UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAFAEL GARCIA

Visual Analytics as a Tool for Deep Learning Engineering

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. João Luiz Dihl Comba
Coadvisor: Prof. Dr. Bruno Castro da Silva

Porto Alegre
June 2019

CIP — CATALOGING-IN-PUBLICATION

Garcia, Rafael

Visual Analytics as a Tool for Deep Learning Engineering / Rafael Garcia. – Porto Alegre: PPGC da UFRGS, 2019.

103 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: João Luiz Dihl Comba; Coadvisor: Bruno Castro da Silva.

1. Deep Learning. 2. Neural Networks. 3. Visual Analytics. I. Comba, João Luiz Dihl. II. da Silva, Bruno Castro. III. Título.

ABSTRACT

Deep Neural Networks are responsible for many groundbreaking results achieved in Artificial Intelligence over the past decade. Several applications from a wide range of domains have begun to employ these models to solve recognition and learning tasks at both scientific and industrial level. Despite the significant growth in usage, the design of effective neural networks still faces multiple challenges, hindering the full potential of such techniques. In particular, the high complexity of such models, that often contain thousands or even millions of parameters spread across several layers, makes it challenging to identify which features the model learned to recognize and how they affect the decision process. Without this knowledge, neural networks become essentially a black-box approach that provides little understanding of what the model does. Such a characteristic is undesirable because it hinders the ability to interpret and evaluate the inner workings of such networks. If we can have more understanding about the model, we can use this information to build models with higher and fairer performance, and that can be employed in more critical tasks — such as medical applications — where understanding how the model comes up to a particular conclusion is extremely important. One way to address these challenges is the employment of visual analytics tools to support the exploration and analysis of features learned by neural networks. Many of such tools were developed over the past few years, each with the intent to address challenges in one or more steps of the neural network design workflow. In this work, we review such visual analytics tools, and we propose a taxonomy to classify them according to whether they provide the analysis of the network's architecture, the evaluation of the training process, or the understanding and interpretation of which features the model learned to recognize. Next, we propose a novel visualization technique to guide the architectural tuning of neural networks. We also demonstrate, with a series of experiments, how our method can provide many insights about whether a network's architecture should be changed and which changes — such as adding or removing layers and increasing or decreasing layer size — the designer should perform.

Keywords: Deep Learning. Neural Networks. Visual Analytics.

Análise Visual como uma Ferramenta para Aprendizagem Profunda

RESUMO

Redes Neurais Profundas são responsáveis por muitos dos mais importantes resultados alcançados na Inteligência Artificial na última década. Muitas aplicações em diversas áreas começaram a utilizar estes modelos para solucionar tarefas de reconhecimento e aprendizado tanto à nível industrial quanto científico. Apesar do significativo aumento na utilização desta técnica, o desenvolvimento de redes neurais efetivas ainda encara vários desafios que acabam freando o potencial destas técnicas. Em particular, a alta complexidade destes modelos, que frequentemente contem milhares, ou mesmo milhões, de parâmetros espalhados por várias camadas, dificulta a identificação das *features* que o modelo aprendeu a reconhecer e como elas afetam o seu processo de decisão. Sem tal conhecimento, as redes neurais se tornam essencialmente uma caixa-preta que dá pouquíssimo entendimento sobre o que o modelo está realmente fazendo. Isso é indesejável pois diminui a capacidade do usuário de interpretar e avaliar os processos internos dessas redes. Se nós conseguíssemos fazer isso, nós poderíamos usar tal informação para construir modelos não só com melhor performances, mas também com processos de decisão mais embasados, de forma que possam ser empregados em tarefas críticas — como aplicações médicas — onde entender como o modelo chega à uma certa conclusão é extremamente importante. Uma forma de abordar esses desafios é o emprego de ferramentas de análise visual que permitam a exploração e análise das *features* aprendidas pelas redes neurais. Muitas dessas ferramentas foram desenvolvidas nos últimos anos com o objetivo de abordar desafios em uma ou mais etapas do fluxo de desenvolvimento de uma rede neural. Neste trabalho, nós revisamos estas ferramentas de análise visual e introduzimos uma taxonomia para classificá-las de acordo com o tipo de análise que elas providenciam, seja a análise da arquitetura da rede, a avaliação do processo de treinamento, ou a análise e interpretação de quais *features* o modelo aprendeu a reconhecer. Após isso, nós ainda propomos uma nova técnica de visualização para guiar a escolha da arquitetura de redes neurais. Nós demonstramos, através de uma série de experimentos, como nosso método pode revelar múltiplas informações sobre como e se a arquitetura da rede deve ser modificada — por exemplo, adicionando camadas ou aumentando o número de neurônios — para melhorar a performance da rede.

Palavras-chave: Análise Visual, Aprendizagem Profunda, Redes Neurais.

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
RL	Reinforcement Learning
DNN	Deep Neural Network
DFN	Deep Fully-Connected Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
VA	Visual Analytics
AU	Architecture Understanding
TA	Training Analysis
FU	Feature Understanding
AVis	Architecture Visualization
AVal	Architecture Validation
EMM	Evolution of Model Metrics
RTA	Real-Time Analysis
FE	Feature Explainability
MI	Model Interpretability

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Artificial Intelligence (AI) is a field of Computer Science that focuses on the development of computational systems able to reproduce the reasoning, the decision-making process, and the learning ability that humans have [Bellman 1978]. These systems have to process information similarly as humans do, which requires them to perform intrinsically subjective tasks whose solutions depend on past experiences. Traditional algorithms struggle to achieve satisfying performance on such tasks because they follow an imperative behavior, i.e., they follow procedural commands that tells them precisely what to do and when to do every step of their calculation. They cannot adjust their behavior to task variations that were not specified by the programmer — even if only slightly different — and they are not able to use their past experience to improve their future performance. Humans, on the other hand, are quite good at performing such tasks. Once a human learns to perform a given task — e.g., to kick a football into a goal — it can quickly expand this knowledge to related but somewhat different tasks — e.g., to kick a bigger ball from a different position into a hole on the floor instead of a goalpost. Over the decades, the AI community has explored different ways to address these kinds of problems. In this work, we address one of such approaches: the employment of *Machine Learning* techniques.

Machine learning (ML) is a subfield of AI that includes any computational technique or method designed to learn how to perform a predetermined task by learning from experience [Mohri, Rostamizadeh e Talwalkar 2018]. Most ML methods resort to the analysis of a vast amount of data and the employment of probabilistic techniques to build a *model* that performs the task. To do so, these models infer knowledge not explicitly exposed in the data, but that can be useful to the prediction process [Samuel 1959]. Because the ML practitioner does not tell the model what to do but instead let it gather the knowledge from the available data, machine learning methods can generalize their behavior even for task variations they were not trained to perform. This ability makes them a viable alternative for many pattern recognition tasks that resort in subjective analysis, such as — but not limited to — object recognition [Guo et al. 2016], sentiment analysis [Ain et al. 2017], and robotic control [Garcia, da Silva e Comba 2017].

Over the past decade, a particular family of machine learning techniques, known as *deep learning* (DL), has been responsible for some of the most groundbreaking results in many recognition and prediction tasks. Deep learning techniques employ models called *deep neural networks* (DNNs). These models contain a sequence of layers

that perform multiple non-linear operations on the input data. Each of these layers contains a substantial— sometimes up to millions — amount of learnable parameters that can be adjusted to make the model perform better at the task at hand. As such, DNNs can model much more complex functions than other machine learning techniques — that usually have much fewer parameters — and, consequently, can achieve better performance in a wide array of difficult recognition tasks [LeCun, Bengio e Hinton 2015]. A non-exhaustive list of tasks and applications where deep models have recently achieved successful results includes image classification [Krizhevsky, Sutskever e Hinton 2012, Simonyan e Zisserman 2014, Szegedy et al. 2015, He et al. 2016], speech recognition [Graves, Mohamed e Hinton 2013, Bahdanau et al. 2016, Amodei et al. 2016], natural language processing [Sutskever, Vinyals e Le 2014, Cho et al. 2014, Kim 2014, Luong, Pham e Manning 2015], robotics and reinforcement learning [Lillicrap et al. 2015, Mnih et al. 2015, Hasselt, Guez e Silver 2016, Silver et al. 2017], object detection [Girshick 2015], image captioning [Donahue et al. 2015, Vinyals et al. 2015], image generation [Gregor et al. 2015, Karras, Laine e Aila 2018], and music generation [Huang et al. 2017, Engel et al. 2019].

Despite the outstanding results achieved by DNNs in the past decade, deep learning still faces some significant challenges that hinder their employment in many applications. First of all, there is no analytic way to find out which network architecture achieves the optimal performance in a prediction task [Goodfellow, Bengio e Courville 2016]. For this reason, DL practitioners often resort to a trial-and-error approach that requires many iterations of model tuning to reach a satisfying performance. Considering that training one iteration may already be significantly expensive — the training of some industrial-scale networks can take weeks, even in GPUs —, finding and training a network whose configuration yields a desirable performance quickly becomes very difficult and time-consuming.

The complexity of DNNs also makes it difficult to understand and to interpret the decision process followed by the model. The vast amount of learnable parameters in a DNN makes it challenging to keep track of all the operations the input sample is going through. For this reason, understanding which role each of the model's components plays in the decision process is very difficult or even outright impossible without the proper tools. Due to these limitations, DNNs are often seen as black-boxes [Liu et al. 2017], where little is known about how the model makes predictions. In most cases, it is difficult even for ML experts to explain why a model is not working correctly and what should be

changed to improve it.

Additionally, even if a model has excellent performance on a test set, understanding how the model comes up to a conclusion — e.g., which features of the input are considered and how relevant they were for the prediction — brings many advantages. It provides users with more confidence that the model is reasoning about the task just like humans would, and it allows the employment of these models in critical applications where knowing what led the model to a decision is essential, such as medical applications and autonomous driving. To put in perspective how important it is to achieve more interpretable models to the growing of deep learning, according to Fei-Fei Li — one of the most renowned researchers in the ML community — interpretability is one of the biggest bottlenecks that AI and DL face today and one of the research topics requiring more effort in the next years [Zeng e Wang 2017].

All of the challenges mentioned above raise several questions, so far, only partially answered by the state-of-the-art in DL research. Achieving efficient ways to answer these questions during the design of a DNN model is an essential step to reach more efficient ways to design and employ neural models in many real-world applications. Below, we list three questions that we believe are more related to the techniques addressed in this contribution.

- *Which features did the model learn to recognize*? Much of the success found in DNNs lies in their ability to transform the input attributes into new and more abstract features that are more suitable for the prediction task. As such, DNNs can model much more complex functions, but it also makes it more challenging to understand the meaning of these functions. The high-level features learned by the network — i.e., the internal representations of the input sample — are often not entirely obvious for designers and users, which makes it hard for them to understand what the model learned to recognize in the input samples. This fact severely limits the interpretability of such models, because users have no way to know what exact information in the sample the model takes into account when performing the prediction task. Additionally, it makes models much less trustworthy, as there is no way to build confidence that the model is going to behave as expected in unknown situations. Looking at the model's performance in a validation set may be enough for regular applications, but it is not when the model in hand performs a critical task where mistakes may not be tolerable.
- *How does the model use such features to build predictions*? Once the designer

knows which features the model learned to recognize, a question that follows naturally is how does the model uses these features to make predictions. Neural networks contain a sequence of layers, each applying a non-linear operation to the data received from the previous one. In such a setting, what each layer essentially does it to transform the representation of the data into a new set of features — usually more abstract and more high-level. As such, it is not trivial to understand how the network transforms one set of features into the next one and how this process leads to the final prediction. Understanding how the model builds high-level features from low-level ones and uses them to predict labels is a central aspect of achieving a satisfying degree of model interpretability. Additionally, such an understanding allows the designer to make more confident assertions about whether the current decision process is adequate or if the model should be modified to achieve a more effective one.

- *How did the training process converged to that configuration*? Neural networks resort to non-convex optimization techniques to improve their prediction performance, which makes it impossible to ensure that such training process will lead to the optimal configuration. Additionally, the training of DNNs is computationally expensive, making it prohibitive to perform it too many times. Due to these limitations, it is desired that designers could quickly spot issues on the training process and fix them. However, such analysis is complicated due to the already mentioned complexity that DNNs often display. Alternatives to mitigate these issues and allow designers to find out problems in the training process — or in the resulting parameter configuration — is an essential asset to increase the usability of DNNs.

The concerns above were raised by the DL community multiple times over the past few years, and there is a clear belief that future advancements in the field require better approaches to address these issues [Ribeiro, Singh e Guestrin 2016, Lipton 2016, Samek, Wiegand e Müller 2017, Zeng e Wang 2017, Marcus 2018]. Such concerns led to the research of alternatives to provide better interpretability, understanding, and training guiding to a DNN designer [Vellido, Martín-Guerrero e Lisboa 2012, Ribeiro, Singh e Guestrin 2016, Samek, Wiegand e Müller 2017, Arras et al. 2017].

From this initial effort, a particular class of techniques has stood out as an efficient tool to achieve such model understanding: *Visual Analytics* (VA) techniques. These techniques have proved to be very useful in giving insights to designers and users about the inner workings of DNNs at multiple levels of their design workflow [Garcia et al.

2018, Hohman et al. 2018]. Indeed, the support of DL engineering has become one of the most researched topics in the VA community recently. As an example, around one-quarter of the publications presented at IEEE VAST 2017 — one of the main venues for VA research — addressed VA support of neural networks [IEEE-VAST 2017 Symposium 2017]. The proposed techniques are quite diverse, addressing several of the problems encountered when developing DNNs — in particular, the three questions listed above.

In particular, some VA tools provide support to analyze the network's *architecture*, evaluating how each component (neurons or layers) is contributing to the prediction performance and how they can be modified to improve such performance [Liu et al. 2017,Liu et al. 2018, Garcia et al. 2019]. Another focus is the analysis of the *training process* of DNNs, helping designers to understand how the DNN arrived at some prediction behavior and giving insights about issues during the learning process [Qi et al. 2017, Cashman et al. 2017]. Nonetheless, the task in which visualization tools have found more popularity is the analysis of the *features* learned by the model. These VA tools provide ways to interpret the model decision process and to explain which features the model is considering in its predictions, both from a model and a sample perspective [Liu et al. 2017, Kahng et al. 2018, Pezzotti et al. 2018, Rauber et al. 2017, Zeiler e Fergus 2014, Yosinski et al. 2015].

Such tools have already proved to be very useful for the analysis of DNNs. They can aid designers to reach some key and non-trivial conclusions about the inner workings of DNNs and to validate some previously widely-held beliefs that lacked empirical evidence. For example, with the help of VA tools, it was possible to discover that even neurons from hidden layers contain interpretable and class-specific features whose role in the decision process can be recognized [Zeiler e Fergus 2014]. Another interesting finding guided by VA tools is that the layers in a DNN often learn implicit class hierarchy present in the training data. This way, samples from very unrelated classes — e.g., cars and trees — often produce far different activations in the first layers of the DNN, while samples from classes that are closely related — e.g., horses and zebras — only create recognizably different activations in deeper layers [Alsallakh et al. 2018].

Despite the effectiveness in addressing some of the major problems faced by DL engineering, the employment of VA techniques in this field is still very recent. Therefore, the related literature still has a noticeable gap concerning surveys and reviews covering visual analytics techniques for deep learning support. In particular, there was no well-defined methodology to classify these tools according to which specific tasks in DL engineering they support. To alleviate this issue, we propose a task-and-technique based

taxonomy that encompasses tools resorting in VA to support the analysis, training, and interpretation of DNNs [Garcia et al. 2018]. Our survey, published at the Computers & Graphics journal [Computers & Graphics 2018], classifies proposed VA approaches according to the tasks (and subtasks) they aim to support and to which model architecture they address. To design this taxonomy, we reviewed around 40 papers and contributions comprising DNN visualization support. Such works were all published in the last decade — and in particular in the previous five years.

While working in the mentioned survey, we identified several challenges in DL engineering that are yet to be adequately addressed by VA techniques. We discuss these open gaps and present arguments for why these challenges can benefit from VA support. One of these open challenges relates to the architectural tuning of DNNs. Most existing VA tools do not provide clear alternatives to answer questions such as: *Does the model need more layers*? *Does it need more neurons in a given layer*? Or even *Does the features learned across its neurons contain redundancy*? To address this issue, we developed a novel VA tool to guide DNN designers in answering such questions during model development [Garcia et al. 2019]. Our method employs a novel visualization technique, called *Activation Occurrence Maps*, to display how often neurons activate to features found on different classes. With such information, designers can identify if the learned features are indeed enough to let the model successfully distinguish all the classes in the training set. When this is not the case, the visualization can act as a guide to identify which components of the network are responsible for the suboptimal decision process and infer which architectural modifications could improve the DNN performance. With a series of experiments, we demonstrate how our method can be used to guide the design of DNN from scratch, as well as improving pre-existent models. This work will be published in the 2019 IJCNN (International Joint Conference on Neural Networks) [IJCNN - International Joint Conference on Neural Networks 2019].

The structure of this dissertation is as follows. In Chapter 2 we describe the related work both in a perspective of general advancements in deep learning research and in how visualization can support the engineering of DL models. In Chapter 4, we introduce our task-and-technique taxonomy to classify current approaches in VA for DL. In Chapter 5, we present a new visualization method to guide the architecture tuning of neural networks and explain how it differs from the pre-existing techniques. Finally, Chapter 6 concludes the dissertation.

2 RELATED WORK

Even though deep learning has produced many real breakthrough only in this decade, the field itself is not exactly new. The idea of using the human brain as an inspiration to learning methods go back as far as the 1940s and 1950s [McCulloch e Pitts 1943, Easterbrook 1959]. The *perceptron* [Rosenblatt 1958], considered to be the first precursor to modern DNNs, was proposed in this period. The perceptron already employed the idea of using learnable parameters to model a function performing a task. However, the model was still relatively simple and could only model linear functions, and thus was unable to handle complex tasks. Notably, Minsky et al. [Minsky e Papert 1969] proved that a perceptron could not possibly learn the XOR function, which significantly decreased the interest in neural networks for a reasonable period.

At this point, the development of more complex ML models was halted both by the need for more powerful computers and the lack of appropriate training methods. The second issue was coped with the advent of the *backpropagation* algorithm [Rumelhart et al. 1988], a significant turning point in the history of ML research. This method allows networks with multiple layers to optimize their weight configuration in a much more efficient way, which effectively means that, theoretically, a big enough network could model any function.

After that, it did not take long for neural networks to achieve successful results in practical settings. Two years later, Le Cun et al. [LeCun et al. 1990] introduced the concept of convolutional layers, demonstrating how it could be used to achieve good performance in simple computer vision problems, such as recognizing handwritten digits in an image. Unfortunately, the excessively computational cost required to train large neural networks meant that the field would still stay away from the spotlight for more than a decade.

Only in the first years of the 21st century that neural networks began to significantly increase in usage, mainly due to the high computational power achieved by modern computers. In particular, the increased use of GPUs was quite beneficial to the training of neural networks, as they allow computers to perform many operations in parallel, which mainly decreases the time needed to train most DNNs [Raina, Madhavan e Ng 2009]. However, the deep learning terminology was not coined until 2006, when Hinton et al. [Hinton, Osindero e Teh 2006] introduced *deep belief networks*. The greedy designing method of these networks increased the number of layers as needed during training,

which brought to life the concept of *deep* neural networks.

The first main breakthrough produced by DL happened in 2012 with the development of the AlexNet model [Krizhevsky, Sutskever e Hinton 2012]. This DNN architecture won the 2012 ImageNet Large Scale Visual Recognition Competition (ILSVRC). In this competition, the goal is to build a classification model on a widely known image classification dataset called ImageNet [Deng et al. 2009], comprised of millions of images, each belonging to one of around a thousand classes. By using convolutions in the first layers of the model, AlexNet achieved a 15.3% top-5 error rate — i.e., how often the correct class is not among the five more likely classes according to the network's prediction — an improvement of 10.8% with respect to the runner up competitor method. In the years following AlexNet's development, several authors built upon the original model to improve even further the performance achieved on the ImageNet dataset [Zeiler e Fergus 2014, Simonyan e Zisserman 2014, Szegedy et al. 2015, He et al. 2016], achieving top-5 error rates as low as 3.6% [He et al. 2016]. The impressiveness of these results established DNNs as the standard technique to perform image classification today.

Deep neural network's exceptional results in image classification served as the kick-off to countless DL approaches achieving state-of-the-art results in other areas as well during the following years [Schmidhuber 2015]. Deep models quickly became the main approach for tasks such as *object recognition* [Girshick 2015] and *video classification* [Ng et al. 2015], mainly due to how effective convolutional layers are in recovering meaningful features from pixel values. The usefulness of DL is not limited to image-related tasks though. With the advent of recurrent architectures [Hochreiter e Schmidhuber 1997, Cho et al. 2014], DL proved to be an effective tool for long-term learning tasks as well, such as *text classification* [Zhang, Zhao e LeCun 2015], *speech recognition* [Hinton et al. 2012, Graves, Mohamed e Hinton 2013], *machine translation* [Cho et al. 2014], along other *natural language processing* (NLP) tasks [Lopez e Kalita 2017, Young et al. 2018].

However, the current advancements in DL are not restricted to practical results, but also the development of novel techniques. Innovations such as the ReLU activation function [Dahl, Sainath e Hinton 2013], dropout regularization [Srivastava et al. 2014], and batch normalization [Ioffe e Szegedy 2015] were essential to many of the groundbreaking results DNN have achieved in the last few years. These techniques significantly improved DNN performance by making them faster to converge, more robust to overfitting, and better at generalizing predictions. Additionally, the launching of several high-

level frameworks aiming DNN modeling [TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems 2015, Al-Rfou et al. 2016] transform DL in a more democratic field, significantly increasing the number of people working in the topic.

Lately, DNNs have been used to address problems outside of the supervised learning scope as well. The advent of generative networks [Goodfellow et al. 2014, Arjovsky, Chintala e Bottou 2017] brought outstanding results in learning representation [Radford, Metz e Chintala 2015] and generative tasks, such as *music* [Huang et al. 2017, Engel et al. 2019] and *image caption generation* [Vinyals et al. 2015]. Additionally, DL is behind many of the recent advances in *reinforcement learning* (RL) problems [Mnih et al. 2015], an ML field that aims to build agents that learn to perform actions according to the state or environment they are exposed to [Sutton e Barto 1998]. By combining traditional RL techniques with the generalization power provide by DNNs, some impressive results were achieved, such as the development of AI agents that can beat world-champions human players in complex games such as Go [Silver et al. 2017].

2.1 Visualization Support for Deep Neural Networks

Despite all the outstanding results of DL approaches, several authors have pointed out concerns about the limits of DNNs. For a start, neural networks have a series of requirements that severely restrict the applications where they can be employed. In his work, Gary Marcus [Marcus 2018] presents a list of challenges currently faced by DL and argues that, unless these challenges are successfully addressed, DL may soon approach a wall that will scarce innovations out. In his list, he makes a stand for the following issues: (1) the training of DNNs requires a vast amount of (labeled) data; (2) transferring knowledge from problem to problem is still hard; (3) difficulty to learn hierarchical structures in an explicitly way; (4) struggle to draw open-end inferences; (5) *lack of model transparency*; (6) difficulty to integrate prior knowledge into the model; (7) models cannot distinguish correlation from causation; (8) lack of robustness to unstable environments; (9) *untrustworthy predictions*; and (10) *model designing is intricate and time-consuming*. Samek et al. [Samek, Wiegand e Müller 2017] and Ribeiro et al. [Ribeiro, Singh e Guestrin 2016] makes additional arguments about the importance of interpretable and explainable models in many applications.

Currently, visual analytics methods are one of the main tools to address some of these issues [Garcia et al. 2018, Hohman et al. 2018], namely challenges (5), (9), and (10)

raised by Marcus, along with the need for interpretability raised by Samek and Ribeiro. The use of VA in DL is mainly motivated by the former's ability in providing the analysis and understanding of high-dimensional [Liu et al. 2016, Maaten, Postma e Herik 2009] and temporal data spaces [Elman 1990], a chief characteristic of DL-related data.

In fact, VA has been a useful tool for ML support outside the DL-scope as well [Liu et al. 2017]. Visualizations have been used in ML to address problems such as: classification [Talbot et al. 2009, Ribeiro, Singh e Guestrin 2016, Rauber, Falcão e Telea 2018, Ren et al. 2017, Alsallakh et al. 2014] and regression performance analysis [Mühlbacher e Piringer 2013], feature selection and engineering [Krause, Perer e Bertini 2014, Rauber et al. 2015, Brooks et al. 2015], clustering evaluation [Tatu et al. 2012], decision tree analysis [Elzen e Wijk 2011] and interactive machine learning [Jiang e Canny 2017, Paiva et al. 2015, Amershi et al. 2014]. Although many of these approaches can also be used in DL engineering — e.g., to evaluate the network's performance —, they do not address the main DL-specific issues mentioned above, which is the main interest in our research.

One of our aims in this work is to design a taxonomy that categorizes VA tools according to which tasks and techniques they address in the engineering of DL models. This is a topic that has not been properly addressed in the literature yet. Before the publication of our survey [Garcia et al. 2018], some tutorials had surveyed over some techniques [Montavon, Samek e Müller 2018, Samek et al. 2017, Yeager et al. 2016, Zeng 2016, Seifert et al. 2017, Grün et al. 2016], but those only tackled a relatively small scope of applications and techniques. Others have addressed related topics, but that do not fall in the same scope of VA for DL support we aim for, such as predictive visual analytics [Lu et al. 2017]. Liu et al. [Liu et al. 2017] address the broader topic of using VA to support ML development in general, but they do not provide an extensive list of tasks on the design workflow of DL models in which VA can be employed. Zachary Lipton [Lipton 2016] provides an interesting and very related discussion about the definition of interpretability in ML, but his work does not aim to build a taxonomy for techniques in the field.

The closest survey to our goal was proposed by Hohman et al. [Hohman et al. 2018]. They follow a human-centered approach to answer questions such as: (1) why to visualize different aspects of the model and its training process?; (2) who employs visualization support for DL?; (3) what should we visualize in DL?; (4) how to visualize it?; and (5) in which step of the design process does the visualization occur? For each of these questions, they describe the motivation for why answering such a question is important and give examples of how VA can provide answers for them. However, they do not pro-

vide an extensive list of the techniques and contributions that have been employed for DL support and which tasks they address, which we do. Additionally, their survey focuses on readers from a ML background that may not have a deep knowledge of VA techniques. We addressed the inverse need. Our survey focuses on explaining the usefulness of VA in DL to VA experts that may not be fully aware of the current state of DL research and challenges.

3 MACHINE LEARNING AND DEEP NEURAL NETWORKS

In this chapter, we give a brief introduction to machine learning as a field. We explain its general concepts and the intuition that led researchers to invest time on such techniques. We also describe the main types of machine learning tasks, their goals, the main algorithms in each group, and in which applications they can be employed. In particular, we address the neural network architectures used in DL and how they differ from classical ML approaches. Finally, we detail the design workflow followed by machine learning practitioners when designing a model for a prediction task. This topic is particularly relevant for this contribution because the VA approaches we consider in our survey — and also the one we propose later — mostly aim to address issues and challenges faced during the design workflow of neural networks.

