Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

A Visual Approach for Probing Learned Models

Jan Tagscherer

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Thomas Ertl

Supervisor: Dipl.-Phys. Qi Han,
Dr. Dennis Thom

Commenced: February 21, 2018

Completed: August 21, 2018

Abstract

Deep learning models are complex neural networks that are able to accomplish a large range of tasks effectively, including machine translation, speech recognition, and image classification.

However, recent research has shown that transformations of input data can deteriorate the performance of these models dramatically. This effect is especially startling with adversarial perturbations that aim to fool a deep neural network while being barely perceptible. The complexity of these networks makes it hard to understand where and why they fail.

Previous work has attempted to provide insights into the inner workings of these models in various different ways. A survey of these existing systems is conducted and concludes that they have failed to provide an integrated approach for probing how specific changes to the input data are represented within a trained model.

This thesis introduces Advis, a visualization system for analyzing the impact of input data transformations on a model's performance and on its internal representations. For performance analysis, it displays various metrics of prediction quality and robustness using lists and a radar chart. An interactive confusion matrix supports pattern detection and input image selection. Insights into the impact of data distortions on internal representations can be gained by the combination of a color-coded computation graph and detailed activation visualizations. The system is based on a highly flexible architecture that enables users to adapt it to the specific requirements of their task. Three use cases demonstrate the usefulness of the system for probing and comparing the impact of input transformations on performance metrics and internal representations of various networks.

The insights gained through this system show that interactive visual approaches for understanding the effect of input perturbations on deep learning models are an area worth further investigation.

Contents

1 Introduction **13**
1.1 Motivation . . . 13
1.2 Research Problem . . . 13
1.3 Contribution . . . 14

2 Foundation **15**
2.1 Visualization . . . 15
2.2 Deep Learning . . . 18

3 Related Work **29**
3.1 Survey of Deep Learning Visualization Approaches . . . 29
3.2 Selected Approaches . . . 33

4 Concept **39**
4.1 Requirements . . . 39
4.2 Design . . . 40

5 Implementation **49**
5.1 Used Tools . . . 49
5.2 System Architecture . . . 51
5.3 Data Processing . . . 53
5.4 Views . . . 60

6 Use Cases **71**
6.1 Choosing a Model for a Specific Task . . . 71
6.2 Analyzing the Effect of Manipulations of Input Data . . . 73
6.3 Defending Against Adversarial Perturbations . . . 78

7 Discussion **83**

8 Conclusion **85**
8.1 Results . . . 85
8.2 Open Questions and Future Tasks . . . 86

A Appendix **87**
A.1 Routes . . . 87

Bibliography **99**

List of Figures

2.1 Reference Model for Information Visualization 16
2.2 Process of Visual Analytics 18
2.3 Architectures of Deep Neural Networks 21
2.4 Examples for Adversarial Perturbations 24

3.1 User Interfaces of Related Applications 34

4.1 Interaction Flow 40
4.2 Input Image Selection Interaction Flow 44
4.3 Checkerboard Patterns in Inception Activations 45
4.4 User Interface Mockups 47

5.1 System Architecture 52
5.2 Interface Overview 60
5.3 Activation Difference Visualization 62
5.4 Activation Visualizations 63
5.5 Input Image Selection 64
5.6 Confusion Matrix at Different Zoom Levels 65
5.7 Confusion Matrix Modes 65
5.8 Confusion Matrix With and Without Diagonal Values 66
5.9 Input Image Predictions 67
5.10 Predictions for Distorted Versions 68
5.11 Detailed Model Performance 69
5.12 Distortion Configuration 69

6.1 Model List (First Use Case) 72
6.2 Detailed Model Performance (First Use Case) 72
6.3 Robustness Radar Chart (Second Use Case) 74
6.4 Confusion Matrices (Second Use Case) 74
6.5 Input Image Selection (Second Use Case) 75
6.6 Detailed Predictions (Second Use Case) 76
6.7 Node Activation Similarities (Second Use Case) 77
6.8 Activation Visualization (Second Use Case) 78
6.9 Model List (Second Use Case) 78
6.10 Confusion Matrix (Third Use Case) 80
6.11 Detailed Predictions (Third Use Case) 81
6.12 Node Activation Similarities (Third Use Case) 81

List of Tables

2.1 Deep Learning Frameworks 26
2.2 Wrappers for Deep Learning Frameworks 27

5.1 Frameworks and Libraries Used in the Backend 49
5.2 Frameworks and Libraries Used in the Frontend 50
5.3 Preset Models 54
5.4 Performance of Included Models 55
5.5 Preset Datasets 58
5.6 Preset Distortions 59

List of Abbreviations

API Application Programming Interface. 51

AUC Area Under the Receiver Operating Characteristic Curve. 22

CNN Convolutional Neural Network. 21

CSS Cascading Style Sheets. 50

DL Deep Learning. 13

DOM Document Object Model. 51

FNN Feed-Forward Neural Network. 20

GPGPU General-Purpose Computing on Graphics Processing Unit. 19

HTML Hypertext Markup Language. 50

HTTP Hypertext Transfer Protocol. 49

ILSVRC ImageNet Large Scale Visual Recognition Competition. 23

JSON JavaScript Object Notation. 51

LSTM Long Short-Term Memory. 21

MNIST Modified National Institute of Standards and Technology Database. 23

NIPS Conference on Neural Information Processing Systems. 23

NN Neural Network. 18

ReLU Rectified Linear Unit. 20

RNN Recurrent Neural Network. 20

SL Supervised Learning. 19

SVG Scalable Vector Graphics. 59

UI User Interface. 41

UL Unsupervised Learning. 19

WSGI Web Server Gateway Interface. 49

1 Introduction

We have entered an era of enormous amounts of data. In 2013, reports estimated that 2.5 quintillion bytes of data were created every day and that “90% of the data in the world today has been created in the last two years” [IBM13]. With Google processing “more than 23 petabytes of data per day” [Joh14], we have long crossed the threshold into a territory where the sheer amount of data is unimaginable, let alone making sense of its structure or contents. Dedicated systems that support these tasks are needed.

1.1 Motivation

The acquisition of raw data has ceased being a problem. Instead, the focus has shifted towards the challenge of extracting valuable knowledge from these vast data sets, and especially automating this process of knowledge extraction and subsequent decision making. This challenge has inspired the development of a multitude of approaches for automatic data analysis. A prominent approach promotes the idea of supervised learning from examples. Using a set of annotated training samples, a model is trained to predict the classification of previously unseen data samples [MCM13]. Algorithms for accomplishing this task include Decision Trees [Qui86], Support Vector Machines [HDO+98], and Artificial Neural Networks [Hay94]. Especially the latter have proven their ability to achieve a high performance [CN06] and have continuously gained in popularity.

As researchers continually improve the performance of these networks, they tend to grow more complex, until their size had risen far enough to warrant calling them Deep Learning (DL) models. Now that not only the size and complexity of data amounts had grown beyond comprehension, but also the size and complexity of models created to make sense of this data, another challenge has emerged: We have to understand how these processes and models work and what knowledge they have internalized to have the tools necessary for “constructive evaluation, correction and rapid improvement” [KAF+08]. As an added bonus, knowing how a model works will allow us to communicate about it meaningfully and gain trust in its decisions.

1.2 Research Problem

The goal of providing insights into the inner workings of DL models spurred the development of a large array of visualization systems, each focussing on a different set of network attributes. They aim to allow users to understand how a model works, how it makes its decisions, and – maybe most importantly – when and why it fails.

Research has shown that distorted input images can be the cause for some of these failures. They can seriously deteriorate the performance of deep neural networks [HCD12]. These input manipulations can consist of operations such as cropping the image, rotating it, occluding parts of it or altering its lighting.

Recently, an even more intriguing type of input manipulation has received increasing amounts of interest. So-called adversarial input perturbations attack deep networks, altering input images in a barely perceptible way while making the model's predictions practically unusable [SZS+13]. Even more fascinating, universal adversarial perturbations are able to fool networks across input images [MFFF17].

Understanding why and how these adversarial perturbations work on different network architectures is not only crucial for defending against those attacks but can reveal inherent structures and attributes of networks. In general, learning about how robust different models are to input transformations and how these transformations affect their internal representations can be useful for gaining insights on how deep neural networks work, providing a first step for improving them.

At the time of writing, there is no visualization system that offers an integrated and flexible way for analyzing the impact of input transformations on deep neural networks, and in particular not for adversarial perturbations. The system developed in this thesis aims to fill this gap.

1.3 Contribution

The contribution of this thesis consists of the following parts:

- A comparative survey of existing visualization systems for understanding, improving, comparing, and explaining deep learning models.
- Advis, a highly extensible visualization system for comparing the robustness of deep learning models to arbitrary input distortions and analyzing their impact on a model's internal representations.
- A set of use cases demonstrating the usefulness of the system for realistic analysis tasks.

2 Foundation

The topic of this thesis is set in the realm of Visual Analytics, applied to the area of Deep Learning models. This chapter will provide the necessary theoretic groundwork.

2.1 Visualization

In order to solve problems of information overload, the research field of visualization can provide valuable tools for making large amounts of data accessible to human beings. In a general definition, Card et al. describe visualization as "the use of computer-supported, interactive, visual representations of data to amplify cognition" [CMS99], with cognition being the acquisition of knowledge from data. According to them, a visualization's purpose is insight, reached through the generation of visual representations from large data sets by potentially transforming data and extracting features to enable the viewer to see its essential structure.

Card et al. [CMS99] identify three main goals of visualization, with the first one being discovery. Allowing an undirected search for structures can generate valuable insights. Visualization tools can aid this exploratory analysis by extracting information content and making previously non-apparent coherences visible by the means of visual representation. On top of that, knowledge obtained through visualization enables better decision making. Finally, data visualized in a meaningful way can assist in explaining its contents to both experts and laymen and therefore improve communication. Together with benefits to other cognitive activities, visualizations are an indispensable tool when confronted with large amounts of data.

Within the area of visualization, two related subfields can be discerned, Scientific Visualization and Information Visualization. The former encompasses the visual representation of scientific data, and especially physically-based data. In contrast, the field of Information Visualization is concerned with abstract data and non-physical information [CMS99]. Commonly, there is no natural interpolation, no given spatial embedding, and few obvious hierarchies and connections, increasing the complexity of creating meaningful visual representations even further.

The task at hand is concerned with visualizing internal representations of learned machine learning models. Therefore, the data to be visualized is abstract and falls into the realm of Information Visualization, which will be outlined in further detail below.

2.1.1 Information Visualization

With a slight variation of the general definition of visualization, the field of Information Visualization can be described as "the use of computer-supported, interactive, visual representations of abstract data to amplify cognition" [CMS99]. Furthermore, Information Visualization systems are especially

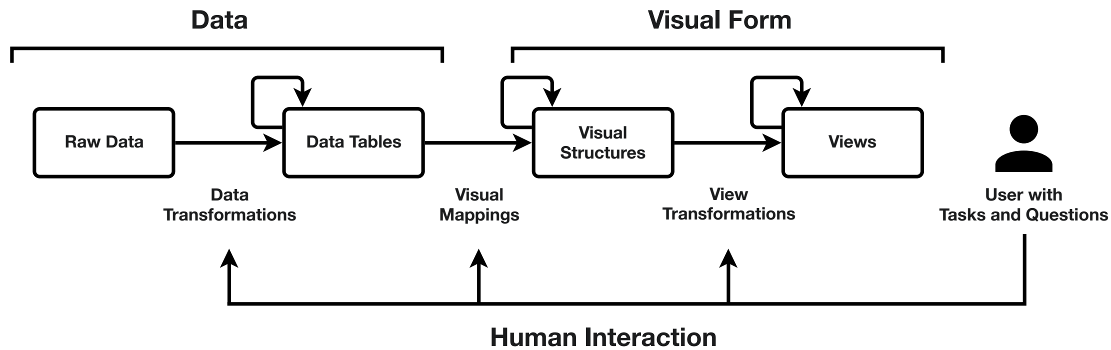


Figure 2.1: Reference model for Information Visualization (Adapted from Card et al. [CMS99]).

well-suited for exploratory tasks, with the user browsing a large information space but having no specific goal or question in mind [FVSN08]. Actual tasks or questions might arise during the process of examining the data. This application of Information Visualization as an exploratory aid can be especially useful when users have a lack of understanding of how a system is structured and prefer an approach with a comparatively low cognitive load to explore the system [Mar97], as can be the case with complex machine learning models. For these reasons, Information Visualization can be a powerful tool for a better understanding of deep learning models.

Card et al. [CMS99] introduce a reference model for Information Visualization as seen in figure 2.1 that describes the necessary steps for creating such a visualization system. This model was applied in various ways during the creation of Advis and will be explained in more detail hereafter.

In the reference model, raw data flows through a series of transformations until it reaches the human user of the system. The user can then decide to adjust each of the transformations using controls that work as an interface between the human and the system.

Data Transformations

Raw data is at the very start of the pipeline. This data is abstract and might be in a peculiar form. It usually cannot be made sense of without any assistance. Therefore, this data is transformed into a set of relations. Along with metadata, these relations can then make up so-called data tables. In these data tables, rows represent variables within the input data and columns represent cases (a set with values for each of the variables).

When creating these data tables, raw data is usually transformed in some way, creating data tables with derived values, structure, or both. Common examples of such data transformations are binning variables into a number of classes or aggregating their values. Variables can be nominal, ordinal or quantitative, and which transformations to apply to which variables has to be carefully considered. These transformations can be chained until the desired structure has been reached.

Visual Mappings

After data tables have been created, they can then be mapped into visual representations. To do so, the designer of the visualization system has a large array of visual structures at their disposal, but depending on the data structure and variable types only some of them might be a good choice. A visual mapping is said to be *expressive* if its representation does not omit any data, but also does not erroneously include more information than what is available in the data. On top of that, a mapping is called *effective* if it can be interpreted quickly and correctly and is sufficiently informative. In order to find such a mapping, attributes of human perception have to be considered. Exemplary features used to create visual mappings are the spatial position of elements, connection and enclosure, their color, shape or size. On top of that, these attributes can be varied temporally to encode more information. Using these features, viable mapping strategies can be found, such as histograms for frequency data, scatter plots for the value distribution over two variables, parallel coordinates for multivariate relations [ID91], representations based on glyphs [BKC+13] or stacked displays like tree maps [Shn92] or cushion maps [VV99] for hierarchical data.

View Transformations

The final step of the pipeline augments these visual structures to increase the amount of information that can be visualized. The structures are packaged into interactive views, turning static visual structures into dynamic visualizations and adding context.

Interaction techniques include brushing and linking, where data values marked in one view affect another view. By controlling the viewpoint by zooming, panning, and clipping, certain areas of interest can be brought into focus. Filtering and sorting can be used to find interesting data values. All of these techniques have been applied in the user interface of Advis.

2.1.2 Visual Analytics

Information Visualization can provide important tools to make sense of data. However, when the rapid extraction of relevant information from a flood of heterogeneous data is necessary, this data analysis process has to be automated. But creating, evaluating, and improving such a process is a difficult task. The field of Visual Analytics tries to tackle this problem by combining visualization, data analysis, and human analysts, therefore harvesting human "flexibility, creativity, and background knowledge with the enormous storage capacity and the computational power of today's computers" [KMS+08]. Visual Analytics "combines automated analysis techniques with interactive visualizations for an effective understanding, reasoning and decision making on the basis of very large and complex data sets" [KAF+08].

Visual Analytics can be described as an iterative process as depicted by figure 2.2. At the very start is the input data, from which a selection is chosen and preprocessed, for example by transforming, cleaning, and integrating or by selecting subsets as needed. This data can then be either visualized directly or used to generate hypotheses that form a model. These models can also be visualized. Conversely, visualizations can be used to generate hypotheses. Given models and visualizations, the user can conclude insights from either one and add them to their knowledge base. Furthermore, they can manipulate visualizations directly, for example by selecting or zooming, and generate new

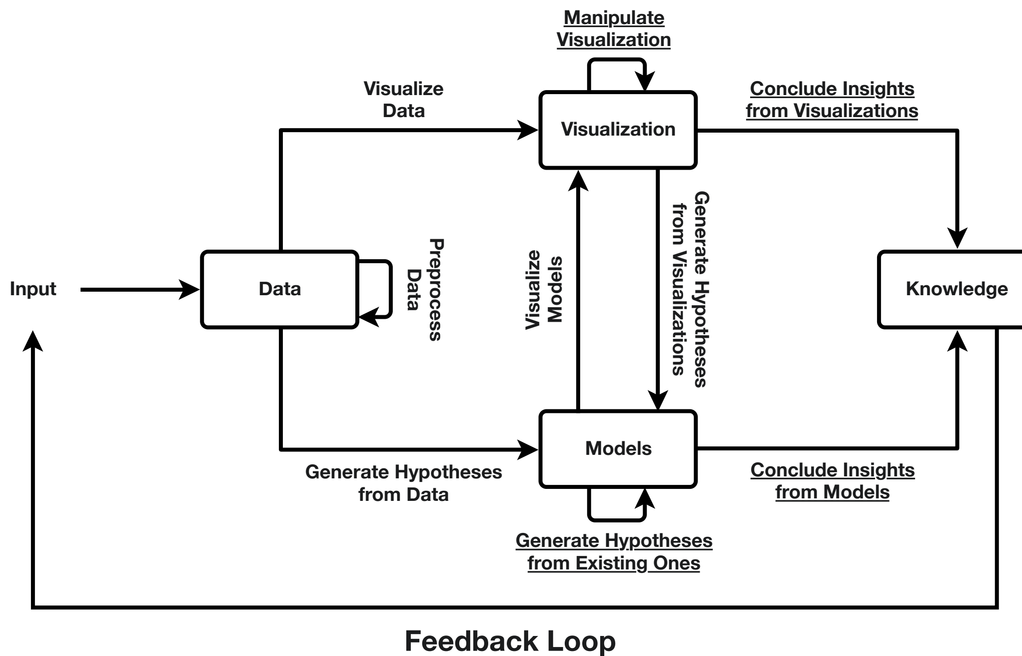


Figure 2.2: Process of Visual Analytics (Adapted from Keim et al. [KAF+08; KMS+08]). User interactions are underlined.

hypotheses from the set of existing ones by adjusting parameters in the analysis process. Finally, this process is not run only once but iteratively until the desired amount of insight has been achieved, using feedback from previous iterations each time.

Building upon the visual information seeking mantra "Overview first, zoom and filter, then details on demand" [Shn96], Keim et al. [KMS+08] formulate a Visual Analytics mantra: *"Analyze first. Show the important. Zoom, filter and analyze further. Details on demand."*

In its current state, Visual Analytics is split up and dispersed in a number of related areas, with each of them applying its pieces as needed. Instead of relying on ad-hoc applications, the field could benefit from infrastructure and standardization that crosses discipline boundaries [KAF+08]. These areas include visualization as described above, the management of large and possibly heterogeneous data which was a challenge during the implementation of Advis, research perception, cognition, and human-computer interaction, as well as data analysis methods like deep learning which will be outlined in the next section.

2.2 Deep Learning

Standard Neural Networks (NN's) can be understood as a collection of simple, connected processors called neurons. Each neuron receives input values and in turn produces a sequence of real-valued activations by multiplying each input with a weight, summing them up and adding a bias value, and passing the result of this calculation through a so-called activation function. Common activation

functions include the hyperbolic tangent function, the Sigmoid function or a linear rectification function. These inputs can be either data from the environment or the outputs of other neurons, received through weighted connections [Sch15].

A network's topology might change over time, but at any given point in time it can be described as a finite set of the aforementioned neurons and a finite set of directed edges between them. Neurons receiving input from external sources and passing their output into the network are part of the *input layer*, and ones receiving input from within the network and whose activation represents the network's result are part of the *output layer*. Neurons in between are said to be part of so-called *hidden layers*.

The network's behavior on a set of input data is determined by its set of weights and biases. In order to make the NN exhibit a desired behavior, one has to find appropriate weights and biases. The process of finding these values is called *Learning* [Sch15].

Simple applications of this idea have first been introduced under the name of *Feed-forward Multilayer Perceptrons* [IL66] and usually had few hidden layers. Since then, many variations and improvements have been developed. An especially promising and rapidly evolving area of research increased the length of the causal chains of computational stages within shallow networks, yielding deep network architectures [Sch15]. This field has since been called Deep Learning (DL). These types of models have achieved breakthrough performances in tasks such as image recognition [KSH12] and speech recognition [HDY+12], leading to a quick adoption. Reasons for their popularity include increased chip processing abilities, e.g. with General-Purpose Computing on Graphics Processing Units (GPGPU's), an increased size of available data that can be used for training, as well as advances in Machine Learning and Information Processing research [DY+14]. Due to its bright future, the visualizations discussed hereafter will focus on DL models.

There is neither a certain depth threshold to distinguish Deep Learners and Shallow Learners, nor is there a final definition for DL. However, most definitions share the themes that models consist of multiple layers of non-linear information processing and that methods for supervised or unsupervised learning find feature representations at increasingly higher and more abstract levels [DY+14]. An example for such an hierarchical abstraction is a digital image. At its lowest level, its contents are represented by an array of pixel values, potentially separated into color channels. A DL model might combine these pixel values into edges with their orientations and locations. At the next level, edges with particular arrangements might be combined into motifs. Multiple of these motifs could make up parts of objects, which could then be combined into whole objects [LBH15].

2.2.1 Methods and Applications

DL models can be designed and trained in various ways. Important training methods include Supervised Learning (SL) and Unsupervised Learning (UL). An example for SL is the classification of an input image into one of multiple classes. The model receives an image as its input data and produces a vector of outputs. These outputs can be scores, each one corresponding to a category and indicating the certainty of the model that the input belongs to that category. When performing SL, one has to supply the desired label of each training input sample, for example the ground-truth category. It is assumed, that input events are independent of each other and earlier output events [Sch15]. Using the model's output and the desired output, a function is computed that measures the error between the actual and the desired output. In order to improve the model, this error value has to

be minimized by modifying the model's internal parameters accordingly. To do so, a gradient vector for each weight is computed, indicating how the error would change when the weight would be adjusted in correspondence with the gradient. By adjusting weights according to the negative value of their gradient, the error value can be reduced, for example using *Stochastic Gradient Descent* [Bot10]. The most popular error minimization technique in current networks is *Backpropagation*, which can be used to train networks with stacks of multiple levels [LBH15]. SL has been dominant in recent competitions. Supervised deep NN's have proven especially powerful and have won many international pattern recognition competitions [Sch15].

In contrast, UL techniques are able to "capture high-order correlation of the observed or visible data for pattern analysis or synthesis purposes when no information about target class labels is available" [DY+14]. Use cases include the generation of encodings for raw data that lend themselves better for subsequent goal-directed learning [Sch15].

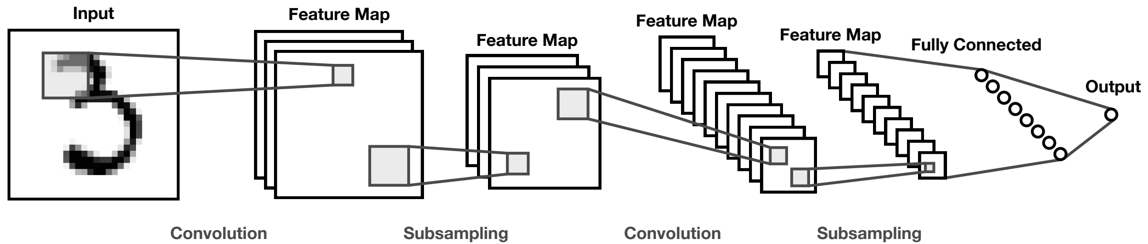
On top of these learning techniques, there are multiple ways in which DL networks can be constructed. The first major class is comprised of Feed-Forward Neural Networks (FNN's), which map a fixed-size input (e.g. an image) to a fixed-size output (e.g. the probabilities of every possible category of object that may be visible in the image). On the other hand, in a Recurrent Neural Network (RNN), external input is processed sequentially one element at a time. Neurons can receive input from other neurons at previous discrete time steps. Due to this property, RNN's are able to maintain a state vector in their hidden units that implicitly contains historical information about earlier input elements [LBH15]. Using the graph notation from above where neurons were understood as nodes connected by weighted directed edges, FNN's would form acyclic graphs and RNN's would form ones containing cycles [Sch15].

In FNN's, the weighted sum of inputs from the previous layer is computed and then passed through a non-linear function onto the next layer. The most popular way of introducing non-linearity is by using a Rectified Linear Unit (ReLU) [NH10], which simply implements the half-wave rectifier. This method turned out to work well and promote fast learning in networks with many layers. When done right, this transformation of input data in a non-linear way can make categories linearly separable by the last layer [LBH15].

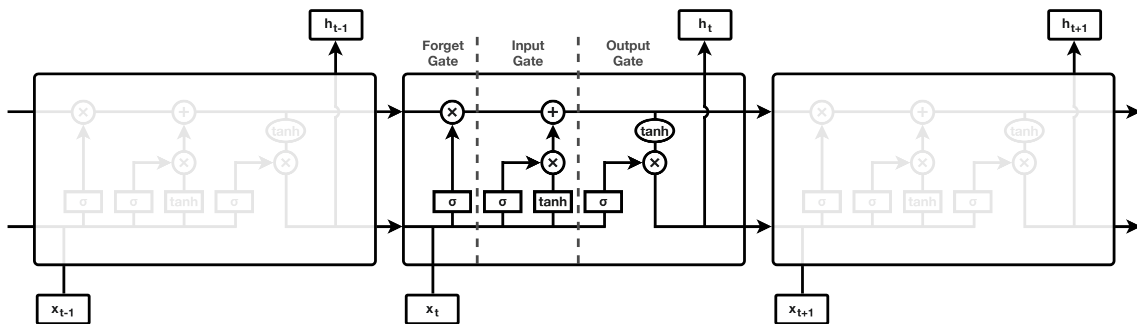
Recently, a special type of FNN called Convolutional Neural Network (CNN) has proven to train more easily and achieve better performances than networks where adjacent layers are fully connected. Their key ideas are local connectivity between layers, shared weights reducing the NN's descriptive complexity [Sch15], pooling, and a large amount of layers.

The architecture of a CNN consists of multiple stages. The first stages contain alternating *convolutional layers* and *pooling layers*. Convolutional units have a typically rectangular receptive field with a given weight vector, a filter. This filter is shifted step by step across a two-dimensional array of input values, producing the unit's output [Sch15]. This discrete convolution efficiently emulates the activations of individual neurons. Instead of having to store and use a bias value and a set of weights for each individual neuron, the amount of parameters can be significantly reduced by sharing them via a convolution filter. The freed up resources can then be invested into increasing the depth of a network. This convolutional approach works well on natural array data such as images because of the high correlation of local groups of values.

The output of these convolutional units is then passed through a non-linearity. A popular choice in modern networks are ReLU's, which simply perform half-wave rectification of their input values.



(a) A Convolutional Neural Network (CNN) processes an input image through repeated steps of convolution and subsampling, leading into fully connected and output layers (Adapted from LeCun et al. [LB+95]).



(b) A schematic of modules of a Long Short-Term Memory (LSTM) network, unrolled in time. The horizontal arrow at the top represents the explicit memory carried through time steps. A module reads the input of a discrete time step, chooses whether to remove parts of its memory or add new values to the memory, and finally creates an output for each time step. Rectangles inside modules represent network layers, ovals represent point-wise operations (Adapted from Olah [Ola15]).

Figure 2.3: Exemplary architectures of a Convolutional Neural Network (CNN) and an Long Short-Term Memory (LSTM).

Afterwards, a pooling layer decreases the data's dimensions by downsampling feature positions, thereby merging semantically similar features [Sch15]. Examples include *Max Pooling*, which returns the maximum value of each cluster, and *Average Pooling*, which returns the average of all values of each cluster.

These stages are followed by further convolutional layers and finally by fully-connected layers, leading up to the model's output [LBH15]. A schematic of how a CNN works on image input can be seen in figure 2.3a.

CNN's work well for data that is in the form of multiple arrays, such as the color channels of a two-dimensional image. By exploiting the fact that many natural signals are made up of compositional hierarchies they score highly in competitions, especially in combination with strong GPU's [LBH15]. Recently, one such model even reached a super-human level of visual pattern recognition [CMMS12] in a competition with the goal of recognizing German traffic signs [SSSI11]. Because they have become the dominant approach for almost all visual recognition and detection tasks [LBH15], convolutional example models will be prominent among the presets for Advis.

RNN's on the other hand are especially useful for tasks that involve sequential inputs with temporal ordering, such as speech and language. Even if hidden units can affect themselves at different time steps, the network can be theoretically unfolded in time, revealing a particularly deep multi-layer network. This both enables RNN's to find solutions to deep problems and makes it possible to

apply back-propagation for training, although this type of training might introduce a new problem: Back-propagated gradients tend to grow or shrink at each time step, leading to them exploding or vanishing over longer time periods. A solution to this problem is augmenting the network with explicit memory in the form of a memory cell. This cell carries memorized values through time steps using a connection to itself with weight 1. This connection is gated by another unit that learns to decide when the memory cell should be cleared. This type of augmented RNN is called LSTM [LBH15]. Figure 2.3b contains a schematic outlining its architecture.

