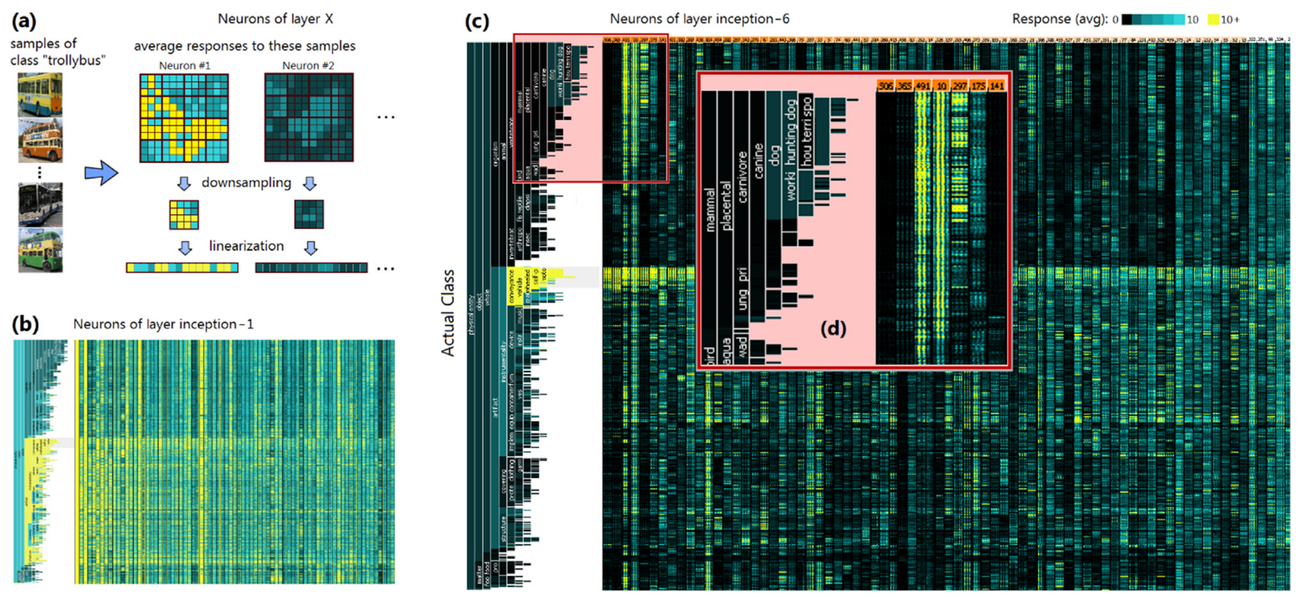


Fig. 8. In their work, Alsallakh et al. [85] use heatmap matrices to visualize how neurons in a user-selected layer respond to inputs of different classes. In their heatmaps (b and c), each row represents one of the possible classes a training example may belong to, and each column represents one of the neurons in that layer. Thus, each cell of the heatmap contains information about the activation produced by a specific neuron given an example from a particular class as input. As convolutional neurons output high-dimensional activations, this information is encoded in a small array computed as described in (a). First, they average the neuron's response for several samples from that class, and then they downsample and linearize the resulting matrix, returning an array with the class activation information condensed in just a few values, if compared to the original matrix. With this technique, they identify that activations of deeper layers build a more clearly defined class hierarchy, with particular groups—e.g., mammals (d)—while shallower layers can identify only more generic groups of classes (b).

it [74]; in such cases, visualizing essential features in a meaningful way is harder than when *X* consists of a single data type.

5.4.1. Model interpretability

One of the main goals of visual analytics in deep learning is to allow experts to interpret deep models. Unfortunately, model interpretability is an ill-posed problem, as there is no universally accepted definition of what exactly means to interpret a model [39]. However, many authors have considered interpreting a model as the ability to understand and explain how the model builds its label decision given a particular input and what is the role of each component of the model on the decision process. One way to attack this problem is to visualize the activation space of each layer or neuron [76] in order to inspect how the decision process work at each component in the model. Unfortunately, such spaces are often high-dimensional, what forbids this approach to be used in complex models.

An alternative approach is to use heatmaps to display how a component behaves for different kinds of input [10,74,80]. Visualizing activation heatmap matrices is one of the most widespread approaches for model interpretability. Here, activations are displayed as a matrix where rows are input samples or sample classes and columns are the neurons or layers of the DNN [74,80]. The colors of the matrix cells encode the activation produced by each input or class (row) to each neuron or layer (column) of the model. When visualizing activations for particular inputs, heatmap matrices work as *instance-based* techniques that show which parts of the neural model learn features for that input [80]. Conversely, when displaying activations for a single class or subset of inputs—done usually by computing the average activation of all considered input samples—heatmap matrices behave as *feature-based* techniques, showing which network parts specialize in recognizing that class or subset [74]. However, one should note that such visualizations can be misleading when inputs belonging to the same class present distinct input features—for instance, in an image classification task, a class *building* may be composed by images of very different buildings (e.g., wood houses and shopping centers), yet they have the same label. Even in a network with high performance on this task, neuron activations in hidden layers may be significantly different for samples of this type (qualitatively different input values but same associate label) [87], and therefore different activation patterns should not necessarily be interpreted as evidence that the network did not successfully learn appropriate intermediate features. An example of such a heatmap matrix produced by Alsallakh et al. [85] is shown in Fig. 8.

Heatmap matrices often do not scale well to datasets with a very high dimensionality—which is often the case in deep neural networks—, as spotting interesting patterns becomes difficult when the number of rows and columns is too big. To overcome this issue, many authors have used dimensionality reduction techniques to visualize the activation space in a bidimensional projection. As outlined in Sec. 2, the activations $\mathbf{a}^l(\mathbf{x})$ produced by a hidden layer l of a DNN for an input sample $\mathbf{x}$ form a high-dimensional vector that captures the different features of $\mathbf{x}$. Comparing such vectors for all samples $\mathbf{x}$ in a training or test set allows one to visualize how the network succeeds (or not) in discriminating between these. To do this, dimensionality reduction methods can be used to create 2D scatterplots of these samples. Close points in these scatterplots indicate samples which are found to be similar by the network [52,67,74,80]. In particular, by projecting activations from multiple inputs, designers can identify instances wrongly predicted and build hypotheses of why this happened by comparing them with inputs with similar activations [52]. Fig. 7 (left) shows such a projection for the well-known SVHN image dataset [112], with points colored by the class label, based on the last hidden layer activations after training [52]. This image allows a designer to quickly observe that data points belonging to each class are divided into two compact clusters—one which represents light digits on a dark background, and one which represents dark digits on a light background. Hence, this visualization lets one see that the network has learned *irrelevant* information of digit-*vs*-background contrast, which is not useful when classifying the digit images—and this is a finding that can help to fine-tune the network to learn more effectively [52]. Additionally, by visualizing how this activation space evolves through the training process and over the layers—Fig. 7 (center and right, respectively)—, they can show how the model gradually learns to separate different classes. The main downside of dimensionality reduction techniques is the intrinsic information loss related to the compression of a high dimensional data to a bidimensional representation. Also, different dimensionality reduction methods—or the same one but with different parameter tuning—can give different and equally useful projections, as datasets may have multiple features that cannot be uncovered in a single projection. To overcome this limitation, several authors have proposed interactive dimensionality reduction approaches [113]. However, given the scalability of those methods and the intrinsic high-dimensionality of activation vectors, dimensionality reduction techniques are an effective way to achieve model interpretability on deep models.