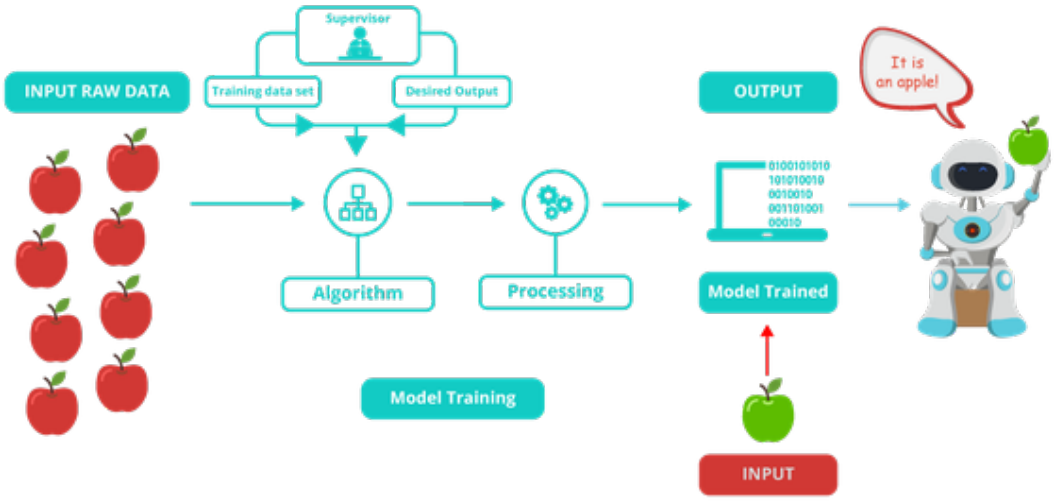
3.1 Classical Machine Learning

Machine learning main goal is to solve problems whose solutions cannot be described with precise instructions [Murphy 2012]. In its most general definition, the ML field includes any technique that acquires the knowledge needed to solve a task from some experience, usually represented by a — often huge— amount of data [Mohri, Rostamizadeh e Talwalkar 2018]. An ML method gathers such knowledge by using statistical techniques to identify patterns in the data. Then, these patterns can be used either to explain some underlying structure present in the data and or to teach the model how to perform the prediction task. If this is done correctly, the ML model can generalize to novel samples, i.e., the model can apply the same reasoning learned with the training data to perform the task on data unavailable during training [Flach 2012].

The learning tasks addressed in ML are very diverse, and their reasoning and goals differ a lot from task to task. Usually, learning tasks are classified according to the *learning scenario* they address [Mohri, Rostamizadeh e Talwalkar 2018]. What defines the scenario to which a task belongs is the type of training data the model has access to and which kind of prediction it aims to perform. In most definitions, learning tasks are classified into at least two main scenarios: *supervised* and *unsupervised* learning [Murphy 2012].

In the former, the samples in the available training data have associated labels stating what is the expected output for that sample. By analyzing the patterns in the

Figure 3.1: Supervised Learning Methods

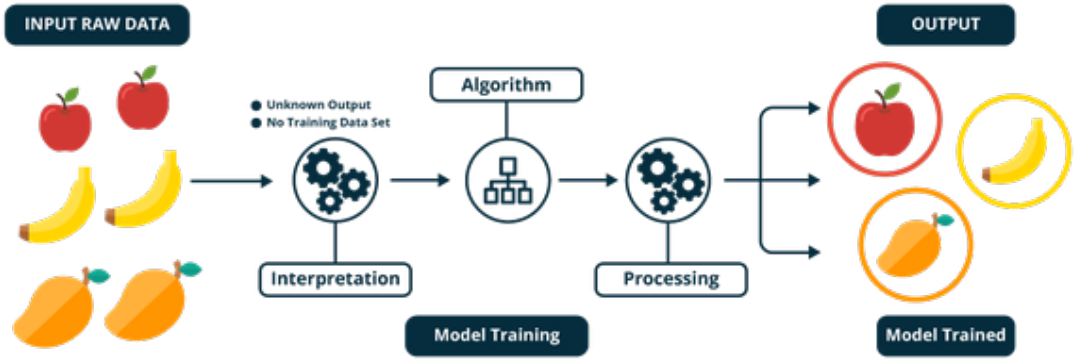


A supervised learning algorithm trains a model to make predictions based on an input sample. During training time, it receives a large amount of labeled data — i.e., both samples and expected labels — and use this data to model a function mapping samples to label. It does so in a way it can generalize predictions to samples the model did not have access during the training process [Vaseekaran 2018].

correlation between samples and labels, a supervised model aims to learn how to predict the correct labels, so it can be used to predict labels to samples whose labels are unknown. Figure 3.1 displays the overall prediction behavior of a supervised model. Supervised tasks include (1) *classification*: to predict which class a sample belongs to — e.g., whether an image depicts a cat or a dog; (2) *regression*: to predict a continuous value associated to the sample — e.g., the price of a house giving several attributes such as number of rooms and location; (3) *forecast*: by analyzing previous values, to predict the next value in a time series — e.g., predicting the price of a company's share in the stock market from its values in previous days; and (4) *ranking*, to learn how to rank a list of samples from better to worst — e.g., a system to recommend web pages according to the user's preference.

In the unsupervised setting, instead, the model does not have access to training labels, and its goal is to find some underlying structure containing in the data. Figure 3.2 displays the overall behavior of an unsupervised model. Some examples of unsupervised tasks are (1) *clustering*: to split the training samples into groups (clusters) whose members share some similarity according to a giving metric; (2) *data generation*: to learn the distribution where the training data lies and to build a model that can generate novel synthetic samples from that same distribution; (3) *anomaly detection*: to identify unique

Figure 3.2: Unsupervised Learning Methods



Unsupervised learning algorithms do not rely on known labels. Instead, they aim to find interesting structures, such as clusters, in the data. Usually, these algorithms analyze all the data points and assign a result value to each of them, which could be, for example, the assigned cluster or a low-dimensional representation for that data point [Vaseekaran 2018].

structures in the training data that does not match the behavior seen in the rest of the training set; and (4) *dimensionality reduction*: to find a more compact data representation that keeps the structures and similarities present in the high-dimensional space.

Although these are the two main learning scenarios described in the literature, others do exist. Some learning tasks do not fit precisely in neither setting, which led to the definition of other, more specific, learning scenarios, such as (1) *semi-supervised learning*: when typical supervised models are trained with data that is not fully labeled — i.e., the associated labels for some samples are unknown —, usually due to how expensive or difficult it may be to gather labeled data in real applications; (2) *self-supervised learning*: a supervised learning task where the label for each sample is taken from the sample itself — e.g., autoencoder techniques [Vincent et al. 2008] have as a goal to learn how to reconstruct the input sample in the output. In this case, the sample is also its label; (3) *active learning*: the model can query an information source — e.g., a human expert — about the correct output during training; and (4) *reinforcement learning*: an agent learns to choose actions to perform according to the environment state it is in, the training happens with no previous knowledge of which are the best actions to take in each state.

Neural networks are, in essence, supervised models. They learn how to predict the correct output by having access to a large number of labeled training examples ex-

plaining how the model should perform. However, they are often employed in other scenarios as well. For instance, the autoencoders mentioned earlier can be used to perform dimensionality reduction if the internal layer has a smaller dimensionality than the original sample [Hinton, Osindero e Teh 2006]. Other unsupervised tasks that DNNs can address include clustering [Caron et al. 2018] and data generation [Radford, Metz e Chintala 2015, Arjovsky, Chintala e Bottou 2017]. The latter is, in fact, one of the leading research topics in the DL today. Architectures such as the *Generative Adversarial Networks* (GANs) [Goodfellow et al. 2014] employ two networks to, respectively, generate data and to discriminate real samples from generated ones. This way, the two networks fight against others during the training process, as the first becomes increasingly better to create synthetic samples similar to real data and the second network becomes increasingly better to distinguish them. Reinforcement learning (RL) is another learning scenario that is benefiting greatly from DL [Lillicrap et al. 2015, Mnih et al. 2015, Hasselt, Guez e Silver 2016]. One of the main problems with traditional RL methods is the unreasonable amount of memory needed to store every state-to-action pair when the number of possible actions or states is too high [Sutton e Barto 1998]. Deep learning mitigates this issue because DNNs can act as a model to the agent, generalizing chosen actions to states that are very similar but not quite the same as states seen previously.

Given that DNNs are intrinsically supervised models, from now on in this work, we will focus on the supervised learning scenario and, in particular, the classification setting. However, it is essential to note that almost all the techniques presented here can be used for DNNs employed in non-supervised tasks as well. We start by giving a more detailed definition of the supervised setting next.

3.2 Supervised Learning Scenario

In the supervised learning setting, each training or test set sample $\mathbf{x} \in X$ — where X is the distribution from where samples are drawn — has an associated label $y \in Y$ that describes what is the expected value that the model should output whenever it receives $\mathbf{x}$ as an input. When Y is a continuous distribution of real-valued numbers, the supervised task is described as a *regression task*. On the other hand, when Y is a finite set of discrete values, the supervised task becomes a *classification task*. In such cases, the labels in Y are called classes. Classes can be seen as groups of elements in $\mathbf{x}$ that share some similarity. Usually, each element $\mathbf{x}$ can be associated with at least one class in Y. For example, in a

typical image classification task $\mathbf{x}$ is an image and each $y \in Y$ corresponds to one of the possible objects depicted in the image. In many practical applications, each label y in a set of d possible labels is represented by a label vector $\mathbf{y} \in 0,1^d$, where the i-th element of $\mathbf{y}$ is set 1 only if $\mathbf{y}$ corresponds to the i-th label in Y. This process is usually known as *one-hot encoding*.

In both supervised learning settings, the main goal of the ML method is to train a model that can successfully replicate the correlation between elements in X and labels in Y — i.e., a model that, given an input sample $\mathbf{x}$, is able to identify the label y associated to $\mathbf{x}$ even for samples in X that were unavailable during training. In a more formal definition, given a training set $D = \{\mathbf{x}_i, y_i\}$ for $i \in \{1,\ldots,N\}$ — i.e., a set of N samples associating inputs in X to labels in Y —, the goal of a supervised learning method is to model a function $f : X \rightarrow Y$ that successfully maps elements in X to their corresponding labels in Y.

Supervised methods often optimize their performance by incrementally modifying the function f to minimize a cost function C that measures how well f can map elements in X to their respective labels in Y. Typical cost functions include *mean squared error* for regression models and *categorical cross-entropy* for the classification setting. In most classical ML techniques, f is easier to learn if the model receives a pre-processed representation of the data sample instead of its raw format. For examples, it is much easier to model a function to classify images if, instead of raw pixel values, the model receives as input information about the object's edges. This pre-processing requires the transformation of inputs $x \in X$ via a feature function $\Psi : X \rightarrow \mathbb{R}^s$, where s is the dimensionality of the transformed representation. Each of the s values in $\Psi(\mathbf{x})$ is called a *feature* of $\mathbf{x}$. The i-th feature of $\mathbf{x}$ is then denoted as $\Psi^i(\mathbf{x})$. In general, manual feature engineering is undesirable as it is hard, time-consuming, and often inefficient. In particular, one of the big advantages of DNNs is the fact that these models can learn appropriate feature representation automatically, avoiding this step in the ML design workflow.

In most ML methods, the designer has to specify the space of possible functions that the model can learn. This space is represented by a set of *model parameters* (*weights*) $\Theta = \theta_1, ..., \theta_m \in \mathbb{R}^m$, where m is the number of parameters in the model. Models with a higher number of parameters can encode more complex functions and therefore, can address more difficult problems. However, a large number of parameters also results in models that are harder to interpret and significantly increases the risk of *overfitting*. When a model overfits, it performs very well on the samples used for training model,

but fails to generalize this performance to novel samples. This problem happens because, if the model has too many parameters, it can easily come up with a very complicated function that can fit almost perfectly every training example, but that does not represent the actual underlying function existing in the data distribution. The overfitting problem is the main reason behind the splitting of the training data into training and test set, as it allows the practitioner to test the model's performance in data whose labels are known by the practitioner but that the model did not have access to during training.

During training, the model adjusts its model parameters Θ, improving how well the function f fits the correlation between X and Y. They do so by searching for a parameter configuration that minimizes the cost function C. At each training iteration, the cost function evaluates how different the predictions made for the samples in D — or a subset of them — are from their actual labels. If the predictions are too different from what they should be, the model modifies Θ significantly, in the hope that this will result in a better performance than before. Otherwise, when the predictions are relatively similar to the expected labels, the parameter configuration does not change much.

Before starting the training of the model, the ML practitioner has to choose the model's *hyper-parameters*. These are fixed pre-determined values that must be set before training and will interfere in the behavior of the model during training. Hyper-parameters vary a lot from algorithm to algorithm. However, an example of a hyper-parameter that appears in almost all machine learning techniques is the *learning rate*. This value affects how much Θ change when a prediction error occurs during training.

Some classical supervised learning techniques other than neural networks include: *k-nearest neighbors* [Murphy 2012]; *random forests* [Breiman 2001]; *logistic regression* [Murphy 2012]; *support vector machines* [Bishop 2006]; and *decision trees* [Hastie, Tibshirani e Friedman 2009].

3.3 Deep Neural Networks

Neural networks differ from classical ML methods regarding how they represent the prediction function f. These models are typically organized in a sequence of $L > 2$ layers. The role of each layer is to further transform the original sample into a new representation that is more suitable for the prediction task. In other words, layers successively extract information from the previous representation to build higher level and abstract *features*. The number of layers, the type and the size of each layer is all chosen by the DL

practitioner before training starts.

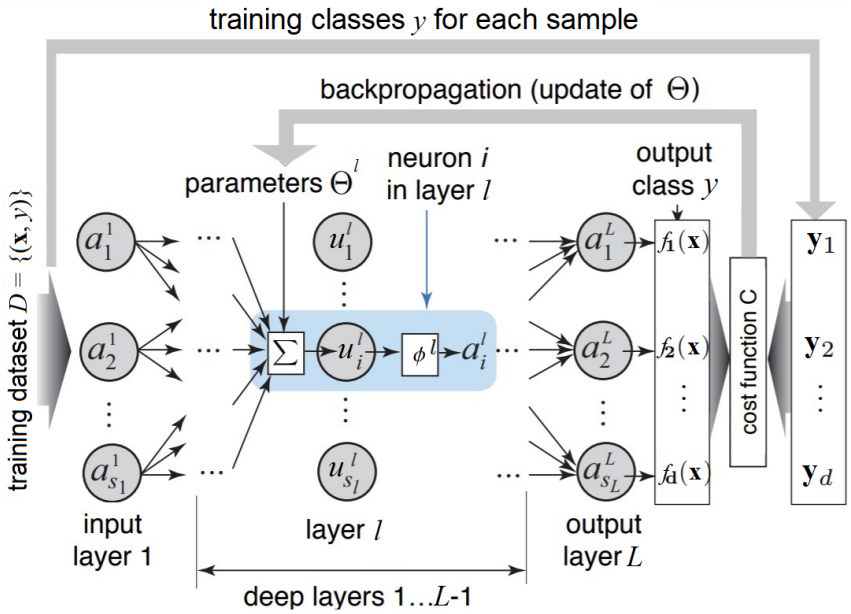
Each layer l in a DNN has a size s^l that defines the number of *neurons* (also known as units) $u^{l,i}$ the layer has. The first layer $l = 1$ in a DNN is called *input layer*. The input layer receives as input the original sample — e.g., an image. Each neuron u in the input layer applies its set of parameters $\Theta^{1,u}$ on the input sample to produce an output vector $a^{l,u}$, usually known as *activations*. The activation of the input layer then serves as input to the second layer $l = 2$ in the model, which, in turn, will also produce its own activation values and pass it to the next layer. This behavior is repeated until the last layer $l = L$, known as *output layer*. The output layer's role is to perform the actual prediction, i.e., the label y associated with the input sample $\mathbf{x}$. Any layer in between the input and the output layer is known as *hidden layer*. The actual operation performed by the neurons of a layer depends on the layer type and the *activation function* — a non-linear operation performed in the output of a neuron, before sending the activation to the next layer. Figure 3.3 shows a more detailed overview of how neural networks work.

As most supervised methods, DNNs are also trained via the optimization of a cost function C, usually via gradient descent. However, the multi-layer structure present in DNNs makes the update of the parameters Θ a bit more complicated. To properly update weights in the first layers of the model, DNNs resort in an algorithm known as *backpropagation* [LeCun et al. 1989]. After updating the parameters in the output layer via gradient descent, this algorithm propagates the gradient of the error calculated in the current layer to the previous layer. It does so for every layer in the model until it reaches the input layer, updating every weight in the Θ.

Most DL applications require a considerably large amount of training data. In such cases, each backpropagation pass — i.e., a single Θ update — would be quite expensive, as it would require the model to predict labels for the entire training set, slowing the training process convergence. A common approach to mitigate this issue is to split the training set in B *mini-batches*. This way, the backpropagation is performed once for every mini-batch, which increases the number of updates, accelerating convergence. The processing of all B mini-batches is called an *epoch*. The number of epochs for which the training occurs is usually chosen as a hyper-parameter by the DL designer.

A neural network can have multiple kinds of *architectures*. The architecture of a DNN defines how many layers the model has, which is the size of each layer — regarding the number of neurons —, and in particular, which *operation* the layer performs. Choosing the right layer configuration is essential to achieve a satisfying performance, as some

Figure 3.3: A Typical Neural Network Architecture



A neural network contains $L > 2$ layers. The first one, the input layer, receives raw samples (e.g., images) as input. By combining the sample with its weights, the first layer produces an activation vector a^1. This activation vector serves as input to the second layer to compute its own activation vector a^2. This process continues through all hidden layers — i.e., all layers that serve neither as input or output to the model — until the activation flow reaches the output layer. The output layer's activation vector a^L is the label predicted by the model. In typical multi-class classification tasks, the output layer contains one neuron for each possible class. When the model makes a prediction, the chosen class is the one whose neuron in the output layer activated the highest activation value. During training time, the model compares the predicted label with the actual one and measures a cost error stating how good is the prediction. Finally, the network updates its learnable weights to minimize such cost error. Differently from other ML techniques, the weights of a neural network are distributed across multiple layers. To properly update all these weights, neural networks have to resort in a backpropagation technique that propagates the error from deeper to shallower layers, updating the parameters in every layer in the model.

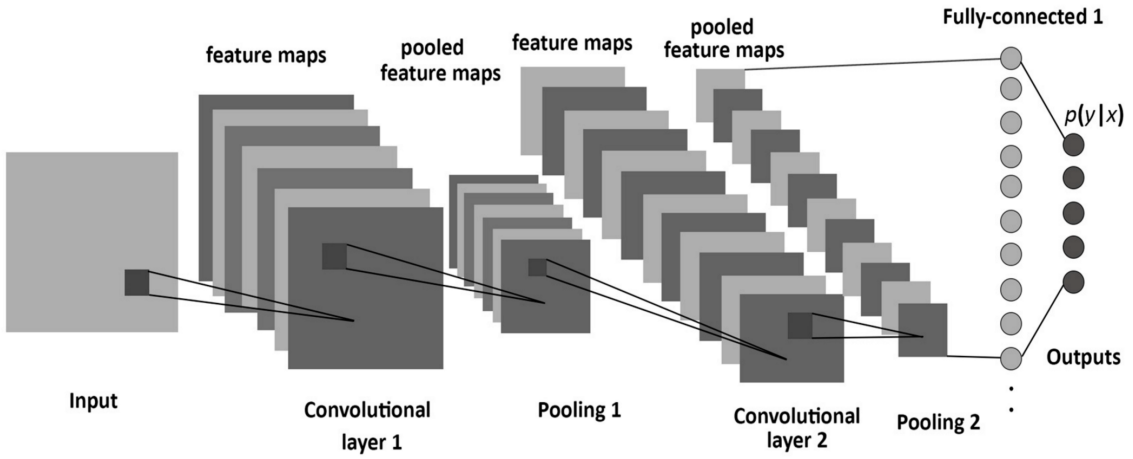
applications often require a specific type of layer. For instance, image classification tasks are usually addressed by networks with convolutional layers, as these layers can better identify spatial features regardless of where they appear in pixel space. In this work, we mainly focus on three typical architecture that is widely used in DL research: *deep feedforward networks* (DFNs): networks that only contain fully-connected layers; *convolutional neural networks* (CNNs): networks with convolutional layers; and *recurrent neural networks* (RNNs): networks with some recurrent layer. It is important to note that network architectures often contain more than one type of layer. For instance, a video classification model may use convolutional layers to learn better representations for each frame, recurrent layers to learn temporal features, and fully-connected layers to perform the classification task in the last layer of the model. Next, we describe in more details each of these three architectures.

Deep feedforward networks: The most traditional DL architecture. In DFNs the neurons are all fully-connected to the adjacent layers. A fully-connected neuron u^l in the l-th layer of the model receives as input the activation vector $\mathbf{a}^{l-1}$ containing all the activations produced by the neurons in the previous layer $l-1$. By applying its parameters Θ^{l,u^l} to $\mathbf{a}^{l-1}$, the neuron u^l produces a single activation value a^{u^l} that will be passed down to every neuron in the next layer $l+1$ — along with the activation values produced by all other neurons in layer l.

An essential characteristic of fully-connected layers is that each neuron outputs a single activation value. This way, a fully-connected neuron's role can be seen as to identify a higher level feature by somehow combining all other features in the previous data representation. Additionally, the number of neurons in a layer defines the dimensionality of the intermediate representation that layer is going to create. If a fully-connected layer has, for example, ten neurons, that means it transforms the data it receives from the previous layer to a new 10-dimensional representation containing ten features. Figure 3.3 shows in more details the architecture of a typical DFN.

Convolutional neural networks: Many recent DL applications in problems such as image classification and object recognition have achieved breakthrough performances by employing CNNs [LeCun et al. 1989]. Any model that falls in this category contains at least one convolutional layer in its architecture. As the name suggests, the goal of a convolutional layer is to apply convolutional operations in the data received from the previous layer to find small-scale patterns that may help the prediction future. By stacking multiple convolutional layers, the model can successively builds more complex features that can

Figure 3.4: A Typical CNN Architecture

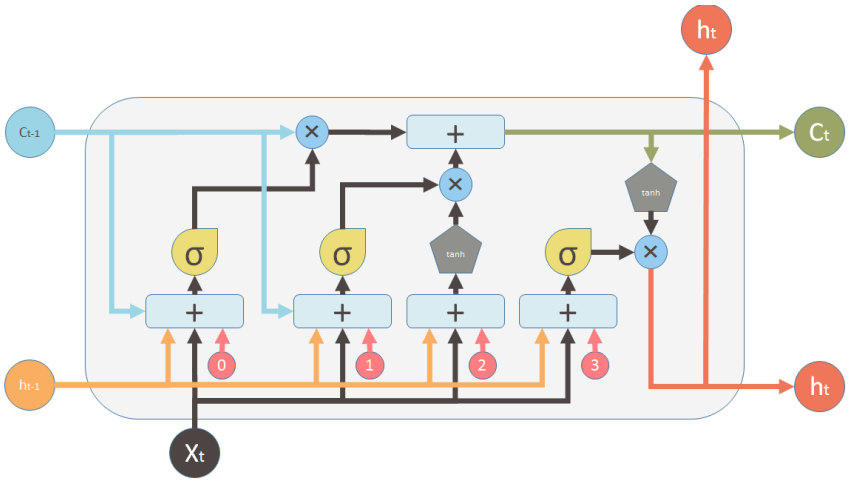


Like DFNs, CNN operate as a sequence of functions, encoded by the layers, that change the data representation to make prediction easier. However, CNNs employ convolutional layers for that. These layers apply convolutional operations to the received data. Their activations are the result of such convolutions. As a result, each convolutional neuron — sometimes called filter — outputs a *feature map* instead of a single activation value. These feature maps usually lie on the same space of the original input — e.g., feature maps from an image classification model are also images. Another important characteristic of CNNs is the usage of pooling layers. These layers often appear after a sequence of convolutional layers and their role is to shrink the number of dimensions in the feature maps, keeping only the relevant values in them. By doing this, the model can compress feature maps without losing essential information that future layers will need. After the convolutional step of the model, the resulting features maps are often flattened to a regular activation vector so it can become the input of a fully-connected layer that further prepares the data representation for classification (or regression) [Yu 2018].

unveil even more abstract patterns present in the hidden process, extracting data representations that allows a much more efficient prediction performance than the original input. For instance, in an image classification CNN, the first convolutional neuron often learns to identify very low-level features such as edge orientation and color contrast. In the following convolutional layers, however, the model can use this low-level features to build more abstract ones, such as complex objects like faces, animals, or vehicles. Convolutional neurons have some important differences from fully connected ones: (1) it has a much smaller number of weights. While fully-connected neurons have one weight for every feature received from the previous layer, a convolutional neuron only has a small kernel that does not depend on the input's dimensionality; and (2) it produces high-dimensional activations — usually named *feature maps* — instead of scalar values [Liu et al. 2017]. Figure 3.4 further details how CNNs works.

Recurrent neural networks: DFNs and CNNs are usually not suitable for applications

Figure 3.5: A Typical RNN Architecture



Recurrent layers keep an internal memory to store information from previous timesteps. The picture depicts an LSTM cell [Hochreiter e Schmidhuber 1997], one of the most popular recurrent layers. This layer stores two vectors while processing a sequence: a hidden state $H = h_0, ..., h_T$ and a cell state $C = c_0, ...c_T$ — where T is the number of time steps in the sequence. Every time the model processes an input x_t (e.g., a word in a sentence) in the t-th timestep, this input x_t is combined with h_{t-1} and c_{t-1} (through several steps of element-wise summation and multiplication) to interact with the internal layers (with sigmoid and hyperbolic tangent activation functions). By doing so, the LSTM cell generates new values h_t and c_t for its internal memories. In some applications, such as sentiment analysis, the LSTM outputs a single activation vector after processing all sequence. When this is the case, such an activation vector is the last hidden state h_T. In other applications, such as machine translation, the LSTM has to output an activation vector for each timestep it received as input. In such cases, the output contains all the hidden state sequence $H = h_0, ..., h_T$. Note that LSTM cells can be stacked, so a sentiment analysis model could employ one LSTM that returns a sequence in the first layer and then another LSTM that returns a single activation vector as the second layer — and possibly more fully-connected layers after that as well. Source: https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714

where the input data contains a temporal or sequential dimension. For example, in a sentence classification task, the model has to analyze a sequence of inputs (words), instead of just a single element. To address problems like this, a different type of architecture exists: recurrent neural networks [Elman 1990]. These networks have specialized neurons with *hidden states*, an internal memory whose values keep information from the previous steps processed by the model. For example, when the model reads a new word in a sentence classification problem, it updates its hidden states to store information that can be important for the previous timesteps or to build the final activation vector after all timesteps have been read. Figure 3.5 shows an example of an RNN.

3.4 Design Workflow of Neural Networks

Designing a ready-to-deploy DNN is a complex task that requires several intermediate steps. To achieve a suitable model in a (relatively) short time, DL practitioners usually follow a multi-phase design workflow. Like most things in DL, this workflow is highly empirical and experimental, with many of its details varying significantly from practitioner to practitioner. Additionally, often one or more steps have to be repeated multiple times due to issues spotted in later stages. Nonetheless, at a higher level, the design workflow of DNNs proceeds as follows (see Figure 3.6).

Data selection: As a first step, the DL practitioner collects a dataset D containing an (usually large) amount of samples $(\mathbf{x}, \mathbf{y})$ and splits it into two disjoint sets: a training set D_{train} and a test set D_{test} (Figure 3.6 (A)). Splitting the training set from the test set is important for any ML technique and even more so for DNNs, as they are easy to overfit due to the large number of parameters they contain. A disjoint test set allows the practitioner to check if the model overfitted by validating its performance in new samples. During the data collecting step, the DL practitioner also has to ensure that the dataset D is adequate for the learning task. Otherwise, several issues may occur during training. For instance, the samples in D may not contain all the variability in the data domain or contain an unbalanced amount of samples for each class.