Applications for DL models are diverse [DY+14] and include but are not limited to:

- Computer Vision and Object Recognition [KSH12]
- Speech and Audio Processing [DLH+13]
- Language Modeling and Natural Language Processing [CW08]
- Information Retrieval [PDS+16]
- Multimodal and Multi-Task Learning [NKK+11]
- Artistic Applications, such as DeepDream [MOT15] or transferring image styles [JAF16] [UVL16]

The scope of the framework developed in this thesis will be limited to models for image classification but can be expanded to more areas in the future.

2.2.2 Model Evaluation

After a network has been trained, its performance has to be measured. The data needed for these performance measures is collected by running the model on the data samples within an evaluation set and recording its output. These outputs can then be compared to the known ground-truth values of the samples. To be able to test the model's predictive capabilities, the test set must not contain samples that have been part of the training set. If the model is able to predict samples it has not seen before, it is able to generalize [LBH15].

There are various performance metrics for classification models [FHM09]. Perhaps the simplest one is the model's *accuracy*, which is defined as the percentage of right predictions among all predictions. In the following, this metric will be either called *top-1 accuracy* or *accuracy* for simplicity's sake, whereas a metric such as the *top-5 accuracy* measures the percentage of model outputs where the right category was among the five predictions with the highest certainty. Given the model's top predictions and the ground-truth labels of samples from the test set, the predictions can be categorized into true positives, false positives, false negatives, and true negatives. Dividing the number of true positives by the number of total positives yields the model's *precision*. Conversely, dividing the number of true positives by the total number of samples predicted as positives (true positives and false negatives) yields the model's *recall*. By calculating the harmonic mean of precision and recall, one gets another important performance metric, the *F1 score*. Another performance metric usually encountered in the context of DL models is the Area Under the Receiver Operating Characteristic Curve (AUC) [Faw06], which "is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance"

[FHM09]. A big advantage of this metric over the simple accuracy is its invariance to potentially imbalanced category distributions within the evaluation set. Originally defined for binary classifiers, these measurements can be extended to classifiers with multiple categories [FHM09].

These metrics measure a model's overall performance but fail to provide a view on where the classifier makes mistakes. Because these insights are crucial for improving a model, confusion matrices [Ste97] have emerged as a popular and intuitive way for visualizing a classifier's predictions on different categories. The columns of such a confusion matrix contain actual classes of input samples, the rows contain classes as predicted by the classifier. Each cell's value corresponds to the amount of samples that are labeled as the cell's actual class, and have been predicted as the cell's predicted class. Therefore, correct classifications are situated across the diagonal of this quadratic matrix, while misclassifications outside of the diagonal represent a classifier's mistakes. Sometimes, cell values are emphasized by coloring them according to a color scale.

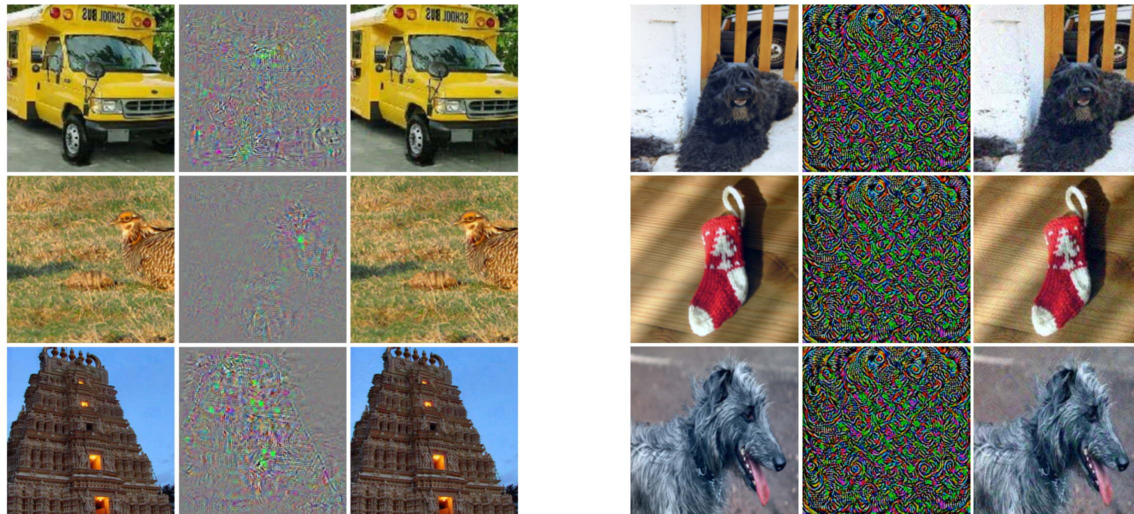
The previously mentioned metrics of precision and recall can be defined intuitively based on the confusion matrix. Each row and column contains one true positive value where the actual class equals the predicted class. Dividing the value of this true positive cell in a row by the sum of all values in the row yields its precision. Conversely, dividing the value of the true positive cell in a column by the sum of all values in the row yields its recall.

Confusion matrices only represent the single top class predictions of a model. They omit important aspects like individual classification scores or error severity, whereas an error can be considered worse if the model has a high confidence in its misclassification. Multiple visualization techniques have been proposed to alleviate this problem, usually by encoding more information in more complex representations. Some examples for these alternatives to confusion matrices will be presented in section 4.2.1.

The aforementioned metrics provided a starting point for researchers to compare the performance of their models and contest each other, accelerating research. In order to establish consistent test sites for DL models, a multitude of challenges has been created where researchers can compete with each other by comparing their model's performance metrics on a given task, sometimes with monetary prices. Perhaps the most basic benchmark for computer vision is the Modified National Institute of Standards and Technology Database (MNIST), a dataset of 60,000 training images and 10,000 test images of handwritten digits between 0 and 9 that have to be classified [LeC98]. In 2013, a DL model achieved a record error rate of only 0.21% for this task [WZZ+13]. Other challenges include the yearly ImageNet Large Scale Visual Recognition Competition (ILSVRC) [RDS+15], where images have to be classified and objects detected on a vast set of 1.2 million input images with 1000 classes which are a subset of the ImageNet dataset [DDS+09]. The annual Conference on Neural Information Processing Systems (NIPS) offers more competitions, with the task of 2017 focussing on adversarial attacks and defenses [KGB+18], a topic close to the focus of this thesis. The platform Kaggle[1] has established itself as a central host for these and many more challenges.

Test datasets from some of these challenges will be used to evaluate models within Advis.

[1] https://www.kaggle.com



(a) Adversarial examples generated for three input images and the AlexNet model [KSH12] (Adapted from [SZS+13]). Each input image has a unique perturbation.

(b) Universal adversarial examples for three input images and the initial Inception model [SLJ+15] (Adapted from [MFFF17]). Each input image is perturbed in the same way.

Figure 2.4: Examples for adversarial perturbations. Left columns contain unmodified input images, central columns contain visual representations of the perturbation added to the input image, right columns contain modified input images causing a misclassification.

2.2.3 Input Image Perturbations

A network's performance can be affected if its input images have been manipulated. For example, the predictions of object detectors can deteriorate if objects are partly occluded or if their size is changed [HCD12].

Contrarily, image transformations can also be used to improve the performance of DL classification networks by data augmentation [How13]. Images from the training set can be cropped, flipped, or their contrast, brightness, and color varied randomly to generate more training samples. These variations can also be used at test time to generate a list of predictions that are then combined into a final prediction, which is hoped to be more accurate than a single prediction on the non-transformed input image.

Recently, a targeted type of image transformation has evoked special interest. By maximizing a network's prediction error, a perturbation of the input image can be found that is hardly perceptible as seen in figure 2.4a but causes the model to misclassify the image [SZS+13]. Even worse, these so-called *adversarial perturbations* tend to generalize well, both across models with different hyperparameters such as their number of layers, regularization, and initial weights, as well as across training sets [SZS+13].

Using a faster method of creating adversarial input data, these samples can be continually added to the pool of training images, hardening the model against adversarial attacks. However, when fooled, the model's confidence of its wrong classification remains quite high [GSS14]. In the advent of

these surprisingly effective attacks, *CleverHans* has been developed, a software library providing standardized reference implementation of attacks for benchmarking an existing model's robustness or improving it through the aforementioned method of adversarial training [PGS+16].

Continuing in this track of adversarial input data, Moosavi-Dezfooli et al. [MFFF17] found a single perturbation that can be applied to all input images of a deep neural network, fooling it with a high probability. By successively aggregating minimal perturbations that send the current perturbed points out of their corresponding classification region, an image-agnostic perturbation is found that fools models reliably but is again barely visible when applied to input images as seen in 2.4b. Using this technique, attacking a model no longer involves solving an optimization problem for each set of input data. Simply adding a pre-computed perturbation vector to the input image suffices, leading to significant consequences for the security of applied Machine Learning systems [BNS+06; HJN+11]. Furthermore, fine-tuning the model using adversarial input images leads to small improvements but not full immunity against those attacks [MFFF17]. This technique of creating universal adversarial input images is available as a preset in Advis.

2.2.4 Frameworks for Implementing Deep Learning

With DL quickly and continually gaining popularity, many frameworks have been released for implementing and evaluating models. Relieving researchers and practitioners of minute implementation details, they offer varyingly high-level interfaces to realize commonly needed concepts of deep neural networks. In combination with good support for modern hardware, these frameworks have become indispensable for DL research and application.

An excerpt of popular frameworks can be found in table 2.1. Moreover, table 2.2 contains some popular high-level wrappers for these aforementioned frameworks, abstracting from some of their implementation details. When designing Advis, the compatibility with models implemented in these frameworks was an important factor and their popularity had to be considered.

Name	Authors	Reference	License	APIs	Website
Caffe	Berkeley Vision and Learning Center	[JSD+14]	BSD	Python, MATLAB, C++	http://caffe.berkeleyvision.org
Caffe2	Berkeley Vision and Learning Center	N/A	BSD	Python, C++	https://caffe2.ai
Chainer	Preferred Networks, Inc.	[TOHC15]	MIT	Python	https://chainer.org
Cognitive Toolkit (CNTK)	Microsoft Research	[SA16]	MIT	Python, C++, C#/.NET, Java	https://www.microsoft.com/en-us/cognitive-toolkit/
Deeplearning4j	Eclipse Foundation	N/A	Apache 2.0	Java, Scala, Clojure, Kotlin	https://deeplearning4j.org
MatConvNet	Oxford Visual Geometry Group	[VL15]	BSD	MATLAB	http://www.vlfeat.org/matconvnet/
MXNET	Apache Software Foundation	[CLL+15]	Apache 2.0	Python, C++, Scala, Julia, Perl, R	https://mxnet.apache.org
PyTorch	Adam Paszke et al.	[PGC+17]	BSD	Python	https://pytorch.org
scikit-learn	David Cournapeau et al.	[PVG+11]	BSD	Python	http://scikit-learn.org
TensorFlow	Google Brain Team	[ABC+16]	Apache 2.0	Python, C++, Java, Go, Swift, and more community-developed bindings	https://www.tensorflow.org
Theano	Theano Development Team	[The16]	BSD	Python	http://www.deeplearning.net/software/theano/
Torch	Ronan Collobert et al.	[CKF11]	BSD	Lua, C	http://torch.ch

Table 2.1: An excerpt of popular frameworks for implementing deep learning models.

Name	Authors	Reference	License	Wrapped Frameworks	APIs	Website
Keras	François Chollet et al.	[Cho+15]	MIT	TensorFlow, CNTK, Theano	Python, community-developed binding for R	https://keras.io
Lasagne	Lasagne Contributors	[DSR+15]	MIT	Theano	Python	http://lasagne.readthedocs.io
TFLearn	Aymeric Damien et al.	[Dam+16]	MIT	TensorFlow	Python	http://tflearn.org

Table 2.2: An excerpt of popular high-level interfaces that wrap frameworks for implementing deep learning models.

3 Related Work

DL models can be used to great effect for solving difficult classification tasks. However, in order to accommodate for the difficulty of these tasks, networks have steadily increased in complexity and size. Their inner workings have ceased being transparent and interpretable. There is no readily available explanation for their decisions, resulting in a lack of trust. Instead, DL systems are often used as black boxes, with the hope that they work well enough for the task at hand.

Equipping users and researchers with the necessary tools to find out how a model works, what it does, when it fails and why it fails can open up this black box, restoring understanding and trust. Only recently, many Visual Analytics systems have been proposed to this extent, enabling users to interpret models, explain their decisions, compare them, debug them, and ultimately improve them. Section 3.1 will outline related systems and their characteristics. Afterwards, section 3.2 will describe a selected set of systems that are the most relevant to the task of this thesis and explain how they differ from Advis.

3.1 Survey of Deep Learning Visualization Approaches

Hohman et al. [HKPC18] introduce a set of attributes that can be used to classify Visual Analytics systems in the context of DL. These attributes include the visualization's goals, what elements of a model are being visualized, how they are being visualized, and when they are being visualized. This classification will be used to provide a structured overview over existing systems.

3.1.1 Visualization Goals

The most common underlying reason for visualizing DL models and processes is to provide *interpretability and explainability* [HKPC18]. By allowing the user to understand decisions of a model and the representations it has learned, they are more likely to trust it. Moreover, while traditional performance metrics allow a quantitative evaluation measure, the model's interpretation can be their qualitative counterpart. Olah et al. [OSJ+18] provide a framework of building blocks that constitute such a system for improving interpretability. Although nearly all systems in this overview share this goal of interpretability, some have made it their focus. For example, some try to build trust by explaining a classifier's predictions [FH03; KPN16; RSG16]. Krause et al. [KPN16] even try to explain them well enough that medical professionals can gain actionable insights to change the prediction of a patient's illness. Others allow users to gain insights into different model's inner workings, either by allowing them to provide input data [SGPR18; YCN+15] or by visualizing the evolution of a model [CPMC17; SWA01; ZBM16].

Other systems understand the creation of a model as an iterative process with many parameters that have to be adjusted and fine-tuned. They aim to support this process of *debugging and improving models* [HKPC18] by helping developers quickly identify and fix problems. In order to achieve this goal, some systems enable users to recognize patterns that might hint at issues within a model's structure or trained parameters which can then be fixed [ACD+15; AHH+14; Bru14; Goo17a; KAKC18; PHV+18; ZBM16]. Another domain where issues might need to be revealed is within the training data [AJY+18; RAL+17]. Other systems allow informed decisions for guiding a model during its training process [CGR+17; CSP+16; ZXZ+17]. Others simply convey a better basic understanding of a model and its mechanisms that might enable users to improve it [KFC16; LSL+17; NQ17]. Talbot et al. [TLKT09] enable the informed creation of ensemble classifiers that are superior to their single components.

Another area where DL visualizations can be helpful is the *comparison and selection of models* [HKPC18]. This area encompasses both the comparison of the same model at different training stages [Goo17a; ZHP+17] and the comparison of multiple different models [AHH+14; HHC17; KFC16; KPN16; MCZ+17; RAL+17; TLKT09]. In practice, some of these systems simply display resulting visualizations of different models next to each other [AHH+14; Goo17a; HHC17; MCZ+17; RAL+17] while others employ more advanced visualizations to compare multiple models in one view [KFC16; KPN16; TLKT09; ZHP+17].

The final reason for visualization is an educational one, the *teaching of deep learning concepts* [HKPC18]. These systems are usually lightweight and contain visualizations that are easy to interpret. Non-expert users can interactively play with aspects of DL, either by manipulating models and their hyper-parameters [Kar14; NQ17; SCS+17] or by supplying input data themselves, for example by writing [CHJO16; Har15], selecting input images [HHC17] or via a webcam [Goo17b]. After manipulating some part of the process, users generally receive quick feedback that can build understanding and intuition.

Advis provides information about layer activation differences and the distribution of misclassifications across categories when transforming input images, supporting the goal of *debugging and improving models*. On top of that, multiple performance metrics of a list of models are generated and visualized, targeting the goal of *comparing and selecting models* that are well-suited for a specific task, especially when robustness against input transformations is of interest.

3.1.2 Visualized Elements

DL models have a set of prominent attributes that lend themselves to visualization. One of these attributes is the model's *computation graph* and *network architecture* [HKPC18]. Often, the network's topology is used as an entry point to the system. Users can click on a node to expand its visualizations. This topology can either be shown explicitly as a graph [Bru14; KAKC18] or explicitly as a more compact table or list of layers [Kar14; PHV+18; YCN+15; ZXZ+17]. Another possibility is encoding relevant information within this graph using visual representations such as size or color of nodes and edges [CSP+16; Har15; LSL+17; SCS+17; SWA01; ZHP+17]. This is especially useful for conveying information that is bound to individual parts within the network's architecture. Moreover, some systems simplify graphs before displaying them to allow for an easier overview [Goo17a; LSL+17].

Another attribute that can be visualized are *learned model parameters* [HKPC18]. These include filters and weights of individual layers [Bru14; CSP+16; Har15; Kar14; PHV+18; SWA01; WGYS18; ZHP+17], for example as colored maps. Learned features might be shown using one of various specialized feature visualization techniques [AHH+14; CGR+17; HHC17; LSL+17; SCS+17; YCN+15; ZHP+17; ZXZ+17], some of which will be outlined later. Architecture-specific parameters include hidden state clusters of RNN's [KPN16].

Furthermore, information about *individual computational units* [HKPC18] might be of interest. These include activations of individual neurons shown as color maps [Bru14; CHJO16; CSP+16; Kar14; PHV+18; WGYS18; YCN+15] or in more advanced ways that emphasize their connection to individual neurons [Har15; SCS+17; ZBM16; ZXZ+17]. Especially for visualizing a unit's training progress, showing error gradients can be useful [CPMC17; Kar14; ZBM16]. Other systems visualize model-specific data such as individual hidden states inside of an LSTM [SGPR18].

Deep neural networks contain a large amount of neurons in each layer, resulting in a high dimensionality of output data. Visualizing these *neurons in high-dimensional space* [HKPC18] in a meaningful way can contribute to a user's understanding. For example, some systems project the neuron's activations of a layer into two-dimensional space [CHJO16; KAKC18; Kar14; PHV+18; ZXZ+17], using dimensionality reduction techniques such as t-SNE [MH08]. Sometimes, this projection is only performed on the last layer, allowing a clustering of input samples [Bru14; ZBM16]. Chung et al. [CSP+16] not only show a two-dimensional embedding of activation maps and gradients, but also of filter coefficients and gradients.

All of these detailed visualizations can be complemented by a more macroscopic view on *aggregated information* [HKPC18] across a model. A common way of providing this information is by displaying traditional performance metrics such as accuracy or AUC [CGR+17; CSP+16; Kar14; KFC16; KPN16; TLKT09; WGYS18]. During training, other scalars such as test and training loss can be of interest [Goo17a; SCS+17]. Some systems try to enrich these conventional metrics by subsuming them with different visualizations that can carry even more information [AHH+14; KAKC18], for example by using the positioning of color-coded boxes and bars [ACD+15; RAL+17]. Confusion matrices are another way of aggregating model predictions [AJY+18; Bru14; TLKT09]. Similarly, other systems simply collect a model's predictions on a test set [CGR+17; Kar14; KFC16; WGYS18] or neuron activations over time [CHJO16; ZBM16]. Furthermore, systems may visualize an overview of the values of all layers using metrics they have defined before, such as discriminability [ZXZ+17] or perplexity [PHV+18]. As further variations, Ming et al. [MCZ+17] display the distribution of RNN model responses to a selected word, Strobelt et al. [SGPR18] match and display similar hidden state patterns of LSTMs from different input samples, Frank et al. [FH03] plot class probability estimates as a function of two of the data attributes, Liu et al. [LSL+17] show multiple important facets of a neuron at once and Streeter et al. [SWA01] give an overview of weight matrices of a network family tree that has been created by an evolutionary algorithm, as well as a fitness graph for generations of a population of networks.

Advis displays the *computational graph*, both as an entry point to more detailed visualizations and to encode information about activation differences using color. The activation of *individual computational units* on original and transformed input images can be viewed and compared. *Aggregated information* is shown in the form of detailed performance metrics of individual models and a large-scale confusion matrix.

3.1.3 Visualization Methods

After elements that should be shown have been identified, a system's designer has to decide on how to visualize them. Of the many possibilities for data visualization, a selection of methods has proven especially prominent among DL visualizations, one of which are *node-link diagrams for network architectures* [HKPC18], as used by Wongsuphasawat et al. [WSW+18]. These graphs can be used as an entry point whereas clicking on a node reveals further information [Bru14; KAKC18; ZHP+17] or they can encode further information about nodes and edges using variables such as color or scale [CSP+16; Har15; LSL+17; SCS+17; SWA01].

Scatter plots and dimensionality reduction [HKPC18] can be used to provide a comprehensive view on activations [CSP+16; KAKC18; Kar14; ZBM16], filters [CSP+16; PHV+18], features [ZXZ+17] or data samples [AHH+14; FH03; Goo17a; SCS+17].

When showing the evolution of a model, *line charts* [HKPC18] are used to visualize changes in temporal metrics, for example a model's training loss [CSP+16; Goo17a; Kar14; PHV+18; SCS+17] or performance [CGR+17; Goo17a; PHV+18; SWA01; WGYS18].

Drilling down from higher-level to more concrete information pertaining to input instances, many systems allow an *instance-based analysis and exploration* [HKPC18]. This can be achieved by allowing the user to select and inspect a single instance [Bru14; CGR+17; CHJO16; Har15; MCZ+17; NQ17; RSG16; SGPR18; ZBM16; ZHP+17] or a subset of instances [AHH+14; AJY+18; HHC17; KAKC18; KFC16; KPN16]. Some systems allow the live creation of input data [Goo17b; YCN+15] or bookmarking and tracking changes across single instances [ACD+15; RAL+17; WGYS18].

Rather than relying on input samples, some systems try to directly visualize a network's learned representations through *attribution and feature visualization* [HKPC18]. The former can be shown using saliency maps [AJY+18; ZBM16]. The latter can be created using a host of different techniques. Popular ones include the so-called deconvolution process which maps activations back into input pixel space by reversing convolution operations [ZF14], visual back-propagation [BCC+16] or by finding pixels in an input image that maximize or minimize the activation of a specific decision [ZCAW17]. Hohman et al. [HHC17], Yosinski et al. [YCN+15] and Zhong et al. [ZXZ+17] make use of these techniques to represent learned features.

Finally, especially educational systems include opportunities for *interactive experimentation* [HKPC18], such as the interactive modification of a network's topology [Kar14; SCS+17], hyper-parameters [HHC17; Kar14; SCS+17], weights [SWA01] or classifier combination of an ensemble model [TLKT09].

Advis extends the *node-link diagram* for computation graphs included in TensorBoard [WSW+18] by simplifying it, improving its layout, and encoding hierarchically-aggregated relevant information in its node's colors. *Instance-based analysis and exploration* is possible by selecting both a distortion method and an individual input image through an interactive confusion matrix.

3.1.4 Visualization Timing

The last characteristics of the systems in this overview is the timing of their visualization. Some systems visualize models *during training* [HKPC18]. In practice, visualizations might update dynamically while the training process progresses [ACD+15; CPMC17; CSP+16; Goo17a; Kar14; PHV+18; SCS+17; WGYS18; ZXZ+17]. Others allow the selection of a training stage, for example with a slider [AJY+18; Bru14; CGR+17; SWA01; ZHP+17].

The majority of systems however visualize models *after training* [HKPC18] by initializing all visualizations on user-supplied models that have already been trained [AHH+14; CHJO16; FH03; Har15; KAKC18; KFC16; KPN16; LSL+17; MCZ+17; NQ17; RAL+17; RSG16; SGPR18; TLKT09; YCN+15; ZBM16] or by letting the user train a model interactively and then evaluate it afterwards [Goo17b; HHC17].

Advis also visualizes models *after training*. However, model modules carry a version attribute that can be used to load and compare multiple snapshots of a model.

3.2 Selected Approaches

Four of the systems outlined in section 3.1 that come the closest to solving the task of probing learned models in the context of input perturbations have been selected. The user interfaces of these systems can be seen in figure 3.1. Their approaches will be described in further detail hereafter.

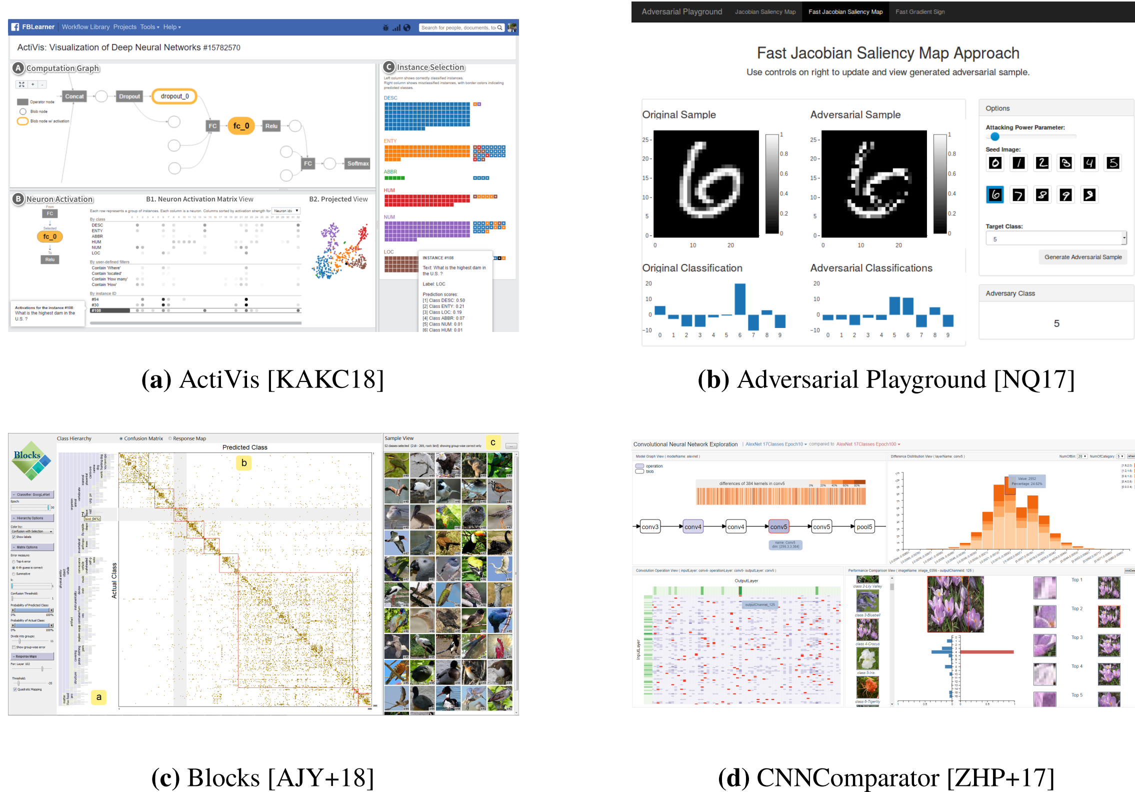
3.2.1 ActiVis

ActiVis [KAKC18] aims to combine instance- and subset-based exploration. Individual instances are usually more familiar to a user and can build a fast understanding of a model, whereas subset-based exploration is more abstract and therefore more effective on large datasets. This goal is achieved through a three-tiered user interface.

A central computation graph provides an overview of the model, allowing users to understand its structure before deciding on which layers they want to analyze further.

After a node from the graph has been selected, the neuron activation view shows up below the graph. Next to the name of the selected node and its neighbors, its central component is the neuron activation matrix view. Columns in this matrix represent neurons, rows represent instances or instance subsets, and cells contain the activation of a neuron on an instance or its average activation on an instance subset. Users can define these subsets using rich queries on raw data attributes, labels, features, output scores or predicted labels. Next to this matrix view, instance activations are projected into two dimensions using t-SNE [MH08]. Instance selections in this view are interactively linked to the matrix. Finally, multiple instances of this neuron activation view can be shown below each other, allowing the comparison of multiple layers.

The instance selection view provides users with an overview of instances and prediction results. Squares represent instances and are vertically grouped based on their true labels. Left columns show correctly classified instances sorted by their prediction scores, right columns contain misclassified



(a) ActiVis [KAKC18]

(b) Adversarial Playground [NQ17]

(c) Blocks [AJY+18]

(d) CNNComparator [ZHP+17]

Figure 3.1: User interfaces of applications that are closely related to the topic of this thesis. Screenshots have been retrieved from the respective proposals.

instances, with each square's fill color representing its true label and its border color representing the predicted label. Clicking on one of these instances adds it as a new row to the neuron activation matrix view.

ActiVis is the most useful for data types other than images, such as text and numerical data. Queries are less effective on image instances and showing a row of neuron activations in the matrix view removes important context contained in the two-dimensional shape of convolutional layers, while their size will seriously overflow this view with content. Furthermore, both inside the matrix and inside the color-coded projected view and instance selection view, users will struggle to discern classes and retrieve useful information when presented with classification tasks that contain a large number of categories, as is the case with the popular ILSVRC's [RDS+15] 1,000 classes. Furthermore, in contrast to Advis, there is no integrated way of perturbing images and no way of comparing multiple models.