5.4.2. Feature explainability

While model interpretability aims to understand the whole decision process of a model, feature explainability aims to answer, given a particular input, which are the features present in it that were relevant for the prediction decision. Several techniques such as code inversion [91], layer-wise relevance propagation [114], and deconvolutional networks [94] were developed to achieve such explainability. These techniques evaluate which attributes (e.g., pixels or words) have information that is considered relevant by a given neuron in order to produce a particular activation value. Attributes that make neurons output high activation values are often more important in the decision process the model learned to perform, particularly in deeper layers. More details about these techniques will be given later in this section.

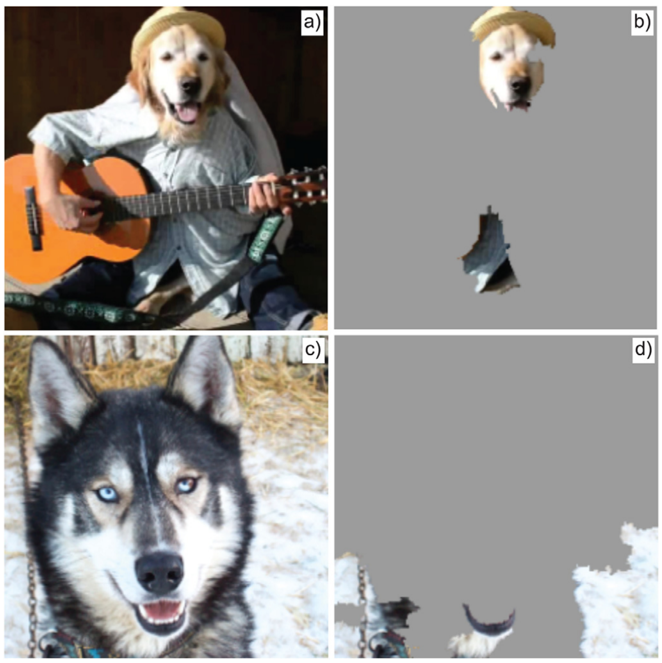


Fig. 9. Image-based visualizations explaining the input elements that lead to a classification outcome [13]. The example on top (a) shows an image which the model is effectively recognizing relevant features to assigned the label *dog* (b), even though the input present features that are not typical in dog images. The bottom image (c), however, is not being correctly learned by the model, as the model is using background features to perform classification and ignoring pixels corresponding to the actual dog (d).

5.4.3. Feature understanding on different models

Deep feedforward networks: In DFN models, we can interpret the fully-connected layers as operations performing non-linear transformations in the multidimensional space the training data lies. For this reason, techniques aiming to visualize high-dimensional spaces such as dimensionality reduction are often good alternatives to analyze DFNs [52]. However, even though such an approach helps to interpret the function of the layer as a whole, it brings little insight about individual units. Techniques that show how neurons respond to different kind of inputs, such as heatmaps [10] are more desirable in such cases, as they give a clear understanding of the role that neuron is playing in the recognition task.

Convolutional neural networks: Visualizing activations is as important to understand CNNs as it is to understand traditional DFNs. Particularly, heatmap matrices have stood out as an effective tool for this task [10,78,84,85], as they can compactly display the activations of neurons for multiple input samples [10] or in multiple convolutional filters [84]. By using a heatmap matrix to compare activation vectors for different inputs, Alsallakh et al. [85] (see Fig. 8) showed that this type of analysis can lead to non-trivial conclusions—such as the identification of a hierarchical structure in the classes present in a training set, given that classes that share similar features usually have similar activations in shallower layers and more distinct activations in deeper ones. Zeng et al. [72] also use heatmap matrices to visualize the differences in the parameters of CNNs with the same architecture but trained with distinct hyperparameters, aiming at understanding how they affect the model's performance.

DR projections have also been used in CNNs to compare activation vectors of different inputs [77]. For instance, Aubry and Russell [88] compare the activation vectors of slightly modified images to understand the manifold created by these modifications in the input (image) space. Nguyen et al. [87] use DR projections of images belonging to the same class in a training set to identify image clusters that are characterized by distinct types of features. This allows identifying different styles, or kinds, of images that must be assigned to the same label by the model, which can be a challenging task for improperly trained models.

Since CNNs are often used in applications that take images as inputs, heatmaps have also been used to show which regions of the input image produce strong activations in a given convolutional filter. For instance, Yosinski et al. [84] depict the filters of convolutional layers as images with the same resolution as the input image, with pixels colored based on how they contribute to the activations produced by that filter. Many variations of such image-based techniques have been proposed for the tasks of CNN feature understanding. We divide these methods into two main types: *instance-based* and *feature-based* methods. Methods from both classes are reviewed and discussed next.

Instance-based techniques: Although deep CNN models can achieve high accuracy in image classification tasks, it is not trivial to figure out which features an image should have to be assigned to some particular class. Understanding this may help in identifying mistakes made by the model, such as considering a recurrent background object or feature as intrinsically part of the class. Fig. 9 shows two examples hereof. Image (a) shows a sample used as input for Google's Inception neural network [8]. Image (b) shows which parts of this input image have been relevant for detecting the class 'Labrador' in the input. We see that such parts contain both relevant information (the dog's face) but also spurious pixels (the bottom part of the character's shirt). Image (c) shows the image of a dog that was incorrectly classified as 'Wolf' using the same neural network [13]. Image (d) shows that this incorrect classification relied solely on the presence of a particular background. Using this insight, the designer of the network discovered that wolf images in the training set all had snow as a background. We call such techniques *instance-based* visualizations, as they emphasize parts of input instances responsible for a high activation vector $\mathbf{a}^l$ in some layer l (see also Fig. 10a).

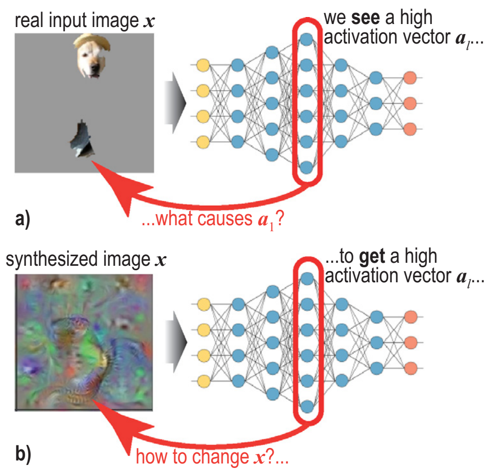

real input image *x*
we **see** a high activation vector a_l...
a)
...what causes a_1?
synthesized image *x*
...to **get** a high activation vector a_l...
b)
how to change *x*?...

Fig. 10. Instance-based vs. feature-based visualizations for feature understanding. In the former, given a particular activation vector $\mathbf{a}^l$, the visualization aims to answer which parts of the real input were more important to generate $\mathbf{a}^l$. In a feature-based visualization, however, an artificial input in synthesized in order to maximize the activation in particular neurons.