Network design: Following the data collecting process, the DL practitioner has to decide the DNN's architecture and hyper-parameters (Figure 3.6 (B)). This phase is critical in the designing process, as any wrong decision may lead to an utterly underperforming model. Unfortunately, these decisions are still mostly based on an empirical basis, as there is no analytical way to find out which architecture and hyper-parameter combination leads to optimal performance [Goodfellow, Bengio e Courville 2016].

Network training: In the third step (Figure 3.6 (C)) comprises training the DNN. During this process, the model uses an optimization technique to calculate the prediction error produced by the network for elements in D_{train} and then employs the backpropagation algorithm to update all weights Θ properly.

Network testing: As a final step, the DL practitioner measures the network's performance on the test set D_{test} (Figure 3.6 (D)). Such evaluation often resorts on some aggregated error metrics — e.g., accuracy, precision, recall, or the area under the received operator characteristic curve (AUROC) [Park, Goo e Jo 2004, Powers 2011]. For classification tasks, another typical way to evaluate the performance is to build the model's *confusion*

matrix [Fawcett 2006]. This matrix displays how often elements from any class A are predicted as belonging to any class B — including when $A = B$. This approach allows the practitioner to learn more details about the model's performance, such as if there is a particular class whose elements are mislabeled more often than others. However, the confusion matrix quickly becomes hard to inspect when the number of classes is too large. The testing phase also includes the task of interpreting the model in applications where model interpretability is important. In such cases, even if the model returns good performance in the test set, it may not have learned the features deemed as important by the human expert, or it may not follow the expected decision process. In any case, it is this step that the DL practitioner has to decide whether the model is ready to be employed in real applications or if it has to receive any modification.

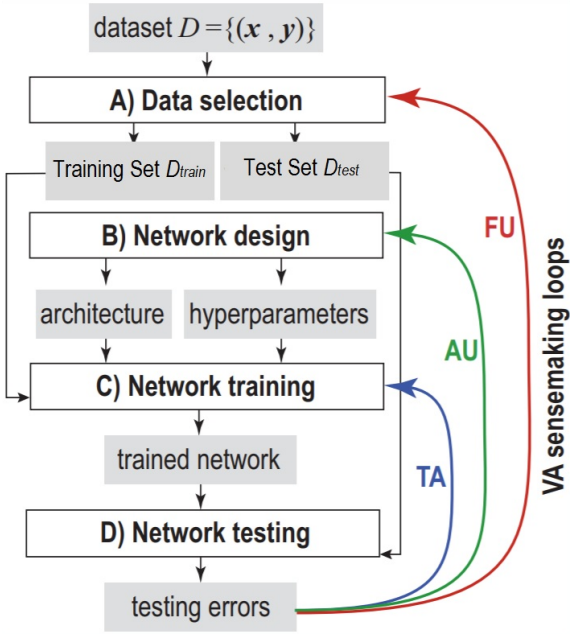
In general, the workflow proceeds by performing each of the four steps above in sequence. Once this is done, if the model displays a satisfying performance, the DL designing process is concluded. However, if the performance is not as desired, the DL practitioner has to repeat one or more steps of this workflow. Unfortunately, this workflow imposes some challenges to the practitioner. First, these steps are not easy to perform. They all require strong support from analytical tools to ensure they were done correctly. Secondly, after identifying an issue in the model designing, it is often hard to tell which changes would suppress such a problem and thus improve the model performance.

For this reason, DL practitioners often have to resort to feedback loops to address several tasks contained in the DNN design workflow. In this work, we discuss three of these tasks, which we call (1) *Architecture Understanding* (AU), (2) *Training Analysis* (TA), and (3) *Feature Understanding* (FU). We chose these tasks because they are the most important tasks that have been addressed by visual analytics tools in recent research, as we discuss in the next chapters. These tasks follow the Brehmer et al. [Brehmer e Munzner 2013] terminology that describes an arbitrary task as a set of domain and data-specific activities at both high and low-level that aim to answer questions at a specific workflow level. We give a brief description of the three tasks below:

Architecture Understanding: an essential step in the *network design* step is to analyze in which ways the DNN's architecture affects its performance. This way, the DL practitioner can have better insights on which modifications it should apply to the model to improve it if needed. However, to do so requires the ability to understand how the decision process works and how which component is aiding the model's predictions.

Training Analysis: the second task relates to analyzing how the model evolved during

Figure 3.6: The Design Workflow of a Neural Network



This picture depicts the overall steps in the design workflow of DNN and which of them are addressed by the three tasks that we present in Section 4 — architecture understanding (AU), training analysis (TA), and feature understanding (FU). The first step (A) consists in collecting a large dataset D — split into training and test set — containing samples and its corresponding labels. In the second step (B), the DL practitioner has to design the network architecture. In this stage, it has to decide how many layers the model has, the type of each of these layers, and how many neurons they should contain. Additionally, it has to make several other architectural choices such as hyper-parameter values or which optimizer the model will use for training. The next step comprises the training process (C). In this stage, the model compares the predictions made for training samples with their actual labels and update the model weights accordingly. In the fourth and final step (D), the DL practitioner tests the model performance in a different set of samples — the test set — to ensure that the model has learned to performed the task, even when it faces samples that were utterly unknown to it. The three tasks address challenges in multiple stages of the design workflow. Architecture understanding helps the practitioner to improve the chosen architecture. Training analysis allows them to find out problems that may occur during the training process as quickly as possible. Finally, feature understanding provides the ability to interpret the representations learned by the model and to explain which features present in the input influenced the decision process to go for a given predicted label over others.

training. Understanding why training did not occur as expected is critical to identify what to modify in the model architecture, hyperparameters, or training technique.

Feature Understanding: finally, another important task is to understand which features in the input data are used by the model to build the respective prediction and, in a higher level, which abstract features the model learned to recognize. With this knowledge, the DL practitioner can validate the model's performance and identify issues with the chosen architecture.

Without proper support for the three tasks above, the engineering of a DNN is reduced to a blind search process, as there are few to none insights about what are the issues and how to remove them. This significantly increases the number of iterations performed during the design workflow, which turns the designing of a DNN into a much costlier process. In the next chapter, we discuss how visual analytics tools can be handy in providing support for these tasks in several aspects [Seifert et al. 2017, Liu et al. 2017].

4 VISUAL ANALYTICS FOR DEEP LEARNING

The analysis of deep neural networks has recently become one of the main focus in the visual analytics community [Garcia et al. 2018, Hohman et al. 2018]. Visualization tools can provide a high-level amount of support to both the designing and the interpretation of deep models. Visual analytics is traditionally used to investigate patterns and behaviors on large amounts of data, in particular, those who are high-dimensional or temporal — i.e., that change over time. Neural networks produce and interact with both types of data. First of all, these models are usually trained with a large amount of very high-dimensional data, such as images, videos, and text data — all of these can easily reach thousands of dimensions. This training data can often be time-dependent as well. For instance, when learning to classify text documents, neural models have to take into account the order in which the words appear in the text, as this is important to identify the correct semantic meaning of the sentences and only that way the model can achieve a satisfying performance in this task. When handling an input sample, neural networks also produce many activation values, which can also be very high-dimensional — e.g., activations from convolutional neurons — or time-dependent — e.g., hidden states in LSTMs. Furthermore, if one aims to analyze how the network evolves over the training process, a second temporal perspective is added to the analysis. Given all the complexity involved in the data produced and handled by neural networks, the study of these models is far from a trivial task, and tools such as visual analytics are very useful in providing the needed support for the employment of these models.

Even though the current interest in VA considerably grown in the past few years, some earlier works in the area date back to the first years of the century [Streeter, Ward e Alvarez 2001]. In this work, Streeter et al. propose a technique named NVIS that uses a heatmap matrix to encode the learnable weights of a neural network. By displaying weights from each layer in a different row of the heatmap matrix, it allows the user to compare patterns that may exist across distinct layers of the model. In another early work, Tzeng and Ma [Tzeng e Ma 2005] use a node-link graph visualization to display the topology of the neural network, setting the color of the nodes according to the activation produced by the respective neuron for a given input. Although these methods provide interesting insights for small networks with few layers and neurons, they do not scale to the size of the networks required for most applications today. Moreover, they fail to properly address issues like model interpretability and training analysis, which are of

valuable importance for the deep learning community. Regardless of these shortcomings, these earlier works set up the foundation for the current research in visual analytics for deep neural networks and several novel approaches make use of similar visualization techniques to address the issues above.

Due to the relative novelty of the field, not a lot of effort has been put into categorizing the existing approaches to support visual analytics for deep learning engineering. During our research, we identified a gap in the literature in regard to a taxonomy to classify the methods proposed recently in this topic. To address this issue, we developed a task-and-technique based taxonomy that categorizes VA for DL methods according to which tasks they address and which model architectures they support.

4.1 A Taxonomy for Visual Analytics Tools Supporting Deep Learning Engineering

We present here a taxonomy that aims to classify visual analytics tools supporting the designing, training, and interpretability of deep neural networks. To build this taxonomy, we searched over 40 existing publications in this topic, classifying these approaches in three tasks: 1) *architecture understanding* (AU); 2) *training analysis* (TA); and 3) *feature understanding* (FU). We choose these three categories because they were the main goals of the VA tools aiming deep learning support that we identified in our research. It is important to note that these three tasks are tightly related to the challenges an ML practitioner usually faces during the design workflow of a deep model, as described in Chapter 3.4. Our taxonomy is designed with the focus on the VA community and to make it easier for them to enter in the area. For this reason, we explain in details why each task is important in the deep learning engineering, but we do not focus too much on how a particular visualization technique works, as we assume the reader already knows it.

In the next section, we describe the methodology we followed during the construction of our taxonomy, explaining how we searched for contributions for this survey. In the following, we describe in details each of the categories (tasks) we propose, which subtasks they contain, what techniques have been proposed to address them, and how they handle the three different neural network architectures, we take into account for this survey: DFNs, CNNs, and RNNs. Table 4.1 displays an overview of our taxonomy, with all the surveyed contributions along with the tasks, subtasks, and architectures they address.

Table 4.1: Taxonomy for VA tools supporting DL engineering

Taxonomy						
Technique	**Tasks**			**Architectures**		
	AU	**TA**	**FU**	**DFN**	**CNN**	**RNN**
[Zeiler e Fergus 2014]			FE		●	
[Liu et al. 2017]	AVis and AVal		FE	●	●	
[Samek, Wiegand e Müller 2017]			FE		●	●
[Montavon, Samek e Müller 2018]			FE		●	●
[Grün et al. 2016]			FE		●	
[Erhan et al. 2009]			MI		●	
[Rauber et al. 2017]		EMM	MI	●		
[Zahavy, Ben-Zrihem e Mannor 2016]			MI	●	●	
[Liu et al. 2018]	AVis and AVal	EMM	FE	●	●	●
[Ming et al. 2017]			MI and FE			●
[Zeng et al. 2017]	AVal and MC			●	●	●
[Kahng et al. 2018]	AVis		FE	●	●	
[Wongsuphasawat et al. 2018]	AVis			●	●	
[Smilkov et al. 2017]	AVis and AVal	RTA	MI	●		
[Chung et al. 2016]	AVis	RTA	FE	●	●	
[Harley 2015]	AVis		FE	●	●	
[Qi et al. 2017]		RTA and EMM		●	●	●
[Pezzotti et al. 2018]	AVal		MI and FE	●	●	
[Zhong et al. 2017]	AVal	EMM	FE	●	●	
[Selvaraju et al. 2017]			FE		●	
[Cashman et al. 2017]		EMM				●
[Yosinski et al. 2015]			MI and FE		●	
[Alsallakh et al. 2018]			MI	●	●	●
[Nguyen et al. 2016]			MI		●	
[Nguyen, Yosinski e Clune 2016]			MI		●	
[Aubry e Russell 2015]			MI		●	
[Simonyan, Vedaldi e Zisserman 2013]			MI and FE		●	
[Wei et al. 2015]			MI and FE		●	
[Mahendran e Vedaldi 2015]			FE		●	
[Mahendran e Vedaldi 2016]			FE		●	
[Zintgraf, Cohen e Welling 2016]			FE		●	
[Dosovitskiy e Brox 2016]			FE		●	
[Zintgraf et al. 2017]			FE		●	
[Li, Mueller e Chen 2017]			FE		●	
[Bojarski et al. 2016]			FE		●	
[Strobelt et al. 2018]			MI and FE			●
[Li et al. 2015]			MI and FE			●
[Rong e Adar 2016]			FE			●
[Ding et al. 2017]			FE			●
[Karpathy, Johnson e Fei-Fei 2015]			FE			●

The table above contains a comprehensive list of the VA contributions we consider for this survey. The table also displays which tasks, sub-tasks, and architectures each tool addresses. Tasks are further refined in the following categories: AU) architecture visualization (AVis), architecture validation (AVal), and model comparison (MC); TA) evolution of model metrics (EMM), and real-time analysis (RTA); FU) model interpretability (MI), and feature explainability (FE).

4.1.1 Methodology

Visualization for deep models is a multi-disciplinary topic that caught the interest of researchers from areas such as visual analytics, machine learning, data science, and computer vision, among other related fields. For this reason, advancements in the topic are published in a wide variety of venues. Therefore, a survey like the one we attempt to do requires a robust paper gathering methodology, allowing us to identify relevant contributions across journals and proceedings from all the related areas.

Our first step was to search through the publications of the most well-evaluated proceedings in the mentioned areas in the last ten years. Some of the researched venues are IEEE VAST, IEEE InfoVis, EuroVis, IEEE Transactions on Visualization and Computer Graphics, ICML, NIPS, ACM SIGKDD, ICCV, and CVPR. From this set of venues, we selected all publications mentioning visualization of deep learning models — or related terminologies — in their title or abstract. We also applied the same search method on platforms like *arXiv* or *Google Scholar*. In the last iteration, we look through the bibliography of all selected papers to find additional contributions that might be useful to the addressed topic.

In our survey, we focus on three main analysis tasks — AU, TA, and FU. These tasks were chosen because they can categorize, with a high degree of precision, the main goals of the vast majority of contributions we found during our research process. Nonetheless, it is essential to note that these three tasks are not an exhaustive list of ways that visual analytics can support deep neural network engineering. Visualization tools can address other challenges, such as training data analysis and performance analysis [Ren et al. 2017, Mühlbacher e Piringer 2013]. However, those are not deep learning specific problems, as they are also relevant when employing other machine learning techniques. For this reason, they are not in the scope of this research.

4.1.2 Architecture Understanding

A typical challenge in deep learning engineering is the understanding of how the model's architecture affects the prediction process and the overall performance of the model. With this knowledge, designers can be more confident about which changes in the architecture would result in a model with better performance or less complexity. To do so, they must analyze the model's components — such as layers and neurons — and in-

terpret how they interact with the input sample and how they contribute to the prediction output. However, due to the large number of parameters deep models contain, such an analysis turns into a difficult task. When employing wide and deep neural networks, practitioners can quickly lose track of what is the purpose of each component. Discovering any correlation between the roles of different components is also challenging to address, particularly when these components interact with many others to produce their activation values or maps.

Many of the visual analytics tools on our survey address the problem of *architecture understanding*. The tools placed in this category mostly aim to help the user to understand which is the role of each network's component — layers and neurons — in the prediction process of the model and how the architectural decisions affect such process. This task can be performed at both a higher and a lower level of abstraction. At a higher level, what VA tools do is to provide ***architecture visualization*** for the analyst. These visualizations make use of conventional graph visualization techniques to display the model's topology in a way that allows designers to quickly keep mental track of the overall architecture structure [Wongsuphasawat et al. 2018, Liu et al. 2017]. These tools can show, for instance, how many layers the model has, the size of these layers, and which operations they perform.

Additionally, by displaying how neurons interact with each other — e.g., by showing the distribution of weight connections or activations between subsequent layers [Liu et al. 2017] — visual analytics tools can help understanding how features learned in previous layers are merged to achieve a more abstract pattern recognition in deeper layers of the model, thus building the overall model's classification process. If combined with the visualization of the whole model structure, such techniques can display a structure-and-data combination of the network. For instance, one way to give more intuition about how the model structure correlates with the behavior present in the model's prediction process is to annotate the neural network's graph visualization with information such parameter vectors, activation values or training statistics [Liu et al. 2017, Liu et al. 2018, Ming et al. 2017]. Displaying such information can help the ML practitioner to identify several inefficiencies that can occur in an underperforming deep model, such as neurons which are not being selective toward any class [Pezzotti et al. 2018, Garcia et al. 2019] or that are displaying a very redundant behavior [Garcia et al. 2019, Rauber et al. 2017]. In brief, what these techniques do is to provide the ***architecture validation*** of a neural network.

Another useful subtask related to architecture understanding is ***model compar-***

ison. As stated in Chapter 3, one of the main challenges in the design of deep neural networks is to identify the best architectural choices for a given learning task. The selection of an adequate architecture and hyper-parameters may be the difference between a model that returns accurate predictions and one that is unsuitable for the task. Unfortunately, identifying the right hyper-parameters is complicated and there is no easy way to infer why a given hyper-parameter choice led to a bad performance. Visual analytics can address this issue by providing an alternative to compare the behavior of two or more models on the same task. This way, the analyst can have insights such as how one model may have achieved a better performance than other or why a given hyper-parameter choice led to a particular model behavior [Zeng et al. 2017, Sedlmair et al. 2014].

4.1.2.1 Architecture Visualization

Most deep learning architectures are directed acyclic graphs (DAGs) where nodes represent neurons and edges represent the weights connecting neurons in different layers [Liu et al. 2017]. Therefore, *graph visualization* techniques have been widely employed as a way to visualize the structure of deep neural networks [Liu et al. 2017, Kahng et al. 2018, Wongsuphasawat et al. 2018, Smilkov et al. 2017, Chung et al. 2016, Liu et al. 2018]. These techniques provide multiple benefits to the analysis of the network. First, they allow for a quick overview of the operations performed at each component of the network, telling the user what happens at each neuron or layer when an input flows through the network [Wongsuphasawat et al. 2018]. Secondly, it gives insights to the machine learning practitioner on how and where the model can be modified to achieve better performance at the task [Kahng et al. 2018].

The effectiveness of such visualizations is heavily dependent on which annotated information is displayed along with the model's components. Although visualizing the overall structure of the network is useful for designers to keep track of the model they are working with — particularly when designing very deep models —, it does not give by itself any additional information that the designer does not already know about the model. On the other hand, by annotating the nodes and edges in the graph visualization with meaningful values that explain the activity of the trained neurons is much more useful. Earlier works did so by making use of the nodes to display heatmap matrices displaying the parameters of the corresponding neurons [Streeter, Ward e Alvarez 2001]. However, parameter values — in particular, those on deeper layers — are tough to interpret. Their actual importance is highly dependent on the weights in other layers. Instead, more in-

sightful approaches have used similar techniques to display neuron activations, either for specific inputs or the average activation produced for each class in the training set [Liu et al. 2017, Kahng et al. 2018].

Neural networks employed in most applications are deep and wide, i.e., they have a large number of layers and neurons, respectively. Such characteristic introduces a scalability challenge to the design of visualization tools aiming to provide architecture visualization of deep models. Authors have addressed this problem with multiple approaches. One solution is to cluster neurons that have similar behavior, i.e., groups of two or more neurons producing similar activations [Liu et al. 2017]. This clustering approach makes the visualization less overwhelming for the user at the same time that it provides useful information about neurons that may be redundant and patterns that may not be adequately learned by the model, which may lead the model to underperform for one or more classes.

A second approach for reducing visual pollution is to add *edge bundling* [Lhuillier, Hurter e Telea 2017] to group connections linking neurons in the same clusters or groups [Liu et al. 2017, Wongsuphasawat et al. 2018]. This technique reduces the visual clutter often found in the visualization of dense graphs and allows users to focus on the important patterns contained in the node connections instead of in many individual connections at once. Another way to decrease the visual burden of visualizing large networks is the omission of non-critical or fixed operations, such as pooling, dropout and activation layers. Although these operations are significant for the performance of a network, they do not contain learnable parameters that are updated during training. Instead, they always perform the same operation. In some analysis, visualizing the output of these layers may not be necessary.

As the last approach to address scalability issues, the highlighting of regions of interest within the network, along with the aid of graph interactive exploration techniques, is also among the proposed alternatives [Wongsuphasawat et al. 2018]. In this approach, the user can quickly identify regions that show relevant behaviors and further explore it, visualizing in more details statistics like activation values or relevant features for that region.

4.1.2.2 Architecture Validation

This sub-task adds a new goal to the role of AU in the analysis of deep models. Here, the main focus is not just to visualize the architecture behavior, but to validate if the chosen structure is indeed the most suitable one. Even though both sub-tasks are

tightly related — indeed, most architecture visualization approaches also provide ways to validate the structure [Liu et al. 2017, Kahng et al. 2018, Smilkov et al. 2017, Chung et al. 2016] —, visualizing the whole architecture as a graph is not always required to validate it. Some authors have used other approaches to transmit such information to the user. In particular, heatmap matrices have been widely used to provide architecture validation [Liu et al. 2017, Kahng et al. 2018, Pezzotti et al. 2018]. The key advantage of such approach is that visualizing heatmap matrices is usually easier and less overwhelming than visualizing graphs, as they can display information more compactly. For instance, some contributions employ heatmap matrices to guide the user in the identification of inert neurons or neurons that activate for too many classes [Garcia et al. 2019, Liu et al. 2018, Kahng et al. 2018, Pezzotti et al. 2018]. In either way, those components end up not affecting the prediction process as they should.

4.1.2.3 Model Comparison

One last AU sub-task worth mentioning, but that has not been sufficiently addressed yet is model comparison. The ability to compare different models trained for the same task gives designers the possibility to analyze why they might be performing differently. One context in which this ability is particularly useful is when choosing the model's hyper-parameters [Sedlmair et al. 2014]. With the comparison of the behavior — both performance and internal activations — produced by models trained with different hyper-parameters, the ML designer can get valuable insights on what are the impacts of the selected hyper-parameter configuration in the final performance.

Another interesting possibility is to compare different snapshots of the same model, but trained until different epochs. This way, the designer can analyze how the additional training impacts the performance and can more easily make assumptions such as whether more training is going to improve the model's performance significantly or not. One of the few contributions addressing this sub-task is CNNComparator [Zeng et al. 2017]. This technique compares different epochs of the training process by using heatmaps and histograms to display the difference between parameter configuration at different epochs of the training process. When used in this context, the model comparison sub-task is highly related to our next task, training analysis, which we review next.

4.1.2.4 Architecture Understanding on Different Models

Deep Feedforward Networks: DFNs only contain fully-connected layers — excluding non-learnable layers —, where each neuron in any given hidden layer l receives as input the activations from all neurons from the previous layer $l-1$ and produces a single scalar value as activation output. This makes it easier to annotate DFN-aimed graph visualizations with statistics such as activations produced at each neuron, as there are fewer data to display. Most techniques do that by using heatmap matrices or color encoding methods to associate activation values with either the nodes [Liu et al. 2018, Harley 2015] or to the edges [Smilkov et al. 2017, Chung et al. 2016] in the graph visualization. The heatmap matrices used in this context mostly use rows to represent the different inputs or classes in the training (or test) set and the columns to represent different neurons in the layer [Liu et al. 2017, Kahng et al. 2018]. Such visualizations help the user to gather insights about how the prediction is occurring in various components of the model at once.

Convolutional Neural Networks: Similarly to DFNs, convolutional networks also have a directed acyclic graph structure. However, they differ from the previous architecture due to the presence of convolutional neurons in the first layers of the model. These neurons — also known as filters — perform successive convolutional operations in the input data, allowing the model to identify much more abstract patterns, which is very useful for some types of data, such as images. Such distinctions bring significant challenges to the architecture visualization of these models. First of all, parameters in the convolutional filter are applied to multiple activation values from the previous layer. Take the example of an image as input. Any learnable weight in the convolutional filter will be applied to all — or almost all — pixels in the image, regardless of its position.

Additionally, conversely to fully-connected neurons that output a single activation value per input, convolutional filters output high-dimensional activations — sometimes named *activation maps* —, often with the same size of the original input. This increase in the dimensionality makes it much harder to represent the activations produced by a convolutional filter with simple heatmaps or color-encoding. Despite these issues, many graph visualization techniques have been proposed to analyze the architecture of convolutional models [Liu et al. 2017, Liu et al. 2018, Chung et al. 2016, Harley 2015].

One typical approach when visualizing the structure of CNNs is to use the nodes of the graphs to display the corresponding filter's weights [Zeng et al. 2017] — i.e., the weights of the filter in the exact order they are applied to the input —, the activation map produced by a single input [Liu et al. 2018, Chung et al. 2016, Harley 2015], or inputs

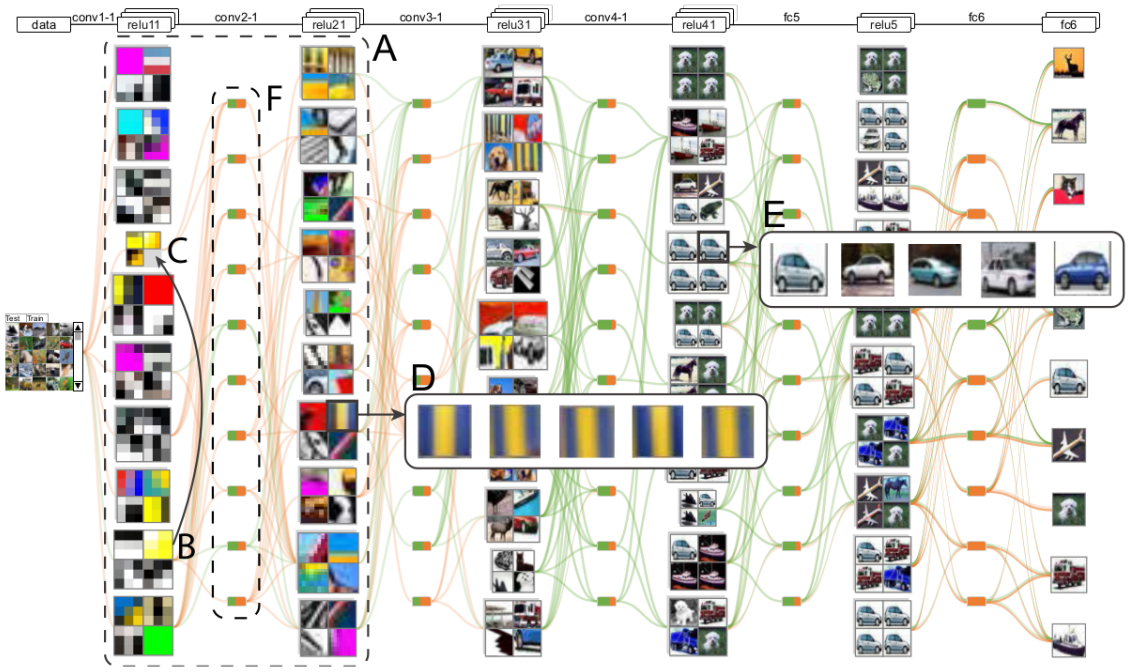
that produce strong activation on the neuron [Liu et al. 2017] — thus, being more likely to belong to the class of inputs that neuron learned to recognize. Figure 4.1 displays CNNVis [Liu et al. 2017], a visualization technique designed for the structural analysis of CNNs. This technique employs clustering to group neurons of similar activations, presenting in the nodes only features that produce strong activations in neurons belonging to those groups, thus helping the recognition of what the network is doing without adding unnecessary visual clutter.