3.2.2 Adversarial Playground

The Adversarial Playground [NQ17] enables a better understanding of how different types of adversarial attacks can fool deep neural networks. It is mainly an educational tool aimed at non-experts that can interactively generate different adversarial samples on-demand. Due to its modularity, it can be extended to other models and attack algorithms and plugged into benchmarking frameworks which might make it interesting to experts and model builders.

The interface of the Adversarial Playground is lightweight and rather simple. An options view shows a set of images, one from each category in the dataset. The user can select such a seed image, the desired attack strength and the target class if they are using an algorithm for targeted attacks. After confirming their input, an adversarial sample is created and displayed alongside the original input image. Under both the original and adversarial input image, the model's classification scores are shown. The options view displays the final prediction for the adversarial image.

The framework contains three preset attack modules, the *Fast Gradient Sign Method* and the *Jacobian Saliency Map Approach* from the *CleverHans* library [PGS+16], as well as a custom improved *Fast Jacobian Saliency Map Approach*.

The Adversarial Playground prides itself in being the first DL visualization system with a focus on adversarial samples, but its educational target audience quickly becomes apparent in its rather limited set of functionality. Contrarily to Advis, no direct comparison between attacks and models is possible and no aggregated information such as performance metrics is shown. There is no graph view to provide an overview for the model architecture and no visualization of the activations of individual computational units. Furthermore, there is no way to inspect more instances than the pre-defined seed images and the classification score diagram will become cluttered and confusing when more than a handful of classes are available.

3.2.3 Blocks

Blocks [AJY+18] supports the analysis of three important data facets – input images, network-internal data, and classification results – to highlight hierarchical patterns of misclassification. Its user interface is separated into four interactively linked views, with the option to either show a large confusion matrix or response map as the central view.

The hierarchy viewer to the side of the central view shows the hierarchy inherent in a large set of categories as a horizontal icicle plot. These hierarchies can either be pre-defined or interactively defined using a seriation algorithm. Individual rectangles within the icicle plot can be colored to encode information about a group, such as a group-level performance metric or the performance difference between two classification result sets when the confusion matrix is active. When the response map is active, they can be colored to show the average response of a selected neuron. Child rectangles can be sorted by size or performance. Clicking on a rectangle selects the corresponding classes in all views.

A central confusion matrix displays a model's classifications, with actual classes as rows and predicted classes as columns. Cells are colored according to their value, with an optional logarithmic color mapping to emphasize less frequent confusions. Correct classifications along the diagonal of the matrix can be excluded to highlight misclassifications. Moreover, single non-zero cells might

disappear due to the large matrix size, which can be mitigated with a halo effect around these cells. This ordered visualization reveals a pattern of block-like misclassifications which can be emphasized by automatically drawing boxes around them. Finally, the matrix contents can be filtered by cell value, top-k results or classification probability. Samples can be selected by drawing a box around them or by clicking on a group in the hierarchy.

Instead of the confusion matrix, a response map can be shown that displays the average response for every class of every neuron in the selected layer, using a heat map of downsampled and linearized activation maps. Individual neuron columns can then be ordered according to their relevance for a specific hierarchy group, revealing potential patterns. Their headers can encode summary information about a neuron using color, such as their average activation for a selected class. Clicking on a profile header makes the sample viewer show images that highly activate the corresponding neuron. Furthermore, latent sub-classes within a single class can be explored in a dedicated window containing a correlation matrix, a sample-level response map, and an ordered list of corresponding samples.

To the right of the interface, a sample viewer shows thumbnails of the selected sample images. They can be grouped by their actual class and filtered by class membership, activation of a selected neuron or classification result. On top of that, the sample viewer can display saliency maps.

Using their visualization system, the authors conclude that early layers are able to separate high-level category groups even after little training, whereas latter layers separate lower-level groups and take more time. To speed up this process, they propose a hierarchy-aware model architecture. On top of that, they are able to highlight quality issues within their dataset.

Blocks allows loading and comparing two sets of classification results, which may be used for a rudimentary analysis of the effect of image perturbations. Other than Advis, these image transformations and especially adversarial perturbations are not integrated and have to be performed outside of the system. Even then, only two classification results can be compared at once and the creation or comparison of multiple distorted versions of input images is impossible. On top of that, there is no graph to give the user an overview of the network architecture. Instead, layers have to be selected by their number using a small slider which is cumbersome and impractical, especially in today's large networks. No immediate performance metrics of models are available. Finally, the two-dimensional structure and potentially important data of neuron activations is lost in the process of downsampling and linearization.

3.2.4 CNNComparator

The focus of CNNComparator [ZHP+17] is the analysis of differences between two model snapshots after different epochs in the same training process. Its user interface is split into four quadrants.

The network architecture view provides an overview of the network structure and of parameter differences among layers, which are encoded as the node's colors. On hover, the overall kernel difference of a layer is shown.

The difference distribution view is situated to the right of the network architecture view, showing the distribution of parameter differences in the selected layer. For clarity in the face of vast amounts of weights, bins are created based on their absolute change. For each bin, the relative change is

quantified into one of several levels which are then shown as a stacked chart. The size of each rectangle in this chart encodes the number of weights it contains, the color encodes the change level. Clicking on a rectangle highlights the corresponding weights in the convolutional operation view.

The task of this convolutional operation view is to show differences in activation. A two-dimensional matrix displays convolution operations, with columns representing kernels and rows representing channels. Cells encode their value difference between the two model snapshots using their color. Users can zoom and pan to explore, and click to show a corresponding kernel matrix. By clicking on a rectangle of the row or column header, corresponding image patches are displayed in the performance comparison view.

This performance comparison view shows a side-by-side comparison of the two model snapshot's classifications using bar charts after the user has selected an input image. On top of that, image patches from this sample are chosen and ranked by their activation value on the selected channel.

While this system can be useful for analyzing the evolution of a model, it is limited to two snapshots from the same model and training process. In contrast to Advis, comparing multiple models with different architectures is impossible, and aggregated information such as model performance metrics are missing. There is no way of performing input image perturbations or visualizing their impact on a model. Activations of individual computational units cannot be shown. Furthermore, state-of-the-art image classification tasks will introduce scalability issues: The classification bar chart will be cluttered with more than a handful of categories and input images can only be selected from a single long list which is impractical with datasets containing tens of thousands of samples.

4 Concept

In this chapter, the requirements will be outlined that have to be fulfilled to solve the task of probing the response of learned models to transformed input images. Afterwards, the system's design and how it meets these requirements will be described.

4.1 Requirements

The following six requirements have to be met in order to solve the concrete task of this thesis while adhering to the guidelines for a high-quality Visual Analytics visualization system [KAF+08].

R1. **Comparison of models.** Users have to be able to compare the performance of models at a glance and retrieve more detailed information on demand. This comparison should be focussed on the model's performance on a set of user-defined input transformations. Using these insights, users should be able to find a model that is robust enough for the task at hand or decide on a model that they can improve further.

R2. **Analysis of the impact of input perturbations on internal representations.** Users have to be supported in the task of visually exploring models and how they react to perturbed input data. They have to be able to gain insights that can be used to debug and improve their model.

R3. **Wide extensibility.** Deep learning models and their application fields are evolving rapidly. A real-world task might involve any kind of dataset and model as well as a specific set of distortions. Therefore, it is essential that the system provides an appropriate interface that allows users to extend it by defining their own datasets, models, and distortions.

R4. **Scale with data volumes and dimensions.** State-of-the-art models contain overwhelming volumes of high-dimensional data. This data has to be made accessible to the user by representing it appropriately. A global overview has to be combined with visualizations of analysis details, with several levels of detail and abstraction in between. Drilling down from the overview allows the user to find areas of interest.

R5. **Intuitive user interface.** The user has to be able to focus solely on the task at hand rather than dealing with an overly complex or distracting interface. Therefore, the interface should be easy to use, consistent, seamless, and intuitive.

R6. **Appropriate infrastructure.** To allow users a smooth interaction with the system, it should respond at least every 100 milliseconds. Therefore, the system needs to be able to manage large amounts of data and perform computations asynchronously and without blocking the interface. Furthermore, the visual representation algorithms have to be fast and dynamic enough to accommodate for large data sizes.

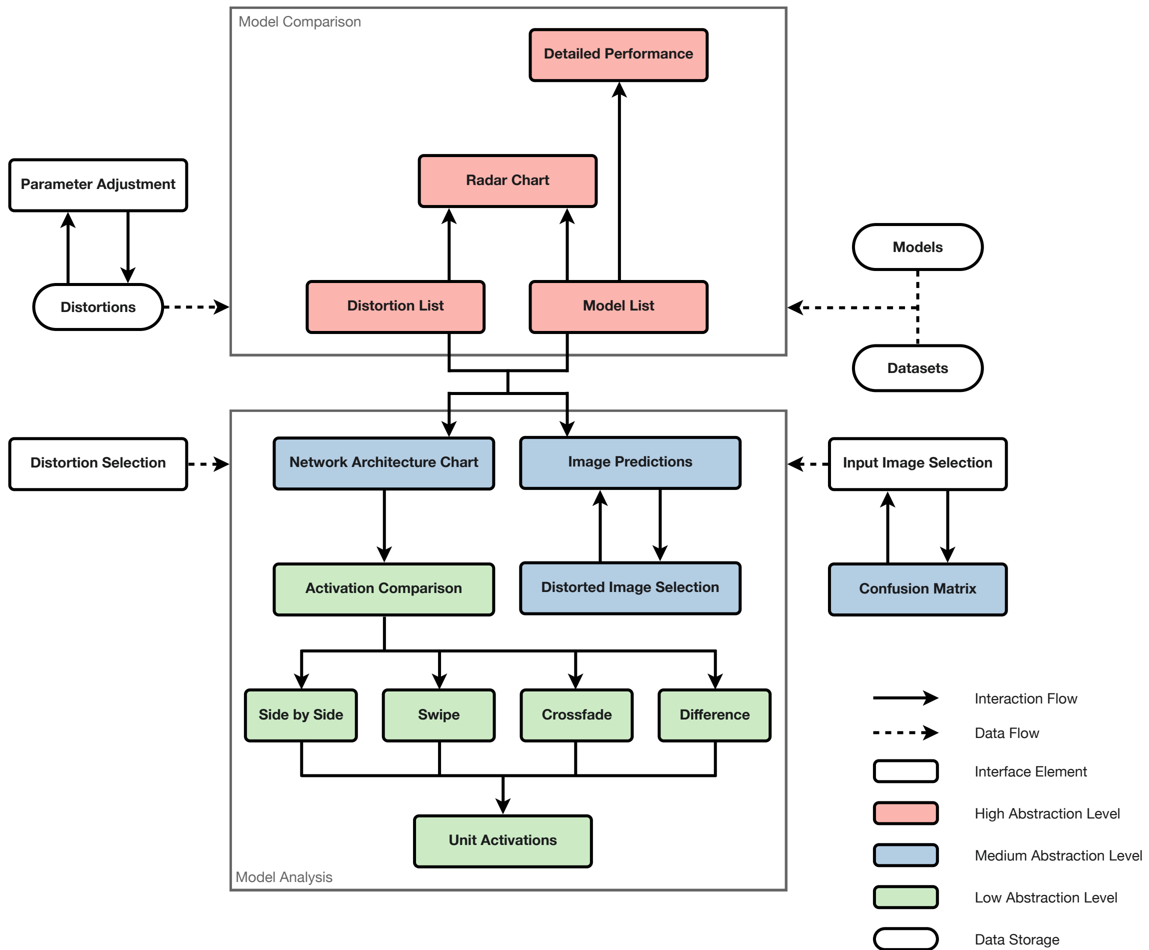


Figure 4.1: A high-level view on the interaction flow within Advis.

4.2 Design

Both the system's architecture as well as its views had to be carefully designed in order to fulfill the requirements outlined above. A process of iterative improvements produced the system design outlined in this section.

4.2.1 Interaction Flow

A defining feature of Advis was adhering to the principle of providing an overview before letting the user dive deeper into more detailed visualizations (R4). This decision yielded an interface of multiple interwoven tiers, where a selection in a higher-level interface element influences the available options in lower-level elements. On top of that, each option in such a view has to represent sufficient information that allows an informed decision about what to select. For example, a user might specify a set of distortions that should be applied to a data set. From there, they might choose a model that they are interested in. Within this model they can then choose an interesting layer and view its activations on an input image they have selected.

An overview of possible interaction flows within Advis is depicted in figure 4.1 and will be described in further detail hereafter.

The interaction flow of Advis begins outside of its graphical User Interface (UI), with users supplying models, datasets, and distortions specific to their analysis tasks (R3). The system has to support state-of-the-art models supplied using an easy-to-use and popular protocol. Furthermore, arbitrary datasets have to be able to be used to evaluate models. These datasets have to include actual image samples, a list of classes and the ground-truth label for each sample. Finally, users have to be able to specify a set of input image distortions that will be used to analyze a model's response and evaluate its performance on perturbed input data. Depending on the analyst's task, these distortions can range from simple image manipulations such as rotating or adjusting brightness, through using custom libraries to perform specialized transformations, to running another specialized network within the distortion that creates the perturbed output image. Moreover, distortions should be able to depend on a set of parameters that can be interactively configured through the UI. Configurations should persist through restarts.

Models, datasets, and distortions are fed into the system and displayed as a list of distortions and models. The distortion list allows users to select a subset of distortions, and the model list displays all models along with a host of performance metrics that signify their robustness to the selected set of distortions. Due to screen space concerns, these metrics have to be displayed in a rather compact form. Therefore, users have to be able to open a view with all detailed performance metrics. These high-level insights into performance metrics of individual models enable users to compare models and select one that is the most robust to the desired list of distortions (R1). However, rather than making users compare numbers, a subset of models can be selected from the list that is then displayed in a radar chart. This radar chart shows the robustness profiles of models on all selected distortions, so users can assess the strengths and weaknesses of a model.

When a user has found a model and distortion combination they want to analyze, they can simply click on its entry in the model list to dive deeper. The model analysis view that appears thereafter needs two types of input. Firstly, the desired distortion has to be selected from a dropdown menu. Secondly, an input image from the dataset associated with the model has to be selected. This input image selection has proven to be difficult to conceptualize in an efficient way. First versions included a simple grid of thumbnails with additional information which was impractical for datasets that often contain thousands of images. A solution to this problem that enables users to make an informed decision on what input image to choose was a confusion matrix alongside a sorted grid of image thumbnails, displaying further information such as the model's prediction certainty for the ground-truth label of that image. Displaying this confusion matrix was a challenge in itself. The commonly used ImageNet dataset [DDS+09] sorts its images into one of 1,000 classes, which would be far too much to display at once in the matrix's rows and columns. Instead, the matrix has to implement some sort of level of detail system itself (R4). This can be achieved by sorting categories into a hierarchy, defined by the dataset. This was especially helpful with the popular aforementioned ImageNet dataset, since its categories correspond to noun leaves in WordNet [Mil95], a lexical database for the English language. A meaningful hierarchy could then be built according to the recursive hypernyms of these nouns.

Before deciding on using a confusion matrix for input image selections, various alternatives had to be considered. These proposals criticize that confusion matrices omit important data by only representing the single top class predictions of a model, rather than its individual classification scores and error severities.

An example for such an alternative is *Squares* [RAL+17]. It represents class predictions as parallel coordinates, with each class displayed as a color-coded column with its class name and optional summary statistics below. Prediction scores of individual instances are binned and shown as bars. Individual instances are represented by boxes, strips or stacks, with errors subsumed by the position and fill pattern of these boxes. The prediction scores of a single instance across all parallel coordinates are displayed on hover, and confusion between classes is shown with spark lines above each class axis. This rather complex representation aims to lower the barrier between performance analysis and debugging root causes of problems within a model.

In another approach, Alsallakh et al. [AHH+14] proposed the *Confusion Wheel*. Classification results for each class are divided into four sets: True positive, false positive, true negative and false negative classifications. The results within each of these groups are then binned and visualized as stacked histograms for each class, centered within circle sectors of the wheel. Individual sample groups are colored according to their classification results. Furthermore, chords of varying thickness between sectors depict class confusions. This visualization encodes more information than traditional confusion matrices, such as the amount of discrimination between classes. The chords between sectors represent class confusions less accurately than confusion matrices but are easier to read at a glance than raw numbers in separated matrix cells.

Although useful for their application domain and worth the consideration, these alternatives to confusion matrices have proven to be inviable for visualizing a model's prediction performance within Advis since they encode even more information than only prediction results. When confronted by datasets with 1,000 classes, the clutter within confusion matrices already became problematic and had to be improved. The aforementioned much more complex alternatives to confusion matrices would have been hopelessly overloaded with this amount of categories.

Therefore, a hierarchy-aware interactive confusion matrix was chosen to represent model predictions. Having retrieved the aforementioned category hierarchy, various display options were considered. The first approach involved only displaying direct children of a hierarchy level within the confusion matrix, with cell values aggregated from the bottom up. Clicking on a cell would recursively update the matrix to show the corresponding sub-hierarchy. This approach faced the problem of omitting misclassifications that fall completely out of the sub-hierarchy that is currently being displayed. To this extent, a variation was introduced that collapses deselected sub-hierarchies rather than hiding them, inspired by the Table Lens [RC94].

The second approach for the confusion matrix displayed all cells at once, color-coded and with dendrograms as row and column headers representing the hierarchy. However, since the hierarchy can be quite deep, these dendrograms can become cluttered.

The third approach – with basic interaction patterns inspired by Clustergrammer [FGR+17], a heat map visualization for high-dimensional biological data – proved the most successful. The whole confusion matrix is again displayed in its entirety as a color map without grid lines or cell values, and therefore not omitting any outliers or patterns like the first approach. In the row and column headers, two levels of the hierarchy are displayed. As the user zooms into the matrix, these headers move outwards, revealing lower levels of the hierarchy until the class level has been reached. Headers are also synchronized with the user panning the matrix. When zoomed in far enough, grid lines and cell values fade in. Outside of each header, a smaller rectangle contextualizes the current view by displaying the path through the hierarchy to get to the current level. This approach displays

an overview of all cells and offers a responsive, intuitive and seamless way for drilling down into sub-hierarchies. It can reveal interesting patterns, such as misclassification outliers or buckets like in the figures 6.4 and 6.10.

As the user explores the confusion matrix, the grid of input image thumbnails updates continuously, corresponding to the input samples present in the current viewport of the confusion matrix. Alternatively, the user can shift-click to create a custom selection rectangle. One of the items in the image list can then be selected as the new input image. The schematic in figure 4.2 outlines the process of selecting an input image using this interactive confusion matrix.

The central model analysis view allows access to two types of information. Firstly, users can explore the model's prediction on the selected input image and the average prediction on a set of distorted versions of the input image. These distorted versions are generated by varying ranged parameters, such as for example the amount of degrees an image should be rotated. These predictions are ordered by certainty and displayed side by side, allowing a quick comparison. If users are interested in the performance of the model for individual distorted versions, they can select one of them using the distorted image selection. This view shows a grid of thumbnails of distorted versions alongside the prediction certainty of the model for the input image's ground-truth class. For a better overview, this grid can be sorted by image index, model certainty, and all varied range parameters. Selecting one distorted version shows its predictions in the previous image prediction screen next to the predictions on the original version. This in-depth instance-level exploration can help users explore model predictions on perturbed input images, for example to find out at what point a model fails on a specific input sample outlier (R2).

The second and most important part of the model analysis view starts out with a network architecture chart. This node-link diagram displays the computation graph of the model. Since this graph can be cluttered with many implementation-specific nodes, it is by default simplified. Unimportant nodes are removed while the edge topology of neighboring nodes is preserved. Finally, the graph encodes information about the difference in activation of individual computational units between original and distorted input using color. To be able to compute these colors, a metric had to be found that quantifies activation differences. In a first try, the simple Euclidean distance turned out to be impractical because its values correlated heavily with the dimensions of the activation matrix. Since these dimensions are usually smaller for every subsequent operation in a deep network, this metric failed to reveal activation differences. Another metric turned out to be much more useful for this task: Cosine similarity is invariant to both matrix dimensions and magnitude. Coloring nodes using this metric can reveal interesting patterns and provide an insight into the impact of perturbed input images on internal representations (R2), as can be seen in the usage scenarios in chapter 6.

In the last part of the system, the user can dive even deeper into the activations by clicking on one of the nodes in the graph, thus revealing both its activations on the input image and its average activations on distorted versions of the input images. The specific input image and distortion that is used to evoke these activations are chosen using the aforementioned two selection elements. The activations themselves are displayed as a set of slices of the cubic activation matrix, with individual values encoded as greyscale pixels. Since these activation maps can be similar and difficult to compare in detail, the user is supported by a variety of comparison aids. Complementing a side-by-side view of both activation maps, users can drag a slider across an overlay of both maps, with original activations displayed to the left and distorted activations displayed to the right of the slider. Furthermore, another view can be used to adjust the opacity of both overlaid maps, crossfading between them. Finally, a fourth view calculates the pixel-wise difference of both activation maps

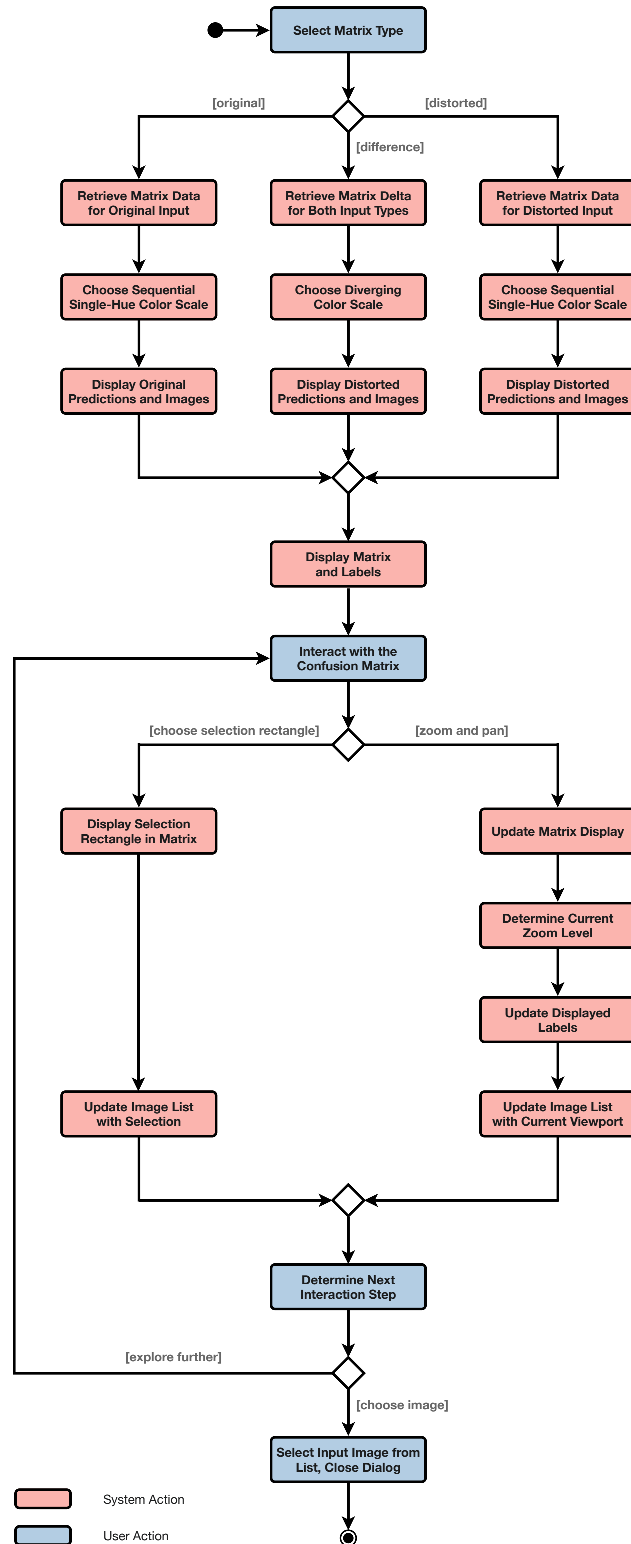


Figure 4.2: An activity diagram detailing how a user interacts with the confusion matrix to select an input image. The user stays in the loop at all times and can iteratively steer the exploration process towards the desired direction.

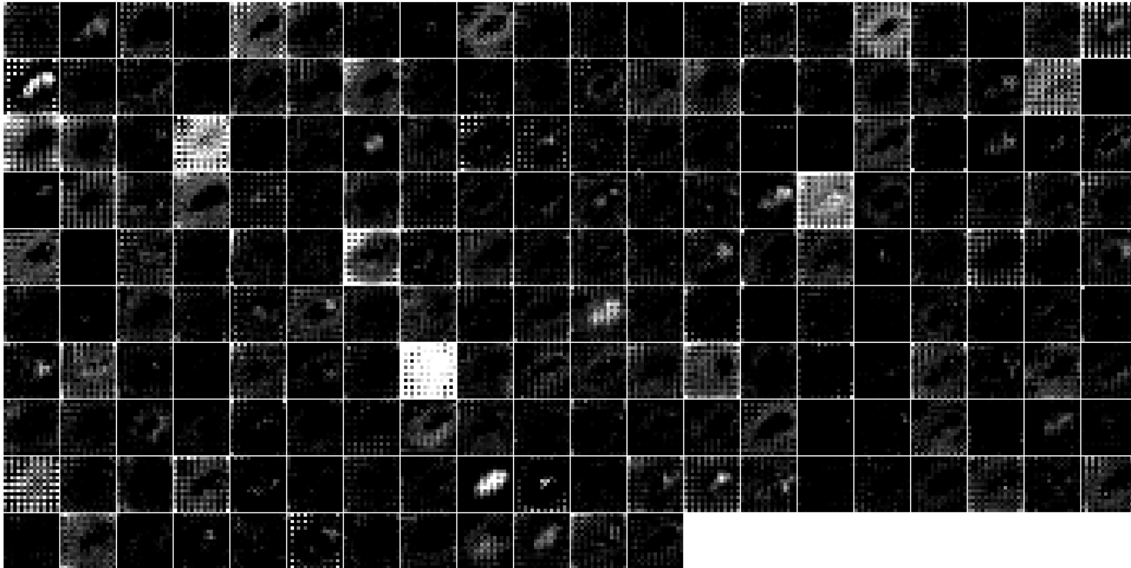


Figure 4.3: Checkerboard patterns in activation visualizations deep within the *Inception V3* model. This exemplary visualization depicts the activations of operation `InceptionV3/InceptionV3/Mixed_7a/Branch_1/Conv2d_0a_1x1/Conv2D`.

and displays it as a heat map. In all of these views, a user can click on a single tile they are interested in to open a zoomed-in version with additional information. This process enables a deep dive into how individual computational units react to input perturbations and how their responses change compared to original input images (R2). By doing so, it can reveal interesting patterns. For example, the activations of early layers in deep neural networks often act like edge detectors, while the information in later layers gets more and more abstract. On top of that, these activation maps can reveal flaws in network architectures. For example, the activation visualizations of the units of some model's layers stay completely black or white for all input images, suggesting that they might be dead and could be pruned. As another example, exploration has revealed that operations deep within the state-of-the-art *Inception V3* model exhibit checkerboard artifacts that can be seen in figure 4.3. These patterns might be caused by overlap within convolutional operations [ODO16]. They can be detrimental for the model's performance, lowering the amount of information passing through the layer. Avoiding these artifacts by appropriately adjusting the stride and size of responsible convolutional operations might be a way to improve the network.

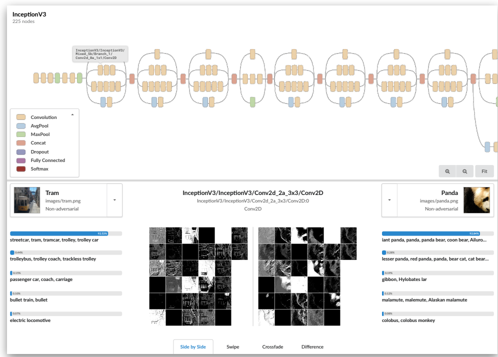
The interface described in this section has been implemented iteratively, with each iteration containing improvements and solutions for problems encountered in the last iteration. Before implementing the interface, mockups were created. An excerpt of the main screens within these mockups can be seen in figure 4.4, depicting the evolution of the interface's structure.