Several examples of such instance-based visualizations exist. Simonyan et al. [89] rank pixels in the input images by how much they contribute to a particular class assignment. Montavon et al. [115] backpropagate the activation from deeper layers to identify relevant pixels in the input. Zintgraf et al. [93,95] iteratively remove different patches of the input image and check whether the model is still capable of recognizing the correct class. More recent works [82,97] analyze the weighted gradient in the last convolutional layer to understand how information is flowing through the model. Li et al., use a two-step algorithm based on *Layer-wise Relevance Propagation* (LRP) [114] to recognize the more relevant pixels to the activation. The results of such approaches are usually displayed as so-called saliency maps [89], i.e., heatmaps where pixels are colored based on their relevance to the classifier's output.

A separate challenge of training deep models is to discover how much of the input information has been lost over the layers. This can be done by measuring whether (and how much) it is possible to reconstruct an image from its activation vector produced in a given layer of the network. As a neural network is a complex composition of nonlinear functions, it is likely that a perfect inverse transformation (from activations to the input) is not possible, especially from the deeper layers. Some of the image's most relevant features may be reconstructed in this way. The main technique to approach this problem is called *code inversion* [91]. To recover a synthetic image $\tilde{\mathbf{x}}$ from the activation vector $\mathbf{a}^l$ in a deep network layer l caused by an actual input image $\mathbf{x}$, gradient descent is used to synthesize an image $\tilde{\mathbf{x}}$ that generates an activation vector as close as possible to the one generated by $\mathbf{x}$. Mahendran and Vedaldi [91] found that activations of lower layers can reconstruct the input image $\mathbf{x}$ more faithfully than those from deeper layers. This supports the hypothesis that deeper layers learn more abstract (less detailed) representations of the input. In some networks, even deep layers can preserve image-specific features like object position and colors [94]. Mahendran and Vedaldi [92] present a comprehensive study measuring the quality of images generated by code inversion according to criteria such as reconstruction similarity, naturalness, interpretability, and classification consistency. Other approaches use deconvolutional or up-convolutional networks to synthesize such approximate images [6,94]. Deconvolutional networks try to produce the inverse operations performed by a CNN and recover an image from an activation vector. By contrast, up-convolutional networks are conventionally trained CNNs that learn to predict images from the activation vectors produced by the CNN under analysis.

Another instance-based technique related to image synthesis is *caricaturization* [92]. In this approach, an input image is modified to sharpen features that cause high activations in some given neurons of a CNN model. This is usually done by an optimization algorithm similar to the one used in activation maximization techniques (see below). However, the aim here is to exaggerate the most relevant features of real input, thereby allowing the designer to find out what the model is learning from that input. Such features were found to predominate in the final (deepest) layers of the CNN model.

An important drawback of instance-based techniques is that they produce insights that are valid only for a single input $\mathbf{x}$. This leaves some important questions unanswered. It is usually not clear if all the instances $\{\mathbf{x}_i\}$ belonging to some class y must contain the features highlighted by the explanation of a single such image $\mathbf{x}_i$—for instance, it is not evident from the examples shown in Fig. 9 that all images that will be labeled as *wolf* will be so due to the presence of background snow. To reach such a conclusion, one needs to manually browse through the explanations of multiple inputs $\mathbf{x}_i$ belonging to the class *wolf*.

Feature-based techniques: Whereas instance-based techniques reflect activation information associated with different inputs, feature-based techniques work in the opposite direction (Fig. 10b): they create synthetic input images $\mathbf{x}$ that cause high activation vectors $\mathbf{a}^l$ in some given layer l, or in a neuron u_i^l thereof. When applied to a neuron on the network's last layer L, this technique produces images that capture the features that the network deems relevant for the class that that neuron is responsible for [89]. This way, one can find which generic features the network searches over the input image space to predict specific labels. This approach can also be applied to hidden layers to give insights on which features these layers are looking for when predicting a class [51,84].

As previously outlined, instance-based techniques are, by construction, limited to showing learning explanations for a single input (e.g., which particular pixels of an input image where deemed relevant for determining its class). By contrast, feature-based techniques do not have this problem as they aim to show features that are typically relevant for a whole class. Some techniques do this by displaying images from the training set or test set that produce high activations for the neuron or layer of interest. It can be hard for the designer to figure out which specific features in the displayed image set are responsible for the high activations [86]. To cope with this, many techniques use optimization methods to synthesize a 'summary' image that maximizes the activation vector of interest. This type of analysis allows designers to uncover many properties of deep models. For example, Nguyen et al. [87] present a case where a neuron in a hidden fully-connected layer activates for different underwater objects, such as sharks, turtles, and scuba divers, indicating that these layers often learn high-level concepts that are present in multiple classes.

One of the first techniques to synthesize images that cause high activations to a particular class is *activation maximization* [51]. As the activation of a neuron can be seen as a nonlinear function $\phi(\mathbf{x})$ where $\mathbf{x}$ is a given input (see also Fig. 2), the input $\hat{\mathbf{x}}$ that maximizes the above function can be found by applying an optimization method—for instance, traditional gradient ascent—on the input $\mathbf{x}$ with a fixed Θ equal to the network parameters learned after training. This is a non-convex optimization problem where convergence to an optimal global value cannot be guaranteed. However, different and meaningful local optima can be found. These often represent different facets, or aspects, of the class of interest [87], such as different properties of an image that may all be equally relevant for determining its class. This technique was first applied to deep belief networks [51] and later generalized to deep

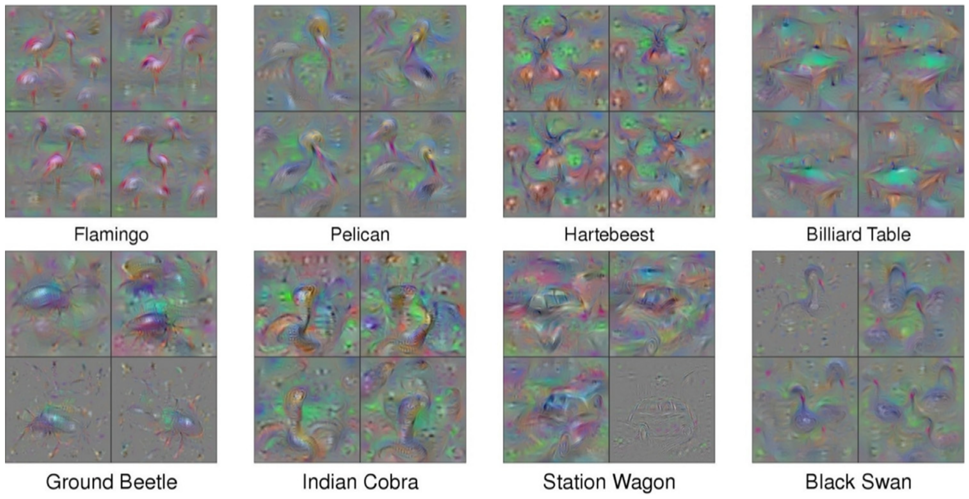


Fig. 11. Activation maximization algorithm applied to eight different classes of the ImageNet dataset [5]. With this technique, experts can visualize which features a neuron is looking for on the input. If applied to neurons in deep layers—those usually return high activations for only one class—one can see images that resemble actual classes on the training set, however in a very unrealistic manner. Image adapted from Yosinski et al. [84].

convolutional networks [89]. Fig. 11 shows the results of the activation maximization technique produced by eight different classes dataset [5]. The produced synthetic images highlight typical features that actual images of the respective classes tend to have.

A drawback of activation maximization is that the synthesized images may be too abstract to interpret, as can be seen in Fig. 11. This is not surprising, given that the space of all images of a given class is too large to be captured by a single 'average' image [89,111]. To address this, several regularization methods have been proposed. These methods constrain the generation of synthetic images by enforcing various criteria that are typically present in real-world images [89–91]. For example, Yosinski et al. [84] propose four regularization techniques that aim to synthesize images with more realistic features: L_2 decay; Gaussian blur; small norm pixel clipping; and small contribution pixel clipping. The two first regularizations aim to remove high brightness amplitudes and high frequencies that rarely appear in natural images; the last two regularizations remove pixels with negligible influences on the activations, letting the designer focus on the important features of the synthesized image. Activation maximization often produces images with repeated features, such as multiple objects or exaggerated objects of the analyzed class, as such exaggerations increase the class-specific activation values. For instance, optimizing an image for a flamingo-recognizing activations leads to multiple pelicans scattered over the synthetic image (Fig. 11 top-left). This is one of the causes of the artificial look of such synthetic images. Nguyen et al. [87] alleviate this problem by using a center-biased regularization penalizes changes close to the borders of the image, thereby forcing the optimization to synthesize features closer to the image center.

While the above improvements create more interpretable synthetic images, they still exhibit non-natural colors and borders. Generating realistic images has, however, been successfully achieved by generative neural networks (GNNs) [63,116]. Nguyen et al. [86] proposed to use a GNN model as a prior to generating realistic images that maximize the activation of a CNN neuron. GNNs are deep models that learn to generate novel samples from the distribution in which the training set lies. In this case, the GNN is trained to generate realistic images from a numeric vector input. After that, this numeric input is optimized to generate an image that maximizes the activation of a given neuron of the CNN.

Another issue faced when synthesizing images with activation maximization is that classes may have very distinct instances. For example, a 'store' class could be represented both by images of the outdoor facade of the store or by images of the inside of the building. Hence, neurons—especially the ones in deeper layers—that recognize features of such classes must be able to activate for very different sets of input features. Such neurons are then said to be *multifaceted* [87]. In such cases, traditional optimization methods like gradient ascent end up mixing features of different facets into the resulting synthetic image, which renders it ambiguous and confusing. To overcome this problem, Nguyen et al. introduce a multifaceted feature visualization [87] that initializes the activation maximization algorithm with an image obtained by averaging instances of the training set that belong to the same class facet. This creates a bias aiming to make the algorithm synthesize an image with the features of that facet.

Recurrent neural networks: One of the main challenges for RNNs, just like for CNNs and DFNs, is to understand the activation patterns produced by specific inputs and which features these inputs are capturing, since being able to do so is key to understanding the behavior of the trained model. Like for CNNs and DFNs, heatmaps have also been used to visualize activations of recurrent models [71,100]. While it is possible to get good insights on which hidden states produce higher activations for a single input by looking at an activation heatmap, it is not easy to understand if the same hidden states share similar behavior for a group of inputs, e.g., words with similar meanings. To address this problem, Ming et al. [71] propose a technique that co-clusters hidden states and input elements—in particular, it creates two cluster models: one of the activation values of a selected hidden layer and one of the input values given to the network; then it tries to identify correlations between types of inputs that consistently generate activations associated with one particular activation cluster.

Another application of heatmaps for RNN models is to measure the importance of each input unit. Li et al. [99] proposed a saliency heatmap that shows the saliency score of each word in the input of a word classification task. This score is calculated by checking how much each input unit contributed to the final label assignment. Ding et al. [101] also use heatmaps to display the relevance of each input word for each output word in a machine translation model. To calculate this relevance, they use a layer-wise relevance propagation (LRP) algorithm. Unlike gradient-based approaches, LRP does not require the activations to be differentiable, which confers additional robustness to the approach.

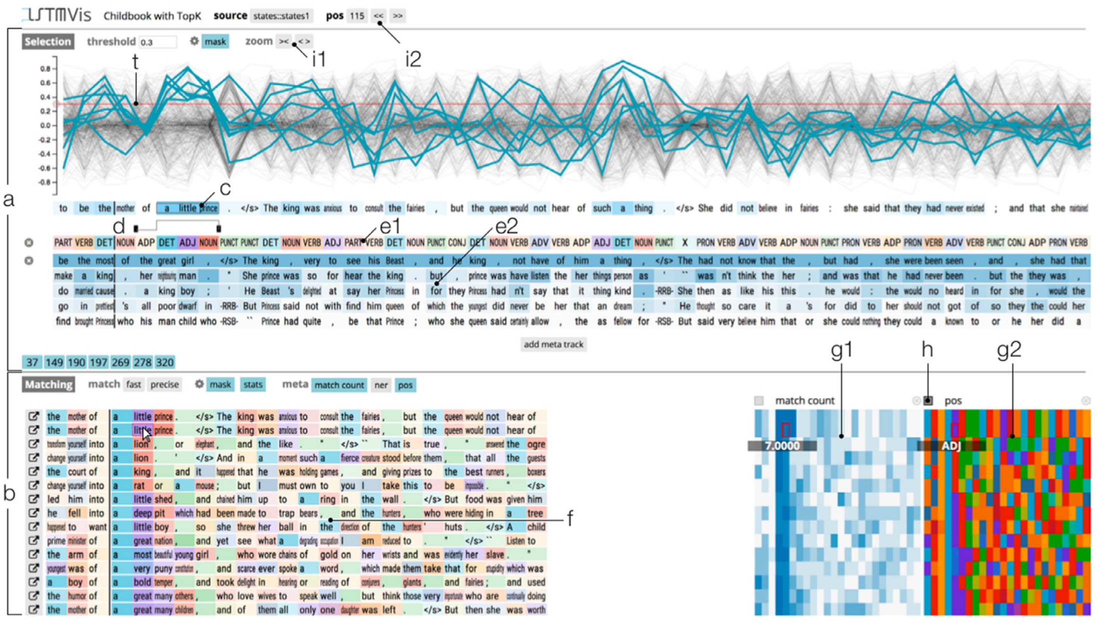


Fig. 12. LSTMVis tool [98] showing the activations of the hidden states in a RNN model for a sequence of input units chosen by the designer. When the designer chooses a text range, the visualization displays how hidden state values changed through the processing of such text range (a). The visualization provides several interactive tools, such as the capability of choosing a threshold for highlighted hidden states (t), the matching and comparison of input sequences resulting in similar hidden states (b), and the visualization of user-defined meta-data about the model (g1, g2).

Heatmaps have also been used to display characteristics of the input text data. For instance, Karpathy et al. [102]—one of the first works proposing the utilization of visualization to understand RNNs—build a heatmap measuring the relevance of each input character or word to the activation of a given hidden unit. This approach was able to identify interpretable semantic units—for instance, one specialized in identifying new lines—in a character prediction application. Strobelt et al. [98] used heatmaps to display designer-defined metrics of interest on the input text, such as to which part-of-speech (POS) a given word belongs shown to (Fig. 12). Conceptually, this visualization is of the same kind of explanatory type as the instance-based visualizations for CNNs (Figs. 9 and 10a), as they highlight parts of an input sample that cause the network to choose a given output class.

RNNs are intrinsically designed to handle sequential input data such as text or time series. When a new input item is processed, the values of the model's hidden units change. This affects the result not only for the current input item but also for a (potentially long) range of subsequent input items. Understanding how each input item changes these hidden units and affects latter computations helps identifying critical aspects of the model that may lead to undesirable performance. Given the sequential nature of RNNs, time series charts emerge as a straightforward way to visualize changes in the hidden units' values. Yet, we have found only a few works in this direction. For instance, Strobelt et al. [98] proposed a parallel coordinates plot (PCP) visualization where each dimension is an input unit, and each polyline displays how a single hidden state varies with the input. Hence, the horizontal PCP axis can be interpreted as a temporal order, and the polylines are analogous to time series. By allowing the designer to highlight units with high activation for a given range of the input, Strobelt et al. were able to find distinct regions of similar behavior in the input text stream, and concluded that such regions had similar semantics.

6. Discussion and open challenges

Although a substantial amount of work has been done over the past few years regarding the use of visual analytics in deep learning techniques, there are still many challenges that need to be successfully addressed by future research. We discuss these challenges from the point of view of the three types of *tasks* that we structured our survey along (i.e., architecture understanding, training analysis, and feature understanding) and, additionally, from the point of view of end-to-end *requirements* that engineers developing deep models face in practice.

6.1. Architecture understanding

Hyperparameter exploration: One problem that is still open even for machine learning experts is how hyperparameters affect the training results. Developers of neural networks usually make architectural choices such as the number of neurons per layer, number of layers, activation function types, training batch size, learning rate, and number of training epochs, by empirical and trial-and-error approaches. This considerably adds to the cost of fine-tuning the training of deep models. In a more broad sense, analysing the parameter space of a simulation model is a topic that has caught a strong interest from the VA community in the past years [73]. However, that is still a lot of space for novel contributions that can help designers to more precisely tune their deep networks. Interactive 'drawing board' solutions where all these aspects can be directly controlled by a designer, while their effects are visualized in real time, exist [76] but cover only very small networks (tens of neurons) with simple two-dimensional inputs.

It would be interesting for machine learning experts to visually analyze the hyperparameter space of a neural model and how values in this space affect the model performance [34]. To do this, visually more scalable techniques (capable of depicting large network architectures) are needed, as well as methods that annotate the hyperparameter space with measured network performance. There is work on the use of Bayesian optimization for actively selecting which hyperparameter values to try next, based on annotating particular settings of such values with the corresponding model performance and trying to infer which ones might work better [117]. Although there is some work about applying such techniques to deep models [118,119], more research is needed in this direction.

Additionally, faster training pipelines, possibly based on multi-scale techniques, are needed to close the sensemaking loop at interactive rates, thereby allowing designers to effectively 'steer' the architecture design as they observe its behavior.

6.2. Training analysis

Training data exploration: To be properly trained, deep models often require large and high-dimensional training sets. However, to date, there are only a few solutions to understanding such training sets, and in particular, which aspects of the input data affect training in specific ways. Training instances may contain hidden biases or mistakes, such as mislabeling, irrelevant correlations of input features with classes [13,52], and, at a higher level, unbalanced coverage of the entire input space X by training samples. All of these aspects can severely harm the effectiveness of the trained model. Visualization techniques, notably the ones focused on the analysis of high-dimensional datasets [14,52,80], are a promising alternative to address this issue. Additionally, a better understanding of the training set characteristics can lead to more efficient choices of architectures and hyperparameters.

Training guiding and interaction: Due to the difficulty in understanding machine learning techniques and the long time required to train deep models, interactive solutions have received significant interest from the machine learning community [36–38]. Visualization methods are key to achieve such interaction, as they can quickly show the designer what is happening during the training process. However, only a few approaches have focused on using visualization to provide real-time feedback to the designer [76,77,79]. The challenge here is directly related to hyperparameter exploration, i.e., providing both visually scalable and computationally scalable (fast) metaphors for the training process.

6.3. Feature understanding

Explainable models: Visual analytics has proven to be an effective tool in explaining the features learned by neural models. Current visualization methods can show which features (from the input data) have been learned by a given model. Many types of features can be considered by these techniques, such as pixels in an image, words in a text document, and value ranges of input data attributes (columns in a data table), each of which computed either per input sample or per set of related instances, e.g., class or class facet. Solutions produced by this type of visualization highlight the most discriminative features for determining a class [13] and follow the intuitions used by earlier methods that aimed at achieving a similar goal but for classifiers that used hand-engineered features [20]. The need for more explainable models, however, is still noticeable. In particular, if one could say *which* input features are responsible for a model's decision, the next step would be to show *how* that decision was made. This involves explaining the responsibilities of groups of layers or neurons of a DNN and how these work together to calculate the final output [34]. In the long run, this will lift the current feature understanding goal to cover the more important goal of understanding how a model as a whole took a given decision.

6.4. Non-functional requirements

Apart from the functional requirements for the visual understanding process of deep learning mentioned above, some non-functional requirements exist, as discussed next.

Fidelity: Modern DNNs can have hundreds of thousands of neurons spread over hundreds of layers and millions of parameters [4,8,9]. The sheer amount of data embedded in, or produced by, such models demands novel VA techniques that scale effectively. The key issue here is that of fidelity or trust: when data is aggregated or simplified, how can we be sure that we trust what the visualization shows? For instance, many approaches use dimensionality reduction (DR) techniques to visualize the high-dimensional data produced by DNNs. Small changes on the hyperparameters of DR methods can lead to massively different visualizations that may easily convey different or wrong insights [120–122]. The goal of optimizing dimensionality reduction so that it accurately conveys the high-dimensional data structure of large datasets (millions of observations, hundreds of dimensions) is an ongoing research endeavor [123,124].

Scalability: The large size of modern DNNs poses two scalability problems. First, we need to develop *visually* scalable infovis metaphors to depict the large amount of high-dimensional, temporal, and relational data spanned by such networks. This is in itself a key challenge in information visualization, for which answers are yet to be found. Separately, we need access to *computationally* scalable ways of performing DNN training, so that insights found in this process can be communicated to the designer at interactive rates, for the VA sensemaking loop to be effectively closed. In cases where interactive-rate training of DNNs is simply not possible due to the size of the problem, approximation methods could be developed that deliver a less accurate, but still insightful, view on the training process at interactive rates.

Different applications: Current research works focus mainly on models handling image classification or natural language processing tasks that use well-behaved training sets. However, more complex applications usually have to handle training sets that may be composed of different types of data and that may come from distinct sources, thus requiring more complicated architectures [74]. Visual analytics tools directing towards these models could be an interesting perspective for future research.

7. Conclusion

In this article, we review works aiming at using visual analytics techniques to understand deep neural networks—a topic that has been widely discussed by the research community in the past few years. We classify these publications into three categories, depending on the particular visualization goal that they aim to achieve: network architecture understanding, visualization to support training analysis, and feature understanding. In particular, we are able to identify that most of the reviewed publications have been mainly focused on understanding which features a given trained model can recognize, how they do it, and how the learning of such features occurs. These visualization approaches prove to be effective in validating the performance of deep models and providing more intuition of their inner workings to the designer of the respective neural network model. However, there is still a lack of contributions aiming at developing techniques that can interactively guide the development of a deep neural network, with only a few approaches addressing this issue. Given that deep networks are usually difficult and slow to train, we consider this as a promising topic of future research, with visual analytics playing a key role in providing such visual interactivity to machine learning experts.

Acknowledgments

Funding: This work was supported by the Conselho Nacional de Desen-volvimento Científico e Tecnológico (CNPq) [grant number 308851/2015-3]; the Research Council of Norway (RCN) [grant number 240862]; RCN and the Nor-wegian Centre for International Cooperation in Education (SIU) [grant number 261645]; and the

Rio Grande do Sul and Oslo collaboration on AI and Robotics (RO-CAIR) [grant number UTF-2016-short-term/10128].

References

[1] Goodfellow I, Bengio Y, Courville A. Deep learning. The MIT Press; 2016.
[2] Samuel AL. Some studies in machine learning using the game of checkers. IBM J Res Dev 1959;3(3):210–29.
[3] LeCun Y, Bengio Y, Hinton G. Deep learning. Nature 2015;521(7553):436–44.
[4] Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. In: Proceedings of the international conference on neural information processing systems, vol. 1; 2012. p. 1097–105.
[5] Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L. ImageNet: a large-scale hierarchical image database. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2009. p. 248–55.
[6] Zeiler MD, Fergus R. Visualizing and understanding convolutional networks. In: Proceedings of the European conference on computer vision. Springer; 2014. p. 818–33.
[7] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. CoRR 2014;abs/1409.1556. arXiv:1409.1556.
[8] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, et al. Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2015. p. 1–9.
[9] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2016. p. 165–73.
[10] Liu M, Shi J, Li Z, Li C, Zhu J, Liu S. Towards better analysis of deep convolutional neural networks. IEEE Trans Vis Comput Gr 2017;23(1):91–100.
[11] Marcus G. Deep learning: A critical appraisal. CoRR 2018;abs/1801.00631. arXiv:1801.00631.
[12] Samek W, Wiegand T, Müller KR. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. CoRR 2017;abs/1708.08296. arXiv:1708.08296.
[13] Ribeiro MT, Singh S, Guestrin C. Why should I trust you? Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD'16; ACM. ISBN 978-1-4503-4232-2; 2016, p. 1135–1144.
[14] Liu S, Maljovec D, Wang B, Bremer P-T, Pascucci V. Visualizing high-dimensional data: advances in the past decade. Comput Gr Forum 2016;23(3):1249–68.
[15] van der Maaten L, Postma E, van den Herik J. Dimensionality reduction: a comparative review. Mach Learn Res 2009;10:66–71.
[16] Cunningham JP, Ghahramani Z. Linear dimensionality reduction: Survey, insights, and generalizations. J Mach Learn Res 2015;16:2859–900.
[17] Sorzano C, Vargas J, Montano AP. A survey of dimensionality reduction techniques. CoRR 2014;abs/1403.2877. arXiv:1403.2877.
[18] Blum MGB, Nunes MA, Prangle D, Sisson SA. A comparative review of dimension reduction methods in approximate Bayesian computation. Stat Sci 2013;28(2):189–208.
[19] Rauber PE, Falcão AX, Telea AC. Projections as visual aids for classification system design. Inf Vis 2018;17(4):282–305.
[20] Rauber P, da Silva R, Feringa S, Celebi M, Falcão A, Telea A. Interactive image feature selection aided by dimensionality reduction. In: Proceedings of the EuroVA; 2015. p. 67–74.
[21] Krause J, Perer A, Bertini E. INFUSE: interactive feature selection for predictive modeling of high dimensional data. IEEE Trans Vis Comput Gr 2014;20(12):1614–23.
[22] Yuan X, Ren D, Wang Z, Guo C. Dimension projection matrix/tree: interactive subspace visual exploration and analysis of high dimensional data. IEEE Trans Vis Comput Gr 2013;19(12):2625–33.
[23] Tatu A, Maas F, Farber I, Bertini E, Schreck T, Seidl T, et al. Subspace search and visualization to make sense of alternative clusterings in high-dimensional data. In: Proceedings of the IEEE VAST; 2012. p. 63–72.
[24] Turkay C, Filzmoser P, Hauser H. Brushing dimensions: a dual visual analysis model for high-dimensional data. IEEE Trans Vis Comput Gr 2011;17(12):2591–9.
[25] Noris B. MLDemos: open source visualization tool for machine learning algorithms. 2017. http://mldemos.epfl.ch.
[26] Gleicher M. Explainers: expert explorations with crafted projections. IEEE Trans Vis Comput Gr 2013;19(12):2042–51.
[27] Gleicher M. A framework for considering comprehensibility in modeling. Big Data 2016;4(2):75–88.
[28] Montavon G, Samek W, Müller K-R. Methods for interpreting and understanding deep neural networks. Digit Sig Process 2018;73:1–15.
[29] Samek W, Binder A, Montavon G, Lapuschkin S, Müller KR. Evaluating the visualization of what a deep neural network has learned. IEEE Trans Neural Netw Learn Syst 2017;28(11):2660–73.
[30] Yeager L, Heinrich G, Mancewicz J, Houston M. Effective visualizations for training and evaluating deep models. In: Proceedings of the International conference on machine learning workshop on visualization for deep learning; 2016.
[31] Zeng H. Towards better understanding of deep learning with visualization. 2016. [M.Sc. thesis], Department of Computer Science and Engineering, Hong-Kong University of Science and Technology.
[32] Seifert C, Aamir A, Balagopalan A, Jain D, Sharma A, Grottel S, et al. Visualizations of deep neural networks in computer vision: A survey. In: Transparent data mining for big and small data. Springer; 2017. p. 123–44.
[33] Grün F, Rupprecht C, Navab N, Tombari F. A taxonomy and library for visualizing learned features in convolutional neural networks. CoRR 2016;abs/1606.07757. arXiv:1606.07757.
[34] Liu S, Wang X, Liu M, Zhu J. Towards better analysis of machine learning models: a visual analytics perspective. Vis Inf 2017;1(1):48–56.
[35] Lu Y, Garcia R, Hansen B, Gleicher M, Maciejewski R. The state-of-the-art in predictive visual analytics. Comput Gr Forum 2017;36(3):539–62.
[36] Amershi S, Cakmak M, Knox WB, Kulesza T. Power to the people: the role of humans in interactive machine learning. AI Mag 2014;35(4):105–20.
[37] Bernardo F, Zbyszynski M, Fiebrink R, Grierson M, et al. Interactive machine learning for end-user innovation. In: Proceedings of the designing the user experience of machine learning systems (AAAI Spring Symposium Series); 2017.
[38] Sacha D, Sedlmair M, Zhang L, Lee JA, Weiskopf D, North S, et al. Human–centered machine learning through interactive visualization. In: Proceedings of the European symposium on artificial neural networks, computational intelligence and machine learning; 2016.
[39] Lipton Z. The mythos of model interpretability. CoRR 2016;abs/1606.03490. arXiv:1606.03490.
[40] Hohman F, Kahng M, Pienta R, Chau DH. Visual analytics in deep learning: An interrogative survey for the next frontiers. CoRR 2018;abs/1801.06889. arXiv:1801.06889.
[41] Murphy K. Machine learning: a probabilistic perspective. The MIT Press; 2012.
[42] Bishop CM. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag New York; 2006.
[43] Hastie T, Tibshirani R, Friedman J. The elements of statistical learning: data mining, inference, and prediction. Springer; 2009.
[44] Breiman L. Random forests. Mach Learn 2001;45(1):5–32.
[45] LeCun Y, Boser B, Denker JS, Henderson D, Howard RE, Hubbard W, et al. Backpropagation applied to handwritten zip code recognition. Neural Comput 1989;1(4):541–51.
[46] Elman JL. Finding structure in time. Cognit Sci 1990;14(2):179–211.
[47] Park SH, Goo JM, Jo C-H. Receiver operating characteristic (ROC) curve: practical review for radiologists. Korean J Radiol 2004;5(1):11–18.
[48] Powers D. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. J Mach Learn Technol 2011;2(1):37–63.
[49] Fawcett T. An introduction to ROC analysis. Pattern Recognit Lett 2006;27(8):861–74.
[50] Brehmer M, Munzner T. A multi-level typology of abstract visualization tasks. IEEE Trans Vis Comput Gr 2013;19(12):2376–85.
[51] Erhan D, Bengio Y, Courville A, Vincent P. Visualizing higher-layer features of a deep network. Technical Report 1341. University of Montreal; 2009.
[52] Rauber P, Fadel SG, Falcão A, Telea A. Visualizing the hidden activity of artificial neural networks. IEEE Trans Vis Comput Gr 2017b;23(1):101–10.
[53] Wong PC, Thomas J. Visual analytics. IEEE Comput Gr Appl 2004;24(5):20–1.
[54] Keim DA, Mansmann F, Schneidewind J, Thomas J, Ziegler H. Visual analytics: scope and challenges. In: Lecture notes in computer science (LNCS 4404). Springer; 2008. p. 76–90.
[55] Lu J, Chen W, Ma Y, Ke J, Li Z, Zhang F, et al. Recent progress and trends in predictive visual analytics. Front Comput Sci 2017;11(2):192–207.
[56] Pirolli P, Card S. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In: Proceedings of the international conference on intelligence analysis; 2005.
[57] Keim D, Andrienko G, Fekete J-D, Görg C, Kohlhammer J, Melan con G. Visual analytics: definition, process, and challenges. In: Information visualization – human-centered issues and perspectives. Springer; 2008. p. 154–75.
[58] Keim D, Kohlhammer J, Ellis G, Mansmann F. Mastering the information age: solving problems with visual analytics. Eurographics Association; 2010.
[59] IEEE VAST 2017 Symposium. 2017. http://ieeevis.org/year/2017/info/papers.
[60] Streeter MJ, Ward MO, Alvarez SA. NVIS: an interactive visualization tool for neural networks. In: Proceedings of the SPIE visual data exploration and analysis, vol. 4302; 2001. p. 1–8.
[61] Tzeng FY, Ma KL. Opening the black box – data driven visualization of neural networks. In: Proceedings of the IEEE visualization; 2005. p. 383–90.
[62] Hinton GE, Zemel RS. Autoencoders, minimum description length and Helmholtz free energy. In: Proceedings of the international conference on neural information processing systems. NIPS'93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1993. p. 3–10.
[63] Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, et al. Generative adversarial nets. In: Advances in neural information processing systems, vol. 27. Curran Associates, Inc.; 2014. p. 2672–80.
[64] Hinton GE, Osindero S, Teh Y-W. A fast learning algorithm for deep belief nets. Neural Comput 2006;18(7).
[65] Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, et al. Human-level control through deep reinforcement learning. Nature 2015;518:7540.
[66] Gibson A, Patterson J. Deep learning. O'Reilly Media Inc.; 2017.
[67] Zahavy T, Ben-Zrihem N, Mannor S. Graying the black box: Understanding DQNS. In: Proceedings of the international conference on machine learning; 2016. p. 1899–908.
[68] Liu M, Shi J, Cao K, Zhu J, Liu S. Analyzing the training processes of deep generative models. IEEE Trans Vis Comput Gr 2018;24(1):77–87.

[69] Ren D, Amershi S, Lee B, Suh J, Williams JD. Squares: supporting interactive performance analysis for multiclass classifiers. IEEE Trans Vis Comput Gr 2017;23(1):61–70.
[70] Mühlbacher T, Piringer H. A partition-based framework for building and validating regression models. IEEE Trans Vis Comput Gr 2013;19(12):1962–71.
[71] Ming Y, Cao S, Zhang R, Li Z, Chen Y, Song Y, et al. Understanding hidden memories of recurrent neural networks. In: Proceedings of the IEEE visual analytics science and technology (VAST); 2017.
[72] Zeng H, Haleem H, Plantaz X, Cao N, Qu H. CNNComparator: comparative analytics of convolutional neural networks. In: Proceedings of the workshop on visual analytics for data learning (VADL); 2017.
[73] Sedlmair M, Heinzl C, Bruckner S, Piringer H, Mller T. Visual parameter space analysis: a conceptual framework. IEEE Trans Vis Comput Gr 2014;20(12):2161–70.
[74] Kahng M, Andrews PY, Kalro A, Chau DH. Activis: visual exploration of industry-scale deep neural network models. IEEE Trans Vis Comput Gr 2018;24(1):88–97.
[75] Wongsuphasawat K, Smilkov D, Wexler J, Wilson J, Man D, Fritz D, et al. Visualizing dataflow graphs of deep learning models in tensorflow. IEEE Trans Vis Comput Gr 2018;24(1):1–12.
[76] Smilkov D, Carter S, Sculley D, Vigas FB, Wattenberg M. Direct-manipulation visualization of deep networks. CoRR 2017;abs/1708.03788. arXiv:1708.03788.
[77] Chung S, Suh S, Park C, Kang K, Choo J, Kwon BC. RevaCNN: Real-Time visual analytics for convolutional neural network. In: Proceedings of the ACM SIGKDD workshop on interactive data exploration and analytics (IDEA); 2016.
[78] Harley AW. An interactive node-link visualization of convolutional neural networks. In: Proceedings of the international symposium on advances in visual computing (ISVC). Springer; 2015. p. 867–77.
[79] Qi H, Liu J, Zou X, Tang A. BIDViz: real-time monitoring and debugging of machine learning training processes. EECS Department, University of California, Berkeley; 2017. [Master's thesis].
[80] Pezzotti N, Hollt T, Gemert JV, Lelieveldt BPF, Eisemann E, Vilanova A. DeepEyes: progressive visual analytics for designing deep neural networks. IEEE Trans Vis Comput Gr 2018;24(1):98–108.
[81] Zhong W, Xie C, Zhong Y, Wang Y, Xu W, Cheng S, et al. Evolutionary visual analysis of deep neural networks. In: Proceedings of the international conference on machine learning workshop on visualization for deep learning; 2017.
[82] Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D, et al. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In: ICCV. 2017, p. 618–626.
[83] Cashman D, Patterson G, Mosca A, Chang R. RNNbow: visualizing learning via backpropagation gradients in recurrent neural networks. In: Proceedings of the workshop on visualization for deep learning (VADL); 2017.
[84] Yosinski J, Clune J, Fuchs T, Lipson H. Understanding neural networks through deep visualization. In: Proceedings of the international conference on machine learning workshop on deep learning; 2015. arXiv: 1506.06579
[85] Alsallakh B, Jourabloo A, Ye M, Liu X, Ren L. Do convolutional neural networks learn class hierarchy? IEEE Trans Vis Comput Gr 2018;24(1):152–62.
[86] Nguyen A, Dosovitskiy A, Yosinski J, Brox T, Clune J. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. In: Proceedings of the international conference on neural information processing systems; 2016a. p. 3395–403.
[87] Nguyen A, Yosinski J, Clune J. Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks. CoRR 2016;abs/1602.03616. arXiv:1602.03616.
[88] Aubry M, Russell BC. Understanding deep features with computer-generated imagery. In: Proceedings of the IEEE international conference on computer vision (ICCV); 2015.
[89] Simonyan K, Vedaldi A, Zisserman A. Deep inside convolutional networks: Visualising image classification models and saliency maps. CoRR 2013;abs/1312.6034. arXiv:1312.6034.
[90] Wei D, Zhou B, Torrabla A, Freeman W. Understanding intra-class knowledge inside CNN. CoRR 2015;abs/1507.02379. arXiv:1507.02379.
[91] Mahendran A, Vedaldi A. Understanding deep image representations by inverting them. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2015.
[92] Mahendran A, Vedaldi A. Visualizing deep convolutional neural networks using natural pre-images. Int J Comput Vis 2016;120(3):233–55.
[93] Zintgraf LM, Cohen TS, Welling M. A new method to visualize deep neural networks. CoRR 2016;abs/1603.02518. arXiv:1603.02518.
[94] Dosovitskiy A, Brox T. Inverting visual representations with convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition.
[95] Zintgraf LM, Cohen TS, Adel T, Welling M. Visualizing deep neural network decisions: Prediction difference analysis. CoRR 2017;abs/1702.04595. arXiv:1702.04595.
[96] Li H, Mueller K, Chen X. Beyond saliency: understanding convolutional neural networks from saliency prediction on layer-wise relevance propagation. CoRR 2017;abs/1712.08268. arXiv:1712.08268.
[97] Bojarski M, Choromanska A, Choromanski K, Firner B, Jackel L, Müller U, et al. VisualBackProp: Efficient visualization of CNNs. CoRR 2016;abs/1611.05418. arXiv:1611.05418.
[98] Strobelt H, Gehrmann S, Pfister H, Rush AM. LSTMVis: a tool for visual analysis of hidden state dynamics in recurrent neural networks. IEEE Trans Vis Comput Gr 2018;24(1):667–76.
[99] Li J, Chen X, Hovy E, Jurafsky D. Visualizing and understanding neural models in NLP. CoRR 2015;abs/1506.01066. arXiv:1506.01066.
[100] Rong X, Adar E. Visual tools for debugging neural language models. In: Proceedings of the international conference on machine learning workshop on visualization for deep learning; 2016.
[101] Ding Y, Liu Y, Luan H, Sun M. Visualizing and understanding neural machine translation. In: Proceedings of the annual meeting of the association for computational linguistics (Volume 1: Long Papers), vol. 1; 2017. p. 1150–9.
[102] Karpathy A, Johnson J, Fei-Fei L. Visualizing and understanding recurrent networks. CoRR 2015;abs/1506.02078. arXiv:1506.02078.
[103] Diehl S. Software visualization: visualizing the structure, behaviour, and evolution of software. Springer; 2007.
[104] van der Zwan M, Codreanu V, Telea A. CUBu: universal real-time bundling for large graphs. IEEE Trans Vis Comput Gr 2016;22(12):2250–63.
[105] Lhuillier A, Hurter C, Telea A. State of the art in edge and trail bundling techniques. Comput Gr Forum 2017;36(3):619–45.
[106] Zhang L, Wang S, Liu B. Deep learning for sentiment analysis: A survey. CoRR 2018;abs/1801.07883. arXiv:1801.07883.
[107] Zhang J, Zong C. Deep neural networks in machine translation: an overview. IEEE Intell Syst 2015;30(5):16–25.
[108] Hochreiter S, Bengio Y, Frasconi P, Schmidhuber J. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. IEEE Press; 2001. p. 464.
[109] Maaten Lv d, Hinton G. Visualizing data using t-SNE. J Mach Learn Res 2008;9(Nov):2579–605.
[110] da Silva RRO, Vernier EF, Rauber PE, Comba JLD, Minghim R, Telea AC. Metric evolution maps: Multidimensional attribute-driven exploration of software repositories. In: VMV; 2016.
[111] Nguyen A, Yosinski J, Clune J. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2015.
[112] Netzer Y, Wang T, Coates A, Bissacco A, Wu B, Ng AY. Reading digits in natural images with unsupervised feature learning. In: Proceedings of the neural information processing systems; 2011. p. 5–12.
[113] Sacha D, Zhang L, Sedlmair M, Lee JA, Peltonen J, Weiskopf D, et al. Visual interaction with dimensionality reduction: a structured literature analysis. IEEE Trans Vis Comput Gr 2017;23(1):241–50.
[114] Bach S, Binder A, Montavon G, Klauschen F, Müller K-R, Samek W. On pixel–wise explanations for non-linear classifier decisions by layer-wise relevance propagation. PLOS One 2015;10.
[115] Montavon G, Lapuschkin S, Binder A, Samek W, Müller K-R. Explaining nonlinear classification decisions with deep Taylor decomposition. Pattern Recognit 2017;65:211–22.
[116] Dosovitskiy A, Brox T. Generating images with perceptual similarity metrics based on deep networks. In: Advances in neural information processing systems, vol. 29. Curran Associates, Inc.; 2016b. p. 658–66.
[117] Brochu E, Cora VM, De Freitas, N. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. CoRR 2010;abs/1012.2599. arXiv:1012.2599.
[118] Snoek J, Larochelle H, Adams RP. Practical Bayesian optimization of machine learning algorithms. In: Advances in neural information processing systems, vol. 25. Curran Associates, Inc.; 2012.
[119] Snoek J, Rippel O, Swersky K, Kiros R, Satish N, Sundaram N, et al. Scalable Bayesian optimization using deep neural networks. In: Proceedings of the international conference on machine learning; 2015.
[120] Wattenberg M. How to use t-SNE effectively. 2017. https://distill.pub/2016/misread-tsne.
[121] Martins R, Coimbra D, Minghim R, Telea A. Visual analysis of dimensionality reduction quality for parameterized projections. Comput Gr 2014;41:26–42.
[122] Rauber P, Falcão A, Telea A. Visualizing time-dependent data using dynamic t-SNE. In: Proceedings of the EuroVis – short papers; 2016. p. 137–42.
[123] McInnes L, Healy J. Umap: Uniform manifold approximation and projection for dimension reduction. CoRR 2018;abs/1802.03426. arXiv:1802.03426.
[124] Pezzotti N, Höllt T, Lelieveldt BP, Eisemann E, Vilanova A. Hierarchical stochastic neighbor embedding. Comput Gr Forum 2016;35(3):21–30.