Recurrent Neural Networks: RNNs add a higher degree of challenge to the architecture visualization task: they cannot be precisely described as DAGs. Recurrent neurons act not only at inputs received from previous layers but also with earlier timesteps generated by themselves. This interaction happens due to the internal hidden states such neurons have. Such hidden states are modified for every time step in the input, which results in very high-dimensional and temporal data that has to be visualized for each neuron. Any method aiming to visualize the overall structure of such models has to take this into account. Another challenge for the development of visualization for such architectures is the high variation of output dimensionality these models can have. Some recurrent models take a sequence as input and only output an activation value after processing the whole sequence [Zhang, Wang e Liu 2018]. Other models do so for every time step in the input stream [Zhang e Zong 2015]. In our research, we identified that no methods have adequately addressed those issues, and the architecture understanding of recurrent neural networks is still an open problem that requires further research into it.

4.1.3 Training Analysis

The training of neural networks is a lengthy process that gives little information to the ML practitioner about what is happening there. If the model fails to converge to a configuration with good performance, the practitioner usually has to restart the training process with different hyper-parameters or with a new model architecture. However, to do so efficiently, it is desirable to be able to get some intuition about what went wrong in the previous training process so that this knowledge can help the identification of a better configuration on the next try. Some visual analytics tools have addressed this issue by letting the designer visualize data related to the whole training process, such as the gradient values backpropagated [Pezzotti et al. 2018, Cashman et al. 2017] or intermediate activations [Rauber et al. 2017] at each neuron over the epochs. The analysis of

Figure 4.1: CNNVis



CNNVis [Liu et al. 2017] is one of the VA tools proposed to support DL engineering, in particular of CNN models. The tool displays the network as a directed acyclic graph (DAG). It groups adjacent convolutional layers (A) to better show the level of abstraction achieved at each pooling layer — note that the first layers only recognize simple features, such as border orientation, while the deeper ones recognize very abstract concepts such as vehicles and animals. It also clusters neurons in the same layer according to the feature maps they produce (B). This clustering makes the visualization less overwhelming, and it also allows for the identification of neurons that learned similar or correlated features.

the ***evolution of model metrics*** during the training process allow designers to reach a better understanding of why and how a model achieve a given performance and why or when it began to develop an undesirable or suboptimal behavior and what may prevent it. [Qi et al. 2017, Zhong et al. 2017].

Training time is a significant bottleneck in the design of neural networks. Deep networks may take several days or even weeks to be thoroughly trained and still may not be returning the expected results after that. In a traditional design workflow, the ML practitioner waits for the training process to finish to evaluate it and then decide it if it is adequate or not. If it is not, he or she makes changes in the model's architecture or hyper-parameters and re-train the network from scratch. Of course, this process increases even further the amount of time required to develop a ready-to-use neural network fully.

To mitigate this problem, some VA techniques have aimed to provide designers with ***real-time analysis*** of the model [Liu et al. 2017, Qi et al. 2017]. These methods do

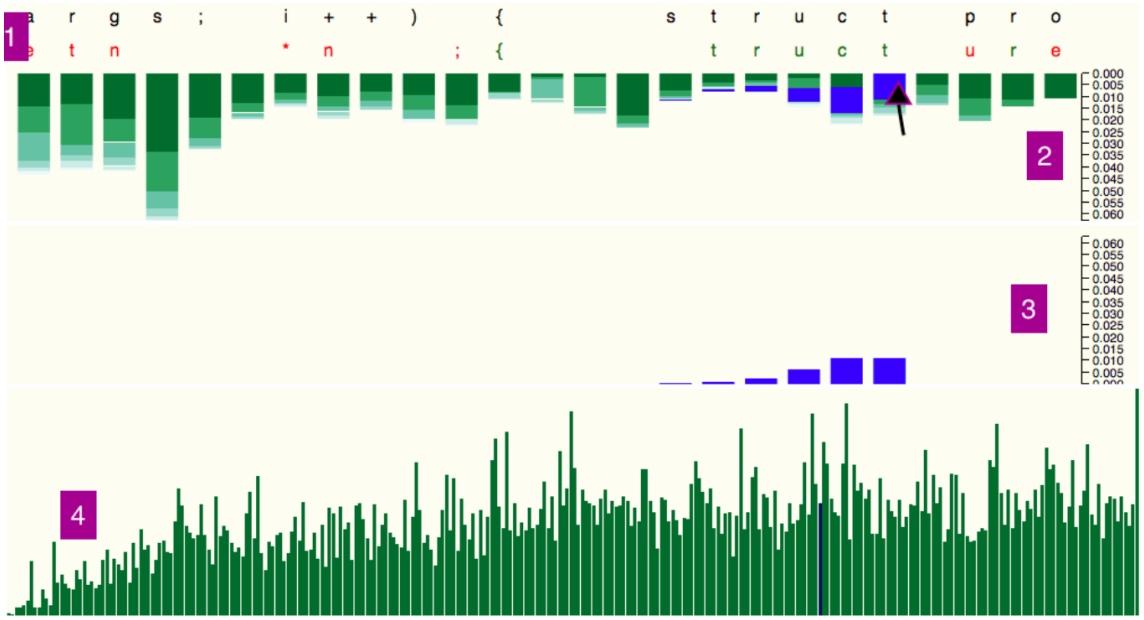
not wait for the training process to end to provide the designer with an interactive analysis of learned features and training statistics. This way, the practitioner can regularly check how the model is converging and decide if it wants to stop the training process before it finishes when a sub-optimal result is visible. Furthermore, it allows designers to make architectural and hyper-parameter changes on-the-fly, allowing the model to be quickly adapted to some need identified during the training process — e.g. if a layer needs more neurons because it has to learn more features than expected. By adjusting the model's architecture on-the-fly without throwing away what it has already learned, the new model configuration can converge much faster to satisfying prediction performance.

Besides applying a gradient-based optimization method to optimize its parameter configuration, a neural network also requires a *backpropagation* algorithm to properly update its weights at each training batch. This additional algorithm is required because the optimization method is based on the output of the last layer — i.e., the actual prediction — and thus can only define how the parameters in the final layer should be updated. By using a backpropagation technique, the neural network can propagate the error encountered in the last layer to earlier layers, and then correctly update the weights across all layers [Goodfellow, Bengio e Courville 2016].

However, two opposite problems related to the backpropagation algorithm may occur during the training process of a neural network: *vanishing gradient* and *exploding gradient* [Hochreiter et al. 2001]. In the first one, the error propagated from the output layer becomes so small at some point before reaching the first layer of the model that it is not enough to apply significant updates to the weights in the first layers. As a consequence, these layers remains unchanged, and the model is unable to learn — or at least, it learns way too slowly. On the other hand, an exploding gradient happens when the gradient error flowing through the model is so big that it always make too drastic updates to the weights, thus never converging to any solution.

Cashman et al. [Cashman et al. 2017] proposed a visualization tool (Figure 4.2) to allow a better understanding of this phenomena and to spot when and why it occurs. In their approach, they use a stacked bar chart visualization to display the magnitude of the weight updates happening at each timestep of the training process, with each bar partition identifying which timestep generated that update. With this visualization, a designer can quickly spot both problems, as they provoke bar partitions to become, respectively, too small or too big very quickly.

Figure 4.2: RNNbow



RNNbow [Cashman et al. 2017] is an example of a VA tool aiming the analysis of the model's training process. It employs stacked bar charts to display the progress of the gradient loss through the process of a sequence input by an RNN. In the top (1), the practitioner can evaluate the performance by comparing predicted and actual labels. Each of the top bars (2) measure the magnitude of the error gradient calculated by the backpropagation at each time step. The slices of each bar represent the source of each fraction of the gradient magnitude (3). Finally, the practitioner can interactively select a training batch to analyse on the bottom part of the view (4).

4.1.3.1 Visualization of Model Metric Evolution

This sub-task includes any visualization tool that aims to support the analysis of the evolution over time of model-related metrics during the training process. Such metrics can be standard machine learning evaluation metrics, such as accuracy or loss, or can be a user-defined metric that can be useful for the development of a particular model.

Some of the methods included in this category focus on the evaluation of gradient updates [Cashman et al. 2017], performance improvements [Zhong et al. 2017], or convergence of activation values [Pezzotti et al. 2018]. Such metrics have significant advantages in comparison to traditional accuracy-and-loss analysis. Besides asserting whether the model is performing well or not, they can be used to answer questions such as why the model is not converging, what is preventing the model from improving its performance and how the model can be modified to address the issues that may be occurring. For instance, in their contribution, Qi et al. [Qi et al. 2017] demonstrate how the tracking of user-defined metrics can be much more useful than standard ones to spot issues in the

training of neural networks.

Additionally, deep neural networks usually contain a large number of components — layers and neurons — that, while connected, still can display a high level of independent behavior, which considerably limits the effectiveness of global metrics in the analysis of such models. For example, while one neuron may have learned handy features to identify whether an input belongs to class *A* or not, another neuron may be struggling to help the prediction process because it is trying to learn selective features for too many classes.

To mitigate this problem, some works have addressed the issue of identifying when a layer or neuron stops learning and what it has learned. Pezotti et al. [Pezzotti et al. 2018], for instance, proposes the use of *perplexity histograms* to identify stable layers whose weights are not updating anymore. This way the user can interactively look at these layers activations and evaluate if they converged as expected or not.

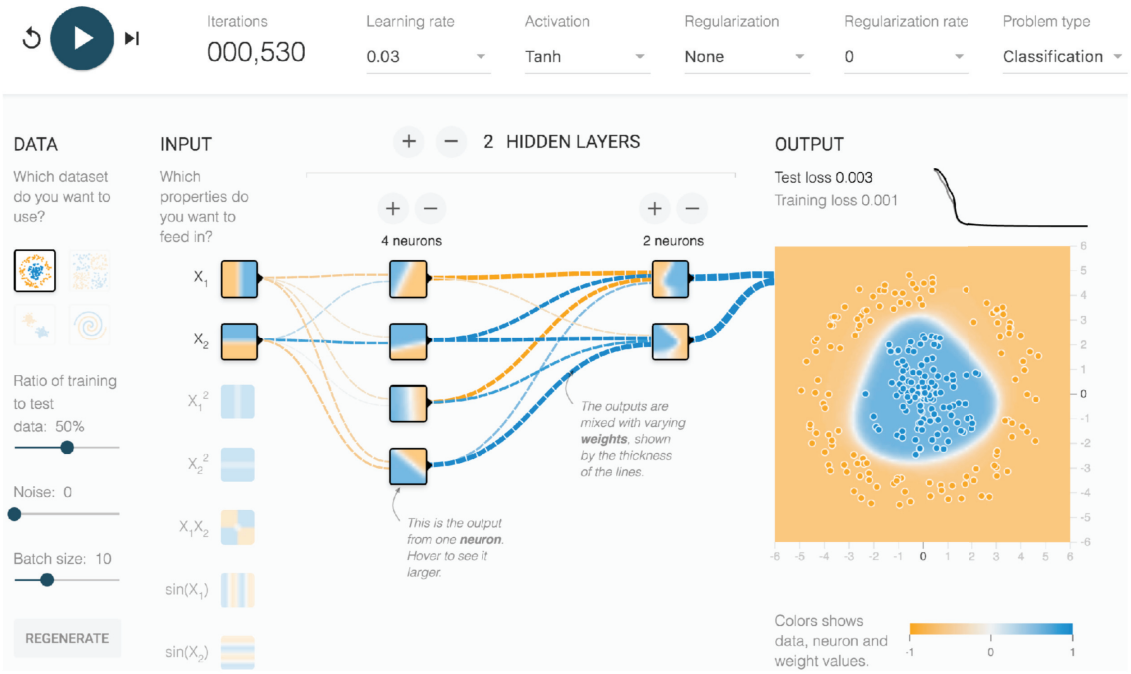
Another approach for tackling this problem, proposed by Zhong et al. [Zhong et al. 2017], employs heatmap matrices to display how neuron's weights and activations change over time. Even more interestingly, their approach allows users to compare different neurons according to two metrics: discriminability — measuring how different the average activation produced by a given class is from the average activation of other classes — and density — evaluating the number of high activations a neuron produces for each class. These measures provide the user with a qualitative way to assess if neurons are working correctly or not, which is essential to make decisions such as what changes should be done to improve the model.

4.1.3.2 Real-Time Analysis

Deep neural networks require a tremendous amount of time to train, which may prevent their utilization in some application where they could have been very valuable. For that reason, tools that mitigate the time spent training DNN are especially welcome. The ability to visualize the model in real-time can help the designers to make conscious decisions about what to change in the model much faster than if they had to wait until the training is complete to analyze if the achieved result is good or not.

Primarily, any visualization task addressed in this survey can be adapted to be performed in real-time. For instance, one can choose to do architecture understanding by visualizing how model's weights and activations vectors are evolving during training [Smilkov et al. 2017], to do training analysis by checking the current state of evaluation metrics or gradient flows [Qi et al. 2017], or even to do feature understanding to

Figure 4.3: TensorFlow Playground



TensorFlow Playground [Smilkov et al. 2017] is a popular on-line tool to learn neural network concepts. It shows both the network's structure and the activation space of each neuron in the model. This visualization allows the user to understand how the network builds its decision process from the features in the original input. More importantly, it is an interactive tool, which allows the user to control several aspects of the network — e.g. architecture, training data, hyper-parameters — and see how these changes affect the model and its performance.

evaluate if the network already learned to recognize interpretable features or not [Chung et al. 2016]. When combined with environments that allow the user to modify the model configuration easily, these tools can prove to be a significant step towards more interactive machine learning models that can take of human expertise to achieve better learning.

Additionally, these tools are also a great way to teach deep learning engineering, as they can make the effects of changes in the model much more tangible. Tensorflow Playgroung [Smilkov et al. 2017] (Figure 4.3), for instance, is a very popular online tool that teaches introductory concepts of neural network design by employing a visualization technique that displays how the neurons split the activation space to help the recognition of simple synthetic datasets.

4.1.3.3 Training Analysis on Different Models

Deep Feedforward Networks: The methods proposed to analyze the training process of DFNs are mostly based on the analysis of metrics evolving through the training process. In particular, they aim to evaluate these metrics for individual components such as layers or neurons instead of considering the whole model. This approach allows the designer to achieve a better understanding of where and why an undesired behavior appeared in the training process and thus can use that information to make effective changes in the model's configuration.

For instance, the approach proposed by Rauber et al. [Rauber et al. 2017] employs a dimensionality reduction technique called t-SNE [Maaten e Hinton 2008] to show how the activation vectors produced by a hidden fully-connected layer for a subset of samples evolve through the training epochs — see Figure 4.5 (middle). By comparing how distant samples from the same class are in the projection in comparison to those from other classes, the designer can evaluate if the layer is becoming selective for every class or if it still struggling to differentiate two or more classes. This approach is particularly interesting when applied to the last hidden layer, as the activation vectors produced by this layer are the ones the model effectively uses to perform classification. If they fail to differentiate samples from any pair of classes, the model will undoubtedly fail to achieve a satisfying performance.

Convolutional Neural Networks: The training process of CNNs is very similar to the training of regular DFNs. The cost function calculates a prediction error in the output layer, and the backpropagation algorithm propagates this error to the rest of the model, updating the weights in such a way that the error is minimized. However, the structural difference between convolutional layers and fully-connected ones do create some differences in the analytical needs of both architectures. For example, activations from convolutional neurons are much harder to visualize due to their high-dimensionality, which limits the applicability of some approaches [Zhong et al. 2017].

To address this limitation, Liu et al. [Liu et al. 2018] propose a VA tool aiming the training analysis of convolutional and generative models. Their approach displays the learning happening in training through line charts showing how many designer-selected statistics evolve over the epochs. By analyzing these plots, designers can spot filters with an abrupt change in the behavior and thus interactively analyze it further to build hypotheses about what could have happened.

Recurrent Neural Networks: The training process of recurrent networks brings consid-

erably more significant challenges if compared to CNNs and DFNs. Due to their recursive structure, they are much more likely to run into problems such as the vanishing gradient during backpropagation. These issues happen because the samples these models receive as input are made of several timesteps and the backpropagation has to propagate the error for each of such timesteps, effectively making the propagation occurs for much longer than in comparison to regular inputs lacking any temporal property [Goodfellow, Bengio e Courville 2016].

For this reason, the ability to recognize when backpropagation-related problems occur is much more valuable with RNNs than with other architectures. Cashman et al. [Cashman et al. 2017] address this problem with a stacked bar chart visualization that displays how the gradient changes at each timestep for different components in the recurrent layers of a model. With their technique, designers can inspect if vanishing or exploding gradient is occurring and get insights about how each training example is affecting the parameters throughout the model, which can be helpful in design decisions such as hyper-parameter modifications.

4.1.4 Feature Understanding

One of the biggest concerns in the deep learning research today is how to interpret the features learned by the model and to explain how it decides for a given output. Highly interpretable models would ensure the usability of deep neural networks in a much more extensive array of applications. First of all, because interpretable models are much more trustable. If the user knows how a model reaches a given prediction, they are much more likely to trust it. Secondly, there are many situations where interpretability is essential, and non-interpretable models cannot be used at all. Critical applications as medical predictions, drug discovery, and autonomous driving (among others) cannot employ models where it is not known how predictions are made [Vellido, Martín-Guerrero e Lisboa 2012].

Hence, much of the machine learning research today focuses on the development of methods to identify which existing features from the input sample the model takes into account to decide for a particular prediction output [Zeiler e Fergus 2014, Nguyen, Yosinski e Clune 2015] and what specific features each of the model's components search for — i.e., which features the input sample must contain to produce high activations in that component [Liu et al. 2017, Rauber et al. 2017].

The difficulty in achieving interpretability with neural networks lies in the fact that these models contain a vast number of learnable parameters and it is tough to make sense what is the purpose of each of them in the prediction process. This problem is particularly true for the parameters in the deeper layers, as they interact only with activation values produced by a — sometimes very long — sequence of previous layers [Lipton 2016].

Several VA approaches have been proposed to address this problem and give more insights about the features learned by neural networks and its components [Liu et al. 2017, Kahng et al. 2018, Nguyen et al. 2016, Hohman et al. 2018]. For this survey, we split the works aiming to understand learned features in two main subtasks: ***model interpretability*** and ***feature explainability***. In the former, the main analytical goal is to understand what features each of the network's components are recognizing and how they use the activation values received from earlier layers of the model to do so [Liu et al. 2017, Rauber et al. 2017, Yosinski et al. 2015]. On the other hand, feature explainability aims to answer which features present in the *input data* the model considers essential to decide which prediction it should output. Such knowledge can make models more reliable, as it can give more confidence to users that the neural network is making predictions as a human expert would [Pezzotti et al. 2018].

Additionally, feature understanding techniques can be classified into two categories: *instance-based* and *feature-based* visualizations. In the first category, the visualization aims to explain the behavior of the model for particular inputs, be it a single one or a subset of related samples — e.g., samples from the same class. This way, the visualization can help designers to figure out how the model process that type of input. For instance, it can explain which neurons are tasked with the recognition of features from those samples and how the model uses such features to determine the output [Kahng et al. 2018].

Feature-based techniques, instead, do not explain the model's behavior from the perspective of particular input samples. Their focus lies on understanding the reasoning of the model itself. They aim to explain, for example, what abstract features a particular neuron learned to recognize or how they interact with the features learned in the previous layers, regardless of in which samples these features might appear or not [Rauber et al. 2017]. These techniques are particularly useful in applications where input samples are not visually interpretable or when the possible features on input samples are so many that is difficult to take any insight from single inputs or group them in a meaningful way [Kahng et al. 2018].

Most VA tools aiming to perform feature understanding do so by looking at the activations produced by the models. Activations produced in hidden layers can be considered as transformed representations of the input sample built by the network so it is easier to perform the prediction task. If we can interpret which information is encoded in the activation values, we can have useful insights about what the model is doing.

Performing this analysis in multiple ways is possible. If the analysis focuses on the activations produced by a single input flowing through the entire network — i.e., an instance-based approach —, the analyst can build strong hypotheses about how the model manipulate the sample features to reach a given prediction. This can be done, for instance, with techniques such as heatmaps or matrix plots [Kahng et al. 2018].

On the other hand, if the analyst focus on which kind of features the input must have to produce strong activations in each of the model's layers or neurons — i.e., a feature-based approach —, the analyst can better understand which patterns the model learned to recognize and make assertions about if they are as expected or not [Rauber et al. 2017] or how the neurons transform data from sample space to activation space [Smilkov et al. 2017] (see Figure 4.3).

4.1.4.1 Model Interpretability

The ability to interpret models is one of the main challenges deep learning faces nowadays [Zeng e Wang 2017]. However, interpretability of machine learning models is an ill-posed problem with no widely accepted definition [Lipton 2016]. However, a key common goal in most definitions is to understand how the model builds its decision process from the information contained in the input sample and which role each neuron plays in such a decision.

One way to achieve interpretability is to analyze the activation space produced by each neuron or layer in the model [Smilkov et al. 2017]. This analysis allows the user to have better insights about how the neurons transform the samples from input space to intermediate representations in multiple activation spaces and how the last of these activation spaces is used to perform the prediction task. In the case of neurons that receive only two or three values as input, it is possible to achieve this analysis by merely plotting the activation space — respectively a plane or a cube — and using color encoding to indicate the activation values produced at each space region — i.e., each combination of input values.

The previously mentioned Tensorflow Playground tool [Smilkov et al. 2017],

which aims to teach deep learning concepts through visualization, follows this approach (see Figure 4.3). Unfortunately, this approach lacks scalability. If the neuron has a high-dimensional input — i.e., the data it receives from the previous layer is high-dimensional — or the neuron itself outputs a high-dimensional activation — which is often the case with convolutional and recurrent layers — this approach becomes prohibitive, as it is not possible to fully visualize high-dimensional spaces.

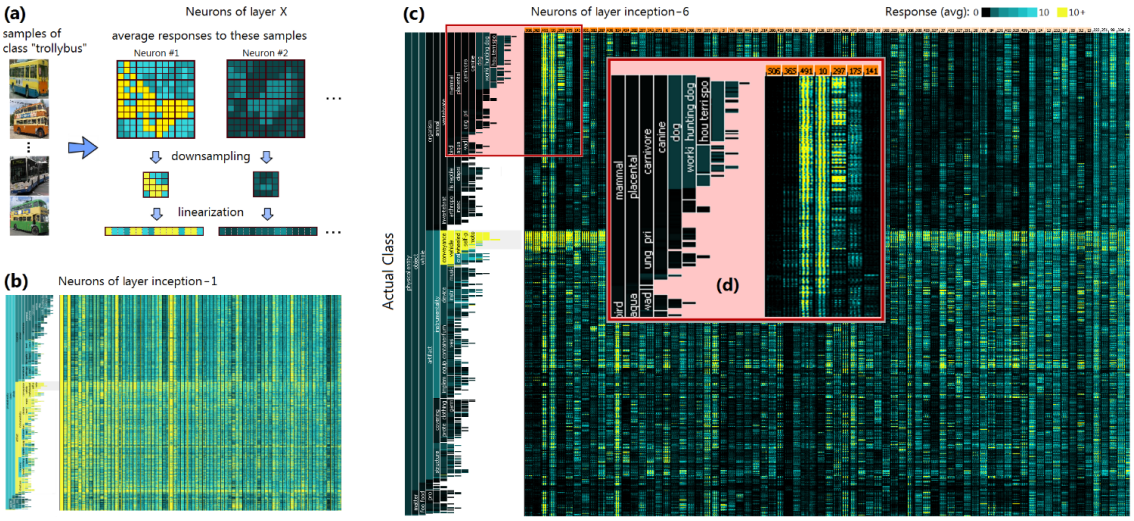
What most visualization approaches do to mitigate this problem is to focus the analysis on a limited number of inputs or in sub-regions of each component's activation space. They do so by using heatmap matrices to show how neurons and layers behave when the user feeds the network with particular inputs [Liu et al. 2017, Kahng et al. 2018, Pezzotti et al. 2018]. In these matrices, columns represent neurons from one or more layers in the network and rows represent either single input samples — thus behaving as an instance-based technique — or each of the possible labels samples can have — behaving as a feature-based technique. In any case, the matrix cells display the average activation values produced by each input sample or label at each neuron. From such visualization, the designer can take insights about which kind of input samples a given neuron activates the most and use this information to build hypotheses about the decision process learned by the model. Combining this visualization technique with graph visualization tools can provide the user with both AU and FU at the same time [Liu et al. 2017, Liu et al. 2018]. An example of such heatmap matrices is shown in Figure 4.4.

In general, averaging the activation value (or vector) produced by same-class samples is a good strategy to find out how the model learned to predict the class. However, there are situations where this can mislead the designer to false assumptions. In some applications, two same-class samples may be very different and may share no similar feature at all.

For instance, suppose an image classification problem where one wants to identify images of *grocery store*. The training set may contain pictures depicting a grocery store's façade while other pictures may depict the interior of these buildings. In either case, the correct label is *grocery store*, but the features a model has to recognize in the first group of pictures to predict the right label is quite different from the features it has to recognize in the second to predict the very same label. Classes with this property as often called multivariated [Nguyen, Yosinski e Clune 2016].

To be able to understand how the model handles such differences during its decision process is an interesting interpretability problem that is not properly addressed when

Figure 4.4: Visualizing Activations with Heatmap Matrices



Several tools employ heatmap matrices to visualize activation values. One of them, shown above, was designed by Alsallakh et al. [Alsallakh et al. 2018]. In their work, they use a heatmap matrix to display how neurons in a given layer respond to inputs belonging to different classes. In their heatmap matrices (B and C), rows represent each class in the training set and columns represent each neuron in the layer. This way, each of the matrix cells encodes the information about the activations produced by that particular neuron to members of a particular class. The value of each cell is computed as described in (A). They first average the activation returned by the neuron for all samples belonging to that class. This average matrix goes through a downsampling plus linearization process, becoming an array that condenses the information about the class average activation. With this tool, the authors identified that the activations produced across the model often mirror implicit hierarchical structure present in the training data (B).

the VA tool averages the activation values of same-class samples. Using an instance-based heatmap matrix may be useful when the designer is aware of the presence of this characteristic in the dataset — and can select representative elements of each group to display in the matrix rows — but in many situations, this assumption may not be accurate.

Additionally to the issue depicted above, heatmap matrices do not scale well, which makes it challenging to uncover interesting patterns when the number of samples or classes is too big or if the layer has too many neurons. An alternative approach followed by multiple authors is the employment of dimensionality reduction techniques to visualize activation vectors. In a neural model, the job of each hidden layer is to transform the data received as input to another representation, keeping — and highlighting — features that are important for the prediction process and throwing away those that are not. When the model is well trained, same-class samples should produce relatively similar activation vectors in deeper layers. Following the same reasoning, examples that are not from the

same class should create considerably different activations as well, so the last layer can perform the classification properly.

For this reason, comparing the activation value produced by the same layer to multiple inputs is an effective way to infer how the layer behaves to samples from different classes. Dimensionality reduction is an effective way to do so when these vectors are very high-dimensional. Several contributions have employed them to achieve model interpretability [Rauber et al. 2017, Zahavy, Ben-Zrihem e Mannor 2016, Kahng et al. 2018, Pezzotti et al. 2018].