4.2.2 Principles

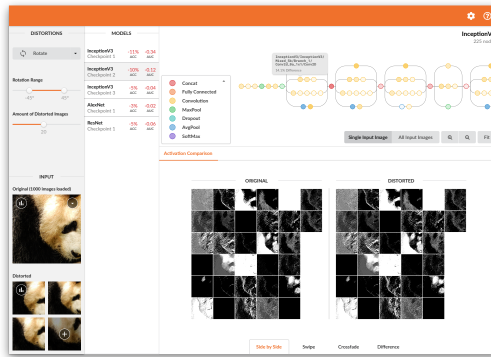
In order to implement this visualization system, the necessary infrastructure had to be designed (R6). An appropriate architecture had to include caching mechanisms in order to be able to quickly access the results of computations that have been executed before. Furthermore, splitting up the system

into a server and a client can speed up visualizations. The server can be equipped with powerful hardware and does the heavy lifting, while the client only needs to display the visualizations and handle user interactions. Implementing the client's frontend as a browser application simplifies access and enhances the range of potential clients further. Of course, both the server and the client can run on the same device if so desired. The system architecture will be described in chapter 5.

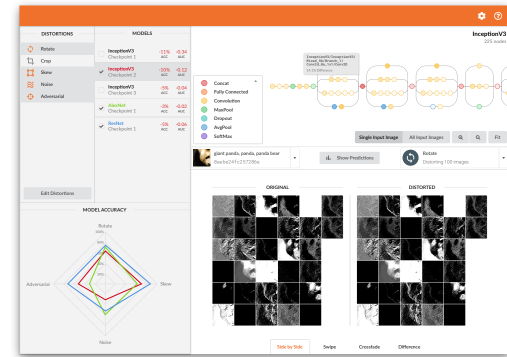
The frontend also had to adhere to a host of principles (R5). The UI had to always be responsive and communicate with the user. All computations are performed asynchronously, and while waiting on data from the server, visualizations and interface elements display progress bars or spinners. Due to the hierarchical nature of the visualizations described before, sometimes elements are empty if no selection has been made on a higher hierarchy level. In this case, empty state views describe why an element is empty and what the user has to do to fill it with content. High-density visualizations such as the confusion matrix have been specifically implemented in a way that makes use of high-resolution displays. Every dialog in the system always opens and closes with a zoom animation from and to the target element that caused its activation, intuitively informing the user about where the dialog belongs to. Finally, all elements of the UI are consistently styled, modular and obtain data through dynamic data binding, allowing easy and fast changes to the interface.



(a) February 2018: The first version of the interface already contained the basic structure of the central analysis view, with a network graph overview and an activation comparison view. This first version was completely standalone. There was no focus on input distortions and no way of comparing different models or viewing their performance metrics. Instead, users could only select one of a handful of precomputed input images and display a layer's activations on this input image. The five top predictions were shown inline.



(b) March 2018: This intermediate version assumed the design of TensorBoard [Goo17a] and introduced a sidebar. In the left section of this side bar, a single distortion can be selected and its parameters can be adjusted. Below the parameters, the current input image and a subset of its distorted versions is displayed. Clicking on either reveals more options, such as an input image selection dialog. The right part of the sidebar contains a list of all models along with some performance metrics, allowing a basic comparison. The activation visualization shows the activations of the selected computational unit for both the original input image and the distorted versions.



(c) June 2018: Two major changes took place in this final version. Rather than limiting the user to selecting a single distortion, a list now allows the selection of multiple distortions at once, with their average effects being visualized in the other views. Configuring distortions has been moved into its own dialog, accessible through a button in the distortion list. Furthermore, the input image selection has been moved out of the sidebar and on top of the activation visualization, where predictions can be accessed as well. The space that has been freed up in the side bar is used for a radar chart displaying the robustness profiles of individual models to the list of selected distortions. Models that should be displayed in this chart can be selected using checkboxes in the model list.

Figure 4.4: The evolution of the user interface of Advis, represented as mockups that were created before implementing the corresponding version.

5 Implementation

This chapter will introduce the implementation details of Advis as well as all important views within the system. Furthermore, it will explain how models, datasets, and distortions are being processed within the system.

5.1 Used Tools

Creating Advis would not have been possible without the support of numerous libraries and frameworks, applied throughout the whole application. Both the frontend and the backend are built and assembled using *Bazel*[1], a fast and scalable build system for a large variety of programming languages, and with many community-supported extensions.

An overview of tools used in the backend and frontend as well as their licenses and URL's can be seen in the tables 5.1 and 5.2 respectively.

5.1.1 Backend

The backend was developed using *Python 3*[2]. It relays Hypertext Transfer Protocol (HTTP) requests between the client and server using *Werkzeug*, a utility library for implementing Web Server Gateway Interface (WSGI) applications.

Models are being run, predictions retrieved and activation maps generated using *TensorFlow* [ABC+16]. This machine learning framework was chosen from the selection shown in table 2.1 due its popularity and its usage in many wrapper APIs. Furthermore, it offers the first-party visualization

Name	License	URL
TensorFlow	Apache License 2.0	https://www.tensorflow.org
Werkzeug	BSD-3-Clause	http://werkzeug.pocoo.org
NumPy	BSD-3-Clause	http://www.numpy.org
scikit-learn	BSD-3-Clause	http://scikit-learn.org
scikit-image	BSD-3-Clause	https://scikit-image.org
Pillow	PIL Software License	https://python-pillow.org

Table 5.1: Frameworks and libraries used in the backend of Advis.

[1] https://bazel.build

[2] https://www.python.org

Name	License	URL
TensorBoard	Apache License 2.0	https://github.com/tensorflow/tensorboard
Polymer	BSD-3-Clause	https://www.polymer-project.org
Polymer Elements	BSD-3-Clause	https://github.com/PolymerElements
Paper Range Slider	MIT License	https://github.com/IftachSadeh/paper-range-slider
Chart.js	MIT License	http://www.chartjs.org
Palette.js	Apache License 2.0	https://github.com/google/palette.js
Chroma.js	BSD-3-Clause	https://github.com/gka/chroma.js

Table 5.2: Frameworks and libraries used in the frontend of Advis.

framework *TensorBoard* [Goo17a] which provides a flexible plugin system. On top of that, users can use one of many available conversion tools if their models are present in a non-TensorFlow format. TensorFlow allows users to create and run static computation graphs. The multi-dimensional matrices flowing through the operations of these graphs are called tensors within the framework.

For various matrix operations and calculations outside of TensorFlow, *NumPy* is used. Furthermore, parts of the machine learning library *scikit-learn* [PVG+11] are put to use in order to calculate the cosine similarity of activations and model performance metrics such as F1 score, precision, and recall.

Two other libraries ease the task of handling and processing input images. *Pillow* is employed for converting pixel arrays into valid image data, for overlaying and blending a set of images, and for compositing individual activation visualization slices into larger activation maps. Using *scikit-image*, input images are loaded from disk and preprocessed for feeding them into a model. On top of that, it is used within the implementations of preset distortions, perturbing input images.

5.1.2 Frontend

The frontend's code was developed using a mixture of three languages. The interface's structure was described using Hypertext Markup Language (HTML) and styled via *Sass*[3], an extension to Cascading Style Sheets (CSS) with syntactical sugar such as variables and nesting. While building, it cross-compiles to plain CSS. The interface's scripting was done using *TypeScript*[4], which is a statically typed superset of JavaScript that cross-compiles to standard JavaScript.

The frontend is based upon the infrastructure of *TensorBoard* [Goo17a], a framework for visualizing various kinds of data a TensorFlow model can output during its training process. TensorBoard requires users to programmatically add so-called summary operations to their computation graph. These write data to a logging directory that can be read and visualized afterwards by TensorBoard.

[3] https://sass-lang.com

[4] https://www.typescriptlang.org

This infrastructure was not extensible enough for creating the dynamic model visualizations Advis offers. Instead, Advis builds on top of TensorBoard and runs models and distortions while the visualization is active, allowing users to configure distortions within the frontend.

The UI is implemented using *Polymer*. Every part of the UI is a custom module, often containing even more nested modules. These modules encapsulate their own Document Object Model (DOM), a style sheet and TypeScript code. The latter includes a set of observable properties, functions, and observers. Communicating properties between modules and from code into the DOM is achieved using data binding, either one-way or two-way. Observing these design patterns leads to an interface that is modular, easily modifiable and extensible, containing properly encapsulated data.

Standard UI elements in a consistent design language have been retrieved from the *Polymer Elements* repository. Since this repository was missing a range slider with two knobs for specifying minimum and maximum values, this component was retrieved from another open source repository[5].

Multiple JavaScript libraries are used for single visualizations within the system. *Chart.js* is used to display the dynamic radar chart. Its data interface has been wrapped and the component itself has been embedded in a Polymer component. *Palette.js* is used to create high-quality color palettes, of which each model is assigned a color. Finally, color scales for heat maps are created using *Chroma.js*.

5.2 System Architecture

The system's architecture is split into two parts, a server and a client. An overview of how these parts are connected can be seen in figure 5.1.

Advis is implemented as a highly customized plugin for TensorBoard. When the system is started, a `logdir` is specified using a command line parameter. This directory contains all user data, consisting of models, datasets, and distortions. The system launches a server that reads this data and sets up a set of routers. Each of these routers has a specific set of features, such as retrieving an image from a dataset, running a model to predict the class of an input image or higher-level operations like calculating model performance metrics, comparing activation matrices or creating confusion matrices. A cache component stores all kinds computation results for faster access at a later time. The contents of this cache are persisted to disk periodically in an asynchronous manner if not specified otherwise. A special cache router allows eagerly caching all important data at once to make computations during the visualization nearly immediate.

These routers offer their services through an HTTP Application Programming Interface (API), returning data in JavaScript Object Notation (JSON). Due to this decoupled architecture, clients can use the information retrieved through the backend in any way they desire. The routes provided by the backend are documented in section A.1.

The frontend accesses visualization data by querying these routes. The UI itself consists of a large array of nested decoupled components as seen in the schematic in figure 5.1, each of them containing their own build target, HTML content description, style sheet, and TypeScript code.

[5] `https://github.com/IftachSadeh/paper-range-slider`

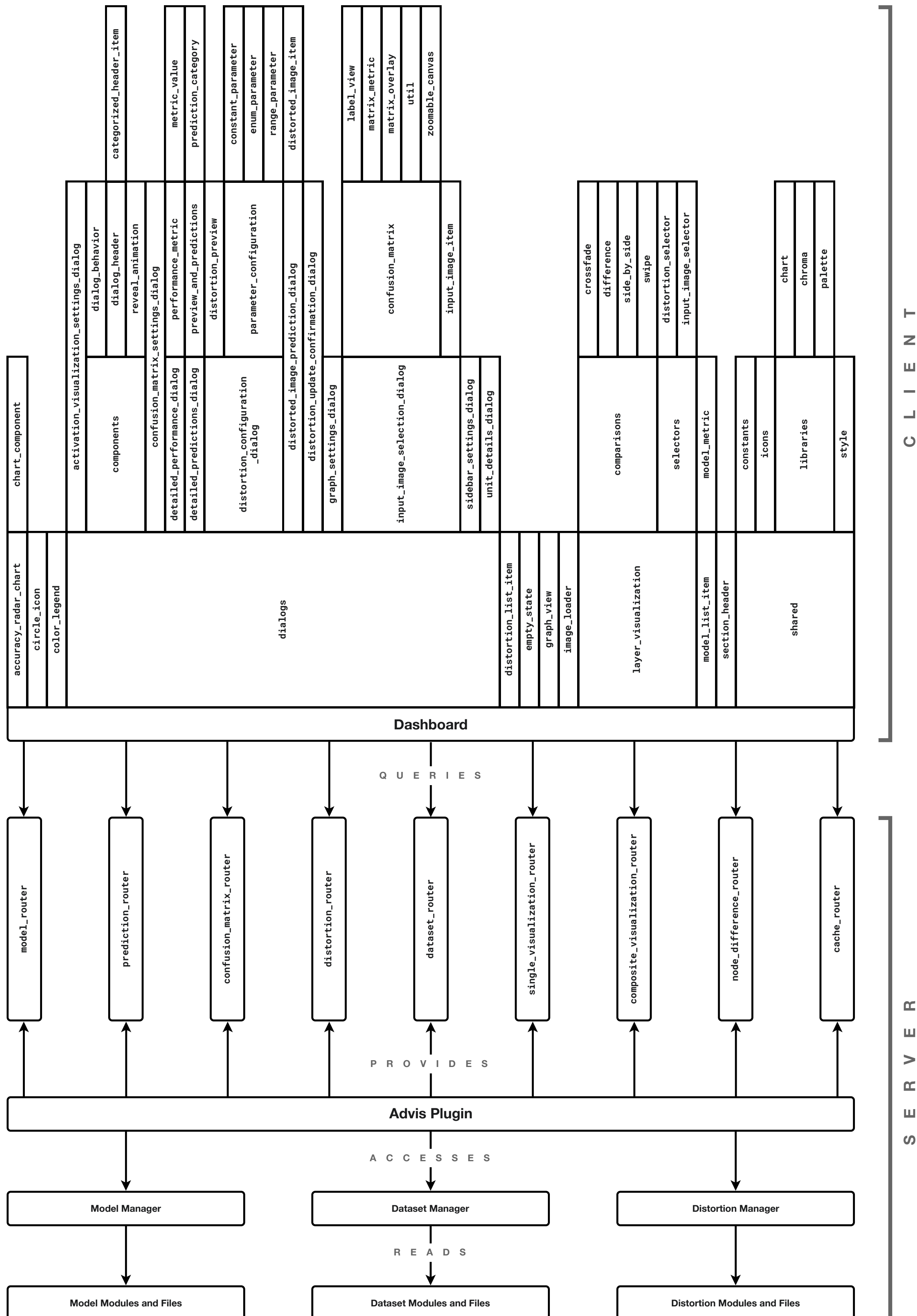


Figure 5.1: A high-level view of the system architecture of Advis.

Special modules include a set of shared configuration constants, icons, styles, libraries, and a highly customized graph view component, originally proposed by [WSW+18], that allows encoding further information using node color scales.

5.3 Data Processing

The data processing within the system's architecture had to be carefully considered in order to achieve the goal of enabling users to apply Advis for all kinds of tasks involving transforming input data for DL models. Models, distortions, and datasets available within the system had to be flexible and extensible. A straight-forward way of allowing this kind of flexibility was by letting users define their own elements using Python modules that are then read and evaluated by the system. With this approach, users can rely on any library, data base or code snippet they desire.

Upon starting the system, the plugin creates managers for models, datasets, and distortions. These managers read and instantiate all modules within the working directory, making them accessible to be used for creating visualizations. Furthermore, Advis delivers some preset modules when started on an empty working directory. These include a set of state-of-the-art models with popular evaluation datasets, as well as some image distortions. Models and datasets have been reformatted to be compatible with Advis using the external demo data repository, with models being initialized using the *Slim* infrastructure [GS16]. These presets can be used as a template for custom modules, for demonstrating the system, and for initial exploration.

The structure of each individual module type and the presets included within Advis will be described hereafter.

5.3.1 Models

The main data of models is contained within their trained checkpoint, which has to be available in the standard TensorFlow format. This model is then loaded and evaluated. Furthermore, every node within the computation graph whose activations should be visualized is annotated with a set of tensor transformation operations, creating image slices from activation tensors. After a model has been loaded and annotated for the first time, the final computation graph is cached to speed up later system starts.

Models are Python modules containing the following functions:

- `get_display_name()`: Returns a human-readable name of the model.
- `get_version()`: Returns a version number of the model. This has no actual effect on the visualization but can be used to distinguish between different iterations of the same model.

Name	Description
VGG 19	Very deep network proposed by Simonyan et al. [SZ14], trained checkpoint retrieved from http://download.tensorflow.org/models/vgg_19_2016_08_28.tar.gz.
Inception V1	Modular convolutional network proposed by Szegedy et al. [SLJ+15], trained checkpoint retrieved from http://download.tensorflow.org/models/inception_v1_2016_08_28.tar.gz.
Inception 5h	Inception model optimized for inference, trained checkpoint retrieved from https://storage.googleapis.com/download.tensorflow.org/models/inception5h.zip.
Inception V3	Improved Inception architecture proposed by Szegedy et al. [SVI+16], trained checkpoint retrieved from http://download.tensorflow.org/models/inception_v3_2016_08_28.tar.gz.
ResNet V2	Residual network proposed by He et al. [HZRS16b], trained checkpoint retrieved from http://download.tensorflow.org/models/resnet_v2_101_2017_04_14.tar.gz.
MobileNet V1	A light-weight network proposed by Howard et al. [HZC+17], trained checkpoint retrieved from http://download.tensorflow.org/models/mobilenet_v1_2018_02_22/mobilenet_v1_1.0_224.tgz.

Table 5.3: Models that are included as preset modules. For an overview of the performance of these models, please refer to table 5.4.

- `get_checkpoint_directory()`: Returns the directory where the model's checkpoint has been saved using the standard `tf.train.Saver`[6]. The input has to be a floating point placeholder with the shape `[image_size, image_size, 3]`. The output of this function has to be a dictionary with the elements `type` (which should be `custom` for non-preset models) and `directory` which points to the checkpoint.
- `get_dataset()`: Returns the identifier of a dataset that this model works with.
- `get_input_image_size()`: Returns the size of quadratic input images as an integer. Has to match the input placeholder's shape.
- `get_input_node()`: Returns the name of the node that the input image should be fed into.
- `get_output_node()`: Returns the name of the node that contains the model's final classification prediction.
- `annotate_node(node)`: Should return `True` if and only if the given node is of interest and can be visualized. Most of the time, this includes nodes with operations such as `Conv2D`, `Relu`, `MaxPool`, `AvgPool` or `ConcatV2`.

[6] https://www.tensorflow.org/api_docs/python/tf/train/Saver

Name	Proposal	Accuracy Top-1	Accuracy Top-5	Precision	Recall	F1 Score
VGG 19	[SZ14]	68.11%	87.86%	68.69%	68.11%	68.69%
Inception V1	[SLJ+15]	67.16%	87.72%	68.02%	67.16%	66.71%
Inception 5h	N/A	66.70%	87.37%	68.09%	66.70%	66.40%
Inception V3	[SVI+16]	76.02%	92.85%	76.72%	76.02%	75.71%
ResNet V2	[HZRS16b]	70.53%	89.58%	70.88%	70.53%	70.05%
MobileNet V1	[HZC+17]	66.36%	86.93%	68.38%	66.36%	66.07%

Table 5.4: Models included in Advis, ordered by date of proposal or availability. The values of the reported performance metrics have been calculated using Advis by evaluating all models on all 50,000 images from the ILSVRC validation set [RDS+15], cropped to their biggest possible central square and then resized to fit the input dimensions of each respective network.

The following six exemplary CNN's for image classification are included within the presets of Advis and ready for analysis. They are listed in table 5.3. All trained networks have been retrieved from [GS16] with the exception of *Inception 5h*, whose trained configuration has been retrieved from [Ten15]. An overview along with performance metrics can be found in table 5.4. The following section will outline these model's architectures and peculiarities.

VGG 19

The VGG architecture [SZ14] is characterized by an increased network depth with more convolutional layers. This depth is made feasible by only using small convolution filters with a receptive field of up to 3×3. Its architecture includes a stack of convolutional layers with these small filters and *Max Pooling*. These convolution layers are followed by three fully-connected layers, with the last one containing an amount of channels that equals the desired amount of classes to be categorized. The final prediction is created by the following *Soft Max* layer. All hidden layers also contain ReLU's.

This network is available in five slightly different configurations, with the last one containing the most weight layers. This configuration with 19 weight layers is used in Advis.

VGG's architecture is appealingly straight-forward and achieves a good performance, but this comes at the price of computationally expensive evaluations.

Inception V1

The first iteration of the Inception architecture [SLJ+15] aims to increase the network's depth and width while keeping the computational budget constant, using readily available dense convolutional components.

Its architecture is made up of stacked Inception modules. These consist of parallel 1×1, 3×3, and 5×5 convolutions, each using rectified linear activation and dimensionality reduction where needed in order to reduce computational load, as well as *Max Pooling* layers. After each module, the outputs of all parallel branches are concatenated. Occasional *Max Pooling* layers between stacks halve the grid resolution. For better memory efficiency during training, these modules are only used at layers further down the network's depth, with earlier layers using a standard convolutional setup.

Szegedy et al. [SLJ+15] create a particular instance of this architecture and dub it *GoogLeNet*, which is equivalent to what is called *Inception V1* within this thesis. It adopts the architecture described before and ads an *Average Pooling* layer before the classifier. Moreover, auxiliary classifiers, made up of smaller convolutional networks on top of the output of Inception modules, are positioned in the middle of the network. Their output is considered during loss calculation and used to combat problems of effectively back-propagating gradients through all layers of this deep network, improving convergence during training.

Inception 5h

The *Inception 5h* model has nearly no documentation but its topology is very similar to *Inception V1*. It has been optimized for inference, evaluates input images fast and its saved training data is smaller. In turn, it trades these improvements for a slightly worse accuracy.

This model has been included in Advis because it has been used in the code accompanying the proposal for creating universal perturbations [MFFF17]. Therefore, it is of interest to find out how it performs on adversarial input data and how this performance compares to that of *Inception V1*.

Inception V3

The *Inception V3* model [SVI+16] aims to improve the performance of the first iteration described above while keeping the computational cost low. It does so by improving some of the original topologies. Most notably, expensive convolutions with larger patch sizes are replaced by multiple chained smaller ones. For example, a 5×5 convolution can be replaced by two 3×3 convolutions. The computational cost freed up by using this technique can be invested into larger filter banks. Furthermore, the original architecture used convolutions with a stride of 1 before pooling which again is computationally expensive. *Inception V3* introduces a technique to reduce grid sizes more efficiently.

Moreover, so-called “label-smoothing regularization” [SVI+16] is introduced. By estimating the marginalized effect of label dropout during training, the classifier layer can be regularized.

Finally, the fully connected layer of the auxiliary classifier is also batch-normalized, instead of just the convolutions.

ResNet V2

When increasingly deep networks start converging, their accuracy can get saturated and start degrading rapidly. To solve this problem, deep residual networks [HZRS16a] were proposed that explicitly let layers fit a residual mapping by introducing so-called shortcut connections that perform identity mappings, skipping over one or more layers. With this idea in mind, residual units can be constructed that allow input to either flow through a weight layer, a batch-normalization layer, a ReLU, another weight layer and batch-normalization layer and then into an element-wise addition operation and from there on through another ReLU to the output, or the input can skip the previous layers and flow directly into the addition operation. By stacking these residual units a deep residual network can be constructed.

The included *ResNet V2* [HZRS16b] improves upon this idea by establishing a direct path for propagating information, both forward and backward, through the entire network instead of only within the residual units. This goal is achieved by replacing the final ReLU function with an identity function and altering the order of operations within the unit. In the proposed unit, input can either flow through a batch-normalization layer, a ReLU, a weight layer, another batch-normalization layer, ReLU, and weight layer into an element-wise addition operation, or skip these steps and flow directly to the addition operation. In this order, batch-normalization and ReLU can be understood as the pre-activation of the weight layers instead of post-activation as in the original architecture.

This new architecture has two advantages: Optimization becomes easier and batch-normalization improves the regularization of the model, reducing overfitting.

MobileNet V1

The main goal of *MobileNet V1* [HZC+17] is to build deep but light-weight neural networks that can be used in mobile applications. Standard convolution operations filter features based on convolutional kernels and then combine them to create a new representation. However, these filtering and combination steps can be separated. By factorizing such an expensive standard convolution into a depth-wise convolution and a 1×1 point-wise convolution, computation and model size can be reduced.

The first layer of the architecture of *MobileNet V1* is fully convolutional. It is followed by multiple layers that make use of the described separated convolutions. Finally, an *Average Pooling* layer reduces the spatial resolution to 1, feeding into a fully-connected layer which in turn outputs its classification through a *Soft Max* layer. All of these layers, with the exception of the final fully-connected and output layer, are followed by a batch-normalization layer and a ReLU unit.

On top of that, the architecture defines two hyper-parameters that can be set to trade off latency and accuracy while instantiating a network: The width multiplier α thins the network uniformly at each layer and the resolution multiplier ρ implicitly reduces the internal representation's resolutions by setting the resolution of the input image. The model included in Advis uses the base configuration that offers the best performance, with $\alpha = \rho = 1$.

Name	Description
NIPS 2017	1,000 input images[7] from the NIPS 2017 Adversarial Learning Challenges [KGB+18].
ILSVRC 2012	50,000 input images[8] from the ImageNet Large Scale Visual Recognition Challenge 2012 [RDS+15].

Table 5.5: Datasets that are included as preset modules. Every image in both datasets is annotated with one of the 1,000 ImageNet [DDS+09] categories. The category hierarchy has been reconstructed from the WordNet [Mil95] corpus, using the Natural Language Toolkit[9].

5.3.2 Datasets

Datasets mainly consist of a list of labeled input images, a list of all classes as well as a nested JSON tree representing a hierarchy of all classes to be used for confusion matrices. All included preset datasets are listed in table 5.5. They are represented by Python modules containing the following functions:

- `get_display_name()`: Returns a human-readable name of the dataset.
- `get_categories()`: Returns an ordered list of all available classification categories, with list indices corresponding to the number of each respective category. Usually, this list is loaded from a separate JSON file.
- `get_category_hierarchy()`: Returns a user-defined category hierarchy which will be used to display confusion matrices in a manageable way even if a lot of categories are present. This hierarchy is given as a tree of nested JSON objects, each with a `name` and a list of `children` objects. Leaf nodes should correspond to actual categories and therefore have no `children` but a `category` index. Usually, this hierarchy is loaded from a separate JSON file.
- `get_all_images()`: Returns a list of all images in the dataset. Each image is described by a dictionary with the elements `id` (a unique identifier), `index` (the image's index within the dataset), `path` (the absolute path to this image), `categoryId` (the ground-truth category number), and `categoryName` (the name of the ground-truth category). Again, most of this information is usually loaded from a separate JSON file and enriched with absolute file paths within the script.

5.3.3 Distortions

Distortions receive an image and map it to an output image. They define a set of parameters that allow configuring the distortion. For each distortion step, they are passed a configuration containing current parameter values. This enables the creation of multiple distorted version of a single input image if ranged parameters are available. All included preset distortions are listed in table 5.6.

[7] https://www.kaggle.com/google-brain/nips-2017-adversarial-learning-development-set
[8] http://www.image-net.org/challenges/LSVRC/2012/index
[9] http://www.nltk.org

Name	Description
Adversarial	Add a pre-computed universal adversarial perturbation [MFFF17] to the input image.
Crop	Crop the input image according to a zoom factor and vertical and horizontal position range.
Noise	Add a configurable amount of random noise to an image.
Rotate	Rotate an image according to a configurable rotation range, symmetrically filling in visible background parts.
Skew	Skew an image by a configurable ranged amount, symmetrically filling in visible background parts.
Style Transfer	Apply a painting style to an image with the fast style transfer technique proposed by Johnson et al. [JAF16] and Ulyanov et al. [UVL16]. The network performing the style transfer has been trained using the also included VGG 19 model. Model checkpoints have been retrieved from `https://github.com/hwalsuklee/tensorflow-fast-style-transfer` and include "La Muse" by Pablo Picasso, "Rain Princess" by Leonid Afremov, "The Shipwreck" by J. M. W. Turner, "The Scream" by Edvard Munch, "Udnie" by Francis Picabia, and "The Great Wave off Kanagawa" by Katsushika Hokusai.

Table 5.6: Distortions that are included as preset modules.

Distortions are Python modules containing the following functions:

- `get_display_name()`: Returns a human-readable name of the distortion.
- `get_parameters()`: Returns a list of dictionaries, each describing a parameter. Parameters have to define a unique `name`, a human-readable `display_name`, a `default` value, and a `type`. Parameter types include:
  - `constant`: A numeric value that will stay the same for each distorted image. The parameter has to also define minimum and maximum values as `constraints`.
  - `range`: A range (`lower` and `upper` bounds) from which a numeric value will be sampled. The parameter has to also define minimum and maximum values as `constraints`.
  - `enum`: An enumeration of which the chosen option will be used for each distorted image. The parameter has to also define a list of `options`.
- `distort(image, configuration)`: Returns a distorted version of the input image which is supplied as a NumPy array of shape `[width, height, 3]` with values between 0 and 1. Any library or operation desired by the user can be used here. Current values of the previously defined parameters are supplied via the `configuration` dictionary, which maps parameter names to their values.
- `get_icon()`: Returns a Scalable Vector Graphics (SVG) path as a string which will be used as the distortion's icon within the UI. Icons should be quadratic with all coordinates between 0 and 24 pixels. This function is optional. If it is not defined, a generic placeholder icon will be used.