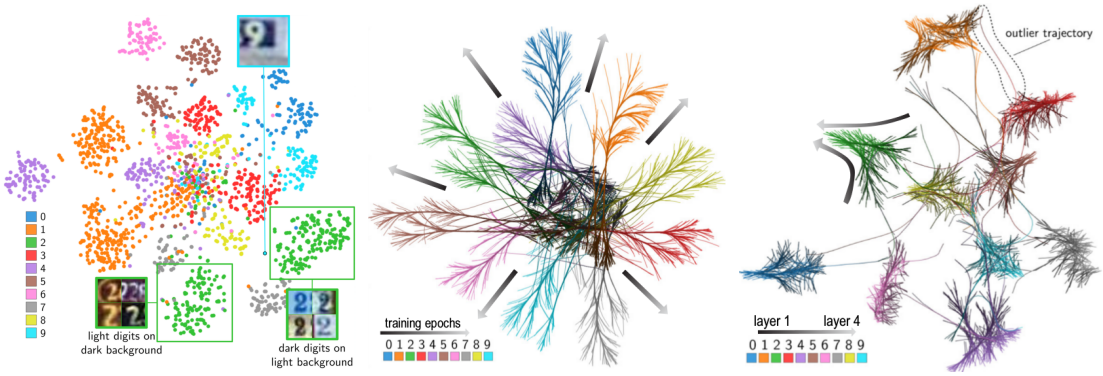
These approaches use techniques such as t-SNE [Maaten e Hinton 2008] to reduce the dimensionality of the activation vectors produced in the layer to 2-D. This projection is then visualized with bidimensional scatterplots whose datapoints similarities mirror how similar the data points — i.e., the activation vectors — are in their original dimensionality. When two or more points lie close to each other in the scatterplot, that means the activation vectors produced by that group of samples in the layer of interest are close in high-dimensional space as well. If they are placed farther away from each other, that means the corresponding activation vectors do not display many similarities at all.

The analysis of a layer's activation projection can bring multiple benefits to the designer. They can, for instance, infer if there is any pattern in misclassified samples. They can also check if the layer is separating the classes as it should or if it is having trouble distinguishing one or more classes. It can as well, quickly identify groups of same-class samples producing very different activations, which could fall on the case of multivariate classes.

Additionally, by visualizing how the activation projection evolves through the layers of the model, the user can have insights about how the decision process occurs, making assumptions about when the model has enough information to identify the correct class of a given sample successfully. Figure 4.5 displays an example of an approach that uses dimensionality projection for model interpretability.

Despite its benefits, dimensionality reduction also has some downsides. When a high-dimensional datapoint is transformed into a bidimensional one, important information may be lost. In particular, it is not possible to infer in which ways two activation vectors are similar to each other, even if their projected datapoints are very close to each other. Additionally, projection techniques often render very different projections for the same data according to the selected hyper-parameters, and it is not trivial to evaluate the quality of each projection.

Figure 4.5: Dimensionality Reduction of Activation Vectors



An interesting way to visualize activation vectors is to project them onto a bidimensional space by using dimensionality reduction techniques. Rauber et al. [Rauber et al. 2017] do so in three different ways. By projecting the activation vectors of a single layer for multiple samples, they explore the learned features and the separability of the classes (left). By projecting activation vectors from several training epochs, they analyze how the training process led to the current feature space (middle). Finally, by projecting activation vectors from multiple layers in the model, they can infer about how the learned features and the class separability evolve along the layers (right).

Although some alternative approaches, such as interactive projections [Sacha et al. 2017], have been proposed to mitigate this problem, such solutions are yet to be employed in the analysis of deep learning models. Nonetheless, due to the scalability and ability to explore high-dimensional data that dimensionality reduction provides, it is an efficient way to address model interpretability on deep neural networks.

4.1.4.2 Feature Explainability

Differently from model interpretability, this subtask addresses the problem of explaining which features from the input sample the model takes into account to define the output label. By relying on the image classification example again, the goal of this subtask is to explain which particular regions of the image contain the information that is important to identify to which class it belongs. For instance, if the image belongs to the class *car*, one would expect the model considers objects such as wheels or the car bodywork as necessary but ignores background pixels as these are not relevant for the concept of what a car is.

Several techniques have been proposed to address this problem. Some examples are: code inversion [Mahendran e Vedaldi 2015], layer-wise relevance propagation [Bach et al. 2015], and deconvolutional networks [Dosovitskiy e Brox 2016]. These methods

evaluate each of the input samples attributes according to how relevant they were for the decision taken by the network. In these evaluations, an attribute in which a big change on its value would not result in a relevant change in the final decision receives a low score, while an attribute that would strongly modify the prediction if modified receives a much higher score. In the sections below we give more details about these techniques.

4.1.4.3 Feature Understanding on Different Models

Deep Feedforward Networks: Fully-connected layers fundamentally transform the input data to another representation space — often high-dimensional — via a non-linear function. For this reason, most tools that address the problem of visualizing high-dimensional data, such as dimensionality projection [Rauber et al. 2017], can be used to analyze the activation vectors produced by fully-connected layers. Fully-connected neurons are rarely visualized alone because they output a single activation value, which makes the analysis much more interesting when it is performed for the whole layer. Heatmap matrices are often used in such situations, as they allow the user to visualize activation values for individual neurons and compare how they differ for multiple input samples or neurons, thus giving more insight about the role of each neuron in the decision process [Liu et al. 2017].
Convolutional Neural Networks: Heatmap matrices have been successfully employed to visualize the activations produced by convolutional filters as well [Liu et al. 2017, Harley 2015, Yosinski et al. 2015, Alsallakh et al. 2018]. These techniques can either focus on the visualization of the activations produced by a single convolutional filter for multiple samples [Liu et al. 2017] or on the activations produced by multiple convolutional filters for a single input sample [Yosinski et al. 2015].

In the former, we have a feature-based approach that aims to understand what is the role of the filter, i.e., which features it learned to identify if the input has or not. The latter is an instance-based approach whose goal is to understand how the model builds its decision process for that particular input.

By employing heatmap matrices, Alsallakh et al. [Alsallakh et al. 2018] (see Figure 4.4) was able to reach interesting conclusions about how convolutional networks implicitly learn hierarchical structures present in the training set. For instance, they show that a convolutional network trained for image classification often already produces very distinguishable activations for images of a car in comparison to images of trees in the first layers. Conversely, it takes much more layers for the same model to distinguish between an image of a car and an image of a truck — which are much more similar objects.

Dimensionality reduction has also been widely applied to the analysis of features learned by CNNs [Chung et al. 2016, Nguyen, Yosinski e Clune 2016, Aubry e Russell 2015]. An interesting experiment, performed by Aubry et al. [Aubry e Russell 2015] uses t-SNE projection to compare the activation vectors of images produced by slightly modifying the same original image in several ways — e.g., rotation, scaling or translation. This helps the user to understand the overall structure produced by the learned representations in that layer. Alternatively, dimensionality reduction can be applied to the training samples [Nguyen, Yosinski e Clune 2016] to identify groups of examples that may belong to the same class but contain very different features (see Figure 4.5).

When the model takes images as input — which is one of the main applications for CNNs — it is also possible to build input-based heatmaps [Yosinski et al. 2015] that shows which regions of an input image contribute the most for the activation produced at any given convolutional filter. Each pixel in these heatmaps corresponds to a pixel in the original image, but with a highlighting proportional to how important the pixel is for the prediction process. Many variations of this approach have been proposed, both instance and feature-based techniques. We discuss them next.

Instance-based techniques: In most image-related tasks, the input image contains more information than what is relevant to perform the task — e.g., background information or structures that do not belong to the object of interest. In such situations, it is essential to check if the model uses the correct information to calculate the prediction. Otherwise, it is not possible to ensure that the model behaves as expected. For instance, one may be interested in design a model that classifies images according to the object it depicts and one of the possible classes is *wolf*. Imagine that during training the model has only access to images of wolfs in a snowy background — but the designer may not be aware of that. The model may end up considering the snowy background as an essential feature of the class *wolf*. In such case, if a different sample depicting a wolf in another background — e.g., a green forest — is presented to the network, it may not recognize it correctly due to the lack of what the model believes to be an important feature. Additionally, if another image displays a different entity — e.g., a dog — in a snowy background, the model may misclassify it as *wolf* as well, due to the strong impact the snowy background has on its decision. Figure 4.6 shows two examples related to this problem.

Several instance-based techniques address this problem by building heatmaps where each pixel in the original input image is highlighted according to the importance that pixel has for the decision taken by the network or for the activation produced in a particular

Figure 4.6: Instance-Based Techniques for Image Classification

The pictures above depict how instance-based techniques can explain which features led to the classification output [Ribeiro, Singh e Guestrin 2016]. In the first example, the model is correctly recognizing the class *dog* (a), as it learned to recognize meaningful features for that class — i.e., facial features (b). In the second example, however, the model is presented with the image of a dog (b) but is incorrectly classifies it as a wolf. Further analysis of the features used for classification (d) shows that the model took into account mostly background features, ignoring the important features of the animal.

neuron. Such heatmaps are often called saliency maps.

By analyzing which regions of the image are more relevant, the designer can infer, for example, if the network takes into account the same features that a human would. Simonyan et al. [Simonyan, Vedaldi e Zisserman 2013] do this by ranking the image pixels according to how much they contribute to the labeling decision. Montavon et al. [Montavon et al. 2017] proposed a backpropagation-like algorithm to infer the relevance of each activation value layer-by-layer. Zintgraf et al. [Zintgraf et al. 2017, Zintgraf, Cohen e Welling 2016] iteractively hides different regions of the image to check if the model is still capable of correctly predict the label. The pixels in this region receive scores based on whether or not the model was able to do so. More recent approaches look at the gradient values to understand how information flows through the model and how important

each feature is [Selvaraju et al. 2017, Bojarski et al. 2016].

Another challenge in feature explainability is to understand which features from the original image were discarded by the model at any given layer. A typical approach for that is to reconstruct the input image from the activation vector it produces at a particular layer. If the reconstructed image is close enough to the original image, it is because the activation vector still preserves most of the original features in an encoded representation. Otherwise, it means that some features may have been lost. However, it is not trivial to perform such reconstruction due to the complexity of the non-linear operations implemented by convolutional layers, which usually are not invertible.

One of the first techniques proposed to address this problem is called *code inversion* [Mahendran e Vedaldi 2015]. Let x be the input image of interest and a^l be the activation vector produced by x in a given layer l. The goal of code inversion is to find out which features in x are still encoded in a^l and which are not. It does so by optimizing, via gradient descent, a synthetic image $\tilde{x}$ that generates an activation vector $\tilde{a}^l$ as similar as possible to a^l. This way, the generated image will likely contain every information present in x that is still encoded in a^l, but it is unlikely to contain any other information that has been already discarded by the model at this point. With this technique, Mahendran et al. [Mahendran e Vedaldi 2015] were able to identify that activations from the first layers of a model usually can reconstruct the original image much better than deeper layers. Although intuitive, this finding validates the widely held assumption that deeper layers identify only very abstract concepts — such as entire objects — in a much robust but less detailed way.

In their following work, Mahendran et al. [Mahendran e Vedaldi 2016] propose several techniques to evaluate the images generated by code inversion. Some of the proposed metrics include, for instance, reconstruction similarity, naturalness, interpretability, and classification consistency. Later, other approaches were proposed to synthesize such images, such as deconvolutional [Zeiler e Fergus 2014] and up-convolutional networks [Dosovitskiy e Brox 2016]. Deconvolutional networks perform operations that approximates the inverse operation performed by the CNN. Up-convolutional networks, however, are CNNs trained with the activation vector a_i^l as training samples and the original images x_i as labels. This way, this new CNN is effectively trained to perform the inverse operation of the one performed by the original network up to the layer l.

A somewhat less-popular but very interesting instance-based approach is *caricaturization* [Mahendran e Vedaldi 2016]. This approach modifies the input image to exag-

gerate or reduce the features according to how important they are for the decision process. They do so by adjusting the original image to maximize the activation produced in a particular neuron. This way, the image keeps the overall structure of the original figure but with the critical features much more salient than unimportant ones.

Instance-based techniques have a significant drawback in which the insights produced are only valid to a single input sample. For this reason, it is impossible to use these techniques to answer questions such as if the features recognized on that particular input also have to appear in any other image for it to receive the same class prediction. Of course, one could analyze every input from such a class to find out that — or at least a significant subset — but this is often impractical, as it quickly becomes very time-consuming and inefficient. Feature-based techniques aim to address this issue, which we discuss next.

Feature-based techniques: Most feature-based techniques aiming to explain the features learned by CNNs do so by synthetizing images that contain the features the model — or some of its components — is looking for. This is particularly useful if applied to the output layer of the model, as this is the layer that returns how likely it is for the input to belong to each class. If we can uncover which features the neuron corresponding to a given class c tries to recognize, we can have a pretty good insight about what features the input must have to be labeled as c by the network [Simonyan, Vedaldi e Zisserman 2013]. When applied to intermediate layers, such an approach would also be helpful to identify the actual role of each of its neuron in the training process [Erhan et al. 2009, Yosinski et al. 2015].

The most basic approach to identify which features a neuron is recognizing is to display the images in the training set that produce high-activation values in that neuron [Nguyen et al. 2016]. By browsing through the images, the user can identify similar patterns among them that may be responsible for the high-activations. Of course, this approach has severe limitations. It does not provide any information about what specific features the model used to produce that activation. It also does not allow to rank the features in this group of images according to how important they are. Due to these limitations, the user can quickly reach a wrong or misleading conclusion about the model.

A much more interesting approach is to synthesize a summary image containing the preferred features for the neuron of interest. This way, the user can not only ensure a more precise insight about the features learned by the neuron but also can interpret them and evaluate if they make sense from a human point-of-view — i.e., if that feature would matter for a human performing the decision task. An example, presented by Nguyen et

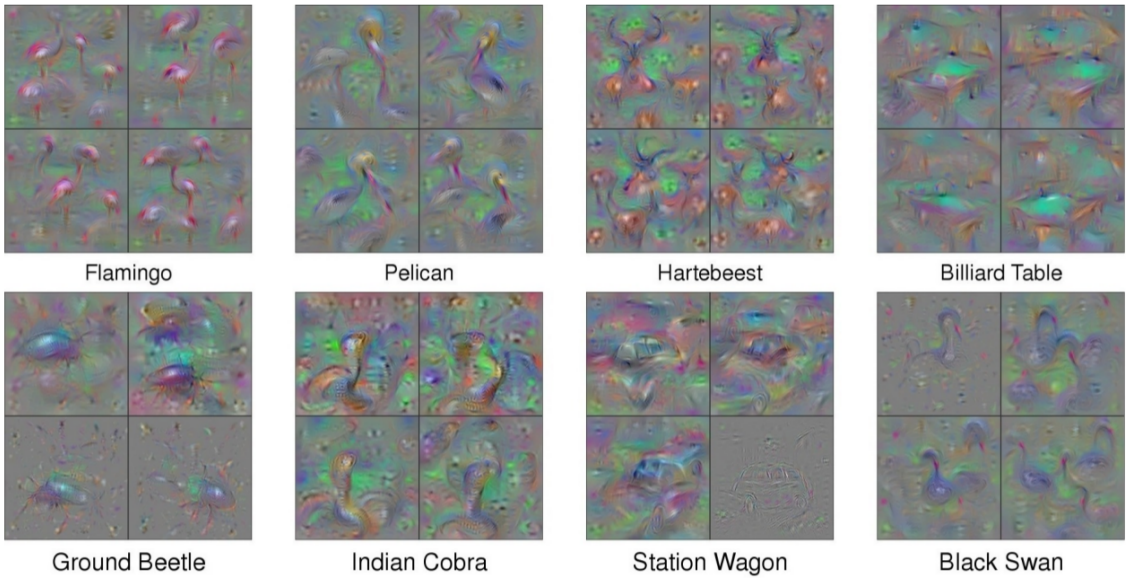
al. [Nguyen, Yosinski e Clune 2016], shows how such visualizations could be used to identify that a particular neuron from an image classification model would only produce high activations for underwater objects such as sharks, turtles, and scuba-divers.

The first technique proposed to synthesize high-activation images were the *activation maximization* algorithm [Erhan et al. 2009]. Considering the activation produced by a given neuron as the output of a non-linear function — i.e., the composition of operations performed by all previous layers plus the neuron itself —, an input the maximizes the activation produced by the neuron can be achieved by optimizing the pixels of an initially random image in order to the output of this non-linear function. This optimization in non-convex, so it is not possible to ensure that the optimization will lead to a global optimum. However, multiple local optima can be equally interesting in this context, as they may represent different features that may produce a high activation in the neuron. The activation maximization technique was first proposed to deep belief networks [Nguyen, Yosinski e Clune 2016] and later generalized to CNNs [Erhan et al. 2009]. Figure 4.7 displays the result of the activation maximization method for output neurons from the same model that learned to recognize eight different classes. Most features learned by this model are interpretable, so it is possible to clearly identify, on the images, the objects corresponding to the respective classes.

One issue with activation maximization is that there is no guarantee that the resulting image is interpretable. The displayed features may be too abstract and hardly recognizable by a human observer, which limits the usability of the technique as it is not possible to assert if the neuron's knowledge is indeed useful or not. This happens because the space of all possible images is too ample and many unnatural images may contain the same structures present in realistic ones, making them also recognizable by the model [Simonyan, Vedaldi e Zisserman 2013, Nguyen, Yosinski e Clune 2015].

Several regularization methods have been proposed to mitigate this issue. Such methods narrow the space of possible generated images by enforcing restrictions that often appear in realistic images [Simonyan, Vedaldi e Zisserman 2013, Wei et al. 2015, Mahendran e Vedaldi 2015]. For instance, Yosinski et al. [Yosinski et al. 2015] introduce four regularization methods to synthesize more realistic images: L_2 decay; Gaussian blur; small norm pixel clipping; and small contribution pixel clipping. The first two techniques remove high brightness amplitudes and high frequencies that are unlikely to appear in real-world images. Conversely, the other two aim to remove pixels whose influence in the produced activation is too small and probably are not useful to the analysis.

Figure 4.7: Activation Maximization Technique



The image above depicts the results achieved by activation maximization algorithm for the output neuron of eight different classes in a model trained on the ImageNet dataset [Deng et al. 2009]. These technique synthesize an image that produces a local maximum activation in a given neuron. When applied to neurons in the output layer, it allows the user to have a good insight of what features the model considers important for the neuron's respective class. The technique can also be applied to hidden neurons to achieve a better understanding of their role in the decision process. In the image, taken from Yosinski et al. [Yosinski et al. 2015], one can see that the neurons are activating for images that strongly resemble actual objects, even though in a quite unrealistic manner.

Another typical problem with activation maximization is that it often wields images with repeated features. This issue happens because in many situations if a feature appears twice in the input image, it will likely produce a higher activation value in the neuron that recognizes such feature than if it appears only once. However, this may turn the synthetic image too difficult to evaluate due to substantial visual pollution or high feature overlap. Figure 4.7 exemplifies this situation quite well. as most images presented there depict multiple instances of the corresponding object. A technique proposed by Nguyen et al. [Nguyen, Yosinski e Clune 2016] mitigate this problem by adding a center-biased regularization to the optimization process of the activation maximization method. This regularization penalizes changes to pixels in the border of the image, thus forcing the optimization to prefer to draw the features in the central region instead.

The improvements above are quite useful to increase the interpretability of synthetically generated images. However, these images are still likely to exhibit non-natural color and border patterns. An intelligent approach that has proved to be very successful

in delivering realistic images though is the use of generative neural networks (GNNs) as a prior to the optimization process. Instead of predicting an output label, these networks learn to generate, usually from an input vector, samples from the same distribution of those contained in the training set and have already been used to generate highly realistic images in several applications [Goodfellow et al. 2014, Dosovitskiy e Brox 2016]. Nguyen et al. [Nguyen et al. 2016] demonstrate that realistic images can be achieved with the activation maximization technique with the help of a pre-trained GNN. First they train a GNN to create realistic images from the same distribution of the images present in the original training set. Then, instead of directly optimizing the image pixels, they optimize an input vector that, once fed to the GNN, will generate a realistic image with high-activation in the neuron of interest on the original model.

Finally, another critical issue faced when synthesizing preferred images is the possibility of multifaceted classes [Nguyen, Yosinski e Clune 2016]. In the last layers of the model, it is very likely that each neuron is highly specialized in a single class — or at least it is desired. If the training set contains a multifaceted class — i.e., a class whose samples may have different and disjoint features among themselves — the traditional activation maximization technique may fail, as it may try to generate an image containing features from multiple facets, leading the image to become hard to interpret.

Nguyen et al. [Nguyen, Yosinski e Clune 2016] proposes an approach to address these issues. In their contribution, they initialize the activation maximization algorithm with an image obtained by averaging training instances from the same facets, i.e., with similar features among themselves, adding a bias to the optimization process that makes it converge to a local optimum also belonging to such facet. By doing so to multiple facets that can appear in images from that class, it is possible to uncover various interesting local optima that can be useful to explain what role the neuron learned to perform.

Recurrent Neural Networks: Feature understanding in RNNs is an even more significant challenge than with CNNs or DFNs. When trying to interpret the knowledge gathered by RNNs, it is not enough to analyze the activations produced by the neurons, but also their internal hidden states. A recurrent layer updates its internal hidden states every time it receives a new time step of the input sample and how these hidden states change according to the sequence they process is key to understand what these layers are performing. Nonetheless, many of the techniques used to visualize the features learned by RNNs are similar to those of CNNs and DFNs.

For instance, heatmap matrices have also been used in the same way as in other

architectures [Ming et al. 2017, Rong e Adar 2016]. These techniques can display, for instance, which hidden states activate for a given input. However, due to the intrinsic high-dimensionality of activations and inputs, coupled with the temporal behavior of RNNs, it is hard to use such visualization to compare multiple data, limiting the insights one can get from it.

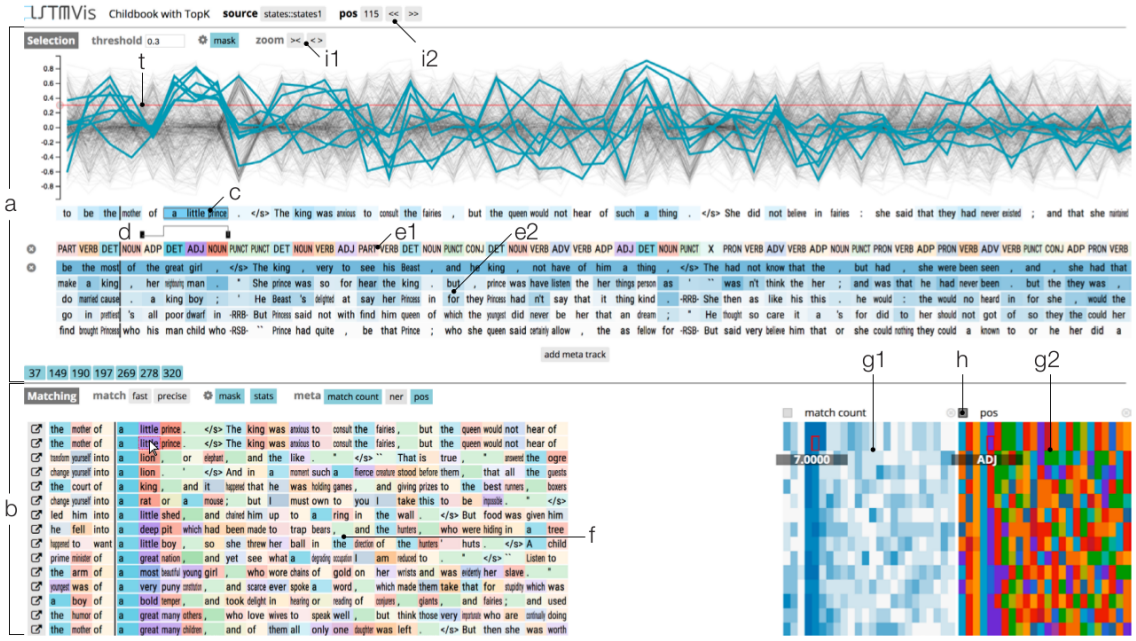
Ming et al. [Ming et al. 2017] proposes an alternative to this problem. They developed a method that co-clusters hidden states and input values — i.e., values from a single timestep — that display some correlation. In other words, each creates clusters of hidden states and clusters of input values and then it links hidden states clusters with input values clusters whose values generate strong activations on that group of hidden states. This way, the user can better identify the role of each hidden state by analyzing which inputs produce higher activations on it.

Heatmaps can also be used to evaluate the importance of each input unit for the hidden state evolution, similarly, as saliency maps do with CNNs. Li et al. [Li et al. 2015] propose a saliency heatmap displaying the so-called *saliency score* of each word in a sentence taken from the training set of a text classification task. They calculate this score by checking how much each timestep contributes to the final label.

With a similar approach, Karpathy et al. [Karpathy, Johnson e Fei-Fei 2015] were able to identify interpretable hidden states with clearly defined roles in the prediction process, such as to determine the existence of new line characters. Strobelt et al. [Strobelt et al. 2018] employ heatmaps to display metrics about the input text, such as to which part-of-speech (POS) each word belongs to, thus aiding the analysis of the hidden state evolution when the network process such words. Ding et al. [Ding et al. 2017] also uses a similar approach in a machine translation task. Such task differ from classification tasks as the output is also a sequence — the input sentence translated to another language — and not a label like in regular classification tasks. For this reason, the heatmap used by them displays the relevance of each input word to the decision of each output word. Thus, the designer can identify if the model is appropriately translating groups of words that cannot be translated adequately without knowing its context. They calculate this relevance value through an algorithm called *layer-wise relevance propagation* (LRP). Unlike gradient-based approaches, LRP does not require differentiable activations, which allows it to be used in models with word embedding layers.

The main goal of RNNs is to make on prediction on sequential data, such as text sentences or time series. When a time step of such sequences flows through the network,

Figure 4.8: LSTMVis



One of the tools proposed to address interpretability of recurrent architectures is the LSTMVis [Strobelt et al. 2018]. It displays the hidden states in a LSTM layer for a selected sequence of time steps. In this example, when the user chooses a text range, the visualization displays the evolution of the hidden states when processing that sequence of elements (a). Additionally, the VA tool provides multiple interactions, such as the ability to set a threshold that highlights hidden states above it (t), the ability to match and compare multiple input sequences that results in similar hidden states (b), and the visualization of user-selected metrics about the model (g1, g2).

it updates several hidden states values that will be kept until future timesteps. This means that time step values can affect the results even after several timesteps in the future. Understanding how each timestep affects the hidden state values and thus future predictions — and for how long they do it — is key to understand how a recurrent network performs. Some works have tried to address this problem by employing time series charts showing how hidden states evolve. Strobelt et al. [Strobelt et al. 2018] uses parallel coordinates to display the evolution of hidden states through the process of a sequence. By supporting this technique with heatmaps showing the POS of each word, they can find out that distinct regions with similar semantics in the text stream produce similar behavior in the hidden states (see Figure 4.8 for more details).

4.2 Open Challenges

Despite a large number of contributions that have been published on the topic recently, there is still a lot of room for improvements in the VA techniques aiming to support the multiple steps in the DL engineering workflow. Below we discuss some possibilities for future research both in the perspective of the three tasks we proposed (AU, TA, and FU) and from the perspective of the end-to-end requirements that ML practitioners often face when designing deep networks.

4.2.1 Architecture Understanding

Hyperparameter exploration: a particularly challenging problem faced in ML is to understand and interpret how the hyperparameters affect the final performance. Although DL practitioners have to make several hyperparameter decisions before training the model — e.g., number of layers, type, and size of each layer, regularization methods, activation functions, learning rates, etc. —, in many situations, these choices are mostly empirical, without a clear notion of which modifications on them would render a better performance. This limitation turns the fine-tuning of deep networks even more costly. Also though the VA community has put a considerable effort in the more general problem of analyzing the parameter space of a simulation model [Sedlmair et al. 2014], there is still a gap regarding specific solutions addressing DL needs. Some *drawing board* solutions where the designer controls all tuning aspects do exists [Smilkov et al. 2017], but they do not scale to large and complex networks.