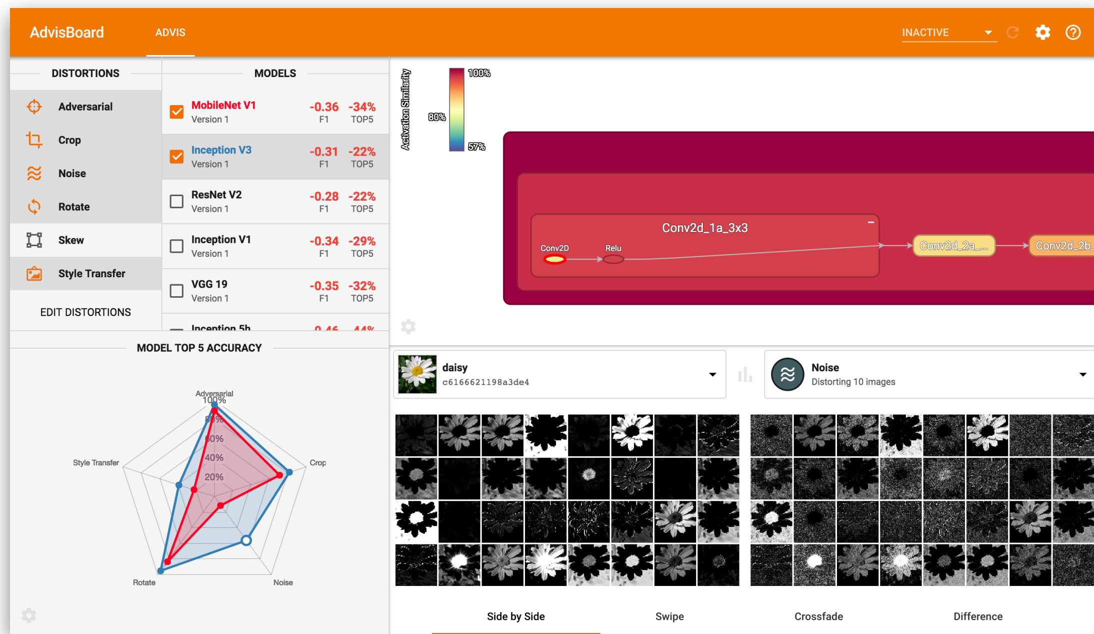


Figure 5.2: The main screen of Advis, with the sidebar to the left and the main view in the center, divided into the network graph at the top and the activation visualization at the bottom.

5.4 Views

When the system has been started and its address has been opened in a browser, a web interface displays a variety of visualizations. An overview of the main screen can be seen in figure 5.2. It is divided into three main views, a sidebar to the left, a network graph in the top right, and an activation visualization in the bottom right. More visualization views can be accessed via buttons in the interface. The individual parts of the interface will be described hereafter.

5.4.1 Sidebar

The sidebar has a light-gray background and can be seen in the left part of figure 5.2. It is the main starting point of the visualization. The top part of the sidebar contains two lists, one with all distortion modules and one with all models. The distortion list shows all available distortion modules and their icons. The user can select a subset of these distortions that will be used in later visualizations. An edit button at the bottom opens the distortion configuration dialog described in section 5.4.7.

The model list shows all models and their versions. Every model displays the average change of their performance metrics between original input images and input images perturbed by the set of selected distortions. Clicking on the metrics of one of these models opens a dialog with detailed model performance metrics, described in section 5.4.6. Clicking on a model list item itself makes the network graph display the model's architecture.

Model items can be toggled using checkboxes to their left. Active model items are assigned a color and displayed in the radar chart at the bottom. In this radar chart, model's change profiles between original and distorted input data for a specific metric are displayed in their respective color. Detailed information is available when hovering over a node. The node corresponding to the currently selected distortion for the activation visualization described in section 5.4.3 is highlighted.

Finally, the cog at the bottom left reveals a settings dialog for configuring the sidebar. Users can select the performance metrics that should be displayed in the model list and the metric whose changes should be represented in the radar chart.

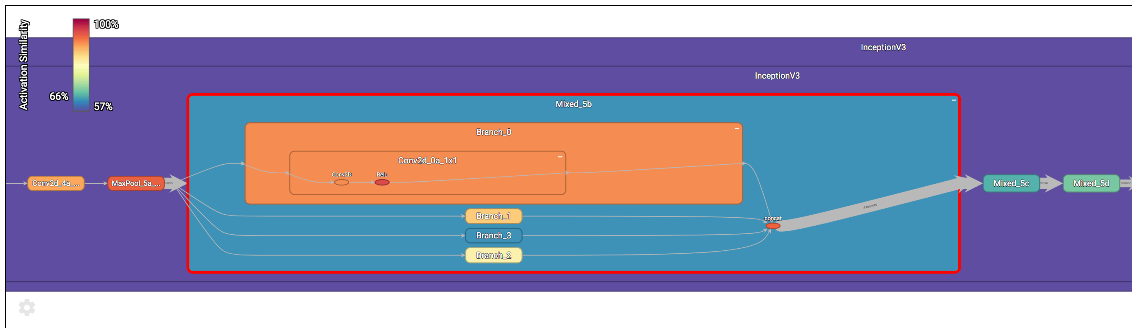
5.4.2 Network Graph

The network graph displays the computation graph of the model that has been selected. For a better overview, nodes are grouped in multiple nested levels of sub-groups. Users can expand sub-groups by double-clicking on them.

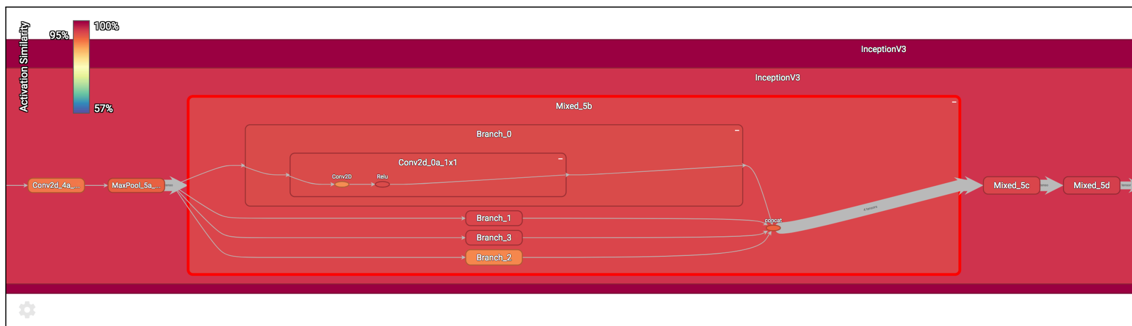
Most computation graphs contain lots of clutter in the form of implementation-specific nodes, such as nodes dedicated to loading input images or trained values. They do not have activations that can be visualized. For this reason, their activation similarity between original and distorted input also cannot be computed. They would appear grey in the network graph. By default, this clutter is combated by simplifying the graph. All nodes that have no activation visualization annotations attached as specified in section 5.3.1 are iteratively removed while the system is being launched. The edge topology is retained during this process. If a node from this directed computation graph is removed, edges between its input nodes and its output nodes are inserted. This simplification is performed on the *Protobuffer* message containing the graph structure and is cached for later launches. Within the UI, the user can switch between the simplified graph and the full graph. Nodes that are not annotated appear grey in the full graph, and node value aggregation only considers child nodes that actually have values.

Nodes are colored according to their similarity in activation between normal and distorted input images. Each leaf node within the simplified hierarchical computation graph that is displayed in the network graph view corresponds to a computation operation that has been outfitted with further operations that extract their activation tensors. This activation tensor extraction is the first step for further operations transforming the activations into greyscale images for the activation visualization in section 5.4.3.

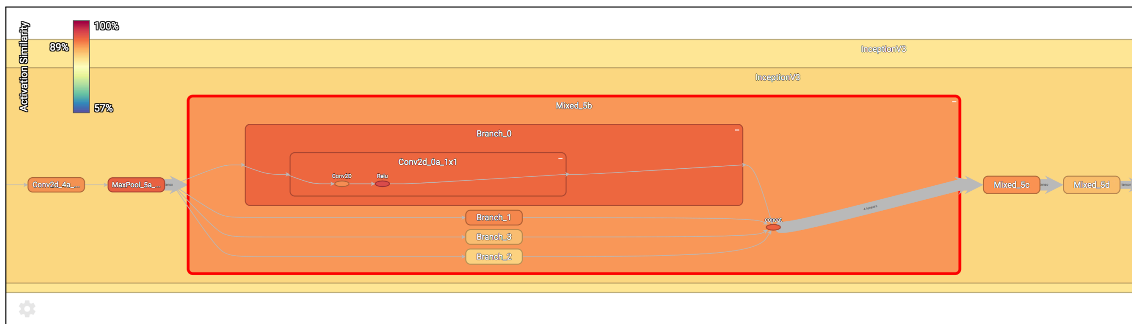
The computation graph can be executed within TensorFlow in such a way that evaluates the activation extraction operations for a specific input image. In order to calculate node similarity values, the system retrieves a configurable amount of input images from the dataset associated with the model that is displayed in the network graph. It then iterates through all of these input images and through all distortions. The system extracts the activation of all nodes for all original input images and all distorted versions of these images. For each pair of an original input image and its distorted version, the cosine similarity of their extracted node activations is computed. This yields a value between -1 and 1 that is invariant to the size and magnitude of activation tensors. Instead, it quantifies insightful activation differences. This value is then normalized to be between 0 and 1 and multiplied by 100 to reflect a similarity percentage. Low values represent large activation differences between original and distorted image versions, while high values represent high similarities. By default,



(a) Minimum Aggregation



(b) Maximum Aggregation

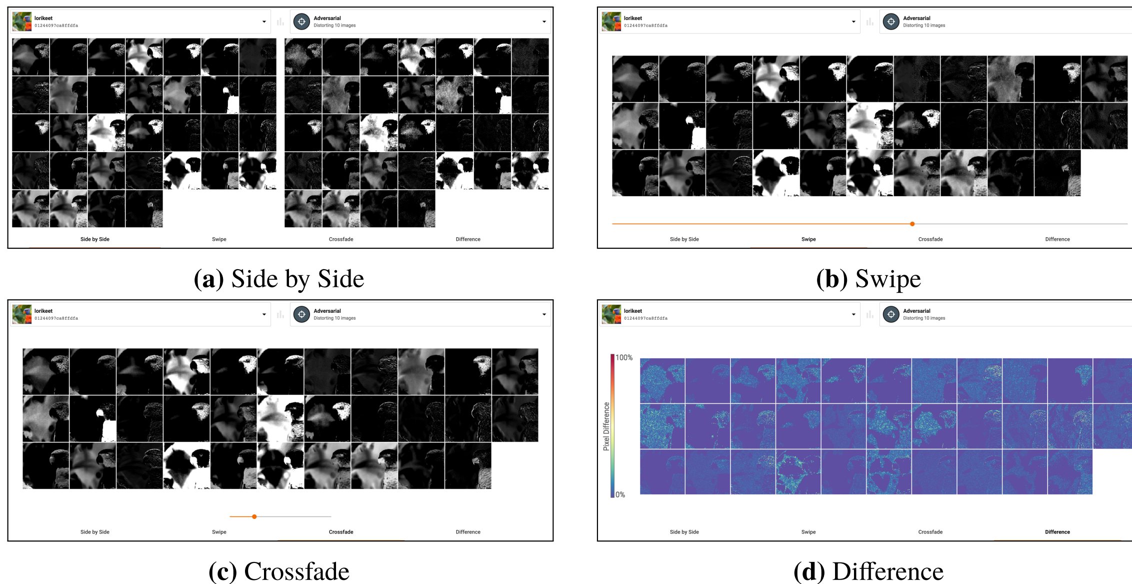


(c) Average Aggregation

Figure 5.3: Different aggregation options for the node activation similarity visualization. The legend in the top left displays the range of activation similarity values to the right and the value of the currently selected node to the left. Different aggregation methods can be preferable for different tasks.

this operation is repeated for all selected distortions and a configured amount of images within the dataset. The activation similarities for each leaf node are then averaged and used to color the node using a configurable color scale.

The color of each parent node is recursively aggregated from all of their child nodes. The legend in the top left displays the color scale, with the present range of minimum and maximum similarity values to the right, and the similarity value of the currently selected node to the left. Clicking on a leaf node displays its activations in the activation visualization at the bottom as described in section 5.4.3.



(a) Side by Side

(b) Swipe

(c) Crossfade

(d) Difference

Figure 5.4: Different comparison views for activation visualizations.

By clicking the cog in the bottom left of the network graph, a settings dialog can be opened that allows the user to configure the view. They can switch between displaying the simplified graph or the original one and specify a color scale for node activation differences as well as an aggregation method for calculating activation difference values for parent nodes. Possible options are the minimum of all child values, the maximum or the average. These options determine how similarity values for higher-level nodes in the graph representation should be aggregated from child nodes. Their effect can be seen in figure 5.3.

Furthermore, users can specify whether a configured amount of input images in the dataset or only the single selected input image should be used for calculating activation differences. The former allows insights about the average activation similarities for the whole dataset, while the latter focusses on a single sample. Similarly, users can select whether all active distortions from the distortion list or only the selected one should be used for this calculation. For the former option, activation similarities for all distortions selected in the distortion list will be averaged. Finally, users can toggle the visibility of the color scale legend, of a box with detailed node information upon selection, and of a mini map of the graph in the bottom right.

5.4.3 Activation Visualization

The activation visualization view in figure 5.4 displays the selected layer's activations for both the original input image and the average activations for all of its perturbed versions. The input image and distortion that should be used for the visualization can be selected using the two dropdown menus above the visualization. Individual tiles are sized and arranged in a way that is optimal for the available screen space. Clicking on a tile opens a larger view of it in a new dialog, along with some additional information. Within this larger view, a single tile can be compared using the same four comparison views that are used for activation maps.

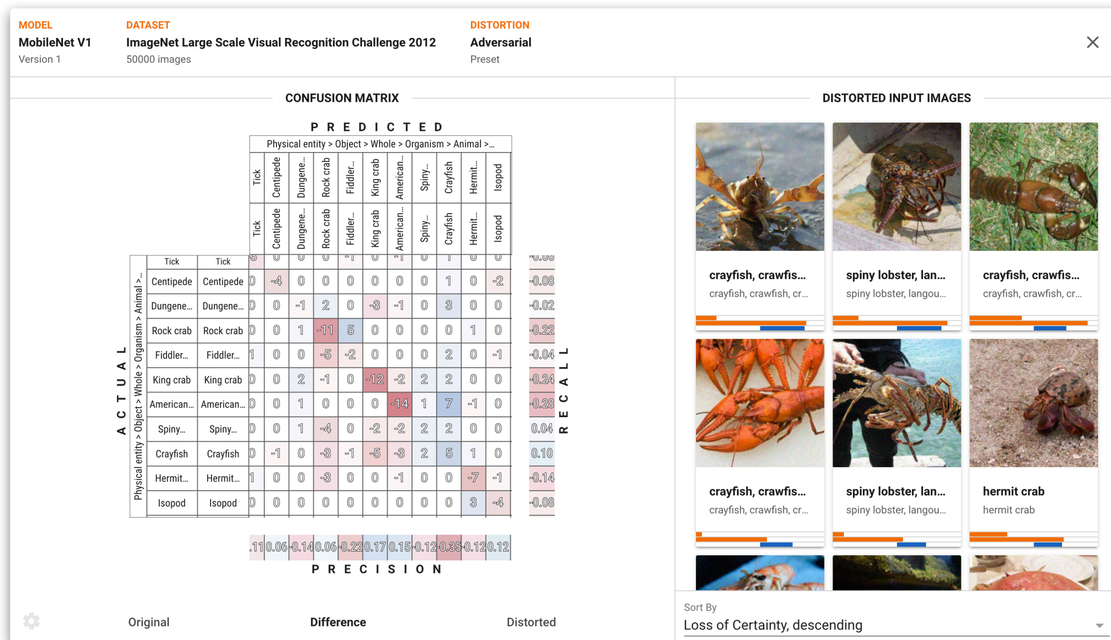


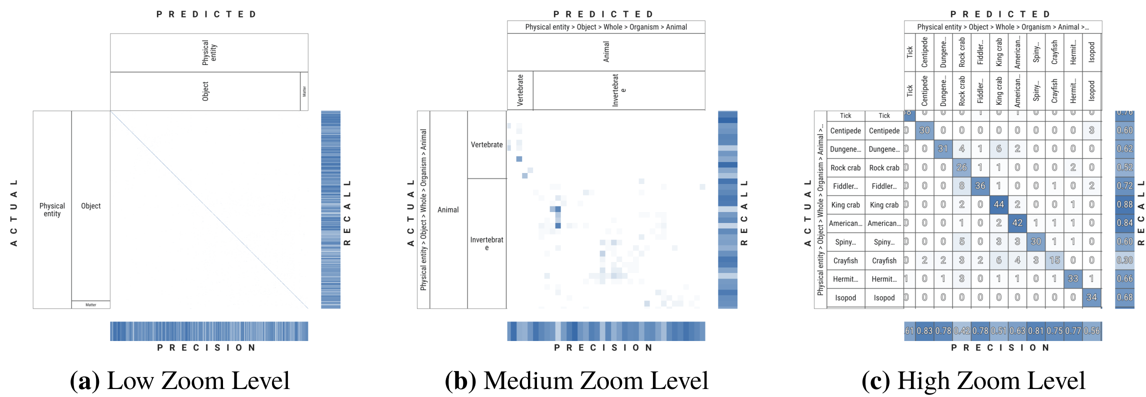
Figure 5.5: A dialog for selecting an input image using an enhanced confusion matrix.

Using the tabs at the bottom, one of four comparison views can be selected to ease the exploration of differences between the two activation maps. The side by side view in figure 5.4a simply displays both maps next to each other. The swipe view in figure 5.4b overlays both visualizations and lets the user control the visibility of them by moving a slider. Parts of the overlay left of the slider will display the original activation map, parts to the right will display the distorted activation map. Similarly, the crossfade comparison in figure 5.4c employs a slider that fades between both activation maps. Finally, the difference comparison in figure 5.4d calculates the per-pixel difference of both maps and displays them as a heat map, with its color scale to the left of it. The color scale that should be used can be selected by clicking on the legend.

5.4.4 Input Image Selection

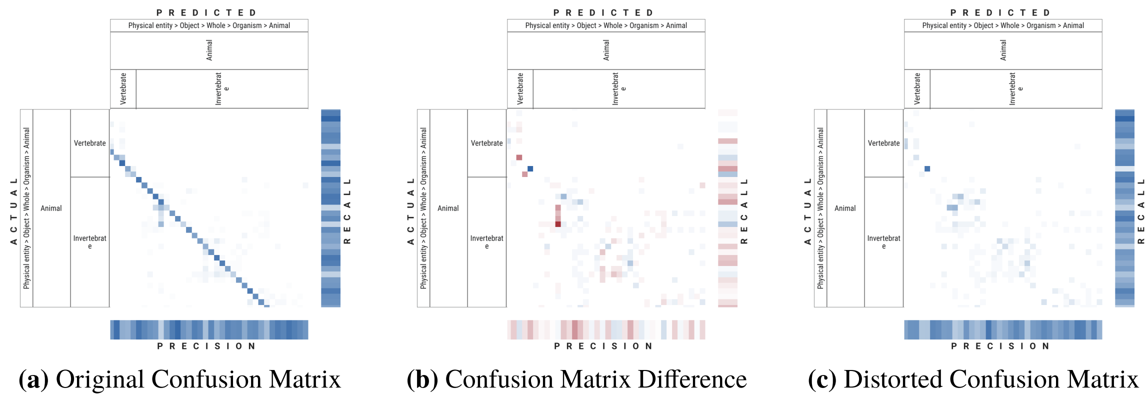
The input image selection dialog as seen in figure 5.5 enables users to make an informed decision on which one of the often large amount of input images to select.

This goal is achieved by employing a confusion matrix. When zoomed out, it acts as an overview and displays a heat map of its cells. To explore the matrix, the user can zoom in and pan, while the column and row headers fluently and interactively display the class hierarchy as an icicle plot [KL83], down to individual categories. As the user zooms in, the labels to the left and on the top of the matrix seamlessly move outwards, revealing labels further down in the class hierarchy. The outer rectangle of both column and row headers contextualize the current viewport by displaying the path through the hierarchy to get there. When zoomed in far enough, grid lines and individual cell values appear. The confusion matrix at multiple zoom levels can be seen in figure 5.6. Hovering



(a) Low Zoom Level **(b)** Medium Zoom Level **(c)** High Zoom Level

Figure 5.6: The interactive confusion matrix at different zoom levels. As the user zooms in, the icicle plot representing the class hierarchy moves outwards and becomes more detailed. At high zoom levels, grid lines and individual cell values fade in. Best viewed in electronic format.

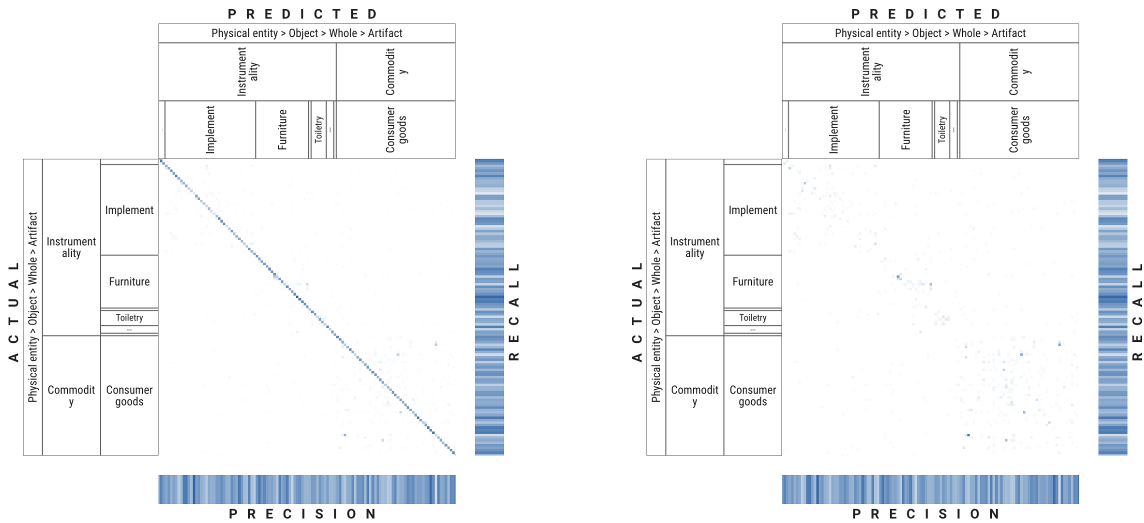


(a) Original Confusion Matrix **(b)** Confusion Matrix Difference **(c)** Distorted Confusion Matrix

Figure 5.7: The confusion matrix can be set to one of three modes. The original and distorted mode visualize cell values using a singe-hue blue color scale, the difference mode visualizes cell value differences between the two using a diverging color scale. Best viewed in electronic format.

over a cell displays its actual and predicted class as well as its value. Hovering over a category label reveals the path through the hierarchy tree to get to it. Moreover, precision and recall values for columns and rows are displayed analogously.

This confusion matrix is available for original input images, distorted ones and the delta between the two using the tab bar below. Original confusion matrices are calculated by making the model predict all images within its associated dataset. Using the known ground-truth label and the retrieved predicted label, the value of the confusion matrix cell corresponding to this actual and predicted class is incremented for each predicted image. Distorted confusion matrices are computed analogously, but input images are distorted before making the model predict their class. Finally, difference confusion matrices are created by computing both the original and distorted confusion matrix and subtracting the values of the latter from the former in a cell-wise manner.



(a) The colors of the diagonal of correct classifications overpower outlying misclassifications.

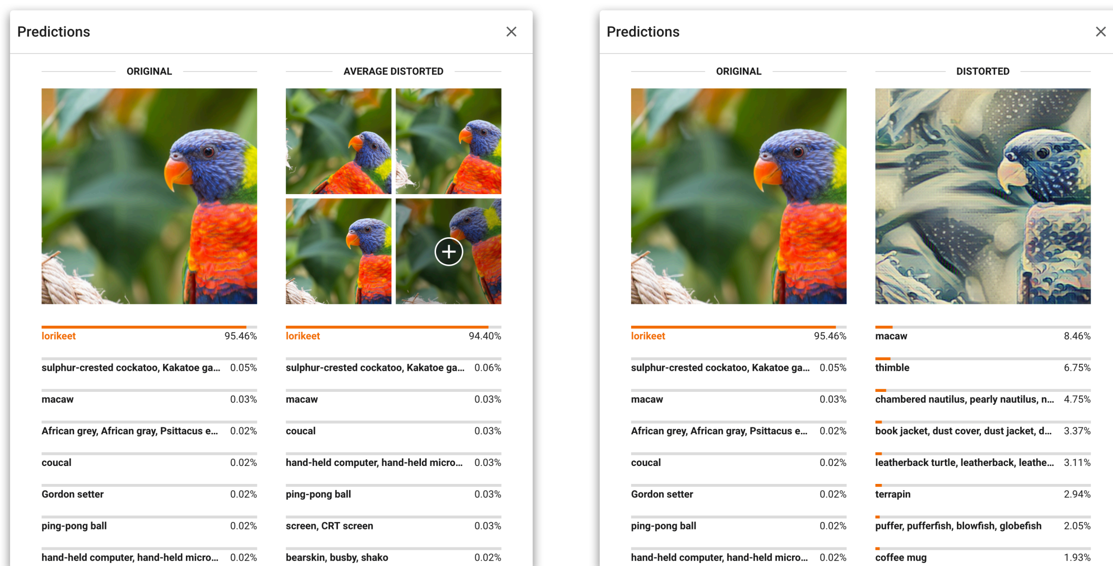
(b) When the diagonal is hidden, interesting block-like misclassification patterns along the diagonal can be made out whose bounds correspond to higher-level nodes in the classification hierarchy. Similarly to the findings by Alsallakh et al. [AHH+14], this indicates that the model is able to distinguish higher level concepts but has trouble with fine-grained classification.

Figure 5.8: The user can specify whether correct classifications along the matrix diagonal should be shown in the color map. Best viewed in electronic format.

After the matrix data has been calculated by the backend, the frontend displays it as a heat map. For the original and distorted confusion matrices, cells are colored according to a single-hue color scale that is white for the cell value of zero and blue for the maximum value within the matrix. For the difference matrix, a diverging color scale is used. It colors cells with the value zero white, while negative values are assigned a red hue and positive ones are assigned a blue hue according to their absolute values. The resulting color maps can be seen in figure 5.7.

In functional classifiers, the diagonal of the confusion matrix containing correct classifications will contain the highest values by far. This might cause outliers and patterns outside of correct classifications to become barely visible, although these are the observations that are actually insightful. To circumvent this problem, users can click on the cog to the bottom left of the matrix and open a settings dialog. Within this dialog, they can choose to exclude the matrix diagonal from the heat map. By doing so, color scales will be adjusted using only maximum and minimum values of cells that are not on the diagonal. Cells on the diagonal will display their values when zooming in but appear white. This exclusion can reveal interesting patterns, as seen in figure 5.8.

While interacting with the confusion matrix, the list of input images to the right updates as the viewport changes to represent the samples in all visible cells. Alternatively, the user can shift-click to create a fixed selection rectangle. This list of input images can be sorted by their loss of certainty in the ground-truth label between original and distorted input images, in ascending or descending order. Alternatively, they can be sorted by their indices. Each item in the input image list shows a



(a) Average predictions for multiple rotated versions of the input image.

(b) Predictions for a single image, perturbed using the style transfer distortion.

Figure 5.9: Class predictions of a model on an original input image and its distorted versions.

thumbnail of the image. Depending on the selected matrix mode, this thumbnail shows either the original input image or its distorted version. Moreover, each item displays its actual class as a bold headline and its predicted class as a caption. When the original matrix mode has been selected, these captions will show the model's predictions on original input images. Otherwise, they will show its predictions on the image transformed by the chosen distortion. At the bottom of each item, three bars indicate the certainty for the original and distorted input image respectively as well as the loss of certainty. Additional information and exact values can be retrieved from tooltips upon hovering. The list itself is populated dynamically while scrolling and only renders items within the current viewport, speeding up the visualization considerably.

Clicking on an item in the input image list highlights it as selected. When the input image dialog is closed, this image is used as the new input for visualizations.

5.4.5 Image Prediction

Clicking on the three bars between the input image and distortion selector in the main screen opens a dialog that shows the model's predictions on both the original and distorted input image, both of which are shown above the predictions. The predictions themselves are sorted by certainty and displayed as a list of bar charts. The label of the ground-truth class is highlighted using its color.

This dialog can either show predictions of an input image and its single distorted version as seen in figure 5.9b or the average predictions of multiple distorted versions as seen in figure 5.9a. Clicking on the distortion preview opens another dialog with a list of all distorted versions. Within this dialog in figure 5.10, thumbnails of distorted versions are shown along with a bar at the bottom representing the model's certainty for the ground-truth label. The list can be sorted by certainty, image index

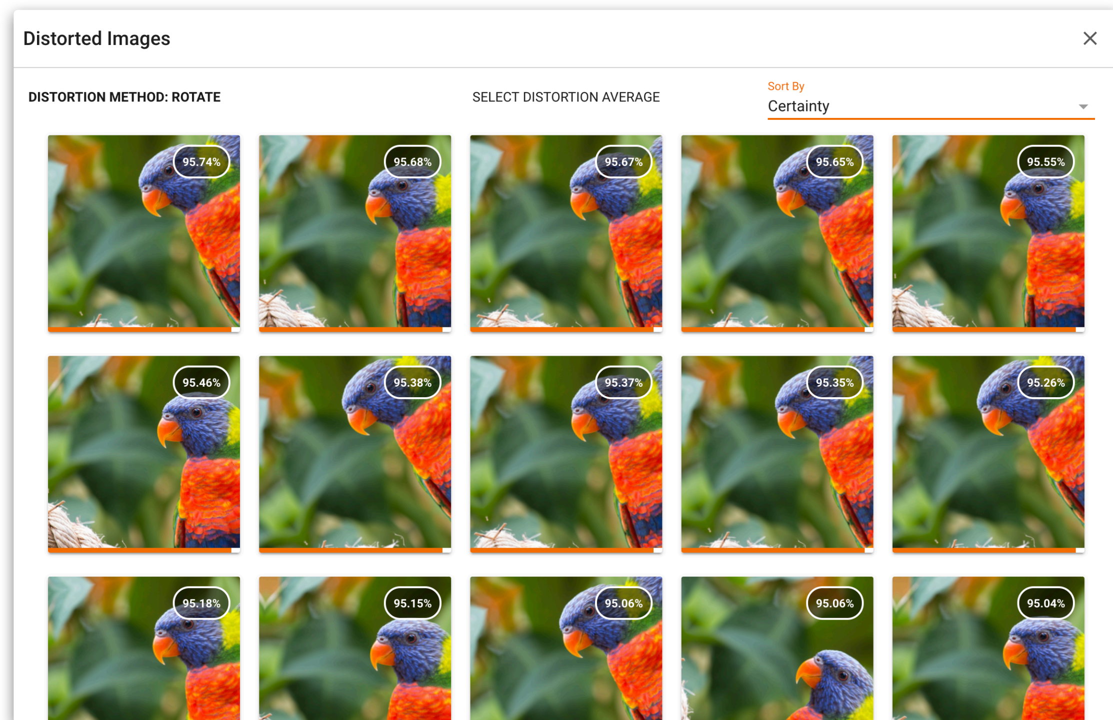


Figure 5.10: Prediction certainties for multiple distorted versions of an input image.

or by any of the ranged parameters present in the selected distortion. Each image's value of the variable that is used for sorting is displayed in the top right of the thumbnail. Clicking on one of these distorted versions closes the dialog and shows its predictions in the previous dialog.

5.4.6 Detailed Model Performance

By clicking on the performance metric of a model in the model list, a dialog can be opened that contains detailed values for all performance metrics. This dialog, as seen in figure 5.11, includes baseline values for undistorted input images, as well as values for different distortions and an average value for all distortions along with indicators that show by how much they differ from the baseline performance.

5.4.7 Distortion Configuration

Distortion modules can define parameters that can be configured by the user. The dialog seen in figure 5.12 provides an interactive and intuitive interface for adjusting these parameters.

Individual distortion modules can be selected from the list to the left. Subsequently, all of their configurable parameters are shown in the center. These parameters can be constants, configured by using a slider, ranges, configured by using a ranged slider, or enumerations, configured by using a group of radio buttons. After any value has been changed, the distortion preview immediately updates, showing four exemplary distorted versions of an input image.

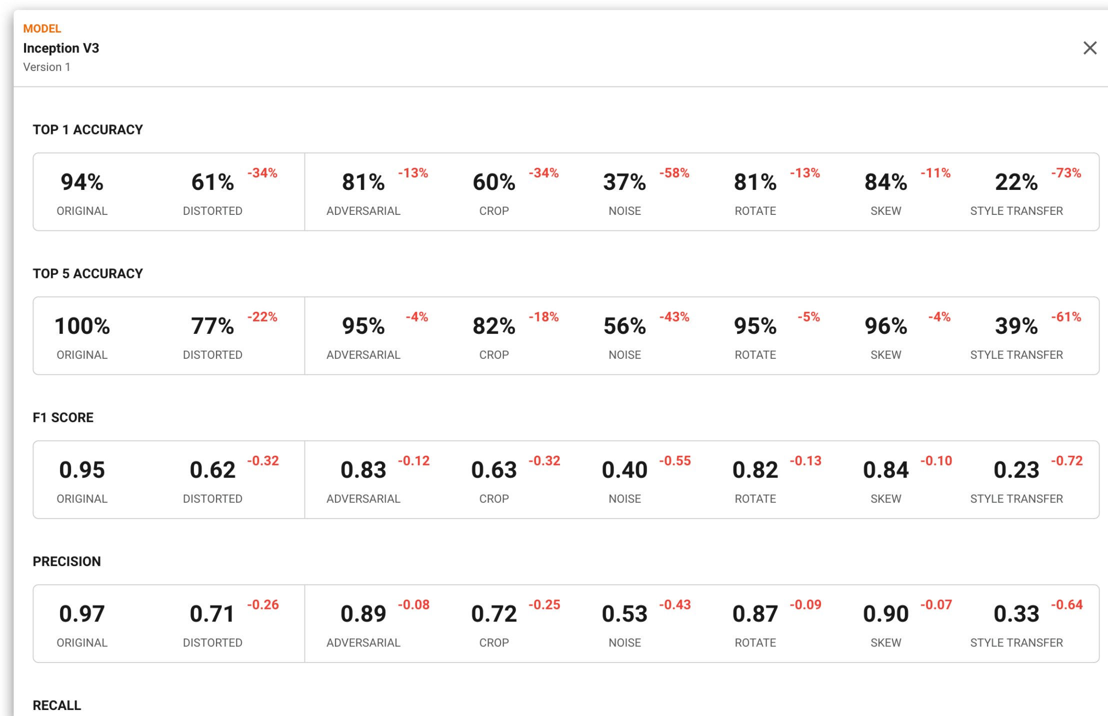


Figure 5.11: A dialog with detailed performance metrics for a single model.

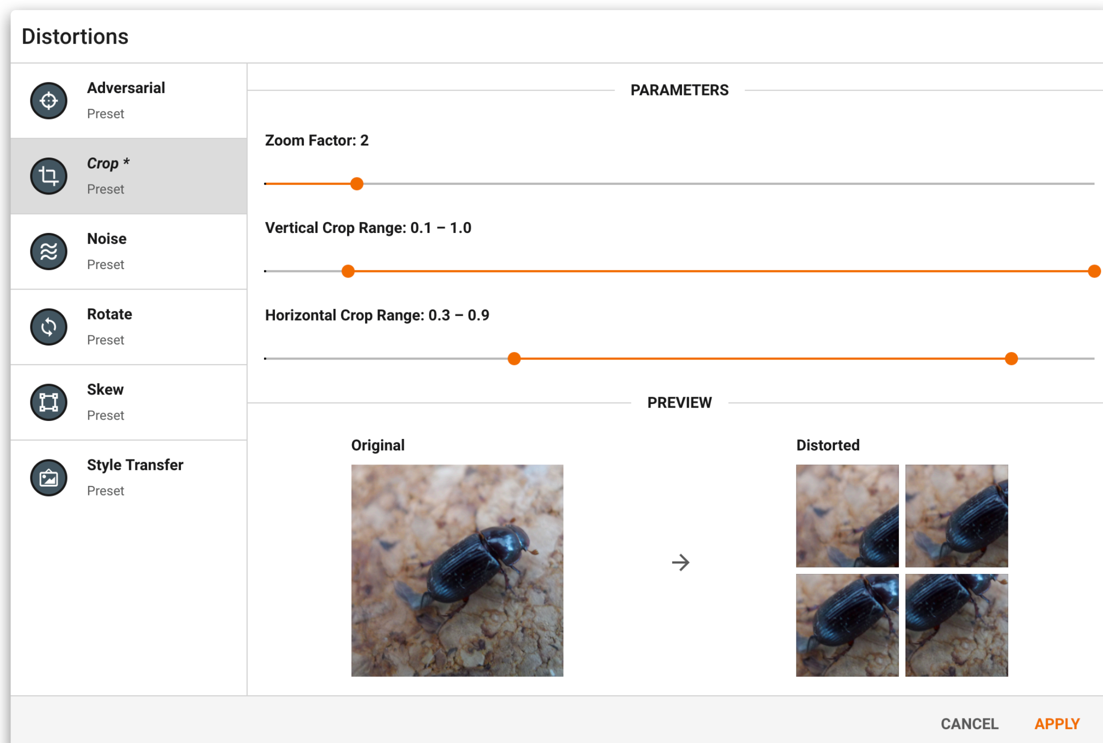


Figure 5.12: A dialog for configuring the parameters of individual distortion modules.

Distortions whose parameters have been modified are marked by a star and italic title text in the list. Clicking apply persistently saves the values that have been selected and invalidates all cached data that relies on distortions that have been changed. Furthermore, the user has the option to rebuild this portion of the cache. If they decide to do so, another dialog with a progress bar indicating the caching progress shows up.

6 Use Cases

Advis has been developed with the focus of supporting a wide array of applications, enabled by the support of highly flexible modules for models, datasets, and distortions.

This chapter will spotlight a selection of usage scenarios, describing how machine learning applicants and researchers might adopt the system in order to gain valuable insights. The use cases rely on preset models (see table 5.3), datasets (see table 5.5), and distortions (see table 5.6). Whenever a user starts Advis on an empty working directory, all of these preset modules are loaded into the directory for demonstration purposes. Models and datasets had to be converted into a compatible format before they could be included. This conversation process was completed using the separate `advis-demo-data` codebase.

6.1 Choosing a Model for a Specific Task

Alice is a software engineer creating a mobile application that allows users to take pictures of food, upload them to their profile and share them with friends. During the process of creating a post, users of her application have to specify what kind of food they have photographed. Since this task can be tedious and boring, Alice would like to streamline the process by offering users a few suggested labels they can choose from. She knows that for the task of finding appropriate labels for an individual image, DL models can be a good choice. However, Alice is not a machine learning expert and would like to use a model that already exists. She decides to use Advis to compare models and select the one that suits her task the best, using all preset models as potential options.

She creates an evaluation dataset using some images from her application that have been manually annotated. She creates a directory with all images and a JSON file connecting image file names with their respective categories. Using this data, she creates a Python module for the dataset and puts it into the working directory of Advis.

On top of that, Alice knows that dishes in the images uploaded by her users can be slightly rotated or cropped. Taking images using a smartphone in low-light conditions like restaurants also introduces noise into the photographs. Fortunately, all of these distortions are available as presets in Advis.

With all modules in place, Alice starts Advis. First of all, she tweaks the configuration of the three distortions she is interested in until the resulting distorted versions are similar to the images uploaded by her users. The live distortion preview helps her in achieving this task.

After applying the changes she has made to the distortion configurations, the system automatically evaluates all models on her input data set. Because Alice aims to show five options to her users, she opens the sidebar options and configures it to display each model's top 5 accuracy change in the model list. Finally, she selects the crop, rotate, and noise distortions from the distortion list.

Figure 6.1: A model list displaying the robustness of individual models to a user-defined selection of distortions, using metrics specified by the user.

Figure 6.2: When opening the detailed model performance dialog, only distortions that have been activated by the user are shown. Indicators show how individual distortions alter the model's prediction performance for different metrics.

Looking at the model list in figure 6.1, Alice can make out the most robust model at a glance: The *Inception V3* network only loses 11% of its top 5 accuracy. She clicks on the metric and opens the detailed model performance values in figure 6.2, revealing an impressive 100% top 5 accuracy on undistorted input images within her dataset. *ResNet V2* comes in as a close second, with a baseline accuracy of 98% and a drop of 13% on distorted images. With all of this in mind, Alice concludes to deploy the *Inception V3* model for her application.

With little effort and programming needed, Advis enabled Alice to comprehensively compare models on a very specific set of data and distortions, allowing her to make an informed decision on which model to use for her task.

6.2 Analyzing the Effect of Manipulations of Input Data

A few months later, Alice's mobile app has found a large audience of users. They are uploading many images every day, so naturally many image labels have to be created. This has put an increasing amount of strain on Alice's server, so she hired Bob, who has applied DL models before and has a deeper knowledge of their internal structure.

Bob notices that Alice has chosen the *Inception V3* network for classifying images. This was a natural choice due to its high performance and robustness to cropped, noisy or rotated input images. However, this robustness comes at the price of a comparatively large model and long inference times. In order to relieve Alice's server, Bob would like to deploy the classification models on the user's phones themselves by packaging them into the app. As another advantage, this would allow users to choose a label even if they have no network connection. However, *Inception V3* is far too large and computationally expensive to run on weaker phones. Instead, Bob would like to use a light-weight model that has been specially constructed for mobile deployment.

To start his exploration, Bob launches Advis using the same set of models that Alice had used before and the NIPS 2017 [KGB+18] dataset for evaluation. This configuration includes *MobileNet V1*, a light-weight model for mobile inference. Although Bob would like to use this fast model instead of the slow and large *Inception V3*, he can immediately make out the reason why Alice did not choose *MobileNet V1* from the same model list that can be seen in figure 6.1. Its average robustness to the distortions of interest is the second lowest in the list, with its top 5 accuracy falling by 20%.

Bob would like to investigate why this model performs so badly on distorted input images. As a first step, he compares the robustness profiles of both *MobileNet V1* and *Inception V3* by selecting their checkboxes in the model list. The radar chart in the sidebar updates and shows the robustness of both models to the three selected distortions, as can be seen in figure 6.3. Interestingly, the robustness of both models to rotation and cropping is comparable. Only on noisy input does *MobileNet V1* fare significantly worse than *Inception V3*. Bob would like to know why.

He selects the *MobileNet V1* model and the noise distortion by clicking on the corresponding node in the radar chart and moves his focus from the sidebar to the central visualization view. As a first step in his model exploration, Bob would like to know which input images cause problems for the model's predictions. He opens the input image selection and inspects the confusion matrix. To get an idea of how predictions change between undistorted and noisy input images, Bob selects the difference tab that displays how a model's classifications have changed through the distortion using a diverging color scale. He notices that correct classifications along the matrix's diagonal

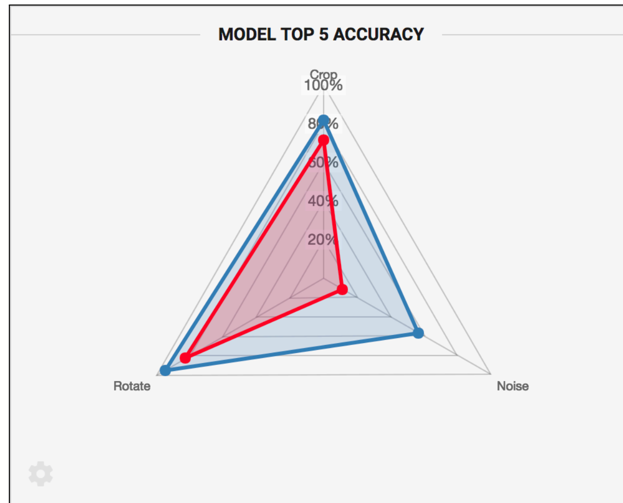
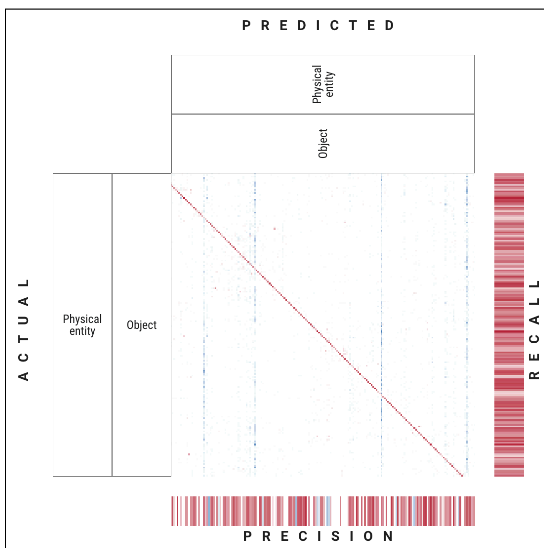
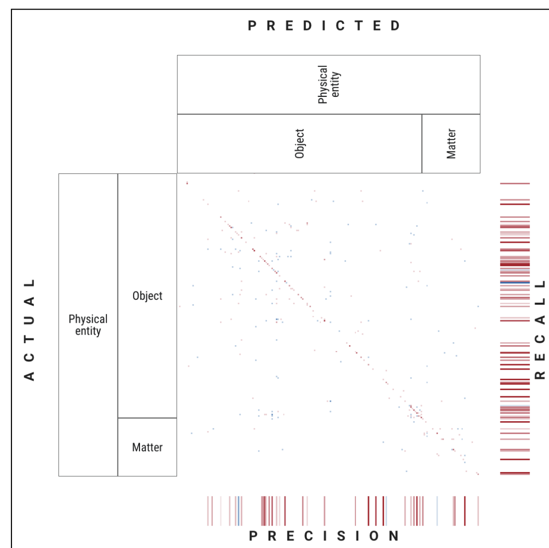


Figure 6.3: A comparison of the robustness profiles of two models to cropped, rotated, and noisy input images, with *Inception V3* in blue and *MobileNet V1* in red.



(a) Unmodified *MobileNet V1* model exhibiting line patterns.



(b) Re-trained *MobileNet V1* model without line patterns but with some misclassification noise, which can be reduced through further training.

Figure 6.4: Confusion matrices for both the original and the re-trained *MobileNet V1* model. Negative cell value changes between original and distorted input images are shown in red, positive ones in blue. Best viewed in electronic format.

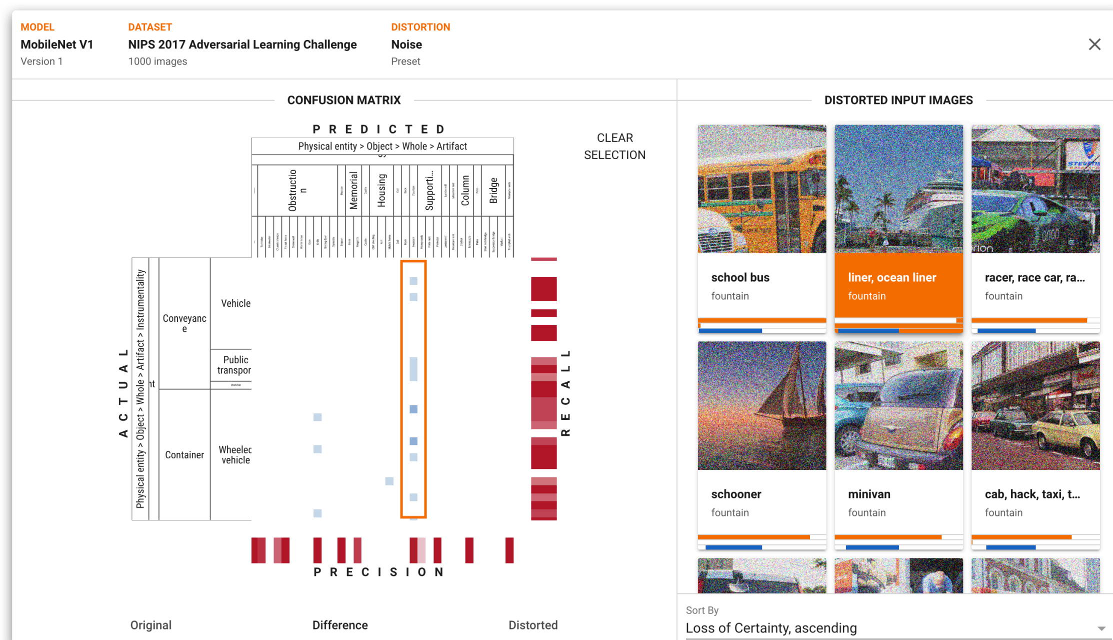


Figure 6.5: The rectangle selection reveals a list of misclassified input images, all predicted to be fountains. Input image thumbnails show distorted versions of input images along with their actual and predicted categories.

deteriorate. On top of that, he recognizes an interesting pattern of misclassifications: Predictions seem to be collected in a handful of bin classes which are completely unrelated to the actual labels of samples, forming vertical lines in the confusion matrix, as can be seen in figure 6.4a. Bob zooms into one of these lines and drags open a selection rectangle over it. He sorts the list of corresponding input images by their ascending loss of certainty. Using this sorted list, he can easily select an input image where the model had a high confidence of its correct classification before the distortion, but where the prediction accuracy has deteriorated completely after adding noise (see figure 6.5). Having made his selection, Bob closes the dialog to inspect the model's predictions on both the original and distorted version using the prediction dialog seen in figure 6.6. As expected, the model produces fairly outlandish class predictions for the noisy input image.

To find out what is happening within the network, Bob inspects the computation graph. First of all, he needs to configure the graph view to suit his needs. He opens the settings dialog using the cog in the bottom left and configures the graph view to use only the currently selected input image and distortion to compute node activation differences. On top of that, he chooses to aggregate activation similarities using the minimum values of all children in order to highlight low similarity values within higher-level nodes. After closing the settings dialog, the node visualization immediately runs the model in the background, retrieves all similarity values and updates the node colors.

Bob expands the network graph and is presented with the expected architecture, consisting mostly of a chain of separated depth-wise and point-wise convolutions, a main idea of *MobileNet V1* for speeding up inference. The node colors represent the similarity of activations between original and distorted input image versions within the corresponding layer. They can provide clues as to where a model fails. Bob knows that the model is fairly robust to rotated input images. Therefore,

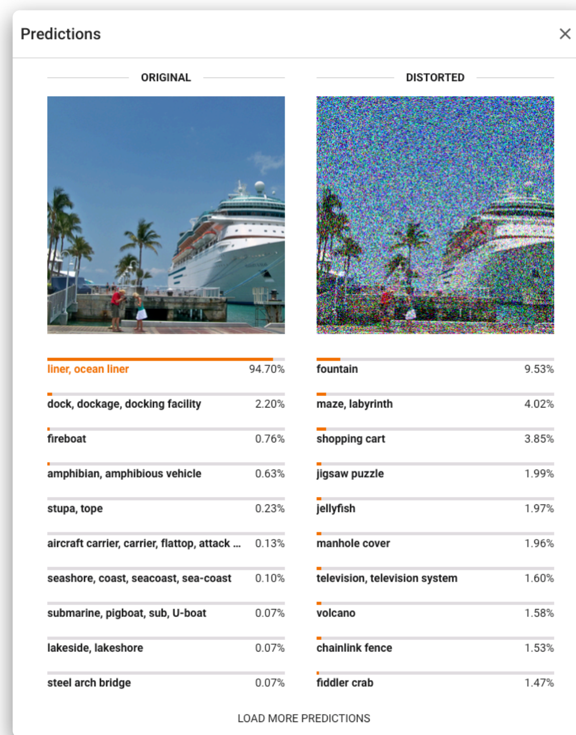
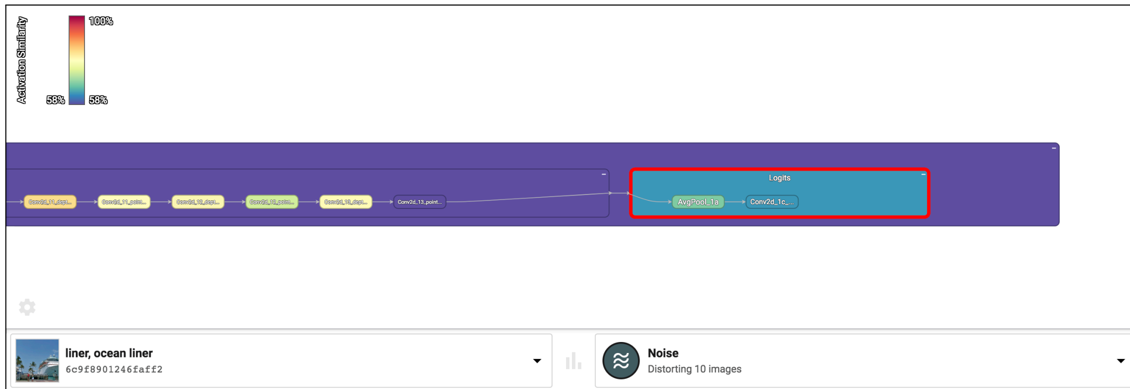


Figure 6.6: The original *MobileNet V1* misclassifies a noisy input image, labeling it with entirely wrong categories. The left half of the dialog contains a thumbnail and predictions for the original input image, the right half contains both for the distorted input image.

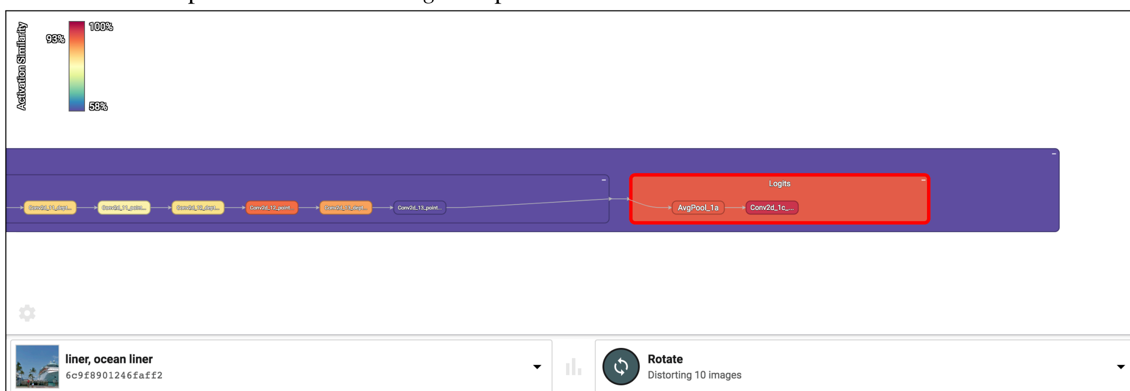
he compares node similarity values for both rotated input images (see figure 6.7b) and noisy input images (see figure 6.7a) by switching between the distortions using the dropdown menu. While doing so, the system retains the current viewport of the graph and smoothly fades between node colors. Inspecting these two specific sets of node activation similarities reveals similar patterns. However, the *Logit* nodes that compute final predictions at the very end of the model stand out. Not only do they have very low similarity values, but in contrast to rotated input images, the final convolutional operation has a lower similarity value than the average pooling operation before it.

Bob suspects that the final layers of the model cannot deal with noisy values propagated through the network. Indeed, clicking on nodes within the main part of the network reveals that they are still able to function with noisy input as seen in figure 6.8. Their activations are neither all minimal or maximal nor are they random noise but instead look similar to the feature extraction steps performed on non-distorted input images. Bob concludes that the model itself is still functional on noisy input images, but the combination of features into the final predictions is broken. He decides to re-train the *Logit* layers to solve this issue.

First of all, Bob needs a training set. He decides to use the ILSVRC evaluation set [RDS+15] with its 50,000 labeled images and makes sure that they do not contain any images that are present in his test set. For each of these images, Bob creates a new input image by adding the same noise he uses in Advis. He then randomly splits the resulting 100,000 images into 80,000 training images and 20,000 validation images.



(a) Activation similarities for noisy input images. Similarity decreases from the average pooling to the convolution operation within the *Logit* scope.



(b) Activation similarities for rotated input images. Similarity increases from the average pooling to the convolution operation within the *Logit* scope.

Figure 6.7: Node activation similarities for individual nodes, for noisy and rotated input images.

Bob loads the original trained *MobileNet V1* model but excludes all weights and biases in the *Logit* scope. Instead, he trains these final layers from scratch on his training dataset using RMSprop [TH12] for gradient descent optimization with a learning rate of 0.01 and a weight decay of 0.00004 for 3000 steps.

Finally, Bob loads his re-trained model as a new Advis module. He increases the module's version number to be able to easily differentiate between both models within the UI. After launching Advis, he recognizes that his re-training has been successful. The model's robustness to noisy input images has doubled, as seen in figure 6.9. On top of that, the vertical line patterns in the confusion matrix have disappeared (see figure 6.4b).

Of course, these are only intermediate results since the training parameters have been chosen intuitively and the amount of training steps is very low for this first test because Bob did not have access to specialized powerful hardware. On top of that, Bob might be interested in adding rotated and cropped input images into the mix of training images. Advis will support Bob through all of these iterative steps of improvement and fine-tuning, providing insights into where changes might be necessary and evaluating their impact.

Figure 6.8: Activation visualization for a layer that is situated deep in the *MobileNet V1* model for both original and noisy input images.

DISTORTIONS
Adversarial
Crop
Noise
Rotate
Skew
Style Transfer
EDIT DISTORTIONS

MODELS

Model	F1	TOP5
MobileNet V1 Version 1	-0.13	-14%
MobileNet V1 Version 2	-0.05	-7%

Figure 6.9: Robustness of both the original *MobileNet V1* model and its re-trained version on noisy input images.

Through deep network analysis, Advis enabled Bob to find the weakness of a specific model and recognize problems at specific points in its computation graph, informing him of steps he can take in order to debug the network.

6.3 Defending Against Adversarial Perturbations

Through Bob's efforts, Alice's app quickly became the largest competitor in its market. To further its growth, Alice has decided to widen the app's scope to images of all kinds rather than just food. To make uploading even easier, she has done away with selectable image labels and has instead

deployed the *Inception 5h* model on her server. This model is a simplified variant of *Inception V1*, optimized for fast inference. When a user uploads an image, the server automatically classifies its contents using the model's top prediction. Other users can then explore images within these categories or subscribe to a few of them to find corresponding images in their feeds.