A valuable contribution in this topic would allow for experts to visually inspect the hyperparameter space of the network and how different points in this space — i.e., different hyperparameter combinations — affect the final performance [Liu et al. 2017]. This requires more scalable techniques that can depict the hyperparameter space of large and complex networks, and that can meaningfully annotate regions of such space according to performance measures. There are contributions using Bayesian optimization to actively select which hyperparameter values try next based on inferences from current model performance [Brochu, Cora e Freitas 2010]. Although some contributions already attempted to apply this idea to deep networks [Snoek, Larochelle e Adams 2012, Snoek et al. 2015], more research in the topic is still needed.

4.2.2 Training Analysis

Training guiding and interaction: Interactive solutions are desirable in DL engineering because they can address two of the main DL problems: the difficulty to understand the model inner working and the long time required to train them [Amershi et al. 2014, Bernardo et al. 2017, Sacha et al. 2016]. Visualization techniques can quickly display what is happening during the training process and, by adding interaction, DL practitioners can inspect the model in real-time from several perspectives — e.g., learned features, convergence rate, class separation, etc. However, few of the current approaches aim to provide real-time and interactive visualization to designers [Smilkov et al. 2017, Chung et al. 2016, Qi et al. 2017]

4.2.3 Feature Understanding

Explainable models: Explaining how a model works is arguably the main VA contribution in DL. Nonetheless, most of the current solutions do not perform well if the training data is not easily interpretable. Many of the proposed visualizations focus on problems such as image classification or natural language processing, whose input data — images and texts — are perfectly interpretable by humans. Although these are two of the main fields where DNNs have been employed, many other applications — e.g., reinforcement learning, robot controlling, signal processing — are also being addressed by DL and would benefit significantly of VA techniques that could help interpret their models.

Different Applications: Existing VA contributions that aim to support DL engineering focus mostly in models employed in classification tasks. Although this is the most usual learning scenario for DL models, neural networks have been successfully adapted to several other scenarios, such as dimensionality reduction, denoising, clustering, and generative tasks. The ability to provide VA support in a way that can make the interpretability of such models easier is a topic that is yet to be fully explored, although a few contributions have already headed in this direction [Liu et al. 2018, Zahavy, Ben-Zrihem e Mannor 2016].

5 A NEW VISUALIZATION TOOL: ARCHITECTURAL TUNING WITH ACTIVATION OCCURRENCE MAPS

As explained in the previous chapter, the main focus of many VA tools proposed recently for DL support is to identify which features a neuron learned to recognize [Simonyan, Vedaldi e Zisserman 2013, Erhan et al. 2009]. However, these approaches open space for a new, still unanswered — in our perspective —, question: *how to evaluate if the learned features are indeed useful or enough for the prediction process*? Interpreting what the model is recognizing may be valuable to understanding how the prediction process works, but without the validation that the learned features successfully cover all the needs to perform the task, is impossible to evaluate in an interpretable manner whether the model is working correctly or not.

In this chapter, we propose a novel VA tool for DL support addressing this issue. Our tool comprises three visualizations: *Activation Occurrence Maps* (AOMs), *Occurrence Difference Maps* (ODMs), *Class Selectivity Maps* (CSMs). These three VA methods aim to guide DL designers in decisions such as: (1) identify neurons that do not contribute to the model's decision process, either because they are not significantly selective — i.e., they output similar activations for too many classes — or because their role is redundant to other neurons in the same layer; (2) to decide whether the size of a layer should be increased due to the current number of neurons not being enough to build selective features for all classes in the training set; and (3) to decide whether more layers should be added to the model because the current number of layers is not enough to have highly specialized neurons — i.e., neurons that activate only for one class. Additionally, we introduce a metric to evaluate the selectivity of a neuron by measuring how the activation produce by it differs from one class to the others. This metric is used by the three methods proposed here.

For the context of this work, we only take into account convolutional neurons — also known as kernels. However, our method can be quite easily modified to handle fully-connected or recurrent neurons. Regarding the tasks presented in Chapter 3.4, our approach lies in the border between AU and FU. While the tool focus on evaluating the features learned by the model (FU task), it does so aiming to identify whether the chosen architecture can be improved in some way (AU task).

This chapter is organized as follows. In Section 5.1 we discuss the modeling tasks that our method aims to address. Section 5.2 details each visualization method employed

in our technique, along with the selectivity metric we propose. In Section 5.3, we present some experiments that demonstrate how our approach can be used to address the modeling tasks in two widely known image classification datasets. Finally, in Section 5.4 we outline some possibilities for future research.

5.1 Modeling Tasks

During the design workflow (see Figure 3.6) of a DNN, a DL practitioner has to make several architectural choices, such as the number, type, and size of each layer [LeCun, Bengio e Hinton 2015]. Such architectural decisions are essential for the model to achieve good performance. Unfortunately, finding the optimal architecture for a DNN to solve a learning task is a difficult problem, and there is no analytical way to find such an architecture. Often, DL practitioners end up choosing an overestimated or underestimated the number of layers or neurons, which leads to the following problems:

- **Too few layers**: When a model has fewer layers than it should, it may fail to distinguish classes with very similar low-level features among them. In general, the first layers of a DNN learn elementary features such as border orientations and color contrasts, and, as deeper as one goes into the DNN's layers, it starts to recognize more complex and abstract objects. If two classes share many simple features — e.g., a *car* and a *truck* class in an image classification dataset —, then the model will likely need a higher amount of layers to build features abstract enough to distinguish between them. If the model does not have that much amount of layers, it will end up mislabeling samples quite often.
- **Too small layers**: Neural networks change the representation of the input sample layer-by-layer, thus iteratively generating more abstract representations that better distinguish the classes in the training set [Rauber et al. 2017]. A DNN's layer does so by taking the features recognized in the previous layer and combine them into more complex features. However, if the layer does not have enough neurons to learn to recognize all important features on that abstraction level in the first place, the next layers will not be able to build more abstract features properly, which will lead the model to mispredictions. Note that, in this case, increasing the number of layers — or even the size of a deeper layer — does not help, as the model does not have, at that point, all the features it needs anymore.

- **Too many layers or too large layers**: Several issues arise from overestimating the optimal number of layers or their sizes as well. Both situations translate to an increased number of parameters, which turns the DNN more prone to overfitting. Additionally, DNNs with too many components (and, consequently, parameters) take longer to train and require much more memory space to be stored, which makes their use quite prohibitive in a system with strict space requirements, such as embedded devices. Given how the use of DL is increasing in areas such as robotics and embedded computing [Han, Mao e Dally 2015], such concerns are becoming even more critical.
- **Ineffective neurons**: Large networks seldom distribute the learned features equally across their components. After training, a layer with a large number of neurons is more likely to contain neurons that recognize unique features — and thus playing an important role in the decision process — and neurons that output similar activations to most or even all classes. Additionally, these neurons in the latter category may also harm the model's performance, as they may introduce some degree of overfitting to the prediction [Han et al. 2015]. Finding ways to identify these neurons is key to build effective models.
- **Redundant neurons**: For similar reasons as the issue above, it is common for DNNs to contain layers where many neurons learned to recognize very similar features, which adds a high degree of redundancy to the model [Denil et al. 2013]. By identifying the neurons where this behavior occurs, the DL practitioner can significantly reduce the size of the model without a drastic decrease in performance.

By taking into account the issues above, we identify three tasks that our VA solution aims to solve:

- **Finding ineffective and redundant neurons**: Finding neurons that are not playing an important — or beneficial — role in the decision process may be very helpful to improve the quality of the model. By removing these neurons, the DL practitioner can decrease the overfitting that may be occurring in the model or even make the model more suitable for limited memory applications. Additionally, these neurons can be reinitialized, instead of removed, in the hope of learning more useful — and not redundant — features.
- **Finding too small layers**: If one can identify a layer that has fewer neurons than it should, it is possible to increase the model's performance in a more optimized way.

By increasing the number of neurons only on that layer, the DL practitioner can learn more important features in that abstraction level — and, consequently, better high level features in deeper layers —, improving the overall model selectivity.

- **Finding too shallow models**: For a model to achieve a satisfying performance, the activations produced by the last hidden layer should contain high selectivity. In other words, the last hidden layer activations of any given class must be completely different from all other classes. If this does not occur, the model will fail to achieve a satisfying performance. If we can identify that the problem happens because the model needs more layers — as opposed to a need for more neurons, for instance — we can more quickly achieve a model with efficient performance.

It is not feasible to perform these tasks in a completely automatic way. It requires to compare the — often high-dimensional — activations produced by all neurons in a layer for all elements in the training set. Not only this is very computationally expensive, but also these tasks lack an exact way to quantify an answer. For example, if two classes display very similar activation patterns in a layer, it sounds reasonable that the model needs more layer to build more abstract patterns that can differentiate those classes. However, it is not so simple to quantify how similar these activations have to be for that to be true. For this reason, we developed a VA approach to address the questions above. We detail this approach in the section below.

5.2 Visualization Techniques

As most VA tools reviewed in Chapter 4 [Liu et al. 2017, Kahng et al. 2018, Simonyan, Vedaldi e Zisserman 2013, Nguyen et al. 2016, Alsallakh et al. 2018], our techniques analyze the activation values produced by the network to evaluate the chosen architecture. This choice is motivated by the fact that it is much easier to understand how each neuron correlates to a particular class by looking at the activations it produces for samples of that same class. The role of a neuron is to identify whether a feature exists in the input or not. If it does, the neuron will output high activation values. Otherwise, low (or zero) activation values are expected.

However, to evaluate if the learned feature is useful or not, a more sophisticated analysis is required. A neuron may learn to identify features that appear in every class and do not contribute to the construction of higher-level features. Conversely, a neuron

may learn a feature that is very specific to a few input samples, and it does not translate as useful knowledge to recognize the class as a whole. Either way, this feature may not be enough — or even helpful at all — to the decision process. In some approaches, this issue is addressed by taking into account the mean activation produced by the neuron for all samples from a particular class [Kahng et al. 2018, Alsallakh et al. 2018].

This approach may sometimes be misleading, though. The activations produced by hidden neuron are taken as input by the next layer in the model, which makes their overall important for the decision process highly dependent on the weights it interact within the following layers. For this reason, a high activation value may not be that important if it only interacts with small weights. Additionally, a pair of redundant neurons may output different mean activation values but do mainly the same thing due to the interaction with the weights in the following layers turning their contribution quite similar. These reasons motivated us to research a new method to analyze activations in a way their roles may become more easily identified and evaluated.

In our technique, we look at the *proportion of positive activations* instead of their actual values. Nowadays, most DNNs use the *ReLU* [Dahl, Sainath e Hinton 2013] activation in their hidden layers. This activation function clamps to zero negative activation values, leaving the actual ranging between 0 and ∞. When a neuron with *ReLU* outputs positive activations, it is telling the next layer that it recognized, to some extent, the feature it searches for. A zeroed activation, on the other hand, means that such a feature is not present in the input sample. Given such characteristic, we can assume that a neuron that learned a feature that is selective for any given class will output positive activations for most, or even all, samples from that class, but seldom will do so for elements of other classes. This way, if we identify neurons producing a high proportion of positive activations for samples of any class c_i and a low proportion for samples of any other class c_j, we can denote this neuron as selective towards the pair of classes (c_i, c_j).

To allow DL practitioners to identify these neurons, to analyze their activation patterns, and to evaluate their role in the decision process, we propose here three visualization techniques: *Activation Occurrence Maps*; *Occurrence Difference Matrix*; and *Class Selectivity Maps*. We explain these techniques in details next. Although we focus on CNNs and convolutional neurons in our explanations, the methods can be used for other types of layers as well.

5.2.1 Activation Occurrence Maps

The goal of any hidden neuron is to identify features present in the sample data and to use them to build a new representation for the original input. When coupled with a *ReLU* activation, this neuron does so by producing a zero activation if the feature is not present in the input or a positive value proportional to how strong the feature's presence is. In the case of convolutional neurons, such a behavior applies for each position in the high-dimensional feature maps as well. For instance, take as example an image classification model. The first convolutional layer with *ReLU* takes as input the original image and returns, for each neuron, another image — i.e., the feature map —, usually with the same (or slightly fewer) dimensions than before. Each pixel in this output image contains a value proportional to whether the feature it looks for is present in that neighborhood in the original image or not [Goodfellow, Bengio e Courville 2016].

To allow for the understanding of how a neuron behaves for samples of different classes or how different neurons behave for samples from the same class, we have to analyze the occurrence of positive activations in different regions of the image output for different samples. A region-based analysis is desired over a global one because different classes may display very similar features, but often in distinct regions of the image [Alsallakh et al. 2018]. To perform such analysis of a given layer, we build an *Activation Occurrence Map* (AOM) for each (neuron, class) pair in that layer.

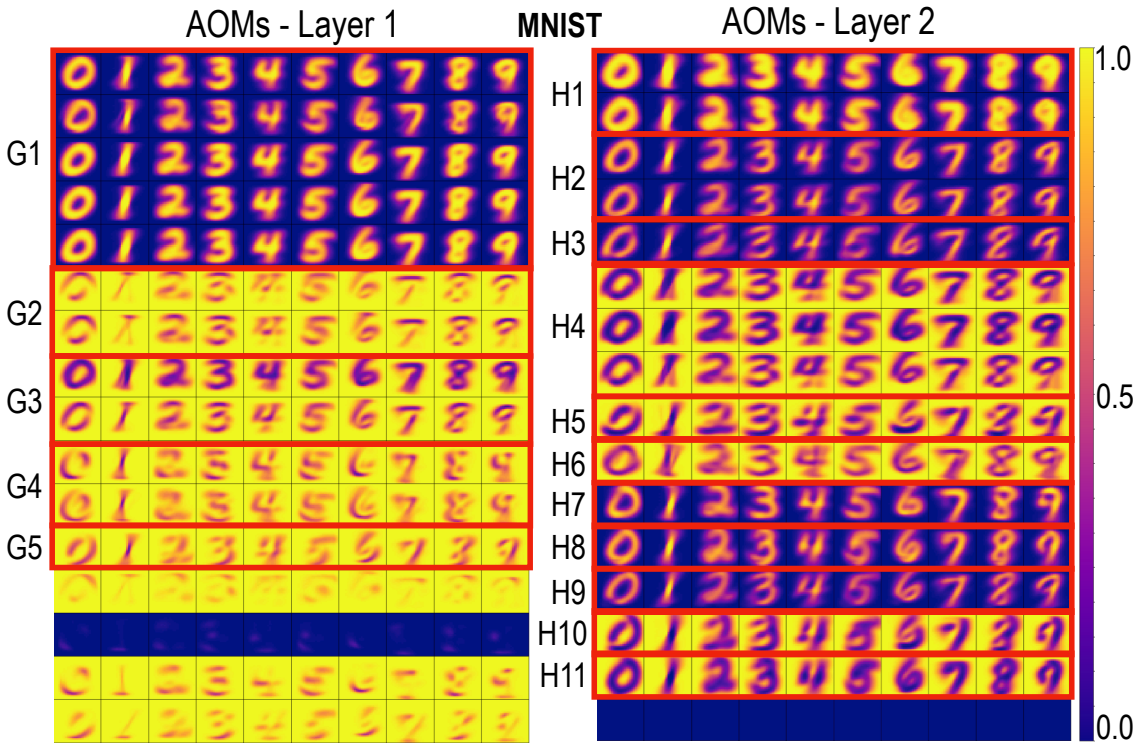
For any particular neuron n and any particular class c in the training set, we compute the AOM $M^{n,c} \subset \mathbb{R}^{WxH}$, where W and H are the width and height of the feature map produced by neuron n when the network process any input image. Each cell $M^{n,c}_{i,j}$ in the AOM denotes the proportion of training images belonging to class c activating positive values at the pixel (i, j) of the feature map produced in the neuron k. This value is calculated as follows:

$$M^{n,c}_{i,j} = \frac{1}{|\mathbf{D}^c|} \sum_{x \in \mathbf{D}^c} U(a^n(x)_{i.j}) \tag{5.1}$$

where $\mathbf{D}^c$ is the set of images in the training set $\mathbf{D}$ that belong to class c; $a^n(x)_{i.j}$ is the activation value produced by n at position (i, j) for the input sample x; and $U(y)$ is a Heaviside step function that outputs 1 whenever y is positive and 0 otherwise.

Figure 5.1 displays the AOMs computed for the neurons in the first (left) and second (right) layer of a CNN trained with the MNIST dataset [LeCun et al. 1998]. In

Figure 5.1: Activation Occurrence Maps for the MNIST Model



The picture above depicts the two AOM views produced by the first and second layer in a model trained with the MNIST dataset (details in Chapter 5.3). In each view, rows represent the 16 convolutional filters (neurons) in the layer and columns represent each of the ten MNIST classes. Red rectangles highlight groups of neurons that often activate for similar features. For instance, groups G1 and H1 seem to identify the digit structure, groups G2, G4, and G5 recognize different border orientations, groups G3 and H4 identify the background, and groups H2, H3, H7, and H9 distinguish features from different handwriting styles.

this visualization, the values $M_{i,j}^{n,c}$ are encoded with an ordinal colormap ranging from blue (no samples activate positively) to yellow (all samples activate positively). In this image, rows represent different neurons in the layer and columns correspond to all ten classes in the dataset. We group rows according to the similarity between AOMs — evaluated via a *hierarchical clustering* technique — to improve the quality of the analysis.

In case a neuron is producing very similar AOMs for all classes, it is likely that the information it passes to the next layer does not help increase the selectivity among classes. In other words, the neuron is probably not useful for the decision process. For instance, if a feature map's pixel produces similar values — whether positive or zero — to most samples across any class, this pixel is unlikely to provide any useful information for the next layer in the DNN. This pattern appears in rows 13 and 15 in Figure 5.1 (left). The corresponding neurons display very similar AOMs for all ten classes, with almost

all positions seldom activating, regardless of class. These neurons are not contributing to the network's decision process and could even be adding some overfitting that harms performance.

The AOMs also allows the DL practitioner to find redundant neurons. The neurons corresponding to the AOMs in the same group in Figure 5.1 (denoted by red rectangles) display such cases. Here, neurons from the same group can individually produce very different feature maps to each of the classes. However, for any class, the neurons in the same group produce very similar AOMs, which demonstrate that these neurons indeed learned redundant features.

5.2.2 Occurrence Difference Matrix

Although AOMs are quite useful, they do suffer from a lack of scalability. To thoroughly analyze a single layer in the DNN, we would have to display $|N| \times |C| \times W \times H$ cells, where N is the number of neurons, C is the number of classes, and (W, H) are the dimensions of each neuron's feature map. This characteristic poses a problem when handling models designed for complex learning tasks, which would often require very large layers. Additionally, these tasks sometimes contain training sets comprising hundreds of classes [Deng et al. 2009], which negatively affects the approach's scalability in an equal manner.

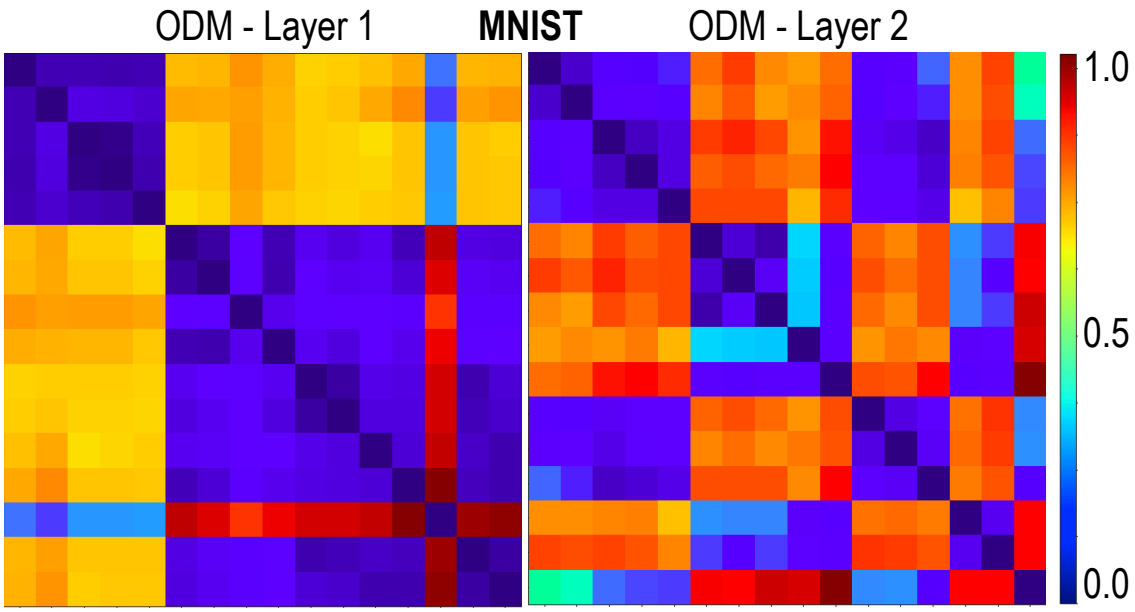
To avoid this problem to some extent, we developed an alternative visualization to analyze a particular layer, the Occurrence Difference Matrix (ODM). This technique employs a symmetric matrix D where each cell $D_{n_i,n_j} \in \mathbb{R}^+$ measures the average difference between the AOMs from neurons n_i and n_j for all positions in the corresponding AOMs for all classes. The ODM D_{n_i,n_j} is computed as follows:

$$D_{n_I,n_J} = \frac{1}{|C| \times W \times H} \sum_{c \in C} \sum_{i=0}^{i<W} \sum_{j=0}^{j<H} |M_{i,j}^{n_I,c} - M_{i,j}^{n_J,c}| \tag{5.2}$$

where C is the set of training classes; W and H are, respectively, the width and the height of the neuron's feature map; and $M_{i,j}^{n,c}$ is the cell (i, j) in the AOM of neuron n for class c.

The ODMs calculated for the AOMs displayed in Figure 5.1 are shown in Figure 5.2. In this visualization, each cell D_{n_I,n_J} encodes, via a blue-to-red ordinal colormap, how similar the AOMs for neurons n_I and n_J are. To easily compare the two visualizations, we use in Figure 5.2 the same order from Figure 5.1. However, in practice, the DL

Figure 5.2: Occurrence Difference Matrix for the MNIST Model



The two ODM views for the first two convolutional layers in the MNIST model. The ODM consists in a symmetric matrix that displays, on each of its cells, the average difference between the AOMs corresponding to each pair of neurons in the layer. It provides a more compact — but also less detailed — overview of how similar different neurons are than the original AOMs. It is particular useful to identify groups of redundant neurons.

practitioner could apply some matrix-reordering algorithm [Behrisch et al. 2016] to sort the neurons in groups of similarity, making it easy to spot groups of redundant neurons. In Figure 5.2, one can note that the neurons with strong similarity in Figure 5.1 keep a similar similarity level in the ODM visualization. The main advantage of the ODMs is its relatively more compact representation, which allows the visualization of larger layers. The gain in scalability is significantly high when handling models whose training set has too many classes, as the ODM's size only increases with the number of neurons in the layer, not with the number of classes. However, although ODMs are quite useful to spot redundant neurons, they are not so suitable for the other proposed tasks, as they do not show the learned features, only if they are similar or not.

5.2.3 Class Selectivity Maps

We also propose a third visualization called Class Selectivity Maps (CSMs). This technique is designed to aid DL practitioners in finding how selective it is each pixel in the feature map of a neuron. Although an individual AOM is quite useful to identify in

which regions the features appear and how often the features appear for that class, they do not provide any insight into how selective that feature is. The CSMs were developed to address this issue. They display how selective each cell in the AOM in comparison to other AOMs from the same neuron but for different classes. They follow the same format of the AOMs, but each cell denotes the *class selectivity* of the pixel. If a pixel activates positive values often for samples from a given class c, but it does not do it for any other class, then this pixel is considered to be selective towards class c. In other words, it learned how to distinguish, to some extent, samples from class c.

A CSM $S^{n,c}$ is represented by a matrix where each element $S^{n,c}_{i,j}$ denotes how selective the position (i, j) of the neuron n's output image is for class c, and is computed as:

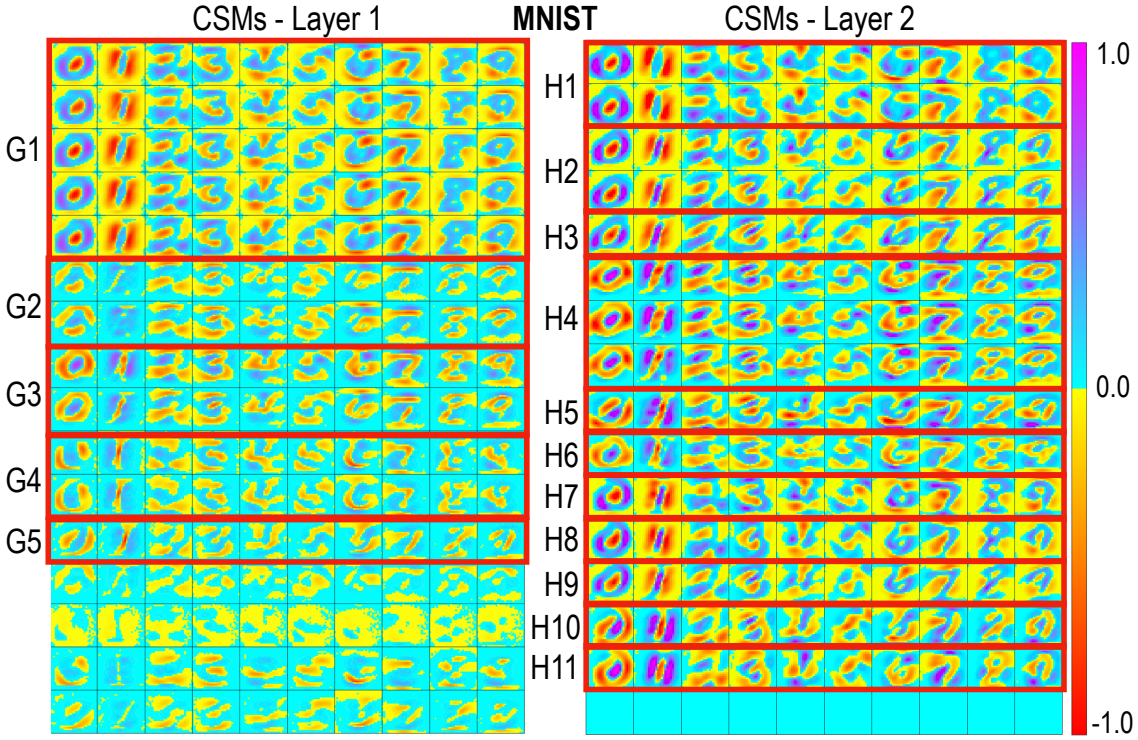
$$S^{n,c}_{i,j} = \frac{1}{|C| - 1} \sum_{d \in C}^{d \neq c} M^{n,c}_{i,j} - M^{n,d}_{i,j} \tag{5.3}$$

where C is the set of all classes and $M^{n,c}_{i,j}$ is the position (i, j) in the AOM corresponding to neuron n and class c.

In Figure 5.3 we show the CSMs produced by the same layers of the AOMs displayed in Figure 5.1. As before, rows represent neurons and columns represent classes. The rows are sorted in the same way as the AOMs for consistency. In this case, we opt for using a two-segment colormap ranging from cyan (0.0) to purple (1.0) for positive values and from yellow (0.0) to red (1.0) for negative values. We do so because we want to emphasize the difference between negative and positive class selectivity values, even when they are closer to zero. The reason for this is because cells with positive values in the CSMs indicate pixels that are selective towards the corresponding class, i.e., these pixel activates more often when the input sample is indeed from that class and seldom does so when it is from any other class. On the other hand, a negative cell value indicates that the corresponding pixel is dismissive towards that class; i.e., it is more likely to activate when the input sample is not from that particular class. This way, CSMs are useful to spot pixels that are selective towards a given class c in both ways: either because a positive activation there means that the input is likely to belong to c or because a positive activation there means that the sample probably does not belong to c.

As an example, it is noticeable in Figure 5.3 (right) that several neurons in the 2þlayer are selective in the areas inside the digit for images from class 'zero'. However, other neurons are more selective in the the digit zero's inner circle — e.g, the one in group

Figure 5.3: Class Selectivity Maps for the MNIST Model



We display above the CSM visualization for the first two convolutional layers in the MNIST model. Similarly to the AOM view, each row corresponds to one of the 16 neurons in each layer, and the columns represent the ten classes in the MNIST dataset. We also keep the same sorting and grouping for consistency. With the CSMs, the user can identify regions in the output that are more selective for some particular classes — i.e., the neuron produces positive activations more often for samples that belong to that class than for others. As an example, the neurons in the H1 group display a significant selectivity towards round-shaped structures in the digits. Conversely, the H2 group displays are more selective towards flat shapes such as straight lines. These regions are more likely to help the prediction process because they create features in the intermediate data representations that effectively distinguish different classes, making classification easier.

H11 —, which indicates that these neurons learned different features of the class and that their selectivity lies in different regions of the feature map.

5.2.4 Neuron Selectivity Metric

The CSMs are quite useful to analyze the selectivity of different regions of a neuron towards each particular class. However, they do not assign an exact selectivity value to each neuron. Such a value would help decide which representative neuron should we keep in the model from a group of redundant neurons. Additionally, it could be used to

help the evaluation of whether a neuron is useful enough for the model or if it could be removed or reinitialized. The ability to do this would allow the DL practitioner to decrease the size and complexity of the model without losing significant performance — or even actually increasing it.

Given the matrix S^n with all the CSMs for neuron n (containing values for every pixel and class), we compute the neuron selectivity $s(n)$ of n as

$$s(n) = \sum_{c_0,c_1 \in C | c_0 \neq c_1} |\text{avg}(S^{n,c_0}) - \text{avg}(S^{n,c_1})| \tag{5.4}$$

where C is the set of all training classes, and $\text{avg}(S^{n,c})$ is the average of all values in the CSM for neuron n and class c. In the next section, we demonstrate, with a series of experiment, how the selectivity metric and the visualizations can be used to address the proposed tasks.

5.3 Experiments

We performed several experiments to demonstrate how a DL practitioner can use our VA tools to address the tasks described in Section 4. In these experiments, we use two widely-known image classification datasets: (1) MNIST [LeCun et al. 1998], comprising a large amount of single-channel images of handwritten digits ranging from 0 to 9 — which denotes the class of each image; and (2) CIFAR10 [Krizhevsky e Hinton 2009], that contains a large amount of images displaying an object belonging to only one out of ten classes — airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. In both cases, we aim to design, from scratch, a DNN that classify the images in their respective classes. These datasets have a considerably distinct degree of difficulty, so our starter model for each dataset is significantly different as well.

For the MNIST model, we first design a simple model containing two convolutional layers with 16 neurons each and 3x3 kernel size; 1 max pooling layer with 2x2 mask size; 1 hidden fully-connected layer with 16 neurons; and the decision layer with ten fully-connected neurons — one for each class. All learnable hidden layers have *ReLU* activation function, while the decision layer uses *softmax*. The model was trained for a single epoch with the *Adadelta* optimizer [Zeiler 2012] and a batch size of 128 samples. The model was implemented using the Keras framework [Chollet et al. 2015]. This model was able to achieve 96.96% test set accuracy after training.

On the other hand, for the CIFAR10 model, we designed a much more complex initial architecture. It contains two convolutional layers with 64 neurons and 3x3 kernel size; a max pooling layer with 2x2 mask size; a dropout layer with 0.25% dropping rate; again two convolutional layers with 64 neurons and 3x3 kernel size; a max pooling layer with 2x2 mask size; a 0.25% rate dropout layer; two convolutional layers with 64 neurons and 3x3 kernel size; a max pooling layer with 2x2 mask size; a 0.25% rate dropout layer; a hidden fully-connected layer with 64 neurons; a 0.5% rate dropout layer; and an output layer with 10 fully-connected neurons, one for each class. All hidden layers with learnable weights have *ReLU* activation function here as well. This model was also implemented with Keras. It was trained by 10 epochs using a Stochastic Gradient Descent optimizer with momentum rate of 0.9 and learning rate of 0.01. After training, this model achieved a test set accuracy of 80.54%.

5.3.1 Layer Size Reduction of MNIST Model

In our first experiment, we use the proposed VA tools to simplify the MNIST model without losing accuracy significantly. After training the model, we built the visualizations shown in Figures 5.1, 5.2, and 5.3 using all the samples in the training set. By following the insights provided by our approach, we remove neurons that do not seem to help the decision process of the network — either because the features they learned to recognize are not useful for any class or because they belong to a group of redundant neurons. Note that, in practical settings, the DL practitioner may choose to reinitialize the neurons instead of removing them. We prefer the later for simplicity.

Our analysis begins with the AOMs of the first convolutional layer — Figure 5.1 (left). At first glance, it is possible to identify a neuron candidate for removal in row 14. Most positions of this neuron seldom activate positive values for any sample, regardless of class. With such behavior, this neuron is unlikely to be useful for the model's decision process. Additionally, it is possible to note multiple groups of neurons with quite a similar activation occurrence patterns. Neurons in G1, for instance, often activate positive values in pixels inside the digit area, and rarely at positions in the background. This behavior means that all these neurons learned to recognize the same feature, the overall digit structure.

Other neurons, however, display a different pattern. These neurons activate positive values more often in background pixels than in digit pixels. These neurons seem to

have learned to recognize features such as the border orientation of the writing. However, they do not all have the same role in the decision process. Some of these neurons activate more frequently in areas above the digit structure — e.g., group G4 —, while others activate more often below the digit structure — e.g., group G2.

Nonetheless, some of these neurons do present a substantial redundancy towards each other. The ODM plot in Figure 5.2 (left) corroborates this hypothesis. In this image, we see several groups of cells with dark blue shades representing groups of neurons that are redundant among themselves.

As follows, we analyze the CSMs produced by these neurons — Figure 5.3 (left). While the AOMs tell us which of the neuron's regions activates more often for each class, the CSMs tell us how selective these activations are. By looking at them, we can notice that the last three rows contain CSMs with no selectivity — or even dismissive — regions, which puts more confidence in our assumption that these neurons are not relevant for the prediction process. Conversely, other neurons, such as those in the G1 group, contain regions in their CSMs with strong selectivity for some of the classes. This observation tells us that these neurons are indeed useful for distinguishing classes and may be necessary for the overall decision process.

Now, we proceed by using the observations we gathered to remove neurons that we believe to be ineffective for the model's decision process. These neurons are represented by the last three rows in Figures 5.1, and 5.3 (left). Additionally, we group neurons with strong similarity, which may denote redundancy, and select a single representative for each group to keep in the network. The red rectangle in Figures 5.1, and 5.3 denotes the groups we chose. The neuron we choose to keep in each group is the one with the highest neuron selectivity metric, as described in Section 5.2.4.

After removing the undesired neurons, we end up we only five neurons in the first layer of the model. To demonstrate how these five neurons are indeed enough to perform the prediction task, we freeze all the weights in these five neurons — i.e., they cannot be modified by backpropagation anymore —, making the layer completely untrainable. Then, we resume the training of the model from the second layer onward. Doing this, we force the model to learn to perform the classification task with access only to the data representation previously determined by those five neurons. If this representation were not enough to perform the task, the model would not be able to recover the original performance, regardless of how long we train. In this case, the deeper layer would not have access to all low-level features they need to build more abstract ones. However, this

is not what happens. After a single retraining epoch, our model achieves 96.93% test set accuracy, which is virtually the same it had initially been with all 16 neurons in the first layer. This result shows that the five chosen neurons indeed cover enough features to achieve the same prediction ability that the model already had with 16 neurons.

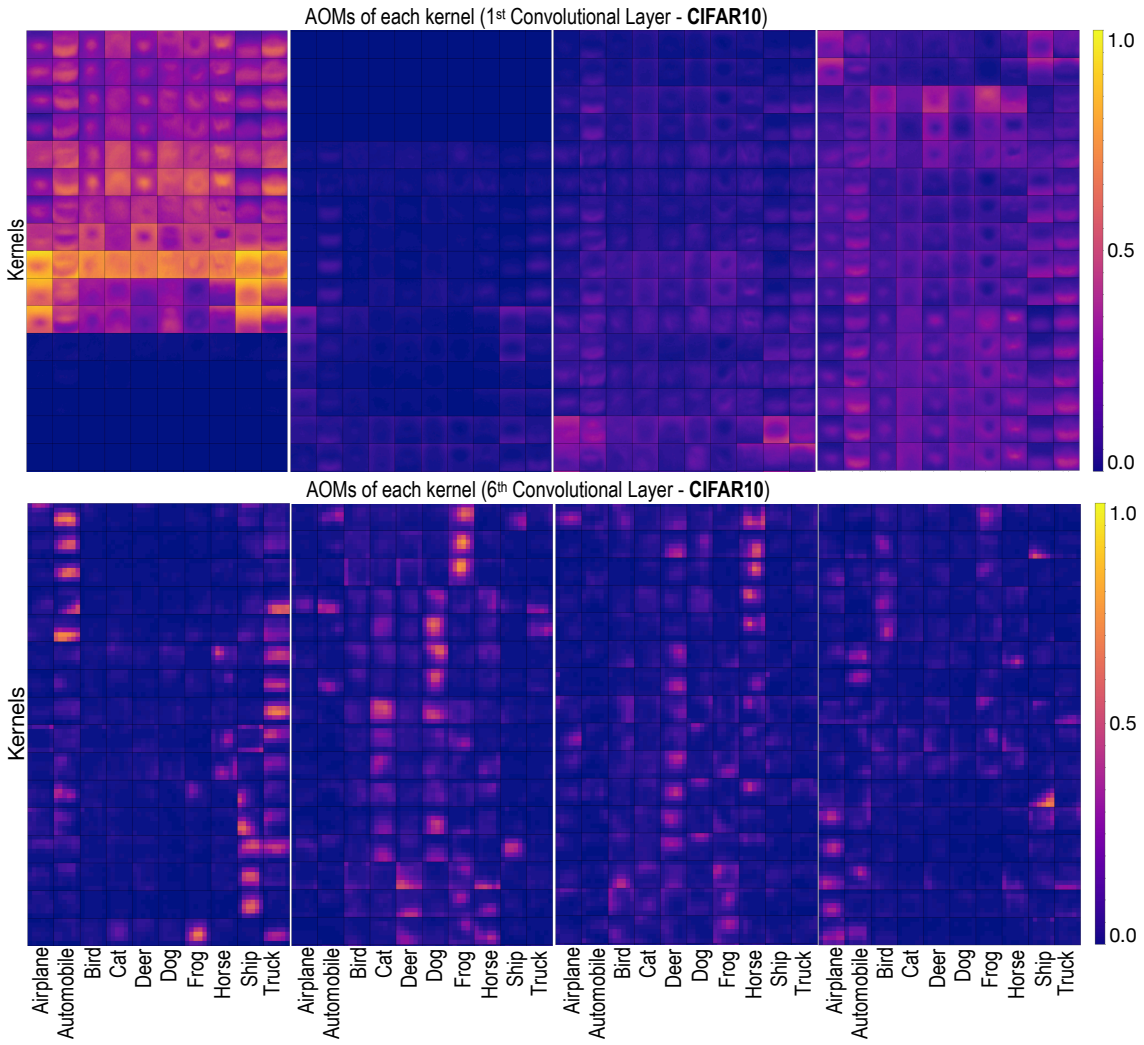
For completeness, we repeat the same process in the second convolutional layer of the MNIST model. By looking at the AOMs for this layer — Figure 5.1 (right) —, we can notice a higher diversity when it comes to features learned across the neurons. For instance, some neurons have specialized themselves in recognition of particular writing styles. The neurons in the H8 group, for example, recognize digits written in 'rounded' writing, while the neuron in H7 recognizes flatter writings. We perform a similar network edit as we did in the first layer, freezing all weights in layer 2 (the weights in the first layer continue to be frozen as well). This way, the model can only modify the fully-connected layers from now on. Even so, after retraining for one additional epoch, the model achieves an accuracy of 97.69%. This result shows that our method not only was able to simplify the model in terms of the number of components without losing performance but also managed to get free from some overfitting that existed in the original configuration.

5.3.2 Layer Size Reduction of CIFAR10 Model

In our second experiment, we apply the same analysis as before to the CIFAR10 model, which tackles a significantly harder learning problem. Figure 5.4 displays the AOMs for the first and sixth convolutional layer in the network. At first glance, we see that this visualization gives us useful insights into the learned features of both layers. For example, several neurons from both layers fail too often activate positively for any class, which implies that the features learned by such neurons are not useful for classification. More interestingly, though, we notice that first layer neurons that activate positive values frequently, do so to multiple classes. This behavior suggests that the features learned by the first layer are too low-level to distinguish individual classes and that the model indeed needs more layers to build features suitable for classification.

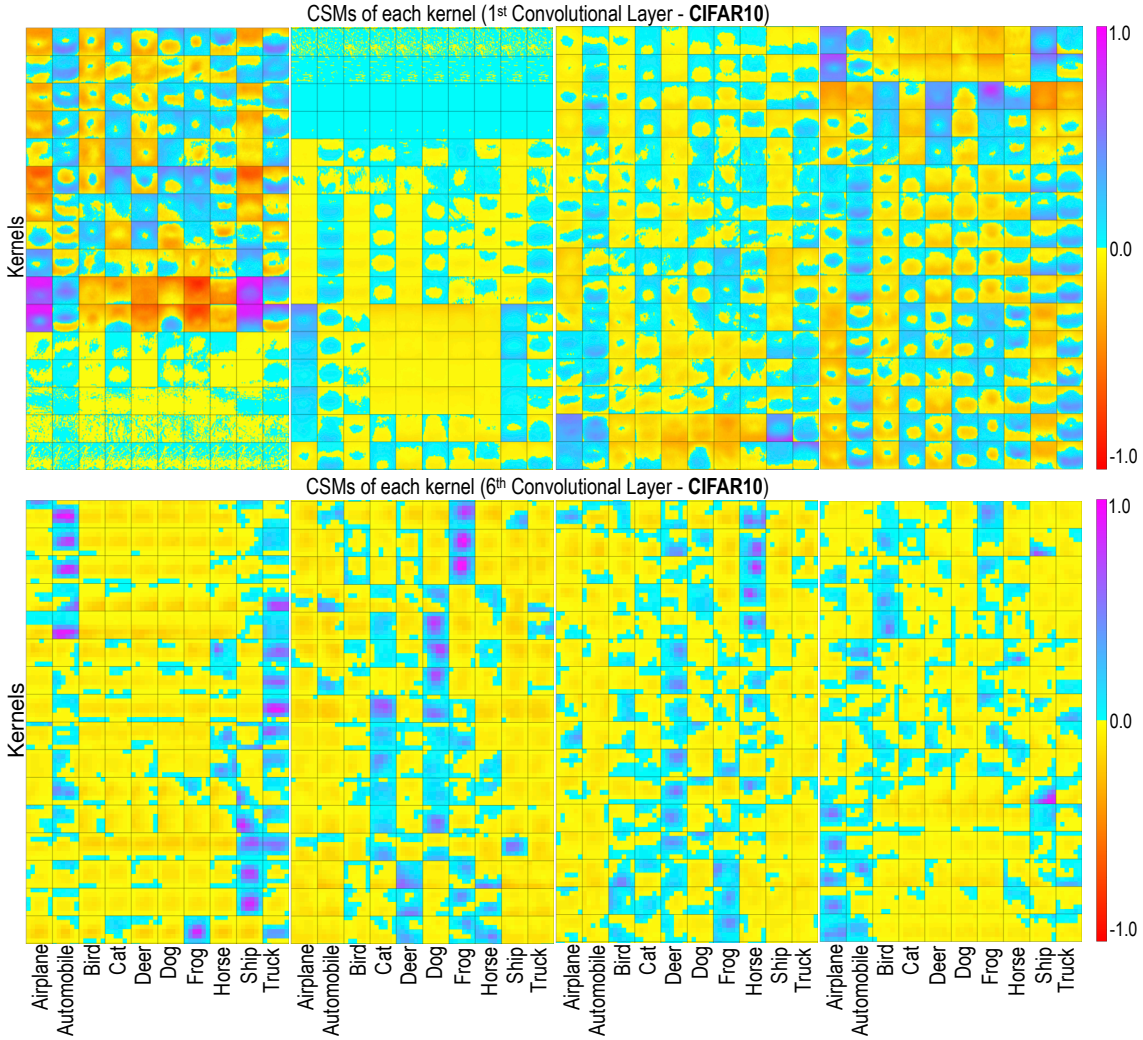
Conversely, the sixth layer shows the exact oppose behavior. In this layer, most neurons are very selective towards one class, i.e., they only activate often for elements belonging to that class. We can see this pattern as an indication that, at this point, the model has already learned features abstract enough to perform classification and to add more convolutional layers is unlikely to improve the performance significantly.

Figure 5.4: Activation Occurrence Maps for the CIFAR10 Model



The picture above shows the AOMs for the 64 neurons in both the first and the sixth layers of the CIFAR10 model. In this view, we sort neurons according to the similarity order computed by applying aggregative clustering in the view's rows. We can quickly notice that the first layer did not learn features that are discriminative enough to distinguish individual classes, which indicates that the model needs more layers to solve the problem. However, even though the neurons in the first layer often activate for elements from most classes, they do not display the same activation pattern for all of them. This behavior is important because it indicates that the layer is making the classes, to some extent, more distinguishable for the next layers. Otherwise, the model would not be able to perform classification successfully, and there would be a need to modify the first layer. The sixth layer, on the other hand, contains more discriminative neurons. Most of the neurons in this layer activate positive values only for a single class. When the model achieves this pattern, it indicates that the model does not need to transform the data representation further to perform classification. This way, adding more layers is unlikely to improve the model significantly.

Figure 5.5: Class Selectivity Maps for the CIFAR10 Model



The picture above displays the two CMS views for the 64 neurons in the first and sixth layer of the CIFAR10 model, following the same order of Figure 5.4. The CSMs can give us some interesting insights that are not so obvious when looking at the AOMs. For instance, even though several neurons from the first layer frequently produce positive activations for all, or almost all, classes, they are usually more selective towards only a few of those classes. This kind of insight gives more confidence to whether the neurons display some selectivity or not.

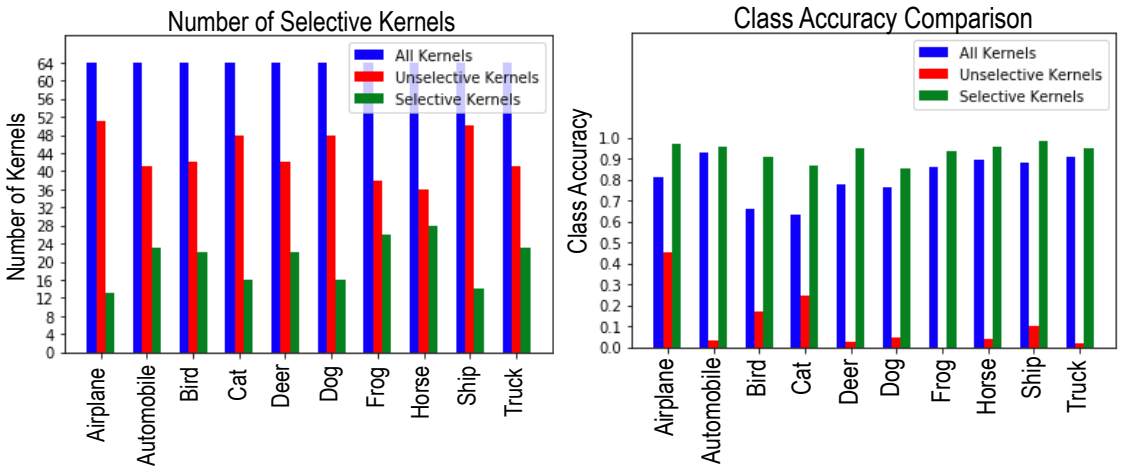
That said, we can also notice that some classes — such as cat and deer — do not achieve a 100% occurrence rate in any neuron, which indicates that the features related to this class that was learned by the model are not broad enough to encompass all kind of samples that these classes may contain. Such a behavior is a strong suggestion that this layer (and previous layers as well) has not learned all needed features and they would probably require to be modified in some way — e.g., increasing the number of neurons or kernel size.

In Figure 5.5 we display the CSMs of the same two layers whose AOMs are shown in Figure 5.4. We can quickly notice that some neurons that activate very often are still not very selective because they do so for most or all classes. However, even in the first layer, we can spot neurons that are very selective towards subsets of classes. For instance, some neurons activate quite often for vehicles, such as ships and trucks, but not so for animals. With this insight, we can assume that, even though these neurons are not selective enough towards individual classes, they do help the decision process by distinguishing subsets of classes, making the decision process easier for the next layers.

With the aid of our insights, we identify neurons that are not contributing to the decision process, whether due to lack of selectivity or substantial redundancy and similarly remove them as we did with the MNIST model in the previous experiment. By doing this, we were able to reduce the size of the first layer from 64 down to 20 neurons. After freezing the weights in the first layer and retraining the model for a single epoch, we were able to achieve a performance of 81.20% test set accuracy. This performance is even higher than the original one of 80.54% accuracy, which strengthens our belief that the removed neurons not only were not useful to the decision process but they were adding some small degree of overfitting to the model.

Following the same process, we reduced the size of the sixth layer down to 35 neurons, which allows us to achieve a test set accuracy of 80.43% after the freezing-retraining process. This performance is only marginally worse than what the model had initially been, but with much fewer neurons. When this happens, the DL practitioner has three options: (1) return to the previous state of the model if the accuracy drop is unacceptable; (2) keep the new configuration if the size-performance trade-off is still worth; or (3) try to reinitialize or modify in some way the dropped neurons to build more interesting features than before.

Figure 5.6: Selectivity Experiment on the Sixth Layer of the CIFAR10 Model



In the graphic on the left, we display the total number of neurons (kernels) in blue, the number of neurons that were considered to be unselective for each class in red, and the number of neurons that were supposed to be selective for each class in green. In the graphic on the right, we display the class accuracy achieved by keeping only the neurons belonging to each of the three groups above. Note that keeping only selective neurons actually increase the performance of the network for that particular class, even though the amount of neurons in the layer is much smaller. On the other hand, if we remove all neurons that were considered selective for that particular class, the network loses most of its ability to successfully recognize that class, significantly lowering its accuracy when faced of samples belonging to such a class. This finding gives us confidence that our metric does capture the selectivity relationship between neurons and classes.

5.3.3 Neuron Selectivity Experiment

This experiment is designed to demonstrate how the proposed neuron selectivity metric indeed captures the intrisic selectivity degree of a neuron. The experiment runs as follows: For each pair of neuron n and class c in a given layer l, we compute the average value $s^{n,c}$ from all cells in the respective CSM $S^{n,c}$. Then, for each class c_i, we remove from the layer all neurons with $s^{n_j,c_i} \leq 0$ — i.e., neurons that are not selective towards c_i — and compute the performance of the resulting model only for the samples in the test set that belongs to c_i. Next, we also do the opposite: For each class c_i, we remove all neurons in l that have $s^{n_j,c_i} > 0$ — i.e., neurons that are selective towards c_i — and compute the resulting performance using only test set samples from class c_i.

We performed this experiment using the sixth layer on the original CIFAR10 model. Figure 5.6 (left) shows how many neurons were kept in the model in each case, as stated above. In Figure 5.6 (right), we compare the test set accuracy for when we keep: all neurons (blue), unselective neurons (red), and selective neurons (blue). We experiment

individually for every class c_i in the CIFAR10 dataset. For all classes, the amount of selective neurons is much smaller than the amount of non-selective neurons. Additionally, the class test set accuracy always drops (significantly) when we remove selective neurons and, conversely, increases when we remove non-selective neurons. Given these results, we can strongly argue that our selectivity metric does capture how much the neuron contributes to distinguishing between classes.

5.4 Future Research

We are confident that this research can lead to even better results in the future, as there is still a lot of room for further study in our technique. First, it would be interesting to study how the visualizations proposed here can be used to improve widely known models such as AlexNet [Krizhevsky, Sutskever e Hinton 2012] and ResNet [He et al. 2016]. Most of these models were trained without the guiding of VA tools and likely could be improved by them. Another interesting experiment would be to train a model from scratch but only adding components according to the insights received from the visualization. For instance, the DL practitioner could start designing with a simple model with a single convolutional layer plus the output layer. After training this initial configuration, insights from the visualization would guide architectural updates such as adding more layers or neurons to the existing layers. This way, a DL practitioner would be able to achieve, from scratch, a model with better performance than what it would be achieved with a blind architectural search.

In this work, we focused on the analysis of activations. However, our technique could also be adapted to analyze the hidden states of recurrent layers, in particular, LSTMs. At each timestep, a recurrent layer takes an element from a sequence of inputs and process it, modifying its hidden states accordingly — and possibly generating an activation value as well. By using our tools to visualize how often positive values appear in each hidden state's position for the same input element, the DL practitioner would be able to build a better understanding about the hidden states store the information needed to affect the model's behavior for future time steps.

Finally, our method still suffers from some scalability restrictions. If the model has too many neurons or is tackling a problem with too many classes, it becomes quite difficult to take insights about patterns in multiple neurons. One possibility to address this issue is to use dimensionality reduction techniques to project the AOMs (or CSMs)

to 2-D space. Doing this would reduce the ability to analyze details of a single map, but would allow for much faster insights regarding how selective a neuron is towards each class. One alternative is to project the AOMs for all (n_j, c_i) pairs of neurons n_j and c_i. By looking at the resulting projection scatterplot, the user would be able to gather insights about the proposed tasks. For instance, one could spot neurons selective towards a single class c_i by looking at a data point from neuron n_j and c_i that lies far from the data points corresponding to the same neuron n_j but to all other classes $c_k | k \neq i$. Another possibility would be to identify classes lacking selective neurons, when all data points corresponding to AOMs from class c_i are close to data points corresponding to AOMs from other classes, thus indicating that these AOMs have very similar features.

6 CONCLUSION

In this dissertation, we explored how visual analytics can be employed to support deep learning applications. This topic has been receiving significant attention in the last few years, mainly due to the increase in the usage of DNNs both in industry and science. As it often happens with new topics, there is still a wide range of open questions about how VA can aid the DL field. In this work we addressed some of these questions, discussing which VA techniques have been proposed for DL engineering and which tasks in the DL design workflow they aim to perform. In particular, we introduced a taxonomy for those tools. Having such a taxonomy is important because it allows DL practioners to have an improved knowledge of what tools they have at hand. Additionally, it gives VA researchers more background about what are the challenges in DL engineering where VA tools are welcome. In our taxonomy, we classified VA tools regarding the task they perform and the type of architecture they address.

Our survey reviewed over 40 papers in the subject. Most of those have been proposed in the past three years or less, which demonstrates how relevant this research field is today. Undoubtely, more contributions are going to follow in the next years, as there is still a significant room for improvements in the current approaches. Furthermore, there are many challenges in the DL design workflow that are not suficiently covered by current VA tools but that would definitely make good use of them. In any case, having a taxonomy like this one can serve as a basis for research of novel VA approaches in the future.

In Chapter 5, we also proposed a novel VA technique for architectural tuning of DNNs. While writing the survey, we identified this designing task as one of the steps in the DL design workflow that still lacks enough approaches addressing them. In this contribution, we proposed the three visualization techniques and discussed how they can be used to treat some of the challenges faced during architectural tuning. We have shown with a set of experiments how our visualizations can be employed to spot ineffective or redundant neurons and to assert whether a layer has learned sufficient features for the classification or not. We also have shown how, in some circumstances, our method can not only aid the removal of neurons that are not helping the decision process but also to spot neurons that are harming the performance.

6.1 Publications

The work done throughout this M.Sc degree— and presented in this dissertation — has also led to the publication of the following papers:

- *Task-based behavior generalization via manifold clustering* [Garcia, da Silva e Comba 2017]: In this paper, we propose a technique to generalize policies learned via reinforcement learning to solve related tasks. We employ a manifold clustering technique to identify discontinuities in the policy space so we can more efficiently select new tasks to be trained via RL. Tasks whose policies lies in more continuous regions of the policy space are then learned via more simpler regression methods.
- *A task-and-technique centered survey on visual analytics for deep learning model engineering* [Garcia et al. 2018]: In this survey, we introduced our taxonomy to classify VA techniques supporting DL engineering according to the tasks and techniques they address. The content of this survey is discussed in Chapter 4.
- *A methodology for neural network architectural tuning using activation occurrence maps* [Garcia et al. 2019]: In this paper, we propose a set of visualization tools to support the architectural tuning of neuron networks. We also introduce a methodology of how these tools can be used to guide the development of novel models. The content of this paper is discussed in Chapter 5.

REFERENCES

AIN, Q. T. et al. Sentiment analysis using deep learning techniques: a review. **Int J Adv Comput Sci Appl**, v. 8, n. 6, p. 424, 2017.

AL-RFOU, R. et al. Theano: A python framework for fast computation of mathematical expressions. **arXiv preprint arXiv:1605.02688**, 2016.

Alsallakh, B. et al. Visual methods for analyzing probabilistic classification data. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 12, p. 1703–1712, Dec 2014.

ALSALLAKH, B. et al. Do convolutional neural networks learn class hierarchy? **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 1, p. 152–162, 2018.

AMERSHI, S. et al. Power to the people: The role of humans in interactive machine learning. **AI Magazine**, v. 35, n. 4, p. 105–120, 2014.

AMODEI, D. et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In: **International conference on machine learning**. [S.l.: s.n.], 2016. p. 173–182.

ARJOVSKY, M.; CHINTALA, S.; BOTTOU, L. Wasserstein gan. **arXiv preprint arXiv:1701.07875**, 2017.

ARRAS, L. et al. Explaining recurrent neural network predictions in sentiment analysis. **arXiv preprint arXiv:1706.07206**, 2017.

AUBRY, M.; RUSSELL, B. C. Understanding deep features with computer-generated imagery. In: **Proc. IEEE International Conference on Computer Vision (ICCV)**. [S.l.: s.n.], 2015.

BACH, S. et al. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. **PLOS One**, Public Library of Science, v. 10, 07 2015.

BAHDANAU, D. et al. End-to-end attention-based large vocabulary speech recognition. In: IEEE. **2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)**. [S.l.], 2016. p. 4945–4949.

BEHRISCH, M. et al. Matrix reordering methods for table and network visualization. In: WILEY ONLINE LIBRARY. **Computer Graphics Forum**. [S.l.], 2016. v. 35, n. 3, p. 693–716.

BELLMAN, R. **An introduction to artificial intelligence: Can computers think?** [S.l.]: Thomson Course Technology, 1978.

BERNARDO, F. et al. Interactive machine learning for end-user innovation. In: **Proc. Designing the User Experience of Machine Learning Systems (AAAI Spring Symposium Series)**. [S.l.: s.n.], 2017.

BISHOP, C. M. **Pattern Recognition and Machine Learning (Information Science and Statistics)**. [S.l.]: Springer-Verlag New York, 2006.

BOJARSKI, M. et al. VisualBackProp: Efficient visualization of CNNs. **CoRR**, abs/1611.05418, 2016.

BREHMER, M.; MUNZNER, T. A multi-level typology of abstract visualization tasks. **IEEE Transactions on Visualization and Computer Graphics**, v. 19, n. 12, p. 2376–2385, Dec 2013. ISSN 1077-2626.

BREIMAN, L. Random forests. **Machine Learning**, v. 45, n. 1, p. 5–32, 2001.

BROCHU, E.; CORA, V. M.; FREITAS, N. D. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. **CoRR**, abs/1012.2599, 2010.

BROOKS, M. et al. Featureinsight: Visual support for error-driven feature ideation in text classification. In: IEEE. **2015 IEEE Conference on Visual Analytics Science and Technology (VAST)**. [S.l.], 2015. p. 105–112.

CARON, M. et al. Deep clustering for unsupervised learning of visual features. In: **The European Conference on Computer Vision (ECCV)**. [S.l.: s.n.], 2018.

CASHMAN, D. et al. RNNbow: Visualizing learning via backpropagation gradients in recurrent neural networks. In: **Proc. Workshop on Visualization for Deep Learning (VADL)**. [S.l.: s.n.], 2017.

CHO, K. et al. Learning phrase representations using rnn encoder-decoder for statistical machine translation. **arXiv preprint arXiv:1406.1078**, 2014.

CHOLLET, F. et al. **Keras**. 2015. <https://keras.io>.

CHUNG, S. et al. RevaCNN: Real-Time visual analytics for convolutional neural network. In: **Proc. ACM SIGKDD Workshop on Interactive Data Exploration and Analytics (IDEA)**. [S.l.: s.n.], 2016.

COMPUTERS & Graphics. 2018. Disponível em: <https://www.journals.elsevier.com/computers-and-graphics>.

Dahl, G. E.; Sainath, T. N.; Hinton, G. E. Improving deep neural networks for lvcsr using rectified linear units and dropout. In: **2013 IEEE International Conference on Acoustics, Speech and Signal Processing**. [S.l.: s.n.], 2013. p. 8609–8613. ISSN 1520-6149.

DENG, J. et al. ImageNet: A large-scale hierarchical image database. In: **Proc. IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2009. p. 248–255.

DENIL, M. et al. Predicting parameters in deep learning. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2013. p. 2148–2156.

DING, Y. et al. Visualizing and understanding neural machine translation. In: **Proc. Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)**. [S.l.: s.n.], 2017. v. 1, p. 1150–1159.

DONAHUE, J. et al. Long-term recurrent convolutional networks for visual recognition and description. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2015. p. 2625–2634.

DOSOVITSKIY, A.; BROX, T. Generating images with perceptual similarity metrics based on deep networks. In: **Advances in Neural Information Processing Systems 29**. [S.l.]: Curran Associates, Inc., 2016. p. 658–666.

DOSOVITSKIY, A.; BROX, T. Inverting visual representations with convolutional networks. In: **Proc. IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2016.

EASTERBROOK, J. A. The effect of emotion on cue utilization and the organization of behavior. **Psychological review**, American Psychological Association, v. 66, n. 3, p. 183, 1959.

ELMAN, J. L. Finding structure in time. **Cognitive Science**, v. 14, n. 2, p. 179–211, 1990.

ELZEN, S. V. D.; WIJK, J. J. van. Baobabview: Interactive construction and analysis of decision trees. In: IEEE. **2011 IEEE Conference on Visual Analytics Science and Technology (VAST)**. [S.l.], 2011. p. 151–160.

ENGEL, J. et al. GANSynth: Adversarial neural audio synthesis. In: **International Conference on Learning Representations**. [S.l.: s.n.], 2019.

ERHAN, D. et al. Visualizing higher-layer features of a deep network. **University of Montreal**, Technical Report 1341, 2009.

FAWCETT, T. An introduction to ROC analysis. **Pattern Recognition Letters**, v. 27, n. 8, p. 861–874, 2006.

FLACH, P. **The Art and Science of Algorithms that Make Sense of Data**. [S.l.]: Cambridge University Press, 2012.

Garcia, R.; da Silva, B. C.; Comba, J. L. D. Task-based behavior generalization via manifold clustering. In: **2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. [S.l.: s.n.], 2017.

GARCIA, R. et al. A methodology for neural network architectural tuning using activation occurrence maps. In: **2019 International Joint Conference on Neural Networks (IJCNN)**. [S.l.: s.n.], 2019.

GARCIA, R. et al. A task-and-technique centered survey on visual analytics for deep learning model engineering. **Computers & Graphics**, Elsevier, v. 77, p. 30–49, 2018.

GIRSHICK, R. Fast r-cnn. In: **Proceedings of the IEEE international conference on computer vision**. [S.l.: s.n.], 2015. p. 1440–1448.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016.

GOODFELLOW, I. et al. Generative adversarial nets. In: **Advances in Neural Information Processing Systems 27**. [S.l.]: Curran Associates, Inc., 2014. p. 2672–2680.

GRAVES, A.; MOHAMED, A.-r.; HINTON, G. Speech recognition with deep recurrent neural networks. In: IEEE. **2013 IEEE international conference on acoustics, speech and signal processing**. [S.l.], 2013. p. 6645–6649.

GREGOR, K. et al. DRAW: A recurrent neural network for image generation. **CoRR**, abs/1502.04623, 2015.

GRÜN, F. et al. A taxonomy and library for visualizing learned features in convolutional neural networks. **CoRR**, abs/1606.07757, 2016.

GUO, Y. et al. Deep learning for visual understanding: A review. **Neurocomputing**, Elsevier, v. 187, p. 27–48, 2016.

HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. **arXiv preprint arXiv:1510.00149**, 2015.

HAN, S. et al. Learning both weights and connections for efficient neural network. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2015. p. 1135–1143.

HARLEY, A. W. An interactive node-link visualization of convolutional neural networks. In: **Proc. International Symposium on Advances in Visual Computing (ISVC)**. [S.l.]: Springer, 2015. p. 867–877.

HASSELT, H. V.; GUEZ, A.; SILVER, D. Deep reinforcement learning with double q-learning. In: **Thirtieth AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2016.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. **The Elements of Statistical Learning: Data Mining, Inference, and Prediction**. [S.l.]: Springer, 2009.

HE, K. et al. Deep residual learning for image recognition. In: **Proc. IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2016. p. 165–173.

HINTON, G. et al. Deep neural networks for acoustic modeling in speech recognition. **IEEE Signal processing magazine**, v. 29, 2012.

HINTON, G. E.; OSINDERO, S.; TEH, Y.-W. A fast learning algorithm for deep belief nets. **Neural computation**, MIT Press, v. 18, n. 7, p. 1527–1554, 2006.

HOCHREITER, S. et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In: ____. **A Field Guide to Dynamical Recurrent Neural Networks**. [S.l.]: IEEE Press, 2001. p. 464.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, MIT Press, v. 9, n. 8, p. 1735–1780, 1997.

Hohman, F. M. et al. Visual analytics in deep learning: An interrogative survey for the next frontiers. **IEEE Transactions on Visualization and Computer Graphics**, 2018.

HUANG, C.-Z. A. et al. Counterpoint by convolution. **ISMIR**, 2017.

IEEE-VAST 2017 Symposium. 2017. Disponível em: <http://ieeevis.org/year/2017/info/papers>.

IJCNN - International Joint Conference on Neural Networks. 2019. Disponível em: <https://www.ijcnn.org/>.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. **arXiv preprint arXiv:1502.03167**, 2015.

JIANG, B.; CANNY, J. Interactive machine learning via a gpu-accelerated toolkit. In: ACM. **Proceedings of the 22nd International Conference on Intelligent User Interfaces**. [S.l.], 2017. p. 535–546.

KAHNG, M. et al. Activis: Visual exploration of industry-scale deep neural network models. **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 1, p. 88–97, Jan 2018. ISSN 1077-2626.

KARPATHY, A.; JOHNSON, J.; FEI-FEI, L. Visualizing and understanding recurrent networks. **CoRR**, abs/1506.02078, 2015.

KARRAS, T.; LAINE, S.; AILA, T. A style-based generator architecture for generative adversarial networks. **CoRR**, abs/1812.04948, 2018.

KIM, Y. Convolutional neural networks for sentence classification. **arXiv preprint arXiv:1408.5882**, 2014.

KRAUSE, J.; PERER, A.; BERTINI, E. INFUSE: Interactive feature selection for predictive modeling of high dimensional data. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 12, p. 1614–1623, 2014.

KRIZHEVSKY, A.; HINTON, G. **Learning multiple layers of features from tiny images**. [S.l.], 2009.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. ImageNet classification with deep convolutional neural networks. In: **Proc. International Conference on Neural Information Processing Systems**. [S.l.: s.n.], 2012. v. 1, p. 1097–1105.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, Nature Research, v. 521, n. 7553, p. 436–444, 2015.

LECUN, Y. et al. Backpropagation applied to handwritten zip code recognition. **Neural Computation**, v. 1, n. 4, p. 541–551, 1989.

LECUN, Y. et al. Handwritten digit recognition with a back-propagation network. In: **Advances in neural information processing systems**. [S.l.: s.n.], 1990. p. 396–404.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, Taipei, Taiwan, v. 86, n. 11, p. 2278–2324, 1998.

LHUILLIER, A.; HURTER, C.; TELEA, A. State of the art in edge and trail bundling techniques. **Computer Graphics Forum**, v. 36, n. 3, p. 619–645, 2017.

LI, H.; MUELLER, K.; CHEN, X. Beyond saliency: understanding convolutional neural networks from saliency prediction on layer-wise relevance propagation. **CoRR**, abs/1712.08268, 2017.

LI, J. et al. Visualizing and understanding neural models in NLP. **CoRR**, abs/1506.01066, 2015.

LILLICRAP, T. P. et al. Continuous control with deep reinforcement learning. **arXiv preprint arXiv:1509.02971**, 2015.

LIPTON, Z. The mythos of model interpretability. **CoRR**, abs/1606.03490, 2016.

LIU, M. et al. Analyzing the training processes of deep generative models. **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 1, p. 77–87, 2018.

LIU, M. et al. Towards better analysis of deep convolutional neural networks. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 91–100, 2017.

LIU, S. et al. Visualizing high-dimensional data: Advances in the past decade. **Computer Graphics Forum**, v. 23, n. 3, p. 1249–1268, 2016.

LIU, S. et al. Towards better analysis of machine learning models: A visual analytics perspective. **Visual Informatics**, v. 1, n. 1, p. 48 – 56, 2017.

LOPEZ, M. M.; KALITA, J. Deep learning applied to nlp. **arXiv preprint arXiv:1703.03091**, 2017.

LU, J. et al. Recent progress and trends in predictive visual analytics. **Frontiers of Computer Science**, v. 11, n. 2, p. 192–207, 2017.

LUONG, M.-T.; PHAM, H.; MANNING, C. D. Effective approaches to attention-based neural machine translation. **arXiv preprint arXiv:1508.04025**, 2015.

MAATEN, L. v. d.; HINTON, G. Visualizing data using t-SNE. **Journal of Machine Learning Research**, v. 9, n. Nov, p. 2579–2605, 2008.

MAATEN, L. V. D.; POSTMA, E.; HERIK, J. Van den. Dimensionality reduction: a comparative review. **J Mach Learn Res**, v. 10, p. 66–71, 2009.

MAHENDRAN, A.; VEDALDI, A. Understanding deep image representations by inverting them. In: **Proc. IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2015.

MAHENDRAN, A.; VEDALDI, A. Visualizing deep convolutional neural networks using natural pre-images. **International Journal of Computer Vision**, v. 120, n. 3, p. 233–255, 2016.

MARCUS, G. Deep learning: A critical appraisal. **CoRR**, abs/1801.00631, 2018.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

MING, Y. et al. Understanding hidden memories of recurrent neural networks. In: **Proc. IEEE Visual Analytics Science and Technology (VAST)**. [S.l.: s.n.], 2017.

MINSKY, M.; PAPERT, S. Perceptron: an introduction to computational geometry. **The MIT Press, Cambridge, expanded edition**, v. 19, n. 88, p. 2, 1969.

MNIH, V. et al. Human-level control through deep reinforcement learning. **Nature**, Nature Publishing Group, v. 518, n. 7540, p. 529, 2015.

MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. **Foundations of machine learning**. [S.l.]: MIT press, 2018.

MONTAVON, G. et al. Explaining nonlinear classification decisions with deep Taylor decomposition. **Pattern Recognition**, v. 65, p. 211 – 222, 2017.

MONTAVON, G.; SAMEK, W.; MÜLLER, K.-R. Methods for interpreting and understanding deep neural networks. **Digital Signal Processing**, v. 73, p. 1–15, 2018.

MÜHLBACHER, T.; PIRINGER, H. A partition-based framework for building and validating regression models. **IEEE Transactions on Visualization and Computer Graphics**, v. 19, n. 12, p. 1962–1971, Dec 2013.

MURPHY, K. P. **Machine learning: a probabilistic perspective**. [S.l.]: MIT press, 2012.

NG, J. Y.-H. et al. Beyond short snippets: Deep networks for video classification. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2015. p. 4694–4702.

NGUYEN, A. et al. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. In: **Proc. International Conference on Neural Information Processing Systems**. [S.l.: s.n.], 2016. p. 3395–3403.

NGUYEN, A.; YOSINSKI, J.; CLUNE, J. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In: **Proc. IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2015.

NGUYEN, A.; YOSINSKI, J.; CLUNE, J. Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks. **CoRR**, abs/1602.03616, 2016.

PAIVA, J. G. S. et al. An approach to supporting incremental visual data classification. **IEEE transactions on visualization and computer graphics**, IEEE, v. 21, n. 1, p. 4–17, 2015.

PARK, S. H.; GOO, J. M.; JO, C.-H. Receiver operating characteristic (ROC) curve: Practical review for radiologists. **Korean Journal of Radiology**, v. 5, n. 1, p. 11–18, 2004.

PEZZOTTI, N. et al. DeepEyes: Progressive visual analytics for designing deep neural networks. **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 1, p. 98–108, 2018.

POWERS, D. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation. **Journal of Machine Learning Technology**, v. 2, n. 1, p. 37–63, 2011.

QI, H. et al. **BIDViz: Real-time Monitoring and Debugging of Machine Learning Training Processes**. Dissertação (Mestrado) — EECS Department, University of California, Berkeley, 2017.

RADFORD, A.; METZ, L.; CHINTALA, S. Unsupervised representation learning with deep convolutional generative adversarial networks. **arXiv preprint arXiv:1511.06434**, 2015.

RAINA, R.; MADHAVAN, A.; NG, A. Y. Large-scale deep unsupervised learning using graphics processors. In: **Proceedings of the 26th Annual International Conference on Machine Learning**. [S.l.]: ACM, 2009. (ICML '09), p. 873–880.

RAUBER, P. et al. Visualizing the hidden activity of artificial neural networks. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 101–110, 2017.

RAUBER, P. et al. Interactive image feature selection aided by dimensionality reduction. In: **Proc. EuroVA**. [S.l.: s.n.], 2015. p. 67–74.

RAUBER, P. E.; FALCÃO, A. X.; TELEA, A. C. Projections as visual aids for classification system design. **Information Visualization**, v. 17, n. 4, p. 282–305, 2018.

REN, D. et al. Squares: Supporting interactive performance analysis for multiclass classifiers. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 61–70, Jan 2017.

RIBEIRO, M. T.; 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**. [S.l.]: ACM, 2016. (KDD '16), p. 1135–1144. ISBN 978-1-4503-4232-2.

RONG, X.; ADAR, E. Visual tools for debugging neural language models. In: **Proc. International Conference on Machine Learning Workshop on Visualization for Deep Learning**. [S.l.: s.n.], 2016.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, American Psychological Association, v. 65, n. 6, p. 386, 1958.

RUMELHART, D. E. et al. Learning representations by back-propagating errors. **Cognitive modeling**, v. 5, n. 3, p. 1, 1988.

SACHA, D. et al. Human-centered machine learning through interactive visualization. In: **Proc. European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning**. [S.l.: s.n.], 2016.

SACHA, D. et al. Visual interaction with dimensionality reduction: A structured literature analysis. **IEEE Transactions on Visualization and Computer Graphics**, v. 23, n. 1, p. 241–250, Jan 2017. ISSN 1077-2626.

SAMEK, W. et al. Evaluating the visualization of what a deep neural network has learned. **IEEE Transactions on Neural Networks and Learning Systems**, v. 28, n. 11, p. 2660–2673, 2017.

SAMEK, W.; WIEGAND, T.; MÜLLER, K. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. **CoRR**, abs/1708.08296, 2017.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM Journal of Research and Development**, v. 3, n. 3, p. 210–229, July 1959. ISSN 0018-8646.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. **Neural networks**, Elsevier, v. 61, p. 85–117, 2015.

SEDLMAIR, M. et al. Visual parameter space analysis: A conceptual framework. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 12, p. 2161–2170, Dec 2014. ISSN 1077-2626.

SEIFERT, C. et al. Visualizations of deep neural networks in computer vision: A survey. In: **Transparent Data Mining for Big and Small Data**. [S.l.]: Springer, 2017. p. 123–144.

SELVARAJU, R. R. et al. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In: **ICCV**. [S.l.: s.n.], 2017. p. 618–626.

SILVER, D. et al. Mastering the game of go without human knowledge. **Nature**, Nature Publishing Group, v. 550, n. 7676, p. 354, 2017.

SIMONYAN, K.; VEDALDI, A.; ZISSERMAN, A. Deep inside convolutional networks: Visualising image classification models and saliency maps. **CoRR**, abs/1312.6034, 2013.

SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. **CoRR**, abs/1409.1556, 2014.

SMILKOV, D. et al. Direct-manipulation visualization of deep networks. **CoRR**, abs/1708.03788, 2017.

SNOEK, J.; LAROCHELLE, H.; ADAMS, R. P. Practical bayesian optimization of machine learning algorithms. In: **Advances in Neural Information Processing Systems 25**. [S.l.]: Curran Associates, Inc., 2012.

SNOEK, J. et al. Scalable bayesian optimization using deep neural networks. In: **Proc. International Conference on Machine Learning**. [S.l.: s.n.], 2015.

SRIVASTAVA, N. et al. Dropout: a simple way to prevent neural networks from overfitting. **The Journal of Machine Learning Research**, JMLR. org, v. 15, n. 1, p. 1929–1958, 2014.

STREETER, M. J.; WARD, M. O.; ALVAREZ, S. A. NVIS: an interactive visualization tool for neural networks. In: **Proc. SPIE Visual Data Exploration and Analysis**. [S.l.: s.n.], 2001. v. 4302, p. 1–8.

STROBELT, H. et al. LSTMVis: A tool for visual analysis of hidden state dynamics in recurrent neural networks. **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 1, p. 667–676, Jan 2018.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to sequence learning with neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2014. p. 3104–3112.

SUTTON, R. S.; BARTO, A. G. **Introduction to reinforcement learning**. [S.l.]: MIT press Cambridge, 1998. v. 135.

SZEGEDY, C. et al. Going deeper with convolutions. In: **Proc. IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2015. p. 1–9.

TALBOT, J. et al. Ensemblematrix: interactive visualization to support machine learning with multiple classifiers. In: ACM. **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. [S.l.], 2009. p. 1283–1292.

TATU, A. et al. Subspace search and visualization to make sense of alternative clusterings in high-dimensional data. In: **Proc. IEEE VAST**. [S.l.: s.n.], 2012. p. 63–72.

TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>.

TZENG, F. Y.; MA, K. L. Opening the black box – data driven visualization of neural networks. In: **Proc. IEEE Visualization**. [S.l.: s.n.], 2005. p. 383–390.

VASEEKARAN, G. **Machine Learning: Supervised Learning vs Unsupervised Learning**. 2018. Disponível em: <https://medium.com/@gowthamy/machine-learning-supervised-learning-vs-unsupervised-learning-f1658e12a780>.

VELLIDO, A.; MARTÍN-GUERRERO, J. D.; LISBOA, P. J. Making machine learning models interpretable. In: CITESEER. **ESANN**. [S.l.], 2012. v. 12, p. 163–172.

VINCENT, P. et al. Extracting and composing robust features with denoising autoencoders. In: **Proceedings of the 25th International Conference on Machine Learning**. New York, NY, USA: ACM, 2008. (ICML '08), p. 1096–1103. ISBN 978-1-60558-205-4.

VINYALS, O. et al. Show and tell: A neural image caption generator. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2015. p. 3156–3164.

WEI, D. et al. Understanding intra-class knowledge inside CNN. **CoRR**, abs/1507.02379, 2015.

WONGSUPHASAWAT, K. et al. Visualizing dataflow graphs of deep learning models in tensorflow. **IEEE Transactions on Visualization and Computer Graphics**, v. 24, n. 1, p. 1–12, 2018.

YEAGER, L. et al. Effective visualizations for training and evaluating deep models. In: **Proc. International Conference on Machine Learning Workshop on Visualization for Deep Learning**. [S.l.: s.n.], 2016.

YOSINSKI, J. et al. Understanding neural networks through deep visualization. In: **Proc. International Conference on Machine Learning Workshop on Deep Learning**. [S.l.: s.n.], 2015.

Young, T. et al. Recent trends in deep learning based natural language processing [review article]. **IEEE Computational Intelligence Magazine**, v. 13, n. 3, p. 55–75, Aug 2018.

YU, A. **How To Teach A Computer To See With Convolutional Neural Networks**. 2018. Disponível em: <https://towardsdatascience.com/how-to-teach-a-computer-to-see-with-convolutional-neural-networks-96c120827cd1>.

ZAHAVY, T.; BEN-ZRIHEM, N.; MANNOR, S. Graying the black box: Understanding dqns. In: **Proc. International Conference on Machine Learning**. [S.l.: s.n.], 2016. p. 1899–1908.

ZEILER, M. D. Adadelta: an adaptive learning rate method. **arXiv preprint arXiv:1212.5701**, 2012.

ZEILER, M. D.; FERGUS, R. Visualizing and understanding convolutional networks. In: SPRINGER. **Proc. European Conference on Computer Vision**. [S.l.], 2014. p. 818–833.

ZENG, H. **Towards Better Understanding of Deep Learning with Visualization**. [S.l.]: Dept. of Computer Science and Engineering, Hong-Kong Univ. of Science and Technology, 2016. MSc thesis, Dept. of Computer Science and Engineering, Hong-Kong Univ. of Science and Technology.

ZENG, H. et al. CNNComparator: Comparative analytics of convolutional neural networks. In: **Proc. Workshop on Visual Analytics for Data Learning (VADL)**. [S.l.: s.n.], 2017.

ZENG, Y.; WANG, L. Fei-fei li: Artificial intelligence is on its way to reshape the world. **National Science Review**, Oxford University Press, v. 4, n. 3, p. 490–492, 2017.

ZHANG, J.; ZONG, C. Deep neural networks in machine translation: An overview. **IEEE Intelligent Systems**, v. 30, n. 5, p. 16–25, 2015.

ZHANG, L.; WANG, S.; LIU, B. Deep learning for sentiment analysis: A survey. **CoRR**, abs/1801.07883, 2018.

ZHANG, X.; ZHAO, J.; LECUN, Y. Character-level convolutional networks for text classification. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2015. p. 649–657.

ZHONG, W. et al. Evolutionary visual analysis of deep neural networks. In: **Proc. International Conference on Machine Learning Workshop on Visualization for Deep Learning**. [S.l.: s.n.], 2017.

ZINTGRAF, L. M. et al. Visualizing deep neural network decisions: Prediction difference analysis. **CoRR**, abs/1702.04595, 2017.

ZINTGRAF, L. M.; COHEN, T. S.; WELLING, M. A new method to visualize deep neural networks. **CoRR**, abs/1603.02518, 2016.