Unfortunately, popularity can beget envy and rivalry. Through an informant, Alice has gathered that Eve – her fiercest competitor – is planning on attacking her system in order to damage her app's reputation. As a matter of fact, shortly thereafter new images with clearly visible contents are uploaded that are put into entirely wrong categories, cluttering her user's feeds with posts they are not interested in. Alice suspects that Eve might be at fault for these errors and quickly starts investigating.

First of all, she analyzes the misclassified photos. She recognizes them from before, realizing that they are copies of older images uploaded to the system but with very slight alterations. Eve might have chosen this approach to make it harder for Alice to remove the images of her attack but leave actual user content unharmed in the process. Alice compares the original and distorted images and recognizes that the latter seem to have been created by always adding the same pattern of values to the image pixels. This reminds Alice of a paper about universal perturbations [MFFF17] she has read recently, where an adversary was able to fool a deep neural network by adding a barely perceptible pattern to input images, without the need of access to the network or performing an optimization problem for each input sample. Alice extracts the changes made to each image and creates a new Advis distortion module that performs these same changes. She starts Advis with her *Inception 5h* model and her new distortion and opens the interface.

Alice selects her adversarial distortion and immediately notices from the sidebar that her model is very vulnerable to this attack. To find out what exactly is happening, she selects the network and opens the image selection dialog. Within this dialog, she selects the difference confusion matrix and notices an interesting pattern. As seen in figure 6.10, correct classifications along the diagonal deteriorate significantly. Instead, classifications of distorted input images form a vertical line across the confusion matrix. This pattern suggests that a large amount of completely unrelated input images are put into one bucket class. Alice zooms in on the matrix and selects a few samples within this line using the rectangle selection tool. As a matter of fact, she sees a large list of input images with entirely different ground-truth labels, all predicted to be "strainers". This result corresponds to the aforementioned technique of universal adversarial perturbations which aims to iteratively push samples beyond their classification regions and into bucket classes [MFFF17]. To investigate further, Alice sorts the list and selects one with a large loss of certainty.

Closing the input image selection dialog sets the selected image as the new input sample. Alice validates her suspicion by opening the detailed prediction dialog. As seen in figure 6.11, this dialog reveals that the distorted input image is indeed classified as a "strainer". The visual difference between the original and distorted image is barely perceptible, but the model is still very confident of its misclassification.

To investigate further, Alice takes a look at the network graph. She configures it to only use the single selected input image and distortion method for activation similarity calculations and sets the aggregation method to use minimal values. Interestingly, the activation similarities decrease steadily as the input data flows through the network, as can be seen in figure 6.12. In the first layer, the very similar adversarial version of the input image produces very similar activations, whereas these activations start diverging increasingly until the final layer, which produces entirely

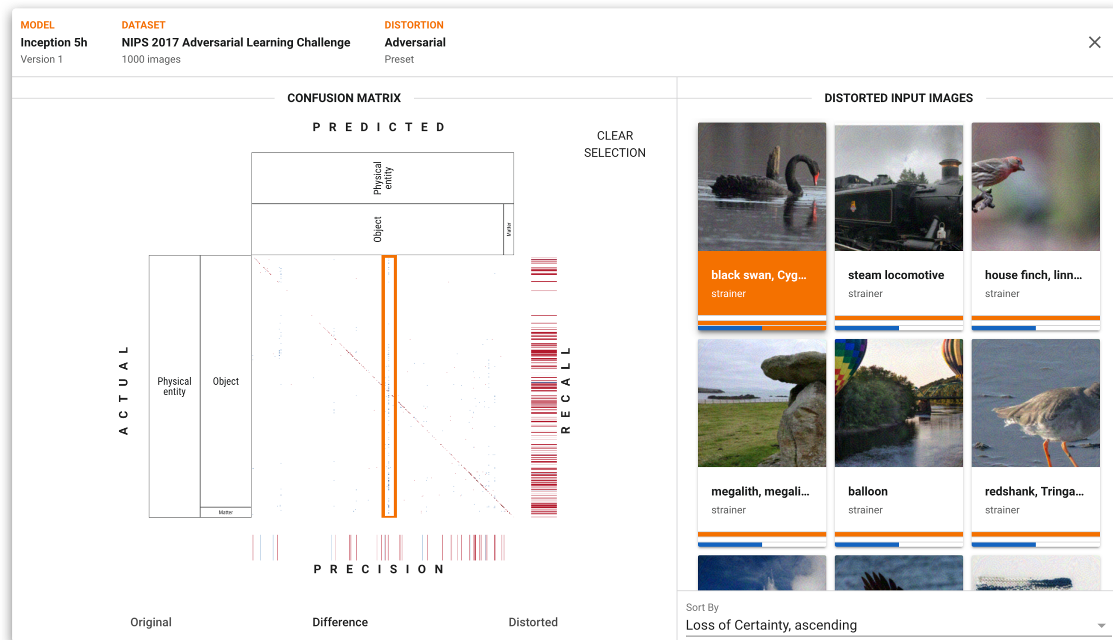


Figure 6.10: Differences in the confusion matrices for original and adversarial input. Negative cell value changes between original and distorted input images are shown in red, positive ones in blue. A single vertical line becomes apparent. The input image list to the right shows thumbnails of adversarial input images within the selection rectangle, all of which are classified as a "strainer". Best viewed in electronic format.

wrong predictions. The technique of optimizing universal adversarial samples [MFFF17] seems to have been successful on her model. Alice concludes that there is a structural problem within her network, making it vulnerable to adversarial input samples. Potential solutions for this problem include increasing a model's robustness through adversarial training [MMS+17] or defending against adversarial perturbations by adapting the technique of neural network distillation [PMW+16]. As another approach, Alice might try to detect adversarial samples from artifacts they produce within the network [FCSG17]. However, all of these approaches require significant effort and Alice wants to stop the ongoing surge of misclassifications as quickly as possible. Therefore, she postpones these more profound solutions, noting that Advis will support her in evaluating whether they have made the network more resistant.

As a quick fix, Alice collects all the insights about the characteristics of adversarial samples she has gained through Advis. Using this knowledge, she decides to remove all recent images that have been classified as a "strainer" and restricts new images with these characteristics from being uploaded. With her categorization mechanism back in order, her users are happy again and Alice can start hardening her network against adversarial perturbations.

With its highly flexible modular distortion and model system, Advis enabled Alice to find patterns within the large-scale malfunction of her model, discovering how adversarial perturbations affect the internal representations of her network and giving clues as to how to defend against them.

Figure 6.11: The *Inception 5h* network spectacularly misclassifies an adversarial input image.

Figure 6.12: Node activation similarities within the *Inception 5h* network decrease steadily for adversarial input images.

7 Discussion

The visualization system for deep neural networks presented in this thesis allows a comprehensive comparison of models and their performance, focussed on the robustness of these models to input perturbations. Insights gained by this functionality can be especially useful while evaluating how well a model handles distorted input data compared to related models. Furthermore, it enables users to make informed decisions on what network to use for their task.

Going beyond the comparison of robustness metrics, the system lets users dive into the internal components of a network, analyzing how distortions affect internal representations. An interesting input image can be selected using an interactive confusion matrix that is able to handle large amounts of data classes. The network itself can be explored using an interactive color-coded node-link diagram. Selecting a layer reveals its activations on the original input image and its distorted versions, as well as a set of advanced comparison methods. Finally, users can choose to display the model's predictions on an individual input image and its distorted versions in detail. All of these visualizations can be useful for gaining an understanding of how networks function and how they react to distorted input data.

All visualizations are contained within an intuitive, consistent, and highly responsive UI. They are implemented by an architecture that separates server-side computations and client-side visualizations, supporting scaling the system when confronted with ever increasing model and data sizes, retaining responsiveness.

Finally, the infrastructure lets users define their very own models, datasets, and distortion methods in code for their specific use cases in a highly modular manner, extending the amount of potential use cases. It includes a comprehensive set of state-of-the-art models and datasets as well as some distortions that user can build upon. To the author's best knowledge, this is the first system for visualizing deep learning models that allows this range and ease of extensibility, as well as the first system that makes use of it to visualize the impact of adversarial perturbations in a non-educational way.

However, in its current state the system's scope is limited to deep neural networks for image classification. Although an interesting and lively research topic, there are many more application fields such as object detection and different types of input data such as audio or text that is not covered by the system. These tasks often go beyond assigning a single class to input data. Moreover, non-feed-forward networks are not supported in the current state of the system.

Furthermore, a host of features included in related visualization systems can be useful for furthering a user's understanding but is missing from Advis. This includes visualizing learned model parameters such as weights, filters or features. The display of neuron activations in high-dimensional space via dimensionality reduction is possible through TensorBoard's native embedding projector functionality but could be included into the interface of Advis in a way that makes it easily accessible and

connects it to the other visualization views. Finally, the system supports instance-based analysis and exploration but misses out on the potential of letting users define subsets of multiple instances that share interesting characteristics.

Fortunately, the system has been implemented with important software engineering principles like data encapsulation, modularity, loose coupling, and maintainability in mind, so these features can be easily implemented and integrated into the frontend in future work.

8 Conclusion

Deep learning models are steadily gaining in both popularity and complexity. While deep neural networks continue to break records in computer vision tasks, their internal structure and learned representations have become opaque. This is detrimental for both researchers and applicants. The process of building and improving networks far too often relies on guesswork and trial and error. The resulting network is often seen as an inexplicable black box, mapping input data to the desired output values more or less successfully.

Many research projects have set out to alleviate this problem through visualization systems and techniques. However, none of them offer an integrated way of analyzing the impact of input data transformations on both internal representations as well as a model's performance. In particular, a better understanding of the mechanics of adversarial perturbations can generate valuable insights on how to defend against them and on the inner workings of deep neural networks in general. This thesis set out to fill this gap.

8.1 Results

As a first step, this thesis has outlined the theoretical foundation of Visual Analytics and how it can be useful for gaining insights into Deep Learning models. Furthermore, a comprehensive set of visualization systems for Deep Learning has been listed and described. These systems, most of which originate from relatively recent research, have been compared according to common characteristics, such as the visualization's goals, what elements of a model they visualize in what way, and whether these visualizations take place during or after training. Finally, four specific systems that are the closest to the topic of this thesis have been selected and described in more detail.

The main contribution of this thesis is Advis, a visual approach for probing Deep Learning models that goes beyond the surveyed system's abilities in terms of visualizing the impact of input transformations. It offers both a comprehensive comparison of models and their prediction's robustness to input transformations as well as deeper insights into the structure of deep neural networks and how their layer's activations change when distorting input images. Moreover, the system's highly flexible architecture has been described, which includes an easy way for users to define their own models, datasets, and distortions.

Finally, three use cases have been presented that demonstrate the usefulness of the system for gaining insights into the impact of input transformations on deep neural networks. The first use case has shown the system's ability to support comparing models and their robustness to input distortions, aiding in the selection of a network that suits a task's specific requirements the best. The second and

third scenario presented ways in which the system allows users a deeper look into their networks and the impact of input distortions on their internal representations, offering support for the task of debugging and improving networks.

8.2 Open Questions and Future Tasks

The system was able to provide valuable insights into deep neural networks and uncover intriguing patterns within their learned representations. But despite the system's large array of functionality, some questions remain unanswered. These questions will have to be addressed in future work.

Possible extensions of Advis include the visualization of learned parameters, for example using feature visualization. Adding this functionality can give users insights into the representations a model has learned. Furthermore, model exploration based on instance subsets and bookmarked activations or layers can be a useful extension of the present exploration based on single instances, supporting the process of making sense of a model's decisions and mistakes. Finally, future research might address the question of how the scope of Advis can be widened to include Deep Learning models from other application areas than image classification, which have different types of input data and produce outputs that are more complex than a simple class mapping. Retrieving structured feedback from Deep Learning experts evaluating the system will be useful for determining the future path of development.

On top of its already present functionality, the architecture of Advis can provide a flexible and future-proof foundation for all of these potential extensions, aiming towards the ultimate goal of improving the understanding of the inner workings of deep neural networks.

A Appendix

A.1 Routes

The backend of Advis defines a set of HTTP routes that are used by the frontend to retrieve all necessary data. These routes, their input parameters, and their outputs are documented in this section.

A.1.1 Models

Retrieve information about all models that have been loaded.

`/models`

Fetch a list of all loaded models.

Parameters: None.

Response: A list of all loaded models with their name, display name, version number, and the name of the dataset they are associated with.

A.1.2 Graphs

Retrieve the computation graphs of models.

`/graphs`

Fetch the graph structure of a specific model.

Parameters:

- `model`: The name of the model whose graph structure should be retrieved.
- `mode`: Can be either `full` (all nodes and edges of the graph) or `simplified` (all nodes that cannot be visualized are removed and their edges transferred to their parent and child nodes).

Response: A *Protobuffer*[1] string with the graph structure of the specified model.

[1] https://developers.google.com/protocol-buffers/

A.1.3 Predictions

Make models predict the class of an image and retrieve information concerning these predictions.

`/predictions/single`

Fetch a model's prediction of a single input image.

Parameters:

- `model`: The name of the model that should perform the prediction.
- `imageIndex`: The index of the input image within the dataset associated with the model that should be classified.
- `distortion` *(Optional)*: The name of a distortion that should be applied to the input image before performing the classification.
- `distortionAmount` *(Optional)*: If the applied distortion should be repeatable rather than random, this parameter defines the total amount of distorted versions that should be generated.
- `distortionIndex` *(Optional)*: The index of the distorted version within the set of all versions. Should be between 0 and `distortionAmount`.
- `predictionAmount` *(Optional)*: The amount of predictions that should be returned. The default value is `5`. If this parameter is set to `-1`, all available predictions will be returned.
- `onlyCategory` *(Optional)*: If only the prediction certainty of a specific classification category should be returned, this parameter can be used to set the index of this category.

Returns: Information about the input data and potentially the distortion that has been applied, as well as a list of predictions returned by the model with their category number, name, and certainty, ordered by certainty.

`/predictions/average`

Fetch a model's average predictions on a set of randomly distorted versions of a single input image.

Parameters:

- `model`: The name of the model that should perform the prediction.
- `imageIndex`: The index of the input image within the dataset associated with the model that should be classified.
- `distortion`: The name of a distortion that should be applied to the input image before performing the classification.
- `distortionAmount`: The amount of distorted versions of the input image that should be generated.

Returns: Information about the input data and the distortion that has been applied, as well as a list of predictions returned by the model with their category number, name, and certainty, ordered by certainty. These predictions have been generated by running the model on all distorted versions of the input image and calculating the average certainty of each category.

`/predictions/accuracy`

Fetch some metrics about the model's performance on a set of input images which may be distorted if so desired.

Parameters:

- `model`: The name of the model that should perform the prediction.
- `inputImageAmount`: The amount of random input images that should be chosen from the dataset associated with the model and used to calculate the performance metrics.
- `distortion` *(Optional)*: The name of a distortion that should be applied to the input images before performing the classification and subsequent performance metric calculation.

Returns: Information about the model and input data as well as the model's top 1 and top 5 accuracy, F1 score, precision, and recall when classifying the input images.

A.1.4 Confusion Matrices

Retrieve data for displaying a model's predictions on original or distorted input images as a confusion matrix.

`/confusion/matrix/full`

Create a confusion matrix with all input images and all categories within a model's dataset.

Parameters:

- `model`: The name of the model that should perform the predictions.
- `distortion`: The name of a distortion that should be applied to the input images before performing the classification and confusion matrix generation.
- `mode`: The confusion matrix generation mode. Can be either `original` to use predictions of original input images, `distorted` to use predictions of distorted input images or `difference` to perform both of the aforementioned predictions and return each cell's difference.

Returns: The rows and columns of the confusion matrix as a two-dimensional array, its value range, the list of labels in hierarchical ordering as well as two arrays with precision and recall values of columns or rows.

`/confusion/matrix/superset`

Create a confusion matrix with all input images and the child categories immediately below a specific superset category within a model's dataset. Individual cell values will be aggregated from the bottom to the child categories.

Parameters:

- `model`: The name of the model that should perform the predictions.
- `distortion`: The name of a distortion that should be applied to the input images before performing the classification and confusion matrix generation.
- `mode`: The confusion matrix generation mode. Can be either `original` to use predictions of original input images, `distorted` to use predictions of distorted input images or `difference` to perform both of the aforementioned predictions and return each cell's difference.
- `superset` *(Optional)*: The name of the superset to be used. If none is given, the most high-level superset will be used.

Returns: Information about the input, the rows and columns of the confusion matrix as a two-dimensional array, two arrays with precision and recall values of columns or rows, as well as a list with the amounts of children categories of each cell.

`/confusion/images/superset`

Fetch all images that fall into cells within a specific superset confusion matrix.

Parameters:

- `model`: The name of the model that should perform the predictions for the confusion matrix.
- `distortion`: The name of a distortion that should be applied to the input images before performing the classification and confusion matrix generation.
- `superset` *(Optional)*: The name of the superset to be used. If none is given, the most high-level superset will be used.
- `sort`: The way that the list of images should be sorted. Can be either `ascending` to sort by an increasing amount of certainty change, `descending` to do the same but in reverse or `index` to sort by image indices.

Returns: A list of all images within the cells of the specified superset confusion matrix as well as their prediction certainties for original and distorted input.

`/confusion/images/subset`

Fetch all images that fall into a subset of actual and predicted categories in the full confusion matrix. All four parameters for defining this subset have to be given as category indices in hierarchical order.

Parameters:

- `model`: The name of the model that should perform the predictions for the confusion matrix.
- `distortion`: The name of a distortion that should be applied to the input images before performing the classification and confusion matrix generation.
- `inputMode`: Defines the confusion matrix that should be used for compiling the image list. Can be either `original` to use the confusion matrix created by predicting original input images or `distorted` to use the confusion matrix created by predicting distorted input images.
- `sort`: The way that the list of images should be sorted. Can be either `ascending` to sort by an increasing amount of certainty change, `descending` to do the same but in reverse or `index` to sort by image indices.
- `actualStart`: The starting ground-truth category index of the subset.
- `actualEnd`: The ending ground-truth category index of the subset.
- `predictedStart`: The starting prediction category index of the subset.
- `predictedEnd`: The ending prediction category index of the subset.

Returns: A list of all images within the cells of the specified subset of the confusion matrix as well as their prediction certainties for original and distorted input.

A.1.5 Distortions

Retrieve information about loaded distortions, apply them on an input image and update distortion parameters.

`/distortions`

Fetch a list of all available distortion methods.

Parameters: None.

Returns: A list of all distortion methods, with their name, display name, type, icon, and detailed information about their parameters.

`/distortions/single`

Distort a single input image and retrieve the resulting image.

Parameters:

- `distortion`: The name of the distortion method that should be used.
- `dataset`: The name of the dataset that contains the desired input image.
- `imageIndex`: The index of the desired input image within its dataset.
- `distortionAmount` *(Optional)*: If the applied distortion should be sequential rather than random, this parameter defines the total amount of distorted versions that should be generated.
- `distortionIndex` *(Optional)*: The index of the distorted version within the set of all versions. Should be between 0 and `distortionAmount`.
- `parameters` *(Optional)*: If different parameter values rather than the currently active configuration of the distortion should be used, they can be supplied as a JSON list. This list must contain all parameters that are needed by the distortion. They are only used for distorting the input image and will not change the configuration of the distortion.

Returns: The input image, distorted using the specified method.

`/distortions/update`

Update the parameter values of a list of distortions. After the update has been performed, all portions of the cache concerning these distortions will be invalidated and might have to be re-cached.

Parameters:

- `distortions`: The JSON dictionary that maps all distortions that should be updated to their set of new parameter values.

Returns: None.

A.1.6 Datasets

Retrieve information about all loaded datasets or retrieve individual input images.

`/datasets`

Fetch a list of all available datasets.

Parameters: None.

Returns: A list of all datasets, with their name, display name, and amount of images in the dataset.

`/datasets/categories/list`

Retrieve a list of all categories within a dataset.

Parameters:

- `dataset`: The name of the dataset whose categories should be retrieved.
- `ordering` *(Optional)*: The way the list of categories should be ordered. Can be `index` if the categories should be ordered in the way they appear in the list or `hierarchical` if they should be ordered in the way they appear in the category hierarchy. Defaults to `index`.

Returns: A list of all categories within the dataset in the specified order.

`/datasets/categories/hierarchy`

Retrieve a hierarchy of all categories within a dataset.

Parameters:

- `dataset`: The name of the dataset whose categories should be retrieved.

Returns: Hierarchically nested dictionaries of all categories within the dataset.

`/datasets/images/list`

Fetch a list of all input images within a dataset.

Parameters:

- `dataset`: The name of the dataset whose images should be retrieved.

Returns: A list of all images within the dataset, with their index, ID, ground-truth category number, and category name.

`/datasets/images/image`

Fetch a single input image from a dataset.

Parameters:

- `dataset`: The name of the dataset that contains the desired input image.
- `index` or `id`: Either the index of the image within the dataset or its ID.

Returns: The image as specified by its index or ID.

A.1.7 Layer

Retrieve visualizations of activations within a single model layer that appear when the model is used to classify an input image.

`/layer/single/image`

Fetch the activation visualization of a single slice within a single network layer of a model.

Parameters:

- `model`: The name of the model that contains the layer that should be visualized.
- `layer`: The name of the layer which corresponds to its TensorFlow node name.
- `imageIndex`: The index of the input image within the dataset associated with the model.
- `unitIndex`: The index of the slice whose activation visualization should be retrieved.
- `distortion` *(Optional)*: The name of a distortion that should be applied to the input image before being fed into the model.
- `imageAmount` *(Optional)*: The amount of distorted versions that should be created and fed into the model. The resulting visualization will be the average of the activations on all distorted versions. This parameter is only needed if a distortion has been specified.

Returns: The activation visualization image of the specified slice when the model is run on the specified input image.

`/layer/single/meta`

Fetch meta information about a network layer which can be visualized.

Parameters:

- `model`: The name of the model that contains the layer that should be visualized.
- `layer`: The name of the layer which corresponds to its TensorFlow node name.
- `imageIndex`: The index of the input image within the dataset associated with the model.
- `distortion` *(Optional)*: The name of a distortion that should be applied to the input image before being fed into the model.
- `imageAmount` *(Optional)*: The amount of distorted versions that should be created and fed into the model. The resulting visualization will be the average of the activations on all distorted versions. This parameter is only needed if a distortion has been specified.

Returns: The amount of slices within the resulting visualization.

`/layer/composite/image`

Fetch the composite activation visualization of all slices of a network layer which can be visualized. In a composite activation visualization, the single visualizations of all slices are combined into a collage in a grid-like fashion. This is done in a way that ensures that the final collage's size matches a desired width and height. For example, the width and height might be specified as the size of a frontend container that should display the visualization in an optimal manner.

Parameters:

- `model`: The name of the model that contains the layer that should be visualized.
- `layer`: The name of the layer which corresponds to its TensorFlow node name.
- `imageIndex`: The index of the input image within the dataset associated with the model.
- `width`: The desired width of the resulting composite visualization.
- `height`: The desired height of the resulting composite visualization.
- `distortion` *(Optional)*: The name of a distortion that should be applied to the input image before being fed into the model.
- `imageAmount` *(Optional)*: The amount of distorted versions that should be created and fed into the model. The resulting visualization will be the average of the activations on all distorted versions. This parameter is only needed if a distortion has been specified.

Returns: The resulting composite activation visualization image.

`/layer/composite/meta`

Fetch meta information about the composite activation visualization of all slices of a network layer which can be visualized. In a composite activation visualization, the single visualizations of all slices are combined into a collage in a grid-like fashion. This is done in a way that ensures that the final collage's size matches a desired width and height. For example, the width and height might be specified as the size of a frontend container that should display the visualization in an optimal manner.

Parameters:

- `model`: The name of the model that contains the layer that should be visualized.
- `layer`: The name of the layer which corresponds to its TensorFlow node name.
- `imageIndex`: The index of the input image within the dataset associated with the model.
- `width`: The desired width of the resulting composite visualization.
- `height`: The desired height of the resulting composite visualization.
- `distortion` *(Optional)*: The name of a distortion that should be applied to the input image before being fed into the model.

- `imageAmount` *(Optional)*: The amount of distorted versions that should be created and fed into the model. The resulting visualization will be the average of the activations on all distorted versions. This parameter is only needed if a distortion has been specified.

Returns: Information about the input data that has been specified as well as a list of all tiles within the composite image with their indices, row and column numbers, as well as their pixel bounds.

A.1.8 Node

Retrieve information about how the activation of single nodes differs when input images fed into the model are being distorted. Individual activation tensors are compared using their cosine similarity. Contrary to the Euclidean distance, this metric is invariant to the size of the activation tensors.

`/node`

Fetch the average difference in activation of a single layer between original input images and their distorted versions being fed into the model.

Parameters:

- `model`: The name of the model that contains the layer.
- `layer`: The name of the layer which corresponds to its TensorFlow node name.
- `distortion`: The name of a distortion that should be applied to the input image before being fed into the model.
- `inputImageAmount`: The amount of input images that should be used to compare activations. They will be chosen randomly from the dataset associated with the model. Afterwards, each one will be distorted randomly once and the layer's activation on the original and distorted version will be compared. To calculate the final activation difference, individual differences for single input images will be averaged.

Returns: Information about the input data as well as the average difference in activation, represented as a single scalar value.

`/node/list`

Fetch the percentual differences in activation of all annotated layers within the model and higher-level nodes between original input images and their distorted versions being fed into the model.

Parameters:

- `model`: The name of the model that contains the layer.
- `distortion`: A comma-separated list of the names of all distortions that should be applied to the input images before being fed into the model.

- `inputImageAmount`: The amount of input images that should be used to compare activations. They will be chosen randomly from the dataset associated with the model. Afterwards, each one will be distorted randomly once for each distortion and the layer's activation on the original and distorted version will be compared. To calculate the final activation difference, individual differences for single input images will be averaged.
- `accumulationMethod`: The method that should be used to combine the activation difference values for a set of child nodes into their parent node. Available methods are `minimum`, where the lowest of all values will be used; `maximum`, where the highest of all values will be used; and `average`, where the average of all values will be used.
- `percentageMode`: Defines whether percentages should be calculated as the position of the value on the scale between 0 and the highest activation difference value (`absolute`) or on the scale between the lowest and highest activation difference values (`relative`).
- `imageIndex` *(Optional)*: If this parameter is defined, only node difference values in activations for this specific image within the dataset will be computed. If it is not defined, average node difference values for a set of input images will be computed.
- `outputMode` *(Optional)*: Defines whether the node names and resulting values should be returned as a simple node-value mapping (`mapping`) or as a JSON dictionary hierarchy, representing the computation graph of the model (`graph`). The default value is `mapping`.

Returns: Meta information about the input data and the value range, as well as the percentual activation differences of all annotated layers using all specified distortions.

`/node/list/meta`

Fetch meta information about the percentual differences in activation of all annotated layers within the model and higher-level nodes between original input images and their distorted versions being fed into the model.

Parameters:

- `model`: The name of the model that contains the layer.
- `inputImageAmount`: The amount of input images that should be used to compare activations. They will be chosen randomly from the dataset associated with the dataset. Afterwards, each one will be distorted randomly once for each distortion and the layer's activation on the original and distorted version will be compared. To calculate the final activation difference, individual differences for single input images will be averaged.
- `accumulationMethod`: The method that should be used to combine the activation difference values for a set of child nodes into their parent node. Available methods are `minimum`, where the lowest of all values will be used; `maximum`, where the highest of all values will be used; and `average`, where the average of all values will be used.
- `percentageMode`: Defines whether percentages should be calculated as the position of the value on the scale between 0 and the highest activation difference value (`absolute`) or on the scale between the lowest and highest activation difference values (`relative`).

- `imageIndex` *(Optional)*: If this parameter is defined, only node difference values in activations for this specific image within the dataset will be computed. If it is not defined, average node difference values for a set of input images will be computed.
- `distortion` *(Optional)*: A comma-separated list of the names of all distortions that should be applied to the input images before being fed into the model. If it is not defined, all available distortions are used.

Returns: The minimum and maximum activation difference values among all nodes within the model.

A.1.9 Cache

Depending on the specified parameters, some of the routes are computation-intensive and might take a long time on weaker hardware. To circumvent this problem and allow an interactive visualization, computation results can be cached and persisted to disk. This cache will be extended lazily after a computation has completed. Additionally, these routes allow creating such a cache for all intense operations eagerly and before using the visualization.

/cache

Eagerly cache graph structures, single predictions, prediction accuracies, node difference values, and confusion matrices for all models, datasets, and distortions and write them to disk. Calculations whose results already have been cached will be skipped.

Parameters:

- `modelAccuracy`: The amount of images that should be chosen to calculate each model's prediction accuracy and performance metrics. The higher this value, the more accurate these metrics will be.
- `nodeActivation`: The amount of images that should be chosen to calculate each node's activation differences. The higher this value, the more accurate these difference values will be.

Returns: The amount of seconds that the caching progress has taken, after the caching process has been completed.

/cache/progress

Fetch the current progress of an active caching process, possibly one that has been initiated through the `/cache` route.

Parameters: None.

Returns: The status message of the caching process that is currently in progress, as well as the total number of steps needed for completing the caching, the current step, and the current progress represented as a percentage value.

Bibliography

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "Tensorflow: A System for Large-scale Machine Learning". In: *OSDI*. Vol. 16. 2016, pp. 265–283 (cit. on pp. 26, 49).

[ACD+15] S. Amershi, M. Chickering, S. Drucker, B. Lee, P. Simard, J. Suh. "ModelTracker: Redesigning Performance Analysis Tools for Machine Learning". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM. 2015, pp. 337–346. DOI: 10.1145/2702123.2702509 (cit. on pp. 30–33).

[AHH+14] B. Alsallakh, A. Hanbury, H. Hauser, S. Miksch, A. Rauber. "Visual Methods for Analyzing Probabilistic Classification Data". In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 1703–1712. DOI: 10.1109/tvcg.2014.2346660 (cit. on pp. 30–33, 42, 66).

[AJY+18] B. Alsallakh, A. Jourabloo, M. Ye, X. Liu, L. Ren. "Do Convolutional Neural Networks Learn Class Hierarchy?" In: *IEEE Transactions on Visualization and Computer Graphics* 24 (2018), pp. 152–162. DOI: 10.1109/tvcg.2017.2744683. eprint: 1710.06501v1 (cit. on pp. 30–35).

[BCC+16] M. Bojarski, A. Choromanska, K. Choromanski, B. Firner, L. D. Jackel, U. Muller, K. Zieba. "VisualBackProp: Visualizing CNNs for Autonomous Driving". In: *CoRR* abs/1611.05418 (2016). arXiv: 1611.05418. URL: http://arxiv.org/abs/1611.05418 (cit. on p. 32).

[BKC+13] R. Borgo, J. Kehrer, D. Chung, E. Maguire, R. Laramee, H. Hauser, M. Ward, M. Chen. "Glyph-based Visualization: Foundations, Design Guidelines, Techniques and Applications." In: *Eurographics (STARs)*. 2013, pp. 39–63. DOI: 10.2312/conf/EG2013/stars/039-063 (cit. on p. 17).

[BNS+06] M. Barreno, B. Nelson, R. Sears, A. Joseph, J. D. Tygar. "Can Machine Learning Be Secure?" In: *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*. ACM. 2006, pp. 16–25. DOI: 10.1145/1128817.1128824 (cit. on p. 25).

[Bot10] L. Bottou. "Large-scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186 (cit. on p. 20).

[Bru14] D. Bruckner. *ML-o-Scope: A Diagnostic Visualization System for Deep Machine Learning Pipelines*. Tech. rep. California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, 2014. DOI: 10.21236/ada605112 (cit. on pp. 30–33).

[CGR+17] J. Chae, S. Gao, A. Ramanthan, C. Steed, G. Tourassi. "Visualization for Classification in Deep Neural Networks". In: *Workshop on Visual Analytics for Deep Learning (VADL)*. 2017 (cit. on pp. 30–33).

[CHJO16] S. Carter, D. Ha, I. Johnson, C. Olah. "Experiments in Handwriting with a Neural Network". In: *Distill* (2016). DOI: 10.23915/distill.00004. URL: http://distill.pub/2016/handwriting (cit. on pp. 30–33).

[Cho+15] F. Chollet et al. *Keras*. https://keras.io. 2015 (cit. on p. 27).

[CKF11] R. Collobert, K. Kavukcuoglu, C. Farabet. "Torch7: A MATLAB-Like Environment for Machine Learning". In: *BigLearn, NIPS Workshop*. EPFL-CONF-192376. 2011 (cit. on p. 26).

[CLL+15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *CoRR* abs/1512.01274 (2015). arXiv: 1512.01274. URL: http://arxiv.org/abs/1512.01274 (cit. on p. 26).

[CMMS12] D. CireşAn, U. Meier, J. Masci, J. Schmidhuber. "Multi-column Deep Neural Network for Traffic Sign Classification". In: *Neural Networks* 32 (2012), pp. 333–338. DOI: 10.1016/j.neunet.2012.02.023 (cit. on p. 21).

[CMS99] S. Card, J. Mackinlay, B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999 (cit. on pp. 15, 16).

[CN06] R. Caruana, A. Niculescu-Mizil. "An Empirical Comparison of Supervised Learning Algorithms". In: *Proceedings of the 23rd International Conference on Machine Learning*. ACM. 2006, pp. 161–168. DOI: 10.1145/1143844.1143865 (cit. on p. 13).

[CPMC17] D. Cashman, G. Patterson, A. Mosca, R. Chang. "RNNbow: Visualizing Learning Via Backpropagation Gradients in Recurrent Neural Networks". In: *Workshop on Visual Analytics for Deep Learning (VADL)*. 2017 (cit. on pp. 29, 31, 33).

[CSP+16] S. Chung, S. Suh, C. Park, K. Kang, J. Choo, B. C. Kwon. "ReVACNN: Real-Time Visual Analytics for Convolutional Neural Network". In: *KDD'16 Workshop on Interactive Data Exploration and Analytics*. 2016 (cit. on pp. 30–33).

[CW08] R. Collobert, J. Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning". In: *Proceedings of the 25th International Conference on Machine Learning*. ACM. 2008, pp. 160–167. DOI: 10.1145/1390156.1390177 (cit. on p. 22).

[Dam+16] A. Damien et al. *TFLearn*. https://github.com/tflearn/tflearn. 2016 (cit. on p. 27).

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei. "ImageNet: A Large-scale Hierarchical Image Database". In: *IEEE Conference on Computer Vision and Pattern Recognition, 2009.* IEEE. 2009, pp. 248–255. DOI: 10.1109/cvpr.2009.5206848 (cit. on pp. 23, 41, 58).

[DLH+13] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, et al. "Recent Advances in Deep Learning for Speech Research at Microsoft". In: *ICASSP*. Vol. 26. 2013, p. 64. DOI: 10.1109/icassp.2013.6639345 (cit. on p. 22).

[DSR+15] S. Dieleman, J. Schlüter, C. Raffel, et al. *Lasagne: First Release*. Aug. 2015. DOI: 10.5281/zenodo.27878. URL: http://dx.doi.org/10.5281/zenodo.27878 (cit. on p. 27).

[DY+14] L. Deng, D. Yu, et al. "Deep Learning: Methods and Applications". In: *Foundations and Trends in Signal Processing* 7.3–4 (2014), pp. 197–387. DOI: 10.1561/2000000039 (cit. on pp. 19, 20, 22).

[Faw06] T. Fawcett. "An Introduction to ROC Analysis". In: *Pattern Recognition Letters* 27.8 (2006), pp. 861–874. DOI: 10.1016/j.patrec.2005.10.010 (cit. on p. 22).

[FCSG17] R. Feinman, R. Curtin, S. Shintre, A. Gardner. "Detecting Adversarial Samples from Artifacts". In: *ArXiv e-prints* abs/1703.00410 (Mar. 2017). arXiv: 1703.00410 [stat.ML]. URL: http://arxiv.org/abs/1703.00410 (cit. on p. 80).

[FGR+17] N. Fernandez, G. Gundersen, A. Rahman, M. Grimes, K. Rikova, P. Hornbeck, A. Ma'ayan. "Clustergrammer, a Web-based Heatmap Visualization and Analysis Tool for High-dimensional Biological Data". In: *Scientific Data* 4 (2017), p. 170151. DOI: 10.1038/sdata.2017.151 (cit. on p. 42).

[FH03] E. Frank, M. Hall. "Visualizing Class Probability Estimators". In: *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer. 2003, pp. 168–179. DOI: 10.1007/978-3-540-39804-2_17 (cit. on pp. 29, 31–33).

[FHM09] C. Ferri, J. Hernández-Orallo, R. Modroiu. "An Experimental Comparison of Performance Measures for Classification". In: *Pattern Recognition Letters* 30.1 (2009), pp. 27–38. DOI: 10.1016/j.patrec.2008.08.010 (cit. on pp. 22, 23).

[FVSN08] J.-D. Fekete, J. Van Wijk, J. Stasko, C. North. "The Value of Information Visualization". In: *Information Visualization*. Springer, 2008, pp. 1–18. DOI: 10.1007/978-3-540-70956-5_1 (cit. on p. 16).

[Goo17a] Google. *TensorBoard: Visualizing Learning*. 2017. URL: https://www.tensorflow.org/guide/summaries_and_tensorboard (cit. on pp. 30–33, 47, 50).

[Goo17b] Google Creative Lab. *Teachable Machine*. 2017. URL: https://experiments.withgoogle.com/teachable-machine (cit. on pp. 30, 32, 33).

[GS16] S. Guadarrama, N. Silberman. *TensorFlow-Slim: A Lightweight Library for Defining, Training and Evaluating Complex Models in TensorFlow*. 2016. URL: https://github.com/tensorflow/models/tree/master/research/slim#pre-trained-models (cit. on pp. 53, 55).

[GSS14] I. Goodfellow, J. Shlens, C. Szegedy. "Explaining and Harnessing Adversarial Examples". In: *ArXiv e-prints* abs/1412.6572 (Dec. 2014). arXiv: 1412.6572 [stat.ML]. URL: http://arxiv.org/abs/1607.08022 (cit. on p. 24).

[Har15] A. Harley. "An Interactive Node-Link Visualization of Convolutional Neural Networks". In: *International Symposium on Visual Computing*. Springer. 2015, pp. 867–877. DOI: 10.1007/978-3-319-27857-5_77 (cit. on pp. 30–33).

[Hay94] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1994 (cit. on p. 13).

[HCD12] D. Hoiem, Y. Chodpathumwan, Q. Dai. "Diagnosing Error in Object Detectors". In: *European Conference on Computer Vision*. Springer. 2012, pp. 340–353. DOI: 10.1007/978-3-642-33712-3_25 (cit. on pp. 14, 24).

[HDO+98] M. Hearst, S. Dumais, E. Osuna, J. Platt, B. Scholkopf. "Support Vector Machines". In: *IEEE Intelligent Systems and Their Applications* 13.4 (1998), pp. 18–28. DOI: 10.1109/5254.708428 (cit. on p. 13).

[HDY+12] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97. DOI: 10.1109/msp.2012.2205597 (cit. on p. 19).

[HHC17] F. Hohman, N. Hodas, D. H. Chau. "ShapeShop: Towards Understanding Deep Learning Representations Via Interactive Experimentation". In: *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM. 2017, pp. 1694–1699. DOI: 10.1145/3027063.3053103 (cit. on pp. 30–33).

[HJN+11] L. Huang, A. Joseph, B. Nelson, B. Rubinstein, J. Tygar. "Adversarial Machine Learning". In: *Proceedings of the 4th ACM Workshop on Security and artificial intelligence*. ACM. 2011, pp. 43–58. DOI: 10.1145/2046684.2046692 (cit. on p. 25).

[HKPC18] F. Hohman, M. Kahng, R. Pienta, D. H. Chau. "Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers". In: *IEEE Transactions on Visualization and Computer Graphics* (2018). DOI: 10.1109/tvcg.2018.2843369 (cit. on pp. 29–33).

[How13] A. G. Howard. "Some Improvements on Deep Convolutional Neural Network Based Image Classification". In: *CoRR* abs/1312.5402 (2013). arXiv: 1312.5402. URL: http://arxiv.org/abs/1312.5402 (cit. on p. 24).

[HZC+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861 (cit. on pp. 54, 55, 57).

[HZRS16a] K. He, X. Zhang, S. Ren, J. Sun. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778. DOI: 10.1109/cvpr.2016.90 (cit. on p. 57).

[HZRS16b] K. He, X. Zhang, S. Ren, J. Sun. "Identity Mappings in Deep Residual Networks". In: *European Conference on Computer Vision*. Springer. 2016, pp. 630–645. DOI: 10.1007/978-3-319-46493-0_38 (cit. on pp. 54, 55, 57).

[IBM13] IBM Consumer Products Industry Blog. *2.5 Quintillion Bytes of Data Created Every Day. How Does Cpg & Retail Manage It?* 2013. URL: https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/ (cit. on p. 13).

[ID91] A. Inselberg, B. Dimsdale. "Parallel Coordinates: A Tool for Visualizing Multivariate Relations". In: *Human-Machine Interactive Systems* (1991), pp. 199–233. DOI: 10.1109/visual.1990.146402 (cit. on p. 17).

[IL66] A. G. Ivakhnenko, V. G. Lapa. *Cybernetic Predicting Devices*. Tech. rep. Purdue Univ Lafayette Ind School of Electrical Engineering, 1966 (cit. on p. 19).

[JAF16] J. Johnson, A. Alahi, L. Fei-Fei. "Perceptual Losses for Real-time Style Transfer and Super-resolution". In: *European Conference on Computer Vision*. Springer. 2016, pp. 694–711. DOI: 10.1007/978-3-319-46475-6_43 (cit. on pp. 22, 59).

[Joh14] S. John Walker. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. 2014. DOI: 10.2501/ija-33-1-181-183 (cit. on p. 13).

[JSD+14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM. 2014, pp. 675–678. DOI: 10.1145/2647868.2654889 (cit. on p. 26).

[KAF+08] D. Keim, G. Andrienko, J.-D. Fekete, C. Görg, J. Kohlhammer, G. Melançon. “Visual Analytics: Definition, Process, and Challenges”. In: *Information visualization*. Springer, 2008, pp. 154–175. DOI: 10.1007/978-3-540-70956-5_7 (cit. on pp. 13, 17, 18, 39).

[KAKC18] M. Kahng, P. Andrews, A. Kalro, D. H. P. Chau. “ActiVis: Visual Exploration of Industry-scale Deep Neural Network Models”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 88–97. DOI: 10.1109/tvcg.2017.2744718 (cit. on pp. 30–34).

[Kar14] A. Karpathy. *ConvNetJS: Deep Learning in Your Browser*. 2014. URL: https://cs.stanford.edu/people/karpathy/convnetjs/ (cit. on pp. 30–33).

[KFC16] M. Kahng, D. Fang, D. H. P. Chau. “Visual Exploration of Machine Learning Results Using Data Cube Analysis”. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM. 2016, p. 1. DOI: 10.1145/2939502.2939503 (cit. on pp. 30–33).

[KGB+18] A. Kurakin, I. J. Goodfellow, S. Bengio, Y. Dong, F. Liao, M. Liang, T. Pang, J. Zhu, X. Hu, C. Xie, J. Wang, Z. Zhang, Z. Ren, A. L. Yuille, S. Huang, Y. Zhao, Y. Zhao, Z. Han, J. Long, Y. Berdibekov, T. Akiba, S. Tokui, M. Abe. “Adversarial Attacks and Defences Competition”. In: *CoRR* abs/1804.00097 (2018). arXiv: 1804.00097. URL: http://arxiv.org/abs/1804.00097 (cit. on pp. 23, 58, 73).

[KL83] J. Kruskal, J. Landwehr. “Icicle Plots: Better Displays for Hierarchical Clustering”. In: *The American Statistician* 37.2 (1983), pp. 162–168. DOI: 10.2307/2685881 (cit. on p. 64).

[KMS+08] D. Keim, F. Mansmann, J. Schneidewind, J. Thomas, H. Ziegler. “Visual Analytics: Scope and Challenges”. In: *Visual Data Mining*. Springer, 2008, pp. 76–90. DOI: 10.1007/978-3-540-71080-6_6 (cit. on pp. 17, 18).

[KPN16] J. Krause, A. Perer, K. Ng. “Interacting with Predictions: Visual Inspection of Black-Box Machine Learning Models”. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM. 2016, pp. 5686–5697. DOI: 10.1145/2858036.2858529 (cit. on pp. 29–33).

[KSH12] A. Krizhevsky, I. Sutskever, G. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. 2012, pp. 1097–1105. DOI: 10.1145/3065386 (cit. on pp. 19, 22, 24).

[LB+95] Y. LeCun, Y. Bengio, et al. “Convolutional networks for images, speech, and time series”. In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995 (cit. on p. 21).

[LBH15] Y. LeCun, Y. Bengio, G. Hinton. “Deep Learning”. In: *Nature* 521.7553 (2015), p. 436. DOI: 10.1038/nature14539 (cit. on pp. 19–22).

[LeC98] Y. LeCun. *The MNIST Database of Handwritten Digits*. 1998 (cit. on p. 23).

[LSL+17] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, S. Liu. “Towards Better Analysis of Deep Convolutional Neural Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 91–100. DOI: 10.1109/tvcg.2016.2598831 (cit. on pp. 30–33).

[Mar97] G. Marchionini. *Information Seeking in Electronic Environments*. 9. Cambridge university press, 1997. DOI: 10.1017/cbo9780511626388.003 (cit. on p. 16).

[MCM13] R. Michalski, J. Carbonell, T. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Springer Science & Business Media, 2013. DOI: 10.1007/978-3-662-12405-5 (cit. on p. 13).

[MCZ+17] Y. Ming, S. Cao, R. Zhang, Z. Li, Y. Chen, Y. Song, H. Qu. “Understanding Hidden Memories of Recurrent Neural Networks”. In: *CoRR* abs/1710.10777 (2017). arXiv: 1710.10777. URL: http://arxiv.org/abs/1710.10777 (cit. on pp. 30–33).

[MFFF17] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, P. Frossard. “Universal Adversarial Perturbations”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 86–94. DOI: 10.1109/cvpr.2017.17 (cit. on pp. 14, 24, 25, 56, 59, 79, 80).

[MH08] L. v. d. Maaten, G. Hinton. “Visualizing Data Using t-SNE”. In: *Journal of Machine Learning Research* 9.Nov (2008), pp. 2579–2605 (cit. on pp. 31, 33).

[Mil95] G. Miller. “WordNet: a lexical database for English”. In: *Communications of the ACM* 38.11 (1995), pp. 39–41. DOI: 10.1145/219717.219748 (cit. on pp. 41, 58).

[MMS+17] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, A. Vladu. “Towards Deep Learning Models Resistant to Adversarial Attacks”. In: *ArXiv e-prints* abs/1706.06083 (June 2017). arXiv: 1706.06083 [stat.ML]. URL: http://arxiv.org/abs/1706.06083 (cit. on p. 80).

[MOT15] A. Mordvintsev, C. Olah, M. Tyka. *DeepDream - a Code Example for Visualizing Neural Networks*. 2015. URL: https://ai.googleblog.com/2015/07/deepdream-code-example-for-visualizing.html (cit. on p. 22).

[NH10] V. Nair, G. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 807–814 (cit. on p. 20).

[NKK+11] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, A. Ng. “Multimodal Deep Learning”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 689–696 (cit. on p. 22).

[NQ17] A. Norton, Y. Qi. “Adversarial-Playground: A Visualization Suite Showing How Adversarial Examples Fool Deep Learning”. In: *IEEE Symposium on Visualization for Cyber Security 2017 (VizSec)*. IEEE. 2017, pp. 1–4. DOI: 10.1109/vizsec.2017.8062202 (cit. on pp. 30, 32–35).

[ODO16] A. Odena, V. Dumoulin, C. Olah. “Deconvolution and Checkerboard Artifacts”. In: *Distill* (2016). DOI: 10.23915/distill.00003. URL: http://distill.pub/2016/deconv-checkerboard (cit. on p. 45).

[Ola15] C. Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs (cit. on p. 21).

[OSJ+18] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, A. Mordvintsev. "The Building Blocks of Interpretability". In: *Distill* (2018). https://distill.pub/2018/building-blocks. DOI: 10.23915/distill.00010 (cit. on p. 29).

[PDS+16] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, R. Ward. "Deep Sentence Embedding Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval". In: *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)* 24.4 (2016), pp. 694–707. DOI: 10.1109/taslp.2016.2520371. eprint: 1502.06922v3 (cit. on p. 22).

[PGC+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer. "Automatic Differentiation in PyTorch". In: *NIPS-W*. 2017 (cit. on p. 26).

[PGS+16] N. Papernot, I. Goodfellow, R. Sheatsley, R. Feinman, P. McDaniel. "cleverhans v1.0.0: an adversarial machine learning library". In: *arXiv preprint arXiv:1610.00768* (2016) (cit. on pp. 25, 35).

[PHV+18] N. Pezzotti, T. Höllt, J. Van Gemert, B. Lelieveldt, E. Eisemann, A. Vilanova. "DeepEyes: Progressive Visual Analytics for Designing Deep Neural Networks". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 98–108. DOI: 10.1109/tvcg.2017.2744358 (cit. on pp. 30–33).

[PMW+16] N. Papernot, P. D. McDaniel, X. Wu, S. Jha, A. Swami. "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks". In: *2016 IEEE Symposium on Security and Privacy (SP)* (2016), pp. 582–597. DOI: 10.1109/sp.2016.41 (cit. on p. 80).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on pp. 26, 50).

[Qui86] J. R. Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1.1 (1986), pp. 81–106. DOI: 10.1007/bf00116251 (cit. on p. 13).

[RAL+17] D. Ren, S. Amershi, B. Lee, J. Suh, J. Williams. "Squares: Supporting Interactive Performance Analysis for Multiclass Classifiers". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 61–70. DOI: 10.1109/tvcg.2016.2598828 (cit. on pp. 30–33, 42).

[RC94] R. Rao, S. Card. "The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus + Context Visualization for Tabular Information". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 1994, pp. 318–322. DOI: 10.1145/259963.260391 (cit. on p. 42).

[RDS+15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, L. Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y (cit. on pp. 23, 34, 55, 58, 76).

[RSG16] M. T. Ribeiro, S. Singh, C. Guestrin. "Ẅhy 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*. ACM. 2016, pp. 1135–1144. DOI: 10.1145/2939672.2939778 (cit. on pp. 29, 32, 33).

[SA16] F. Seide, A. Agarwal. "CNTK: Microsoft's Open-source Deep-Learning Toolkit". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2016, pp. 2135–2135. DOI: 10.1145/2939672.2945397 (cit. on p. 26).

[Sch15] J. Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61 (2015), pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003. eprint: 1404.7828v4 (cit. on pp. 19–21).

[SCS+17] D. Smilkov, S. Carter, D. Sculley, F. B. Viégas, M. Wattenberg. "Direct-Manipulation Visualization of Deep Networks". In: *CoRR* abs/1708.03788 (2017). arXiv: 1708.03788. URL: http://arxiv.org/abs/1708.03788 (cit. on pp. 30–33).

[SGPR18] H. Strobelt, S. Gehrmann, H. Pfister, A. Rush. "LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 667–676. DOI: 10.1109/tvcg.2017.2744158 (cit. on pp. 29, 31–33).

[Shn92] B. Shneiderman. "Tree Visualization with Tree-maps: 2-d Space-filling Approach". In: *ACM Transactions on Graphics (TOG)* 11.1 (1992), pp. 92–99. DOI: 10.1145/102377.115768 (cit. on p. 17).

[Shn96] B. Shneiderman. "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations". In: *IEEE Symposium on Visual Languages, Proceedings 1996*. IEEE. 1996, pp. 336–343. DOI: 10.1016/b978-155860915-0/50046-9 (cit. on p. 18).

[SLJ+15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich. "Going Deeper with Convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1–9. DOI: 10.1109/cvpr.2015.7298594 (cit. on pp. 24, 54–56).

[SSSI11] J. Stallkamp, M. Schlipsing, J. Salmen, C. Igel. "The German Traffic Sign Recognition Benchmark: A Multi-class Classification Competition". In: *The 2011 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2011, pp. 1453–1460. DOI: 10.1109/ijcnn.2011.6033395 (cit. on p. 21).

[Ste97] S. Stehman. "Selecting and Interpreting Measures of Thematic Classification Accuracy". In: *Remote Sensing of Environment* 62.1 (1997), pp. 77–89. DOI: 10.1016/s0034-4257(97)00083-7 (cit. on p. 23).

[SVI+16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna. "Rethinking the Inception Architecture for Computer Vision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826. DOI: 10.1109/cvpr.2016.308 (cit. on pp. 54–56).

[SWA01] M. Streeter, M. Ward, S. Alvarez. "NVIS: An Interactive Visualization Tool for Neural Networks". In: *Visual Data Exploration and Analysis VIII*. Vol. 4302. International Society for Optics and Photonics. 2001, pp. 234–242 (cit. on pp. 29–33).

[SZ14] K. Simonyan, A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556 (cit. on pp. 54, 55).

[SZS+13] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, R. Fergus. "Intriguing properties of neural networks". In: *CoRR* abs/1312.6199 (2013). arXiv: 1312.6199. URL: http://arxiv.org/abs/1312.6199 (cit. on pp. 14, 24).

[Ten15] Tensorflow. *Inception 5h Trained Model*. 2015. URL: https://storage.googleapis.com/download.tensorflow.org/models/inception5h.zip (cit. on p. 55).

[TH12] T. Tieleman, G. Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4.2 (2012), pp. 26–31 (cit. on p. 77).

[The16] Theano Development Team. "Theano: A Python Framework for Fast Computation of Mathematical Expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688 (cit. on p. 26).

[TLKT09] J. Talbot, B. Lee, A. Kapoor, D. Tan. "EnsembleMatrix: Interactive Visualization to Support Machine Learning with Multiple Classifiers". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2009, pp. 1283–1292. DOI: 10.1145/1518701.1518895 (cit. on pp. 30–33).

[TOHC15] S. Tokui, K. Oono, S. Hido, J. Clayton. "Chainer: a Next-Generation Open Source Framework for Deep Learning". In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*. 2015. URL: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf (cit. on p. 26).

[UVL16] D. Ulyanov, A. Vedaldi, V. S. Lempitsky. "Instance Normalization: The Missing Ingredient for Fast Stylization". In: *CoRR* abs/1607.08022 (2016). arXiv: 1607.08022. URL: http://arxiv.org/abs/1607.08022 (cit. on pp. 22, 59).

[VL15] A. Vedaldi, K. Lenc. "MatConvNet – Convolutional Neural Networks for MATLAB". In: *Proceeding of the ACM Int. Conf. on Multimedia*. 2015. DOI: 10.1145/2733373.2807412 (cit. on p. 26).

[VV99] J. Van Wijk, H. Van de Wetering. "Cushion Treemaps: Visualization of Hierarchical Information." In: *InfoVis*. Vol. 99. 1999, pp. 73–78. DOI: 10.1109/infvis.1999.801860 (cit. on p. 17).

[WGYS18] J. Wang, L. Gou, H. Yang, H.-W. Shen. "GANViz: A Visual Analytics Approach to Understand the Adversarial Game". In: *IEEE Transactions on Visualization and Computer Graphics* 24.6 (2018), pp. 1905–1917. DOI: 10.1109/tvcg.2018.2816223 (cit. on pp. 31–33).

[WSW+18] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. Viégas, M. Wattenberg. "Visualizing Dataflow Graphs of Deep Learning Models in Tensorflow". In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 1–12. DOI: 10.1109/tvcg.2017.2744878 (cit. on pp. 32, 53).

[WZZ+13] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, R. Fergus. "Regularization of Neural Networks Using Dropconnect". In: *International Conference on Machine Learning*. 2013, pp. 1058–1066 (cit. on p. 23).

[YCN+15] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, H. Lipson. “Understanding Neural Networks Through Deep Visualization”. In: *CoRR* abs/1506.06579 (2015). arXiv: 1506.06579. URL: http://arxiv.org/abs/1506.06579 (cit. on pp. 29–33).

[ZBM16] T. Zahavy, N. Ben-Zrihem, S. Mannor. “Graying the Black Box: Understanding DQNs”. In: *International Conference on Machine Learning*. 2016, pp. 1899–1908 (cit. on pp. 29–33).

[ZCAW17] L. M. Zintgraf, T. S. Cohen, T. Adel, M. Welling. “Visualizing Deep Neural Network Decisions: Prediction Difference Analysis”. In: *CoRR* abs/1702.04595 (2017). arXiv: 1702.04595. URL: http://arxiv.org/abs/1702.04595 (cit. on p. 32).

[ZF14] M. Zeiler, R. Fergus. “Visualizing and Understanding Convolutional Networks”. In: *European Conference on Computer Vision*. Springer. 2014, pp. 818–833. DOI: 10.1007/978-3-319-10590-1_53 (cit. on p. 32).

[ZHP+17] H. Zeng, H. Haleem, X. Plantaz, N. Cao, H. Qu. “CNNComparator: Comparative Analytics of Convolutional Neural Networks”. In: *CoRR* abs/1710.05285 (2017). arXiv: 1710.05285. URL: http://arxiv.org/abs/1710.05285 (cit. on pp. 30–34, 36).

[ZXZ+17] W. Zhong, C. Xie, Y. Zhong, Y. Wang, W. Xu, S. Cheng, K. Mueller. “Evolutionary Visual Analysis of Deep Neural Networks”. In: *ICML Workshop on Visualization for Deep Learning*. 2017 (cit. on pp. 30–33).

All links were last followed on August 17, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature