EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät

Computergrafik

Masterarbeit

NeuralVisUAL: Deep Neural Network Visualization in the Unreal Engine for Interactive Fly-through Exploration

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Computergrafik
Jannik Hofmann, jannik.hofmann@student.uni-tuebingen.de, 2021

Bearbeitungszeitraum: 01.03.2021 - 30.09.2021

Betreuer/Gutachter: Prof. Dr. Hendrik Lensch, Universität Tübingen
Zweitgutachter: Prof. Dr. Martin V. Butz, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Jannik Hofmann (Matrikelnummer 3959967), October 5, 2021

Zusammenfassung

Tiefe neuronale Netze verfügen über außergewöhnliche Fähigkeiten, die in vielen Bereichen die des menschlichen Geistes übertreffen. Aufgrund ihrer zunehmenden Komplexität, mit Hunderten Millionen, teilweise sogar Milliarden trainierbarer Parameter in modernen Architekturen, ist es praktisch unmöglich, intuitiv zu verstehen, wie genau diese Algorithmen zu solch bemerkenswerten Ergebnissen kommen. Die Forschungsbereiche der Interpretierbarkeit neuronaler Netze und der Netzwerkvisualisierung zielen darauf ab, einen besseren Einblick in die inneren Prozesse solcher künstlicher Intelligenzen zu erlangen.

NeuralVisUAL (Deep **Neural** Network **Vis**ualization in the **U**nreal Engine for Inter**a**ctive Fl**y**-through Exploration) ist der erste Schritt in Richtung eines erweiterbaren Frameworks für die Visualisierung neuronaler Netze, der ein besseres Verständnis dieser Technologien durch freie Erkundung ermöglicht. Es handelt sich um eine modulare Open-source-Anwendung, die in Python, C++ und Unreal-Engine-Blueprints geschrieben ist und jedes in TensorFlow und Keras entwickelte, tiefe neuronale Feed-Forward-Netz visualisiert. NeuralVisUAL nutzt den kräftebasierten Algorithmus ForceAtlas 2 mit einigen Erweiterungen, um ein sinnvolles zweidimensionales Layout des gegebenen Netzwerks zu erstellen. Dieses Layout bestimmt, wo es Objekte in einer virtuellen Spielumgebung erzeugt, welche die Benutzer frei erkunden und nutzen um mit dem Netzwerk zu interagieren. Darüber hinaus visualisiert die Anwendung die Kernel von Faltungsschichten in faltenden neuralen Netzen, die entsprechenden Kernelaktivierungen, Salienzkarten und integrierte Gradienten, entsprechend den nutzer-spezifischen Präferenzen.

NeuralVisUAL besteht aus verschiedenen individuellen Modulen, die durch präzise definierte Schnittstelleninteraktionen miteinander verbunden sind. Unter anderem ermöglicht dieser Ansatz eine Trennung zwischen einem Server, der mit den neuralen Netzen interagiert, und einem Unreal Engine 4-Client, der die Visualisierung für die Benutzer zur freien Erkundung rendert.

Abstract

Deep neural networks have astounding capabilities, surpassing human abilities in many disciplines. Due to their increasingly complex nature, with modern architectures consisting of hundreds of millions, sometimes even billions of trainable weights, it is virtually impossible for researchers to intuitively understand how exactly these networks come to produce such incredible results. Research in neural network interpretability and network visualization aims to provide more insight into the inner workings of artificial intelligence.

NeuralVisUAL (Deep **Neural** Network **Vis**ualization in the **U**nreal Engine for Inter**a**ctive Fl**y**-through Exploration) is the first step towards an extensible framework for neural network visualization, facilitating a better understanding of these networks through exploration. It is a modular open-source application, written in python, C++, and Unreal Engine blueprints, that visualizes any feed-forward deep neural network developed in TensorFlow and Keras. NeuralVisUAL utilizes the force-based algorithm ForceAtlas 2 with some modifications to calculate a meaningful layout of the given network on a two-dimensional plane. This layout determines where to spawn objects in a virtual game environment, which the user can freely explore, interacting with the network through this application. Furthermore, the application visualizes the kernels of convolution layers in convoluational neural networks, the corresponding activation maps, saliency maps, and integrated gradients according to user-defined preferences.

NeuralVisUAL consists of several distinct modules connected by precisely defined interface interactions. Among other advantages, this allows for a separation between a server interacting with the neural network and an Unreal Engine 4 client that renders the visualization for the user to explore freely.

Acknowledgments

First of all, thank you very much Prof. Dr.-Ing. Hendrik Lensch for giving me the opportunity to work on such an exciting project.

I also thank Prof. Dr. Martin Butz for showing such interest in my work and for taking the time to review my thesis.

Mark Boss, thank you so much for your amazing guidance as my supervisor, patiently supporting me in our meetings, and giving me great advice on how to tackle this project.

Furthermore, thanks to Julian Wanner, Florian Martin and all the other people who took the time to patiently answer my questions and give me feedback throughout this project.

In general, I am truly thankful to my family, friends and my girlfriend Sharon, the great people in my life that encourage me and help me overcome difficult times. I will always be grateful for your extraordinary support! Without you, I would not be where I am today.

Contents

List of abbreviations

AI	artificial intelligence
ML	machine learning
SLP	single layer perceptron
MLP	multi layer perceptron
DNN	deep neural network
RNN	recurrent neural network
FBA	force-based graph drawing algorithm
FA2	ForceAtlas2
UE4	Unreal Engine 4
VR	virtual reality
3D	3-dimensional
2D	2-dimensional
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
GAN	generative adversarial network
RGB	red, green, blue

1 Introduction

deep neural network (DNN)s have an extraordinary proficiency of processing data in a sophisticated manner, often surpassing human abilities [SS18, LWL+17, BMR+20, RDS+15]. Research in the understanding of artificial intelligence (AI) has advanced significantly in the last years and decades; however, some aspects of DNNs, of how exactly they produce these impressive results, are still not understood completely [ZZ18, ZGCH21, SWM17]. Various visualization tools aid in advancing AI research, giving developers an increased understanding of the complex processes within these networks [ZZ18, ZGCH21].

NeuralVisUAL (Deep **Neural** Network **Vis**ualization in the **U**nreal Engine for Inter**a**ctive F**l**y-through Exploration) is a modular DNN visualization application that provides the first step towards an extensible visualization solution, facilitating more direct interaction with the internal mechanisms of neural networks. NeuralVisUAL is open-source[1] and licensed under the GNU General Public License v3.0.

The application analyzes existing feed-forward DNNs, retrieving information like architecture, data of convolution kernels, or neuron activations for chosen example inputs. NeuralVisUAL uses a FBA to position the network's layers to give the user a meaningful virtual representation of the network's structure. Furthermore, its python server module uses the collected information to generate visualizations, such as kernels, activation maps, saliency maps, and integrated gradients. NeuralVisUAL then sends the visualization results to its client module, which processes the data and displays it in an explorable virtual world within UE4, as figure 1.1 shows.

The primary goal of NeuralVisUAL is to provide an intuitive visualization, allowing the user to effortlessly grasp the visual representation of the chosen neural network without needing to interpret raw numerical data.

1.1 Problem Statement

Modern DNNs easily surpass humans in their abilities in many disciplines, such as object recognition with CNNs designed for computer vision [RDS+15], natural language processing with OpenAI's GPT-3-network [BMR+20], lip-reading from

[1] The code for NeuralVisUAL is available for download from https://github.com/cgtuebingen/jannik-hofmann-master-thesis

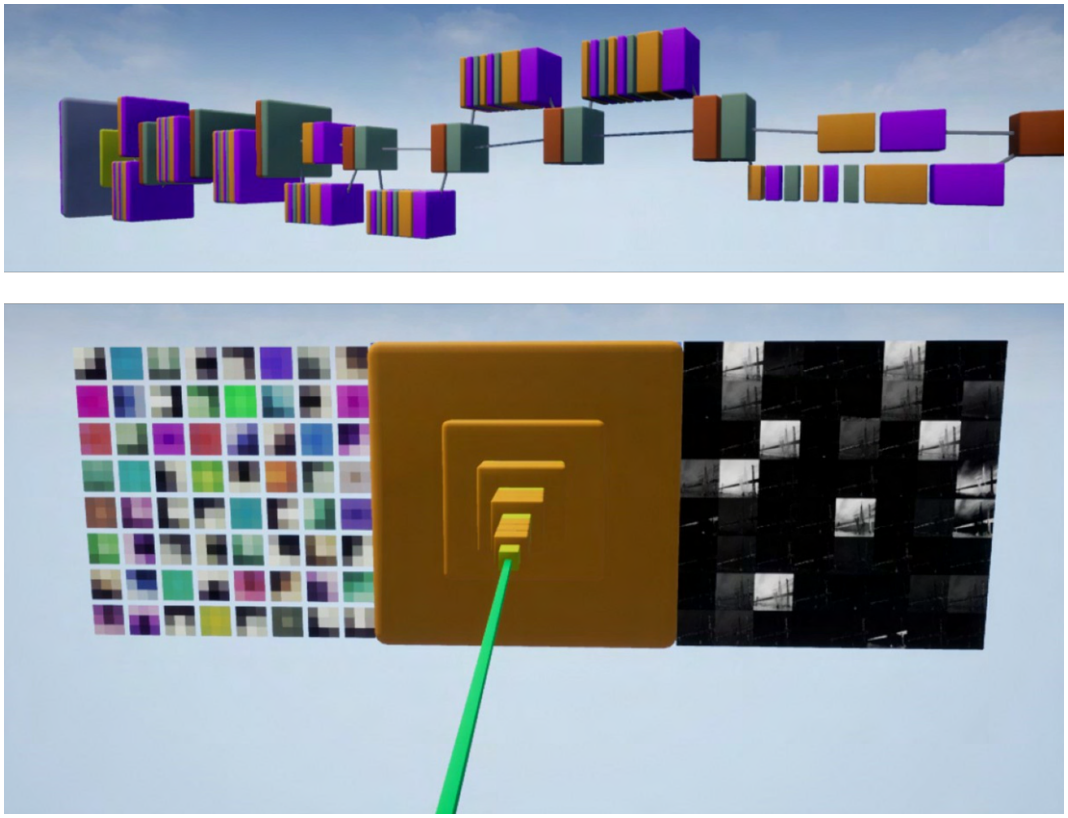

Figure 1.1: Screenshots of NeuralVisUAL, visualizing Resnet-50 (top) and VGG-16 with kernels and activation map of the first convolutional layer (bottom).

video [ASWDF16], controlling their environment from high-dimensional sensory inputs [MKS+15], speech recognition [GMH13], or cancer screening [HBA+19].

However, this highly advanced capability comes with the cost of increased complexity. State of the art DNNs consist of hundreds of millions, or even billions of trainable weight parameters[SZ14, BMR+20, SMM+17]. While modern computers possess the computational power to train and utilize such complex architectures, they become increasingly challenging for humans to grasp fully, especially as it is still not fully understood how exactly DNNs produce such remarkable results [ZZ18, ZGCH21, SWM17].

The research fields of neural network interpretability and network visualizations aim to give developers and researchers a deeper, more complete, intuitive understanding of the inner workings of such DNNs [ZZ18]. Neural network visualizations play a big part in making this information accessible and explorable to researchers and developers, providing more direct interaction with the processes inside these complex algorithms [ZZ18].

Without a profound understanding of the network, it is challenging to debug problems or assess why the network does not produce the desired results [ZZ18, SWM17]. A missing understanding of these processes can lead to decisions based on wrong results [SWM17]. In fields like the medical domain, such issues can ultimately lead to immensely harmful consequences for humans [SWM17]. Explainability in AI is also important for algorithmic fairness, which is crucial to make machine learning (ML) socially sustainable [WCD+18, BH19, NYC19]. Development of neural networks usually does not attempt to facilitate this deep, intuitive understanding, instead it primarily focuses on analytically improving evaluation metrics of neural networks [SSWR18, LS18, nip17, NYC19]. Furthermore, a more profound comprehension helps to overcome bottlenecks in AI research, leading to scientific breakthroughs from novel approaches [ZZ18, NYC19].

Additionally, interactively explorable visual representations of neural networks can be of great use in the educational sector. They can support explaining the mechanisms of DNNs to people unfamiliar with AI, creating an understanding of this usually obscure technology, meanwhile sparking interest and fascination for AI research.

This thesis presents NeuralVisUAL, a modular application for visualizing DNNs that process images as input, utilizing UE4- This application allows users to freely explore feed-forward DNNs, facilitating a more extensive comprehension of complex networks through visualizations.

NeuralVisUAL is the first step towards an extensible solution to further the research into neural network visualization and interpretability, aiding in DNN development, and supporting educational purposes. It can help AI developers reduce the guesswork of how the DNN architecture can be improved and what layers might not be helpful for the desired solution. Furthermore, NeuralVisUAL could also be useful in the educational sector to make AI more tangible through a simple, intuitive visualization.

1.2 Related Work

The research fields of neural network visualizations and network interpretability already have various helpful tools at their disposal to visualize the processes in DNNs and aid in AI development. This section gives an overview of such visualization techniques and frameworks that can be utilized for such purposes, reviewing their advantages and drawbacks.

The most direct method to explore the visual processes within CNN layers is to visualize the kernels [ZZ18]. Gradient-based methods achieve that by calculating a gradient of the analyzed unit, for example, a kernel, over a specific input [ZZ18, ZF14, SZ14, SDBR14, MV15, STY17]. These gradient-based methods include computing saliency maps on a specific input image, and prediction class [SVZ13] and integrated gradients to visually attribute the network results to image features [STY17]. I-GOS utilizes integrated gradients to generate an optimized heatmap correlating with the

influence each image area has on the network's prediction decision, thus generating a visualization that is intuitive to humans while representing the network's decision process [QKL19].

Gradient-based methods are valuable to visualize the features of the given input that certain parts of the network recognize [OMS17, ZZ18]. Furthermore, up-convolutional neural networks can generate images from CNN feature maps, giving a visual representation to these maps that are usually hard to interpret [DB16, NCB+17]. Additionally, Zhou et al. [ZKL+14] demonstrate that object detectors arise naturally within CNNs, making it possible to implement and visualize object localization with networks originally trained for scene recognition.

As an alternative approach, explanatory graphs and decisions trees can disentangle CNN features into interpretable visualizations, providing more intuitive explanations [ZZ18, ZCS+18, ZYMW19]. Explanatory graphs hierarchically visualize the semantic knowledge within CNNs by displaying various patterns that activate the individual convolution kernels [ZCS+18, ZZ18]. On the other hand, decision trees attempt to visually explain how exactly the CNN came to a specific output decision using feature representations, giving users information about how the objects in the input image influenced the network's prediction [ZYMW19].

TensorBoard is an open-source visualization tool that accompanies TensorFlow [AAB+16]. It enables developers to visualize the architecture of TensorFlow networks, show various performance metrics and diagrams of weights, biases and tensors over the training duration, visualize embeddings, among other features [AAB+16]. While it does not provide a 3-dimensional (3D) virtual world, that users can interactively explore, TensorBoard has powerful features to analyze neural networks [AAB+16] and can also render images of kernels, activation maps, and saliency maps, as the interactive explAIner framework demonstrates [SSSEA19].

The analytic results of neural network analysis can be displayed in interactive ways, using tools such as Jupyter notebooks [Jup17], Zeppelin notebooks [Fou17], Kibana [B.V15], Grafana [Lab20], or Tableau [Sof03]. However, these visualizations usually serve to display the analytical information instead of providing an explorable virtual environment [NDB+19].

While not showing the whole network structure, the interactive software of Yosinski et al. [YCN+15] visualizes activation maps of a selected layer, deconvolution of a selected kernel, synthetic input by gradient ascent for that kernel, top correlating training examples and their deconvolution. An especially notable feature of this software is that it can use the frames from a streaming video as input image and dynamically display the changes within the network in realtime [YCN+15].

TensorSpace.js is a browser-based open-source framework to build and visualize TensorFlow [AAB+16] and Keras [Tea20] models in an interactive 3-dimensional environment [Ten19]. It provides intuitive visualizations and can display intermediate layer output [Ten19].

Harley developed an interactive visualization of a handwritten digit classification CNN trained on the MNIST dataset [LBBH98] [Har15]. This application allows the user to paint a new digit and explore the activation patterns of this input in a 3D virtual environment with a high degree of real-time interactivity [Har15]. However, it is limited to visualizing CNNs designed for handwritten digit recognition.

NeuralVis, developed by Zhang et al. in 2019, is a web-based, interactive visualization for DNNs [ZYF+19]. It can visualize network structures, show how the network transforms input data, implement adversarial attack algorithms, and allows users to compare the output of intermediate layers [ZYF+19]. Additionally, the developers evaluated efficiency and efficacy and found out that the selected input has a significant effect on these qualities [ZYF+19].

Although these interactive exploration environments provide various valuable features, the frameworks TensorSpace, Harley's visualization, and NeuralVis are missing more advanced visualizations like saliency maps or integrated gradients [Ten19, Har15, ZYF+19].

Another example of interactive neural network visualizations is the application by Bock and Schreiber [BS18]. It displays the layers and individual neurons of dense layers as a virtual reality (VR) application in Unreal Engine, as shown with the mnist-network [LBBH98] as an example [BS18]. The application provides a relatively simple visualization and works best on small CNNs without residual connections, with few weights in 2-dimensional (2D) dense layers; features such as visual kernel representations, layer connections, or saliency maps are not present [BS18].

The Unity VR application of VanHorn et al. [VZC19] instead lets users interactively develop deep CNNs for image classification, visualizing layer properties, activation maps. It focuses on providing an intuitive environment, that lets users visually build and train such networks on custom datasets, enabling rapid prototyping without writing any code, while giving users valuable insight into the models [VZC19].

CNNVis [LSL+16] visualizes feed-forward deep CNNs in a 2D directed acrylic graph, similar to the layout style of NeuralVisUAL that preserves directional information flow as explained in section 3.2.3. The application displays the network's structure in a hybrid visualization with neurons and neuron interactions, clustering neurons and utilizing hierarchical rectangle packing, matrix reordering and edge bundling algorithms to clear up clutter [LSL+16]. Visualizations for neuron clusters include learned features of neurons, neuron activations, and influence on the network output [LSL+16]. While not taking place in a 3D, virtual environment, CNNvis helps users to diagnose errors and improve their networks with powerful features [LSL+16].

2 Background

This section gives an overview of the background knowledge relevant for NeuralVisUAL.

After a quick introduction to Artificial Intelligence and Machine Learning, section 2.1 an overview of the historical and conceptual basis of DNNs, CNNs in Computer Vision, and information about residual connections and recurrent neural network (RNN)s. Section 2.2 discusses the visualization techniques that NeuralVisUAL utilizes. Finally, section 2.3 introduces FBAs and specifically FA2, which NeuralVisUAL uses to calculate the network layout.

2.1 Neural Networks

The goal of AI is to simulate or reproduce intelligent behavior, often measured in comparison to the intellectual capabilities of human thinking [Goe91, Dob12, Gar17, DW20]. This behavior usually represents itself within intelligent agents that independently adapt to a new environment or varying goals and reach intelligent decisions based on past experience and limited perception [PMG98]. However, the definitions of AI vary widely because the discipline of AI consists of various fields of research, each with its distinct interpretation of these abstract terms. It is difficult to pinpoint what exactly characterizes human intelligence and how to reproduce it. [Fet90, Wan19]

Researchers can create AI systems by utilizing logical approaches [Nil91], bayesian belief systems [AOJJ89], explicitly defined expert systems [Wei76] or ML [Mit97]. Machine learning takes place when a program can produce improved results in a class of tasks with increasing experience [Mit97].

The field of ML consists of supervised, unsupervised, and reinforcement learning. Supervised learning trains with input data labeled with a correct result, allowing the algorithm to compare its performance to the desired outcome. Unsupervised learning happens without labels, forcing the algorithm to find a structure and patterns within the training data. Reinforcement learning happens within a dynamic environment that provides reward-based feedback to the algorithm's actions. [Li17]

Neural networks provide a very prevalent method to implement ML and form the basis of most modern AI research [Li17, HK19].

Due to big data and available processing power, AI is becoming a central part of the modern digital world [HK19]. Recent breakthroughs cover the fields of image recognition [DBK+20, RDS+15, SZ14], speech recognition [GMH13, YL17], natural language processing [BMR+20, SMM+17, SHD14, YHPC18], self-driving cars [GTCM20], vaccine development [VJKH20], protein folding [JEP+21, CLT+18], image and video synthesis [RPG+21, PLWZ19, SSKS17], among other disciplines [SS18].

Computer vision aims to analyze and understand image and video data, modeling the human vision system and retrieving semantic properties for various scientific domains [H+96]. It is closely tied to AI, as most state-of-the-art computer vision algorithms rely on ML to achieve these goals [KRSB18]. Alongside natural language processing, computer vision is one of the typical use cases for deep CNNs [KRSB18, OMK20].

One of the most prominent challenges in computer vision is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which is quantitatively measuring the performance of algorithms for object detection and image classification, based on the ImageNet dataset [RDS+15]. New architectures of neural networks regularly improve on the previous performance, rapidly advancing the state of the art [RDS+15]. Human perception capabilities in computer vision were surpassed by ML algorithms in 2015 [HZRS16, Mar15]. Examples of prominent approaches for networks that achieved remarkable results in the ILSVRC include AlexNet [KSH12], VGGNet [SZ14], GoogLeNet [SLJ+15], resnet [HZRS16], and DenseNet [HLVDMW17], among others [MAK17].

2.1.1 Feedforward and Recurrent Neural Networks

Feedforward architecture implies that the information from any layer exclusively affects later layers and cannot influence any layer before the current one [Kub99, BOU12]. Therefore, a feedforward network can be represented as a directed acyclic graph [SRK95].

In contrast to feedforward-style neural networks, RNNs have cyclic properties and permit the information to flow in any direction between the layers, including from one layer to a preceding layer, as figure 2.1 demonstrates [Gol17, GMH13, GJ14]. If an RNN is executed with a fixed number of iterations, it can be unrolled into a feedforward network, so that each time step feeds into the next one instead of the network feeding back into itself [Gol17, She18].

The internal state of an RNN acts as a memory that permits them to remember information from previous inputs and view the input in a broader context, often utilizing structures specifically designed for that purpose like LSTM or gated recurrent units [Gol17, GMH13].

RNNs utilize sequential or time-series data, which makes them helpful against

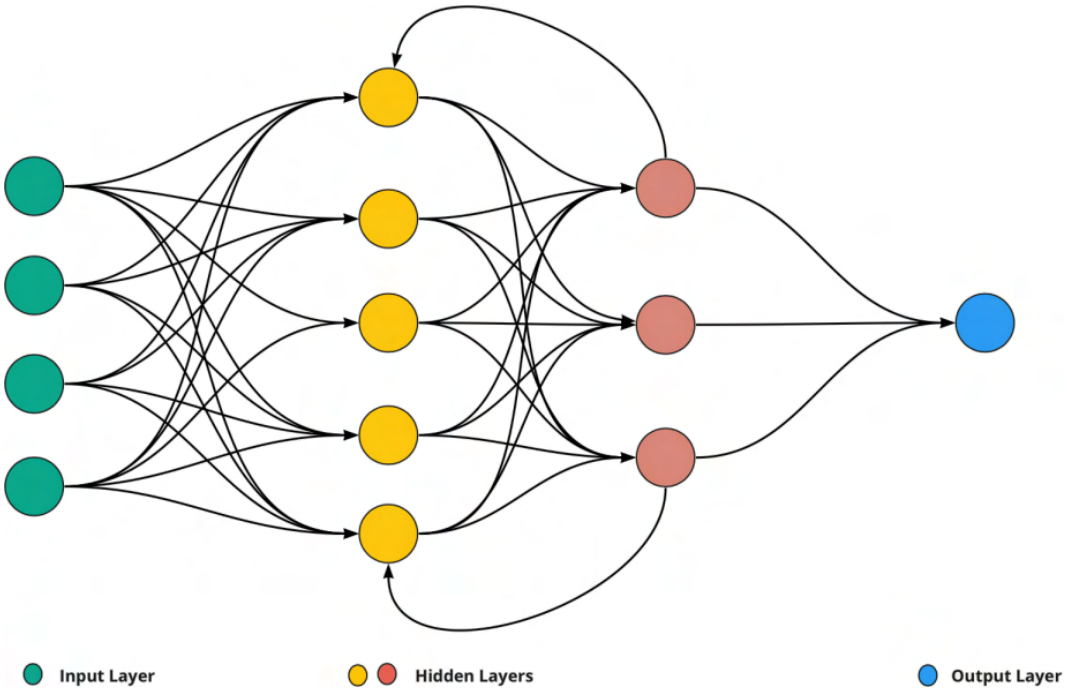

Figure 2.1: Example architecture of a simple RNN, figure taken from [Das20].

temporal problems such as translation, speech recognition, and natural language processing [Gol17, GMH13, GJ14, AGQZ13].

2.1.2 Backpropagation

Feedforward-style neural networks usually train in a supervised manner using backpropagation to optimize their weights towards producing more accurate predictions. This technique incrementally adjusts the parameters towards an optimal state utilizing gradient descent and therefore requires differentiable activation functions throughout the network. Backpropagation serves to make minor adjustments in the network's weights towards the desired outcome of the labeled input data, using relatively large amounts of labeled training data. The changes apply in intensity according to the specified learning rate, which can change dynamically throughout the learning process, depending on the utilized algorithm [YCC95]. This approach strengthens correct predictions while weakening parameters that lead to wrong output. [YEK02, RDGC95, HN92]

Backpropagation calculates the loss function of the predicted output. Such a loss function could, for example, be the mean squared error or cross-entropy between the network's calculated result compared to its desired output for the given input. This loss then propagates back throughout the network towards the input layer, using the derivative of the activation functions and adjusting each weight according

to a minimization algorithm like gradient descent to reduce the loss for the given training example. [Ama93, HN92]

2.1.3 Single layer perceptron

The single layer perceptron (SLP) is the most straightforward, basic idea of a neural network. It was developed by Rosenblatt in 1958 based on biological neurons in the brain, attempting to imitate natural thinking processes. [Ros57, Ros61, nnh16, Fri17]

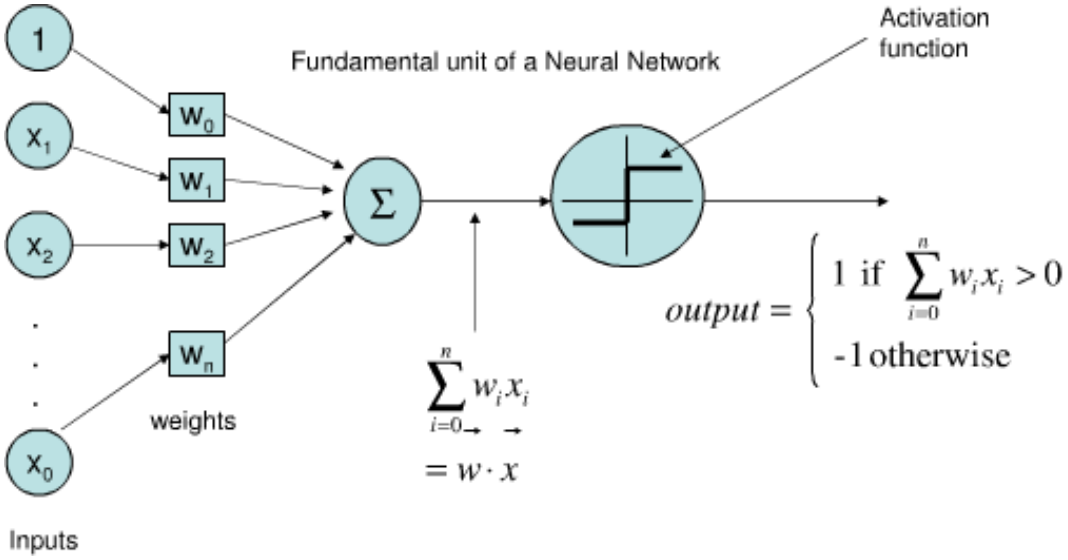

Figure 2.2: Structure of an SLP, figure taken from [VS15].

Figure 2.2 shows the structure of this SLP. It consists of a singular layer and utilizes a vector-matrix multiplication of the input variables with the network's weight data. The weight data within that singular layer determines the perceptron's decision capabilities. The output of the perceptron calculates a weighted sum of the input data plus a bias. [Ros57, DBH92, VS15]

The perceptron trains itself in a supervised manner on labeled data with the delta rule. This approach represents a gradient descent, allowing the network to optimize itself incrementally and move towards a more optimal performance with each training step. Without an activation function, the SLP, therefore, acts as a logistic regression model. [Fri17, DBH92]

Due to only containing one singular layer, the SLP can exclusively represent linearly separable patterns. For example, it is unable to learn a pattern that includes an *exclusive-or* combination of two input variables. [nnh16, DBH92]

Furthermore, the SLP uses activation functions to provide an easily interpretable result. A unit step function provides a distinct, binary result that linearly separates

the different input combinations for classification problems. [Ros61, DBH92]

2.1.4 Multi layer perceptron

In contrast to the SLP, an multi layer perceptron (MLP) contains at least one hidden layer with additional neurons. Due to hidden layers, the network can solve problems that are not linearly separable, making the perceptron significantly more capable. With their multiple layers, MLPs form a basic feedforward network. Even more complex and advanced neural network architectures conceptually originate from MLPs. [Kub99, BOU12, Mac16]

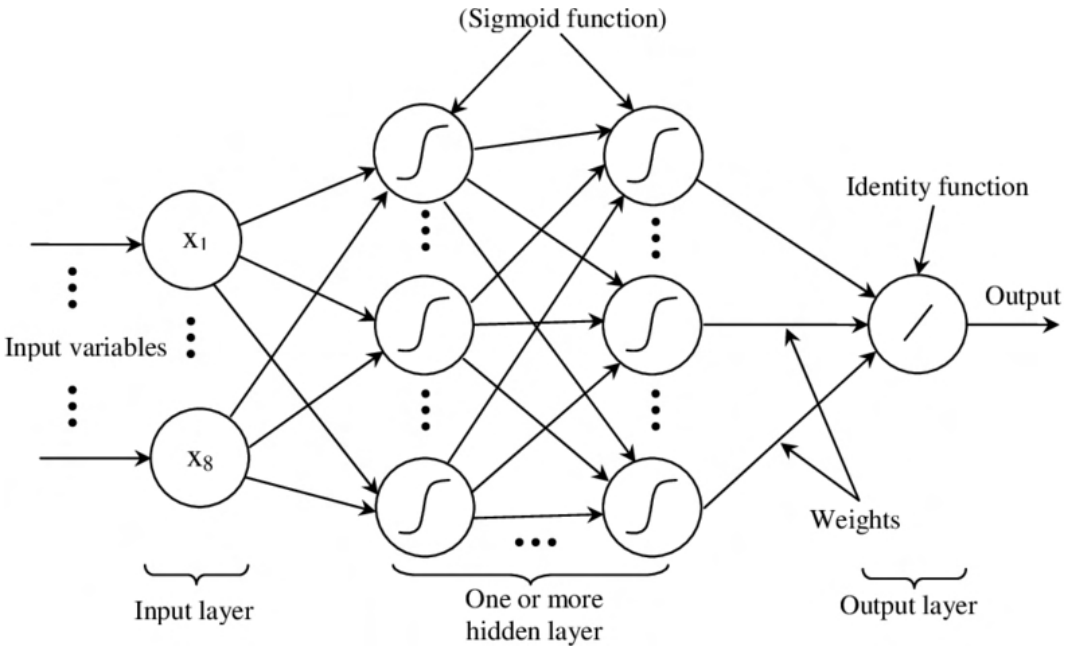

Figure 2.3: Structure of an MLP, figure adapted from [BOU12].

These hidden layers, and the output layer, are fully connected, dense layers. This connection means that per layer, each combination of layer input values with the output neurons comprises its individual weight. Therefore, dense layers in large networks with a relatively high number of neurons can result in an exceedingly high parameter count. [Kub99, BOU12]

Activation functions

In an MLP, each node that is not part of the input layer has a nonlinear activation function, as using linear activation functions would transform the network into a linear regression model, limiting its usefulness [SS17]. For example, typical activation functions include sigmoid, relu (rectified linear unit) or leaky relu functions

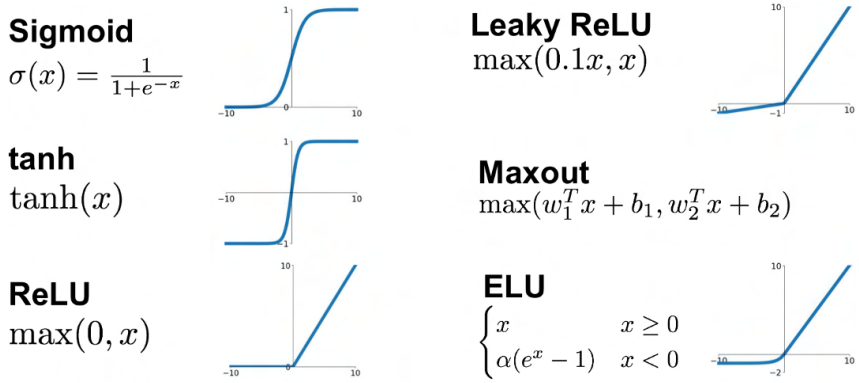

Figure 2.4: Different activation functions for neural networks, figure taken from [Jad18].

[SS17, Jad18], as figure 2.4 visualizes. Maxout selects the maximum of different values [GWFM+13]. Sigmoid and tanh are differentiable functions that give a smooth gradient [SS17]. Relu is more trivial to calculate and therefore computationally more efficient than sigmoid or tanh, while also reducing vanishing gradients [NH10, SS17]. Finally, leaky relu and exponential linear unit are similar to relu but preserve a slight gradient for negative values with aids in training to avoid dead neurons [MHN+13, SS17].

2.1.5 Convolutional neural networks

Convolutional neural networks, developed by Yann LeCun in 1995 [LB+95], incorporate convolutional layers, pooling layers, and relu activations, attempting to resemble the neurons in the visual cortex [ESYF+20, Fuk07].

Convolutional layer

In such a convolutional layer, a kernel convolves over the input data. The kernel has specific dimensions that define the shape of its weights. This layer slides the kernel over the input array instead of densely connecting every input neuron with every output neuron. This approach allows the convolutional layer to have significantly fewer weights than a fully connected layer. The kernel's width and height define its receptive field, the size of the region influencing a singular output neuron. [LB+95, ESYF+20, DV16, Yin19]

The kernel starts in one corner of the input array, processing all input neurons of the receptive field. The kernel multiplies each neuron of that area with its corresponding weight and sums up the products as the output for that kernel position. Subsequently, the kernel horizontally moves its position by the layer's stride, as defined in the network's architecture, and repeats the process for the new receptive field, writing

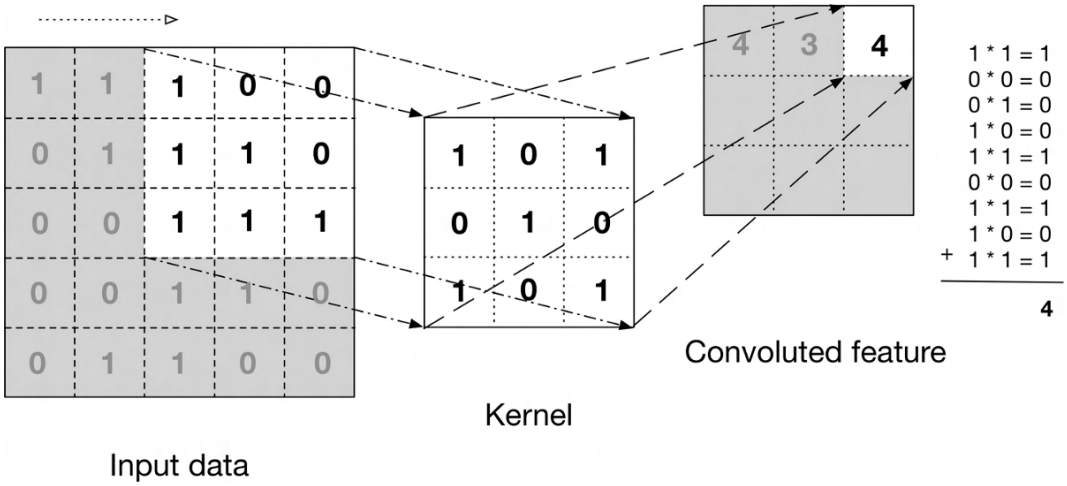

Figure 2.5: Application of a 3×3 convolution kernel without padding and stride 1, figure taken from [Yin19].

the result in the consecutive output neuron. After finishing one row of the input data, the kernel starts at the beginning of the next row, shifting its position according to the stride. Figure 2.5 illustrates this process. The kernel can use padding to extend the input layer to maintain the dimensions of the input layer in the output layer. [ESYF+20, DV16, Yin19]

For example, VGGNet only utilizes 3×3-kernels in its convolutional layers. However, by stacking multiple of these convolutional layers on top of each other, VGGNet achieves a much larger effective receptive field to process the input images, resulting in a deep but straightforward CNN architecture. [SZ14]

Furthermore, alternative kernel designs allow for different operations that do not depend on the multiplication operation, which can be especially useful on less powerful hardware [CWX+20, ECS+21, RORF16, Mog20].

Kernels can also have additional dimensions like depth that cover a multi-dimensional input structure. In computer vision, the kernels usually do not convolve over the depth but rather cover this dimension completely with a kernel of the same depth. This strategy is beneficial when processing a colored red, green, blue (RGB)-image input to the CNN, as figure 2.6 illustrates. Additional kernels with individual weights can convolve over the same data, resulting in output data with a depth larger than one. These kernels slide over the input data in the same manner, but their distinct weights produce different results for each depth dimension of the output. [DV16, Yin19]

Convolutional layers have several advantages over fully connected dense layers. Foremost, due to its sparse nature, a convolutional layer has significantly fewer parameters, making training more efficient due to preventing overfitting. This

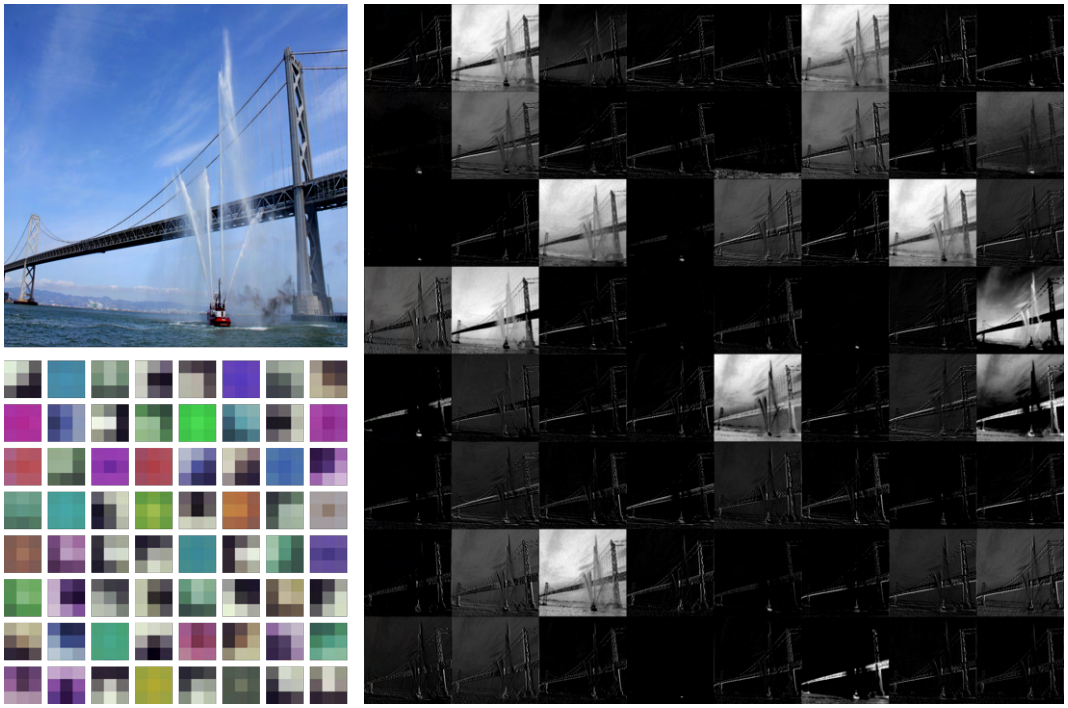

Figure 2.6: Example of an input image (top left), 64 $3 \times 3 \times 3$-RGB-kernels of the first layer in a pre-trained VGG-16 CNN (bottom left) and the corresponding resulting activation maps for that layer (right). Each kernel detects other features due to reacting to edges and color values differently. Figure generated with NeuralVisUAL after loading a pre-trained VGG-16 model.

approach enables CNNs to process relatively large inputs, such as images and videos of relatively higher definition. Furthermore, the kernel's receptive field allows the network to figuratively concentrate on the local neighborhood of a neuron and recognize locally present structures, enabling quickly trainable edge detection and feature extraction. [LB+95, ESYF+20, SLJ+15]

To widen the receptive field exponentially without losing resolution, dilated convolutions collect contextual information on multiple scales. They work by widening the kernel, inserting empty rows between the kernel elements. This technique is especially beneficial for dense predictions, for example, when generating segmentation maps. [YK15]

Transposed convolution predicts a higher resolution output by interspersing the input data with rows and columns filled with zeros, upsampling the information. [DV16]

Pooling layer

Another essential type of layer in CNNs is the pooling layer. As a convolutional layer with a stride of at least two, a pooling layer also decreases the spatial resolution of the layer input, thus downsampling the information as it travels through the network. A pooling layer takes the data, divides it into small regions. From each region, it chooses or calculates a singular value that represents this region, writing it into the output neuron corresponding to that region. [DV16, ESYF+20, NHH15]

For example, max-pooling chooses the highest value, min pooling the lowest, and average pooling calculates the mean of all values within the region. Average pooling with a region size of $n \times n$ equates to a convolution layer with a kernel of size $n \times n$, uniformly filled with the value $1/n$, applied without padding with a stride of n. [ESYF+20, DV16, SSJ+17]

Like transposed convolution, unpooling attempts to reverse the effects of a pooling layer, increasing the spatial resolution, thus upsampling the data. Due to the loss of information from pooling, unpooling reproduces the exact value of the input neuron for all neurons in the corresponding region. If a max-pooling layer records the maximum value's location in the original data, the unpooling layer can restore that maximum value to its original location, leaving the other fields in the region at zero. [NHH15]

2.1.6 Deep neural networks

DNNs are networks with several hidden layers with nonlinear operations, inspired by the complex structures of neurons in the human brain [Ben09, Sch15]. This enables them to generalize from training data and approach complicated problems such as computer vision [DBK+20, RDS+15, SZ14], speech recognition [GMH13, YL17], or natural language processing [BMR+20, SMM+17, SHD14, YHPC18], among others [Ben09, SS18].

In computer vision, DNNs detect different levels of abstraction of image features, depending on the layer depth [OMS17, Ben09]. The early layers usually represent edge detectors, followed by layers with texture and pattern recognition, primary shape detectors, with the last layers finally recognizing whole objects from high-level visual abstractions [OMS17, Ben09].

A DNN with eight layers was already described in 1971 [Iva71]. A typical example of a more modern DNN is VGGNet, which has 16 or 19 consecutive layers, depending on the used variation [SZ14]. Figure 2.7 shows the architecture of the VGG-16 variation [SZ14, FALL17]. According to its creators, this depth permits a higher performance due to more accurate predictions with high generalization ability [SZ14]. VGGNet won second place in the image recognition challenge ILSVRC in 2014, beaten only by GoogLeNet with 22 layers [SZ14, SLJ+15].

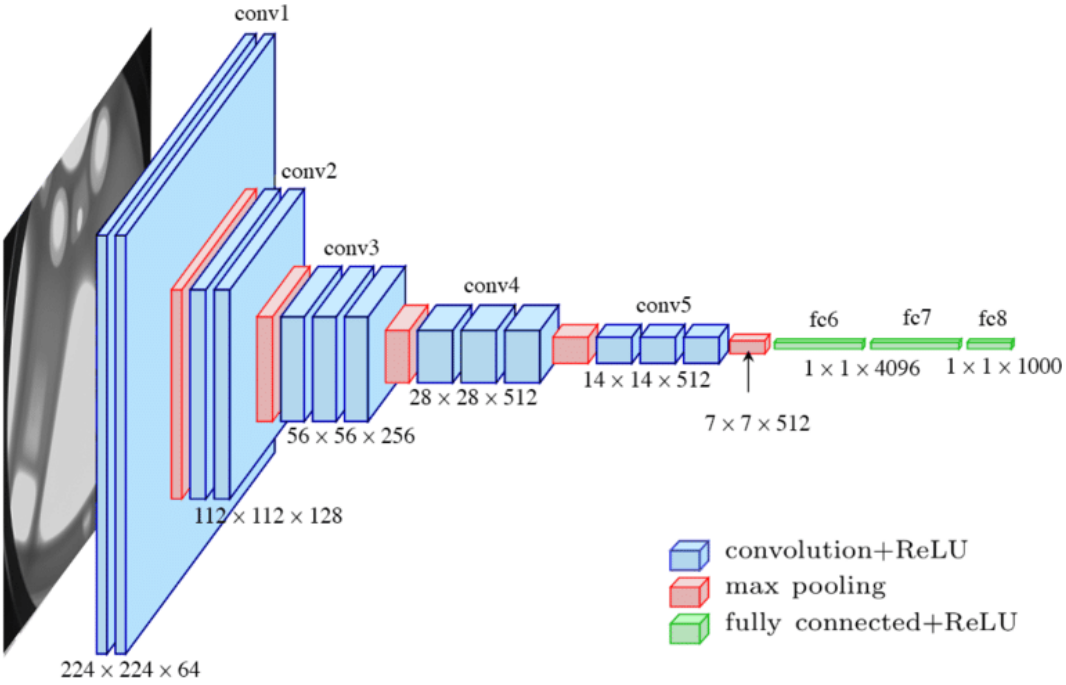

Figure 2.7: Architecture of VGG-16, figure taken from [FALL17].

DNNs can even have hundreds of layers while still producing results with better performance than comparable shallow networks [HSL+16]. However, the training process for exceedingly deep networks raises difficulties with increasing depth. Deeper networks are computationally more expensive to train due to their complexity, requiring more training iterations [Ben09, HSL+16, CPC16]. Even inception can prove problematic on low-power devices, mainly when the network contains many parameters [CPC16]. Furthermore, deeper networks are increasingly prone to overfitting and therefore tend to require regularization methods such as pruning, weight decay, sparsity, dropout, or data augmentation for small training sets [GEH19, Iva71, SHK+14, HSL+16].

Autoencoders use a DNN architecture to obtain a latent space representation through dimensionality reduction, reducing the input to its most essential information while discarding noise. The decoder part of the network then attempts to reconstruct the original input from the latent space representation. [Kra91]

Another example of DNN architecture is the generative adversarial network (GAN), which consists of a generator and a discriminator that simultaneously train against each other. The generator attempts to generate realistic data to deceive the discriminator, whereas the discriminator tries to distinguish the generated data from real training examples. [GPAM+14]

Residual connections

Even if a network is a feedforward network, that does not mean that the information exclusively travels linearly from one layer to the direct successor. Although this was the case with the earliest architectures that evolved from the MLP, nowadays, many CNNs, especially in computer vision, employ residual connections. These connections transfer information from an earlier to a later layer, skipping at least one successive layer in between and adding their original information to the data processed by the intermediate layers. [HZRS16, SIVA17, Kub99]

Residual connections diminish vanishing gradients within a DNN, as the information has to propagate through fewer layers. They typically incorporate double or triple layer skips with nonlinearities and batch normalization in the skipped layers. This approach enables a faster learning rate and diminishes accuracy saturation with increasing layer depth. [SIVA17, HZRS16]

When incorporating several skips in a CNN in parallel, such that every layer directly connects to every other layer in a feedforward fashion, this dense CNN is called a DenseNet [HLVDMW17].

A prominent example of a residual network is U-Net, an autoencoder that employs residual connections between the encoder and decoder halves of the network. These connections enable the U-Net to preserve the high-definition detail of the input and reuse it in the end during upsampling to construct a more precise output. [RFB15]

2.2 Visualizations

Many various tools and approaches exist to visualize CNNs, making them more explainable and more readily understandable. Section 1.2 gives an overview of these tools and frameworks, whereas this section describes the technical background behind generating convolution kernel visualizations, saliency maps, and integrated gradients.

2.2.1 Kernels

The main challenges of visualizing convolution kernels are layout considerations and design decisions about representing matrices visually. Convolution kernels usually are multi-dimensional matrices consisting of floating-point numbers representing their weights. In computer vision, kernels usually have three dimensions: width, height, and a depth corresponding to the depth of the previous layer. The number of kernels within such a layer equates to the output depth of the kernel's convolutional layer. [HND19, Kum19]

The weight values need to be normalized or clipped to visualize these kernels. A simple gradient mapping like grayscale or a more sophisticated color map can

transform these values into colors. [HND19, Kum19]

Often, a CNN takes colored images as input and immediately implements a convolutional layer with kernels of depth three that cover the image's RGB channels. In that case, it is sensible to visualize these first kernels in colors as well, recombining the three values into their respective red, green, blue channels, as figure 2.8 demonstrates. [HND19, FG18]

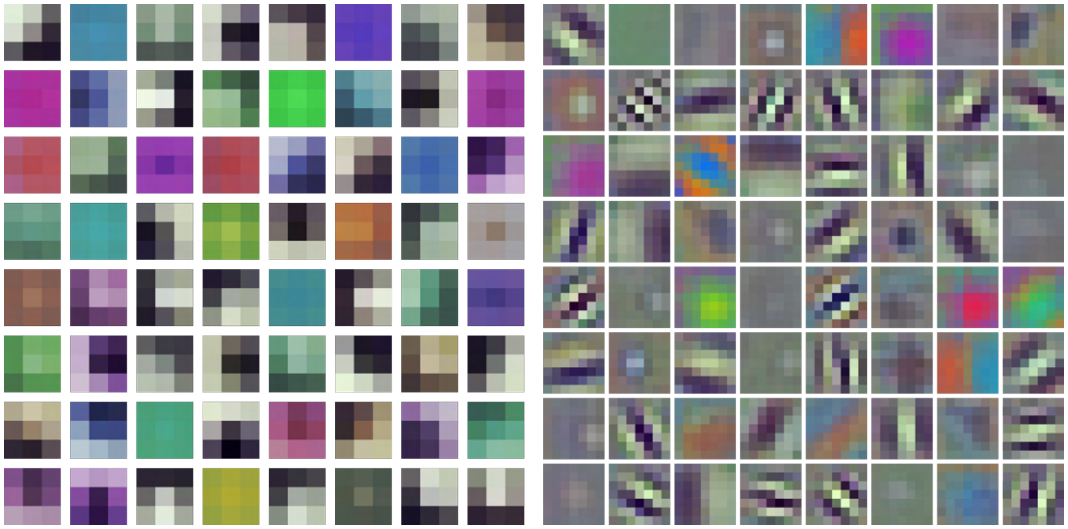

Figure 2.8: Kernels of the first convolutional layer in VGG-16 and Resnet-50, respectively. Figure generated with NeuralVisUAL after loading pre-trained VGG-16 and Resnet-50 models.

Depending on the number of kernels and focus of the visualization, it can present a single kernel or visualize all kernels of a layer in a single layout. When displaying multiple kernels at once, it is reasonable to normalize all kernels in the same manner so that a singular color value in various kernels always corresponds to the same numerical value. [Kum19]

2.2.2 Activation maps

Activation maps show how the CNN perceives the input and processes it throughout its layers. They arise from sending the specified input through the network until the selected convolutional layer applies its kernels to the data, followed by the layer's activation function. Activation maps represent the intermediate state of the information as it travels through the network, visualizing the neurons of individual kernels within a single layer. [Kum19, Kha20]

This data can then be normalized, converted to colors, and rendered as an image similarly to the kernel visualizations, except that the activation maps usually have a higher resolution than the kernels themselves. [Har15, Kha20]

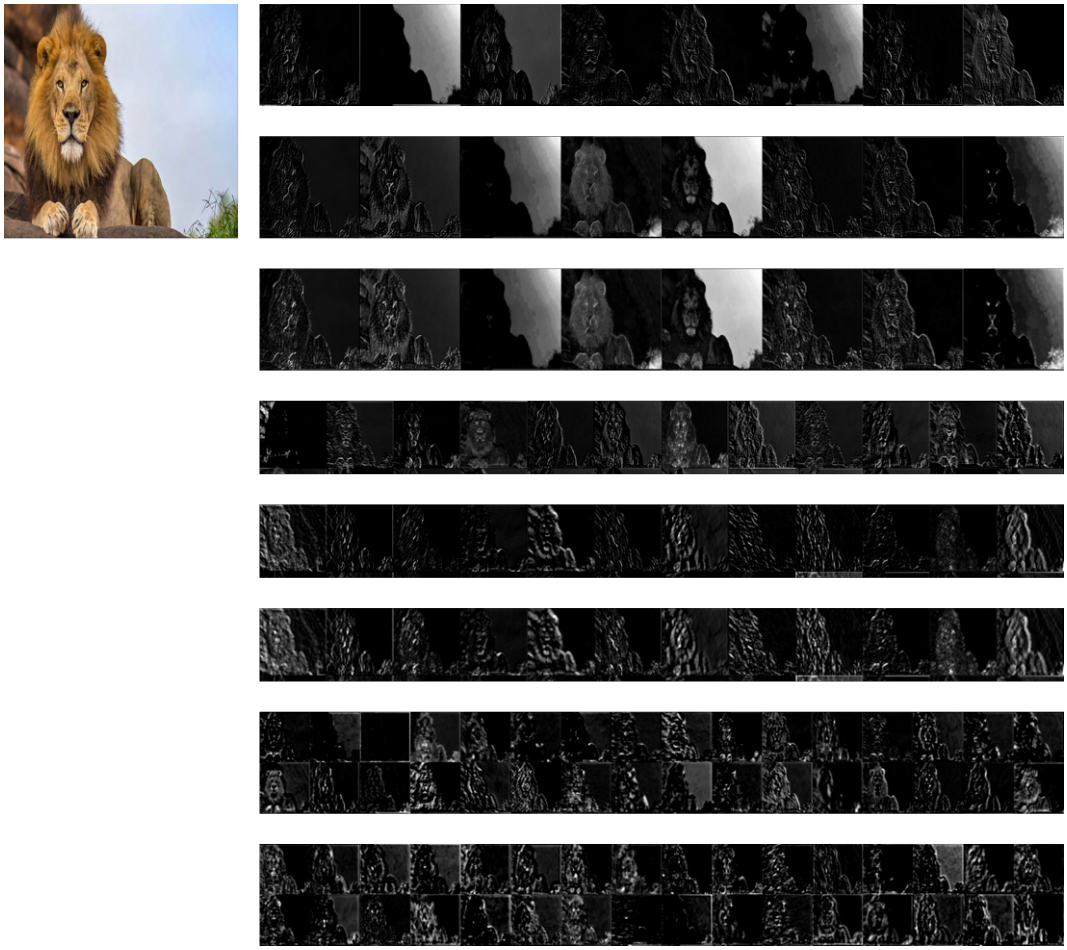

Figure 2.9: Activation maps for the first few kernels of the first eight layers of VGG-16, given the photo of a lion as input. Figure generated with NeuralVisUAL after loading pre-trained VGG-16 and Resnet-50 models.

Activation maps can help display which information the network picks up on which features it recognizes, especially for the first few convolutional layers. Figure 2.9 shows an example selection of kernels within the first eight convolutional layers of the network VGG-16 [SZ14]. [Kum19, OMS17]

2.2.3 Saliency maps

Saliency maps demonstrate what parts of the input have the most influence over the final prediction result. Such a map assigns an individual level of importance towards the result to each singular part of the input, such as an image pixel, as figure 2.10 demonstrates. The main goal of saliency maps is to give researchers a clearer understanding of the decision-making process within the neural network in

an attempt to explain what input features guided the network towards its decision. [SVZ13, SMV+19]

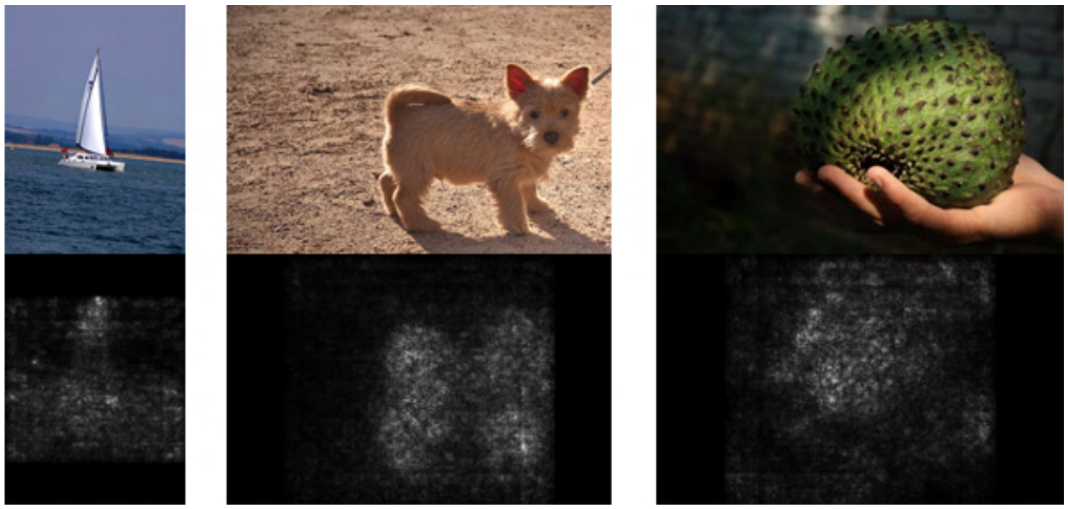

Figure 2.10: Input images and corresponding saliency maps for the predicted class output, figure taken from [SVZ13].

To obtain such a map, the output's gradient over the input is calculated by approximating the class score of the resulting prediction as a linear function with its first-order Taylor expansion. This gradient is then equal to the derivative of the prediction class with respect to the input at a particular position in the input data. Iterating over all possible positions to retrieve the gradient for the desired output produces the saliency map. [SVZ13]

The resulting matrix of values can then be normalized or clipped and visualized as an image with colors representing the saliency either as a single gradient or a more complex colormap. [SMV+19]

2.2.4 Integrated gradients

Integrated gradients fulfill a similar goal as saliency maps, in the sense that they also assign importance to each part of the input, representing how strong its influence toward a given output is, as figure 2.11 demonstrates. Instead of simply calculating the output gradient over the input, this technique integrates the gradient from the baseline to a fully saturated input. The baseline is usually an input completely at zero; this would be a completely black input image for image processing. [STY17, SLL20, SMV+19]

The baseline is usually an input completely at zero; this would be a completely black input image for image processing [STY17, SLL20, SMV+19]. Alternative baselines can be utilized and even combined by averaging over them [SLL20]. For example,

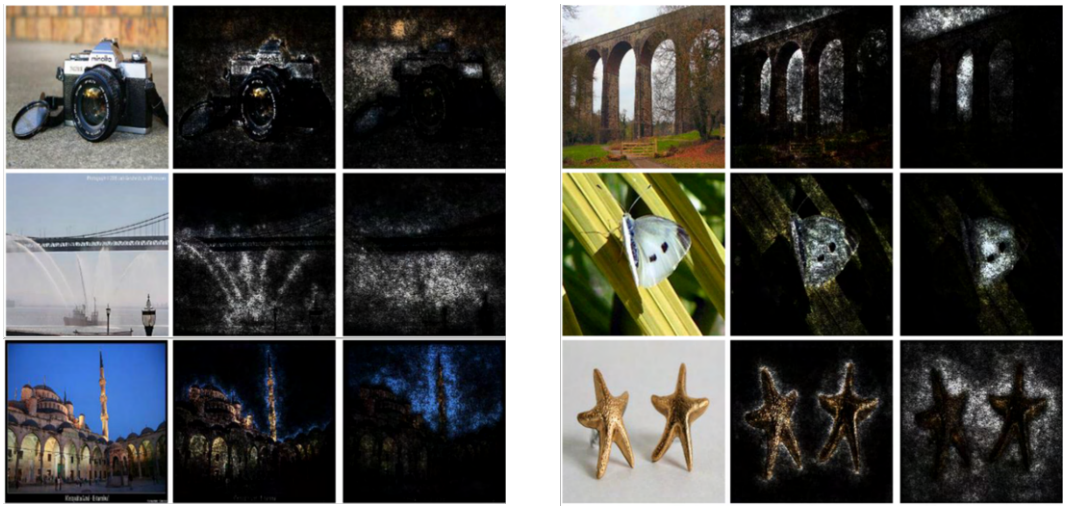

Figure 2.11: Examples of input pictures, their corresponding integrated gradient over the predicted output class compared to standard gradients.

the maximum distance baseline avoids the problem of the gradient being blind to the baseline color and instead generates a baseline as far away from the input as possible [SLL20]. The blurred baseline represents the intuitive notion of missingness by using a blurred version of the image [FV17, SLL20]. A uniform baseline takes a random uniform input to avoid the problem of blurred baseline that is biased towards high-frequency information [FV17, SLL20]. The gaussian baseline takes a gaussian distribution with a specified variance, centered around the original image, adding noise on top of it [STK+17, SLL20]. It is possible to combine multiple baselines by averaging over them [SLL20]. Which baseline or combination of baselines best conveys and visualizes the network's processes depends on the network and visualization goals [SLL20].

This approach satisfies the axiom of sensitivity, meaning that features receive a non-zero attribution if a change in their value to the baseline value would change the network's output value. [STY17]

Saliency maps and integrated gradients are both implementation invariant. This invariance means that two differently implemented networks that consistently produce the same result given the same input will also have the same saliency map and integrated gradient. [STY17, SMV+19, Kok18]

2.3 Force-based graph drawing algorithms

Force-based graph drawing algorithms calculate a visual representation of graphs, often simulating forces that exist in the real world [Kob12]. As part of information visualization research, among dimension reduction techniques and multi-level

approaches, they attempt to visualize complex information in a comprehensible layout [GFV13].

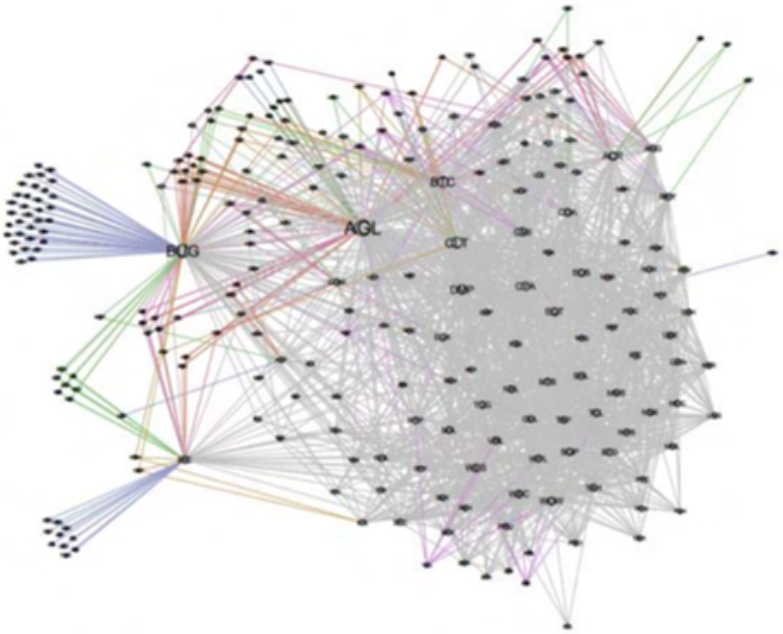

Figure 2.12: Layout of an undirected graph calculated by the FBA FA2, figure taken from [HHW18].

They only utilize the information contained in the graph, without any further assumptions about the nature of the connections or domain-specific knowledge, allowing them to flexibly adapt to any input [Kob12]. This way, these algorithms are versatile and self-adapting, leading to aesthetically pleasing and symmetric results [Kob12, Ead84]. As the forces imitate natural representations of the real world like springs, gravity, or electromagnetic repulsion, the resulting layouts usually have an intuitive, organic character [Kob12]. With these forces, the nodes converge towards a local optimum, reaching a stable equilibrium where all impulses balance each other out [JVHB14, Kob12, GFV13]. Figure 2.12 shows an example of such a layout, calculated by a FBA.

2.3.1 Drawbacks

A drawback of typical FBAs is the fact that they can be unpredictable and non-deterministic due to their inability to search for a globally optimal layout [JVHB14, Kob12, GFV13]. Because of their iterative, explorative nature, minor variations in the initial layout leads to entirely different results, as the nodes might converge to a different local optimum. Users, therefore, do not have direct control over the results, although they can influence the algorithm by adapting the parameters. Also, users

usually have to deal with a trade-off between speed and precision, as the algorithm iteratively calculates each consecutive step. [JVHB14]

2.3.2 Attraction-Repulsion-model

One of the essential distinctive properties of a FBA is the effect of distance between nodes in their calculations. This relationship can represent itself in the *(attraction, repulsion)*-model, which represents the exponents used for the force's proportionality to the distance. For example, one means linearly proportional, two stands for squared proportionality, and a negative value inverts the relationship and represents repulsion forces. As an exception, the value zero in this model represents the logarithm of the node distance. [Noa07]

For example, the LinLog approach naturally has (0, -1), the spring-electric system has the exponents (1, -2), whereas Fruchtermann and Rheingold use the squared proportionality in the attraction force instead, resulting in (2, -1) [JVHB14, FR91].

2.3.3 History and variations of FBAs

In 1963, Tutte introduced a layout algorithm for planar graphs based on barycentric representations [Tut63], which is the first approach towards a FBA [BETT98].

Later in 1984, Eades introduced the spring-electric system, which takes inspiration from real-world interactions [Ead84, JVHB14]. As the name suggests, it utilizes spring formulas for the attractive force, and repulsion forces equivalent to electromagnetic repulsion [Ead84, JVHB14, CS20, GFV13, Kob12]. Therefore, the attraction force between two nodes is linearly proportional to their distance, whereas the repulsion provides a square proportionality [Ead84, JVHB14, CS20]. These relationships mean repulsion has more prominence over geographically close nodes, while attraction takes a comparatively higher priority over wider distances [Ead84, JVHB14]. In the traditional spring layout method, the forces counteract each other, featuring attractive impulses between connected nodes and repulsive movements between any nodes [Ead84].

Seven years after the introduction of this algorithm, Fruchterman and Rheingold extended it with a notion similar to a temperature that cools the movement down for fine adjustments towards the end [FR91, Kob12]. Frick et al. later refined this approach by defining local temperatures instead of using one global parameter [FLM94].

Instead of balancing attraction and repulsion forces, Kamada and Kawai define an optimal distance between the nodes based on graph theory [KK+89]. In contrast, Davidson and Harel introduced more rules to the classic attraction-repulsion algorithm to simulate the annealing of metal in material physics [DH96, KGV83]. The LinLog model of Noack uses a different energy model to better highlight graph

clustering, avoiding graphs with uniform edge lengths [Noa07, GFV13]. Moreover, force-directed algorithms can also utilize genetic algorithms for optimization [Kos91, BBS96].

Algorithms to properly handle graphs with thousands of vertices usually use the multi-scale method [HH01, Kor04, GGK00, GK04, Hu05, QE00, Wal00, HJ04], as they need some adaptation to maintain reasonable performance. This method first organizes nodes locally on a fine-scale level, then relocates the graph on a coarse scale before correcting the fine-scale disorders [HH01]. Another approach for large graphs is stress majorization within multidimensional scaling methods [Kru64, CS20], which permits a mathematical optimization towards the global optimum [GKN04, BP06, Kob12].

Although FBAs usually serve to generate two- or 3D representations, they can also be applied to higher-dimensional spaces [Kob12]. Furthermore, force-based graph drawing can be applied on non-euclidian spaces [Kob12, MB95, Mun97, Mun98, Epp03, Ost96, KW05], including meshes of 3D objects [GGS03].

2.3.4 FA2

FA2 is a continuous force-directed algorithm that uses the accumulated forces model [CS20], initially developed by Jacomy et al. in Java for the open-source network visualization software Gephi [BHJ09]. It calculates a 2D layout that represents structural proximities for undirected graphs of up to 100000 nodes, visualizing the network spatialization process. [JVHB14]

The energy model in FA2 bases itself on real-world forces, with attraction and repulsion between two nodes linearly proportional to their distance, represented as (1, -1) in the *(attraction, repulsion)*-model [JVHB14].

Attraction force

The attraction force between two directly connected nodes is therefore simply equal to their geographical distance: $F_a(n_1, n_2) = d(n_1, n_2)$ [JVHB14].

Alternatively, a LinLog mode within FA2 provides the option of using a logarithmic attraction force for the attraction: $F_a(n_1, n_2) = log(1 + d(n_1, n_2))$ [JVHB14, Noa07].

Furthermore, FA2 recognizes graphs with weighted edges and can use this weight for the to scale the attraction force, using the user-defined edge weight influence δ as an exponent to the weight: $F_a(n_1, n_2) = w(e)^{\delta} d(n_1, n_2)$ [JVHB14].

Another option in FA2 is the mode to dissuade hubs, which aims to place authorities closer to the group's center, while pushing hubs further out: $F_a(n_1, n_2) = d(n_1, n_2)/(deg(n_1) + 1)$ [JVHB14]. Authorities in this context are nodes with a high indegree, whereas hubs have a high outdegree [KNBW11].

Repulsion

The repulsion force depends on the degree of the two nodes, to bring nodes with very few connections closer to highly connected nodes: $F_r(n_1, n_2) = k_r \frac{(deg(n_1)+1)(deg(n_2)+1)}{d(n_1,n_2)}$. k_r specifies the strength of the repulsion force relative to the attraction force. This parameter indirectly affects the spacing between the nodes; a larger k_r expands the graph, making the resulting layout wider. [JVHB14]

To reduce computational complexity and optimize the algorithm's performance for large graphs, FA2 can approximate the repulsion forces with a Barnes Hut optimization, retaining the same general layout shape [BH86, JVHB14].

Gravity

An additional force, gravity, attracts all nodes towards a common center point, thus preventing infinitely increasing geographical separation between disconnected subgraphs: $F_g(n) = k_g(deg(n)+1)$. With the strong gravity option enabled, this force is multiplied by the respective node's distance to the center $d(n)$, forcing an even compacter layout. [JVHB14]

Overlap prevention

To prevent overlapping of nodes, FA2 implements a system to consider the size of nodes and modify the repulsion. In this mode, the distance between two nodes $d(n_1, n_2)$ in formulas is replaced by $d'(n_1, n_2) = d(n_1, n_2) - size(n_1) - size(n_2)$. When this modified distance between two nodes is smaller than 0, FA2 detects overlapping, so it temporarily disables the attraction force between them and uses a stronger coefficient k'_r for the repulsion force. [JVHB14]

Continuously adapting force application

FA2 applies its forces continuously over the whole layout calculation process, instead of acting in distinctly separated phases, to allow Gephi users to stop the algorithm at any time or let it run and perfect the layout for a long time. As an alternative, the algorithm adapts the local and global speed of force-applications dynamically depending on measured oscillation, and traction. [JVHB14]

The automatic adaptation mechanism measures the swinging of each node as the divergence between the currently applied force and the force applied at the previous step. It also monitors each node's traction, the average of the current force and the previously applied force. The closer this traction is to the current force, the less oscillation is happening; a complete swing back results in zero traction. [JVHB14]

Instead of slowly diverging toward a stable position, some nodes tend to oscillate around the optimal position. To optimize convergence, FA2 dynamically adapts

the movement speed of nodes to only permit a user-defined level of oscillation. Additionally, the global speed increase is limited to 50% per iteration step to prevent adverse effects of too high acceleration. With this mechanism, the layout calculations automatically move fast at the beginning and slow into more refined movements when approaching the optimal positions towards the end, optimizing the algorithm's performance. [JVHB14]

Comparison to other algorithms

Because the developers of FA2 chose to implement a continuous algorithm, they could not include several features and optimizations present in other algorithms [JVHB14], such as simulated annealing [DH96], auto-stop [Hu05], phased strategies like in OpenOrd [MBKB11] or graph coarsening [Hu05, Wal00]. Furthermore, FA2 facilitates fluid movement during the layout process [JVHB14], in contrast to the algorithms of Kamada Kawai [KK+89] and GEM [FLM94], which both use non-homogenous forces.

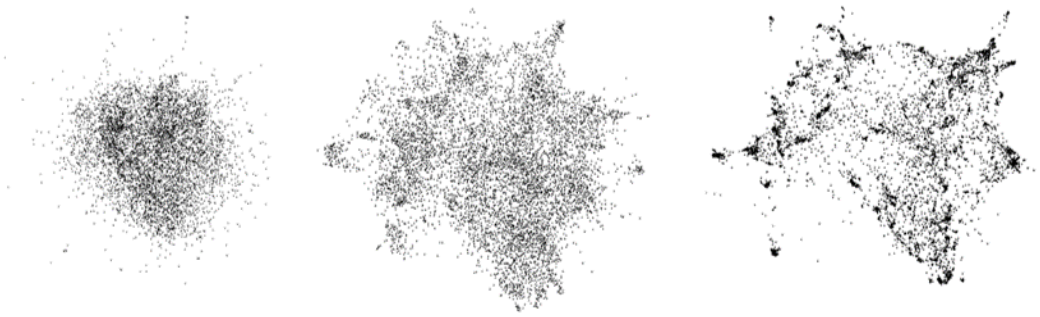

Figure 2.13: Layout of a graph calculated with Fruchterman an Rheingold, FA2, and LinLog mode of FA2. It is visibile how the different (attraction, repulsion)-coefficients result in varying densities of the resulting layout. Figure taken from [JVHB14].

Compared to FA2, the algorithm of Fruchterman and Rheingold [FR91] eventually reaches a better quality in the resulting graph layout, but is exceptionally slow, requiring an optimized version for large graphs. In contrast, the multi-scale algorithm of Yifan Hu [Hu05] is quicker than FA2 while resulting in slightly lower quality. Among these three algorithms, FA2 is the quickest to reach an acceptable level of quality, making it a good candidate for applications where performance is more critical than perfect quality. Empirically, FA2 represents clusters in a manner that is better identifiable for humans. Running FA2 in LinLog mode results in higher quality at worse performance. [JVHB14] Figure 2.13 shows a visual comparison of the layout calculated by Fruchterman an Rheingold, FA2, and the LinLog mode of FA2. acfa2's layouts satisfy aesthetic criteria such as edge crossings, angle sizes, and uniform edge length better than the other algorithms [HHW18].

3 Approach

NeuralVisUAL is an open-source application that interacts with existing feedforward CNNs, which are implemented using Tensorflow and Keras, to visualize them in the game engine UE4 [Inc21]. TensorFlow is a powerful, extensive platform for developing ML models, initially created by Google and released in 2015 under the Apache 2.0 license [AAB+16]. Machine learning researchers across countless disciplines widely use it because of its flexibility and variety [AAB+16]. Built on top of this platform, Keras is an open-source Python API with the primary purpose of facilitating fast prototyping and experimentation of TensorFlow networks [Tea20]. NeuralVisUAL directly loads such a network specified by the user and extracts relevant information like the network structure, kernel data, or layer activations.

The application processes the network structure to calculate a 2D network layout using a FBA based on ForceAtlas 2 [JVHB14], influenced by several layout goals, which the user can adjust in the visualization settings. This resulting layout then places a cuboid for each layer in the 3D virtual world. The free third dimension stays free to render visualizations related to single layers next to its corresponding layer. These visualizations can be convolution kernels, layer outputs, saliency maps, or integrated gradients.

The goal of NeuralVisUAL is to provide an intuitive visualization of the neural network, displaying the vital information about it in a virtual world to be explored by the user.

The architectural design of NeuralVisUAL consists of distinct modules, which developers can easily extend, replace, or adapt for other use cases. One of these modules is the UE4 project, in which the user exclusively interacts with the network. Commands with requests for specific visualizations are sent from this project through a C++ plugin via a WebSocket connection [FM11] to a python server. This python server receives these commands, asynchronously interprets them, and directly interacts with the neural network to obtain the relevant data. It then directly processes this data and uses it to generate a visualization according to the user-specified visualization settings. After finishing these calculations, the server sends this visualization via WebSocket back to the C++ plugin, which relays it to Unreal's blueprint visual scripting system [Inc20b]. There the response is unpacked and used to finally render the visualization in the virtual world.

3.1 Architecture

NeuralVisUAL has a modular architecture with a dedicated pipeline to access and process the data generated by the analyzed neural networks, and display the visualization in the virtual world within the UE4. Unreal Engine is a game engine written in C++ and developed by Epic Games, Inc. It has multiple useful out-of-the-box features that make it a suitable choice for visualization applications like NeuralVisUAL, significantly alleviating the development process of such applications [Lah16, Šmí17].

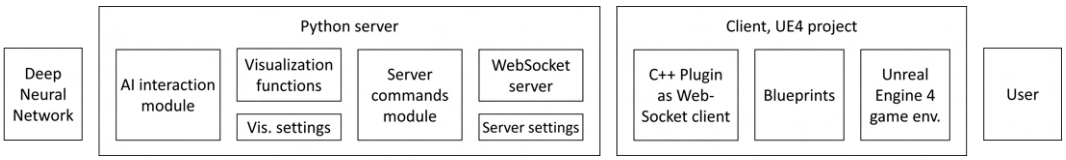

Figure 3.1: Overview of the various modules within the architecture of NeuralVisUAL. The data passes horizontally through each consecutive module. The python server only shows the modules most relevant for data processing.

Figure 3.1 gives a visual overview of NeuralVisUAL's modular architecture. This design of data processing into several distinct steps has several advantages for NeuralVisUAL, which section 3.1.3 explores in more detail.

For a start, NeuralVisUAL offers a high level of modularity, allowing developers to interchange parts of the program with ease. This option enables them to adapt NeuralVisUAL to individual needs or even utilize it for entirely different projects.

Furthermore, this modularity allows running the neural network interaction server, which calculates the visualizations on a separate machine than the UE4 game environment, which renders the visualized network to the user. This possible separation is especially vital as ML research commonly relies on specialized hardware and high-performance cloud computing [CHW16].

Additionally, this pipeline's design aims to ease development efforts while still producing a performant application that can efficiently process raw data collected from the neural networks into visualizations. Developing this part of NeuralVisUAL is more streamlined in python than in C++, because python does not have such lengthy compilation times during debugging [Šmí17]. Moreover, the application is significantly more performant than if this math-heavy module was developed with UE4's blueprint visual scripting system [Inc20a].

Data pipeline

This section presents a quick overview of how NeuralVisUAL processes the information between the user and the neural network to be analyzed, as it progresses

through the modules shown in figure 3.1. Sections 3.1.1 and 3.1.2 describe in more detail how the UE4 client and python server process the data, respectively.

Users explore the selected neural network in an UE4 project; they exclusively interact with the network within this virtual world. For communication between server and client, NeuralVisUAL utilizes the WebSocket protocol. Standardized in 2011 by the IETF as RFC 6455, it is a two-way network communications protocol that works on a single TCP connection, upgrading it from HTTP [FM11]. This protocol provides a simple full-duplex communication interface for NeuralVisUAL, which facilitates a separation between client and server.

Blueprints within the UE4 project form commands that can include new visualization requests and relay them to a C++ plugin that provides an asynchronous WebSocket client. This client sends the commands over a network connection to a python WebSocket server running on the same machine as the analyzed neural network.

The python WebSocket server unpacks this command, asynchronously interprets it, and decides what data to obtain from the neural network. After retrieving this data, the server directly processes it into a visualization, using desired parameters from user-specified settings. The server serializes this visualization with msgpack and sends it back via WebSocket connection to the client.

The WebSocket client of the C++ plugin receives this data and calls blueprint functions during the unpacking process. Blueprints are responsible for caching and interpreting these results, using meta-information to track array positions within the deserialized data. They finally generate materials, spawn object instances, and import textures to realize the desired visualization in the virtual game environment.

3.1.1 NeuralVisUAL in the UE4

Virtual visualization environment

The game engine UE4 renders the visualization results that NeuralVisUAL generates, spawning objects and displaying the information that NeuralVisUAL collects about the chosen DNN, as shown in figure 3.2. This way, UE4 autonomously handles the whole rendering process and provides a pre-existing interface for handling interactions by the user with the virtual world.

The user can fly through this virtual world and freely explore the virtual 3D representation of the neural network. With the keys W, A, S, D, Q, and E the user can move forward, left, backward, right, up, and down respectively, relative to their rotation in space. Moving the mouse without modifier keys changes the players rotation, whereas holding Ctrl during movement rotates the player around the nearest point on the x-axis which traverses through the entire network, allowing them to quickly get a new perspective on the part of the network that is currently in focus. Holding Shift while moving shifts the player along y- and z-axis, and holding

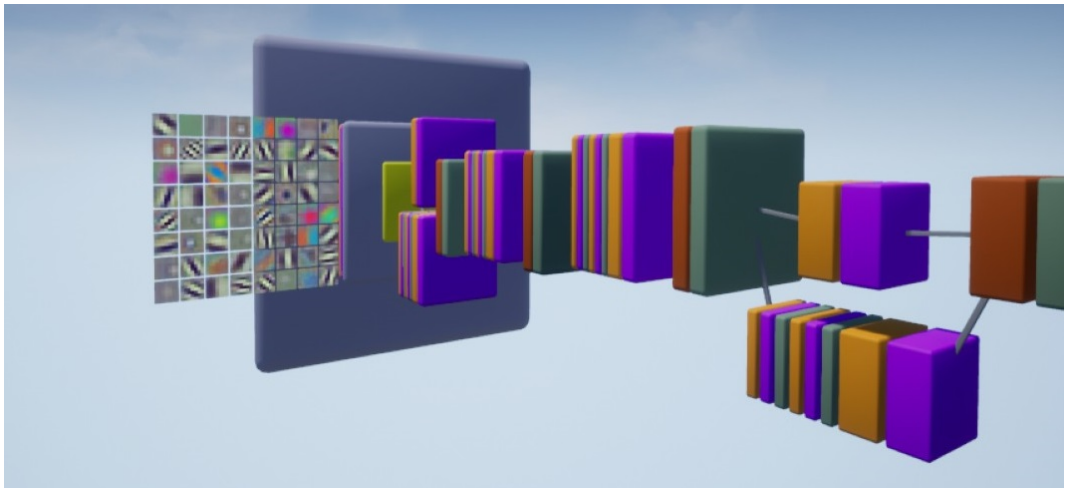

Figure 3.2: Screenshot of NeuralVisUAL's game environment, with a pre-trained Resnet-50 loaded, visualizing its kernels of the first convolutional layer.

Alt during mouse movement shifts along the x-axis. The user interacts with the network exclusively through this application, except when changing preferences in the server or visualization settings.

NeuralVisUAL provides a command console to allow the user to directly interact with the python server from within the game engine. Depending on the user's interaction, the UE4 application sends commands, which include visualization requests. These visualization requests need the retrieval of new information from the network.

Once this request has been sent through the pipeline and the response returns to UE4, this application displays the newly spawned objects and allows the user to explore them.

UE4 blueprints

NeuralVisUAL's UE4 application also utilizes blueprints, the visual scripting language used by the Unreal Engine [Inc20b]. It uses a node-based interface to allow developers to script the gameplay in their projects without developing source code in C++. However, blueprints can integrate with C++ code, as is the case with NeuralVisUAL [Inc20b]. In this application, blueprints relay the user's visualization requests as commands to a UE4 C++ plugin that relays them towards the server.

When receiving the server's response, The C++ plugin calls various blueprint functions using delegate callbacks, relaying the atomic parts of the unpacked response to functions that can handle the various data types. The blueprints organize this data and then modify the virtual world, spawning objects to display the

visualization according to the server's instructions, creating materials and object instances.

Furthermore, the plugin calls specific blueprint functions that mark the start and end of arrays, dictionaries, responses, and WebSocket connections. With these function calls, the blueprints can maintain an overview of the response structure, which is essential for a correct interpretation of the results, for example, when receiving large arrays of varying dimensions.

When sending a specific data value, the plugin calls the respective function for that data type, relaying the value and the original command, first string, and array position. The array position helps the blueprints recreate the original array shape and properly cache the values by knowing the locations of each value within the response structure. The original command can be helpful for verification purposes or to group debug output according to its command. Several server responses have a well-defined structure consisting of a tuple with a string specifying the response type, followed by the rest of the relevant data. Blueprints use this so-called "first string" to identify such a specifically structured response. These responses can be, for example, debug messages, status responses, or instructions to spawn cuboids or image planes in the virtual world.

Developers can extend the blueprints of NeuralVisUAL easily by adding new interactions with the virtual world. Novel command responses can then trigger these, thus allowing developers to augment the functionality of NeuralVisUAL and adapt it to new use cases.

C++ WebSocket client as UE4 plugin

The third part of NeuralVisUAL that runs on the client is a custom C++ plugin. This plugin aims to establish WebSocket connections with the python server to communicate with the other module of NeuralVisUAL via a network connection. This connection can exist within the same machine, simply communicating with localhost on the same network card if the user prefers to run the UE4 visualization on the same machine that executes the neural network.

The C++ plugin relies on the library project boost, a collection of widely used, peer-reviewed C++ source libraries [DAR21]. One of the libraries contained in this collection is the header-only library beast, which is built on boost's asio library and offers a foundation for network protocols, including HTTP/1 and WebSocket [Boo17]. Furthermore, this plugin uses documentation code for an asynchronous WebSocket client [Fal19] provided as part of the beast library. This code was developed by Vinnie Falco between 2016 and 2019 and is distributed under the Boost software license [Fal19].

The C++ client opens a WebSocket connection to the specified address for every command that the blueprints relay to the plugin. Over this connection, it sends the

command to the server and asynchronously waits for responses. Until the server terminates the command-specific session, it can send multiple responses to the client, each packed with the binary serialization format msgpack [Fur19]. These responses can contain status messages, debug information, information responses, float arrays, or image files.

For each response, the C++ plugin validates and deserializes the binary data with msgpack, unpacking it into arrays of various data types. The C++ plugin calls respective blueprint functions during the unpacking process, relaying the atomic data values and the response structure to the UE4 blueprints. These functions include meta methods to notify the blueprints of the start or end of a connection, response, dictionary, or array. The previous section explains this process in more detail.

The C++ plugin can be useful for other projects, which would benefit from a WebSocket interface that facilitates easy interaction between network sources and the virtual world within UE4.

3.1.2 Python interaction server

The python interaction server does the heavy lifting of the NeuralVisUAL application. It receives commands, manages direct interactions with the DNNs, interprets their data, and generates visualizations. These visualizations, among other responses, are sent back to the client to change the virtual world in the UE4 environment.

Python WebSocket server

On the other end of the WebSocket connection sits an asynchronous python server that is already running, awaiting connections from a client. The python server receives these commands and processes them, interprets the space-separated parameters as arguments, and calls the relevant functions in various project modules.

It also relays any responses these processes might generate back to the client. In contrast to the received commands, which are always just a string, the responses are serialized with the msgpack library to standardize the transfer of various data types, arrays, and image files. For example, these responses can be a debug message with verbosity level or a status message informing if the command succeeded, failed, or is still processing. The WebSocket server can also send generated images or other files for caching and instructions to create objects in the virtual world within the Unreal Engine.

Asynchronous execution

As some of the commands, especially the visualization calculations, might take a significant time to process, the python server can asynchronously execute these

commands. That means the server stays reachable, can be interacted with, and processes other commands while still calculating more computationally expensive responses, seemingly running multiple commands in parallel.

To achieve this goal, the python server of NeuralVisUAL utilizes the python library asyncio [Fou18]. Except when a server shutdown or restart interrupts it, an event loop is constantly running, which serves the interactive WebSocket server, waiting for incoming client connections. All following functions that retain the ability to send responses are async functions requiring the await-keyword on their execution. Computationally expensive functions then regularly call an asyncio-specific *sleep* function with a parameter of zero seconds at strategically sensible points. This way, these complex functions yield to other async-processes, allowing them to finish or at least further iterate their calculations before themselves executing another iteration of calculations. The usage of asyncio is not a multithreading solution, as this approach only ever executes one process on a singular thread [Fou21]. However, as the yield points occur relatively often in expensive functions, a human user can hardly distinguish that these processes are not instantly executed but instead wait for such a yield point.

The only major drawback with this one-threaded approach is that certain specific processes within the neural network can block the main thread, making the server unresponsive for a few seconds. Such a situation happens, for example, when a neural network is loaded and imported from another source file into the running python server environment. Moreover, although the data processing and visualization calculations can be equipped with such yield points, retrieving data from the network can block the thread. This interruption is noticeable when calculating the integrated gradient, as it is a computationally expensive task for the neural network.

The relevant yield points that allow other operations to pass through occur at the following places:
The layout algorithm yields at every iteration of its FBA. Section 3.2 explains this algorithm in detail.
When loading a neural network into NeuralVisUAL, the *nn load* command yields before importing the python file that contains the neural network and after, right before importing the Tensorflow library.
When rendering multiple kernels or activation maps within the same layer into one visualization, the function reaches a yield point after caching the visualization for each kernel.
After each draw response with an instruction to spawn an object, even when queuing multiple draw instructions into one grouped response, the application lets other processes pass through.
Another of these points happens when rendering the kernels onto a 2D image texture. Instead of yielding after drawing every single pixel, a variable (*updateEvery*) defines after how many drawn pixels a yield should happen. The default value for this variable is 1000; it is not accessible via the visualization settings but is accessible in

the source code.

The server is equipped with two specific commands to let the user manage the asynchronous tasks running on the server. First, *server info* displays all currently running async-tasks and their state of completion. It also suggests what commands should be executed next, depending on the state of loaded variables and previous visualizations. If the user wishes to interrupt all currently running asynchronous tasks, the command *server stop* achieves precisely that. The user could also utilize *server shutdown* or *server reset*; however, these commands would result in a loss of the state of the server variables, possibly interrupting the visualization progress of a currently loaded network.

The exact locations of the yield points within the program flow have been carefully chosen in such a way that the *server stop*-command does not lead to information loss, corrupt data, or undefined behavior.

Server settings

A dedicated file in the directory of the python server stores all settings that the users can modify to fit the demands of their projects. This section provides a quick, non-exhaustive overview of the most relevant settings for the users. In addition, a second file contains visualization settings that exclusively impact how the server calculates and designs visualizations for the virtual world. These settings are mentioned at the relevant places throughout sections 3.2 and 3.3.

Firstly, users can set the address for the WebSocket server, consisting of IP address and port number, and the maximum size of msgpack-messages send via the WebSocket. Also, users can define how often and at what interval the application tries to establish the WebSocket server at the specified address; this becomes relevant when restarting the server.

Furthermore, it is possible to store a list of paths for selecting neural network models, representing each with a keyword and selecting a default model. The log file path and session-dependant file name can be specified, as well as the paths for file cache and debug output renders.

The user can define any command or list of chained commands as custom macros, facilitating the swift execution of commonly used command combinations. It is also possible to deactivate command chaining and to disallow remote code execution via the *python* and *eval* commands. Additionally, the user can specify whether chained commands should be interrupted upon failure or negative response of an intermediate command within the chain.

It is also possible to define a desired verbosity level for the debug output, finetuning the level of detail that the debug messages provide. Finally, for increased legibility, the user can specify whether the server output console should strive to break lines

at spaces instead of in the middle of words, and whether the output of the server console should use color ANSI codes.

Server commands

The python server can recognize 31 relevant commands, in addition to synonyms for some of these commands, and various testing commands irrelevant for the scope of this work. These commands are separated into general commands, which provide a relatively versatile interface as the basis for this server, and commands specific to neural networks, which facilitate the main functionality of NeuralVisUAL.

General commands: At any point, the user can list all available commands, search through them or display a detailed explanation of a specific command *(help)*. It is possible to prepare specific commands *(prepare)* ahead of time, execute them *(load)* or prompt for a command in the server console *(console)*. The user can view all executed commands *(history)*, seeing which ones succeeded or failed. If permitted in the server settings, it is possible to execute python code *(python)*, evaluate an expression *(eval)* or send the file at a specified path *(send file)*.

The user can view the server's IP and port *(server address)*, change the verbosity level of debug output *(verbosity)*, shut down the server *(server shutdown)*, or restart it *(server restart)*. Furthermore, it is possible to hot-reload python modules without restarting the server, even caching relevant variable data of that module while reloading the functions from source *(server reload)*. Server info shows all asynchronous tasks currently processing on the server, along with tips on what commands should be executed next for successfully loading and visualizing the neural network *(server info)*. The user can also cancel all running asynchronous tasks at once *(server stop)*, review the size of the visual cache per network *(server cache info)* and selectively or completely clear the cache *(server cache delete)*. A version check is helpful to compare server and client command versions *(check version)*. Depending on the level of mismatch on the major or minor version, the server prints a warning or throws a critical error to avoid undefined behavior.

Commands specific to neural networks: Furthermore, the user can initialize a neural network that already exists on the server *(nn load)* or check if one already has been loaded *(nn is loaded)*. After loading a network, they can retrieve the tensorflow version *(tf get version)*, network structure *(tf get structure)*, retrieve any information the server obtained about the network *(tf get vars)*, cache and view its trainable variables *(tf get train vars)*, or reset the cached structure data *(tf reset structure)*. When choosing a specific input image, it can be displayed right on the first layer while printing the prediction result *(tf draw prediction)*.

Finally, for visualization, the server understands commands to draw the layout *(tf draw structure)*, visualize the kernels of any convolution layer *(tf draw kernel)*, or display a layer's activation maps *(tf draw activations)*. Additionally, it can draw a

saliency map *(tf draw saliency)* or visualize the integrated gradients *(tf draw gradients)*.

Most of these commands accept a varying number of parameters to specify the desired behavior of the command. Users can find the exact number, order, and default value of parameters in the server commands module of the server's source code and the help text for each command by prepending *help* to the command, separated by a single space. These parameters are necessary, for example, to change server settings, identify layer or kernel selection for a visualization, define the DNN location, select an input image for visualization, set a prediction class for saliency maps and integrated gradients, or filter the output of variables.

It is possible to chain commands together with an ampersand (&) to execute them successfully. If one of the commands fails or returns a negative value, like, for example, *nn is loaded*, the rest of the command chain is interrupted and prevented from executing.

Retrieval and visualization of data from neural networks

Depending on the command that the server received, it calls specific functions within its AI interaction module. The server can then load the neural network from a specified path directly in python or retrieve and structure data from an already loaded network, usually as NumPy-arrays. NumPy is an open-source python package for scientific computing, distributed under a BSD license, and heavily utilized in ML applications, including Tensorflow [Num20]. For this reason, NeuralVisUAL also relies on this package for data processing.

Depending on the original command, the server might further process this data into a visualization. This visualization could include layout calculations to position many cuboids in 3D space or other visualizations rendered onto a 2D texture. Section 3.2 explains the layout algorithm in more detail, whereas section 3.3 discusses the different visualizations available in NeuralVisUAL. The AI interaction module sends the data retrieved from the neural network to another python module on the server containing the visualization functions. In congruency with the user-defined visualization settings, this module transforms the raw data into visual representations of that data. This visualization aims to give the user the ability to easily grasp the information and understand the data on an intuitive level.

Whichever layout or visualization the server produces in that process, the virtual world has to spawn objects to display these results. For the layout, precise spawn instructions are created and packed into an array with a clear string in its first place that defines this type of instruction. This package is then relayed to the WebSocket module of the server, back towards the UE4 client.

For visualization, the server renders images to avoid spawning hundreds of individual objects with the recommended visualization settings. These images are stored in the server's file cache, serialized as binary file data with msgpack[Fur19] and

sent towards the client. The blueprint functions of the client use these files later as a texture for a new material instance, spawning them onto a plane in the virtual world.

3.1.3 Modularity

The architecture of NeuralVisUAL described here utilizes such a data pipeline to provide a high level of modularity. This modularity gives flexibility to developers who want to extend or modify this application to adapt it to varying use cases, which can be entirely different from visualizing DNNs. Moreover, this pipeline allows the python server and UE4 rendering client to run on different machines, which may be necessary due to the infrastructure used in ML research. The pipeline design also eases development efforts while still resulting in an application that prioritizes performance.

Adaptability to other use cases

As the communication between the various modules of this pipeline is well-defined, especially when passing through the WebSocket connection, it is possible to modify and extend NeuralVisUAL for different use cases efficiently. Developers can reprogram, adapt or replace any module to fit the demands of their visualization project.

For example, as the python server is a distinct module, the client application can connect via WebSocket connection to a different server, which can provide completely different visualizations. If required, the developers can program this server to visualize a completely different project, which is not necessarily associated with AI. As long as this server knows how to send proper commands to spawn virtual objects like cuboids and image planes, the client can visualize these.

On the other hand, developers can replace the UE4 client with a different application, which can send server commands and interpret the visualization responses. This application could, for example, be a different game engine, a mobile app, an interactive website, or simply an interface to render the calculation results in the form of diagrams. This way, the neural network can easily be accessible by different clients, depending on the goal of visualization or desired interaction with the network.

Finally, the singular WebSocket plugin for UE4 could be helpful in various projects, as it simplifies the reception of information to the game engine from a network source. Due to the universal nature of WebSocket connections [FM11], this information can come from applications developed with any programming language, not limited to Python or C++. In combination with the asynchronous python server, developers can easily modify their custom UE4 projects to interpret new commands that they

can implement on the server. Therefore, this module of NeuralVisUAL can also be valuable outside of visualization research, such as actual video games or any other type of application.

Separation of server and client

Because the processed data flow through a WebSocket connection before being rendered in the game engine, it is possible to run the visualization code with direct access to the neural network as a python server on a different machine than the Unreal Engine rendering environment.

This separation is instrumental if the neural network runs on a Linux machine, while the UE4 application is compatible with Windows. It can also prove necessary for performance reasons to separate the two machines if the hardware on one machine is not performant enough to run DNN calculations and render the virtual environment simultaneously.

Additionally, some neural network architectures, especially in research, need high-performance hardware to generate results in a reasonable time frame - or at all [CHW16]. In this case, special servers usually execute these networks with high-performance hardware dedicated to these tasks [CHW16]. They do not necessarily offer the option to run a low-latency game application. The developers can then set up NeuralVisUAL so that the server exclusively runs python code with direct access to the neural network, calculates the visualizations, and streams the resulting data via the WebSocket connection to the client running the UE4 application.

Developments efforts and application performance

In addition, this pipeline eases development efforts. Since compiling the UE4 project in C++ on a personal computer takes a significant amount of time [Šmí17], the interruptions in the debugging process can be minimized by outsourcing the visualization calculations to an interpreted language like python. For example, compiling the C++ code of the NeuralVisUAL-project on an Intel i5-7300HQ @ 2.5GHz with 16GB RAM usually takes 5 to 15 minutes, sometimes up to 30 minutes.

The blueprint visual scripting system also has its limitations and can be laborious for implementing advanced visualization calculations [Inc20b]. Additionally, on a low level, it is slower than C++ or python's NumPy, as blueprints do not work as close to low-level machine code as C or C++. Therefore blueprints are not recommended dealing with large amounts of data, and complicated math-heavy operations [Inc20a], which are needed to process the amounts of data collected from neural networks during visualization. Due to these reasons, blueprints also require more development efforts and produce a slower application than outsourcing the visualization calculations to python.

Lastly, as a separation of server and client is possible with this pipeline, in the vast majority of cases, the server executing the neural network is the more powerful machine among the two that can execute the calculations significantly more efficiently. Therefore it is undoubtedly more sensible to outsource the math-heavy data processing and visualization calculations to the python server instead of the client machine running the UE4 application.

3.2 Layout

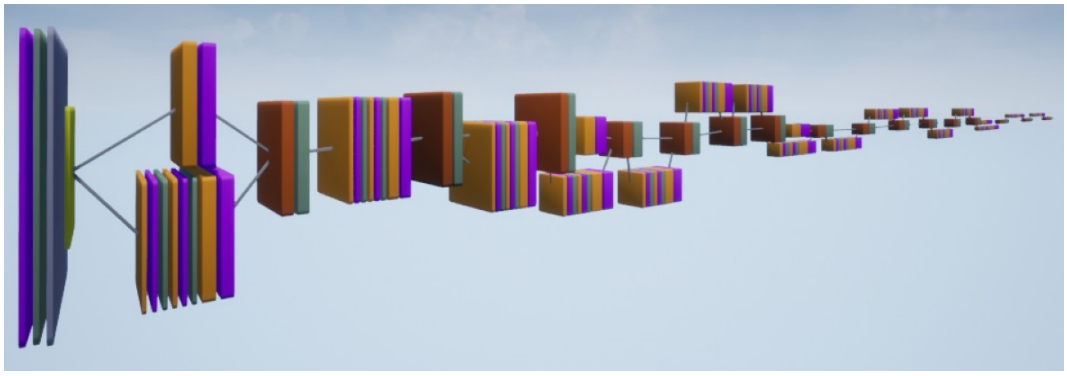

Figure 3.3: Layout of Resnet-50, calculated and visualized by NeuralVisUAL in the UE4 virtual environment.

To visualize a complete neural network in a 3D space, decisions on how to display and align the network's components are necessary. The layout algorithm only focuses on the network's layers as solid objects to simply this process. These objects have specific properties like size, position, and color, depending on the characteristics of their respective layer. The layout algorithm is responsible for deciding on how to distribute and place the layers in 3D space to facilitate the user quickly obtaining an intuitive understanding of the neural network's architecture simply by looking at this collection of objects in the virtual representation, as figure 3.3 demonstrates.

3.2.1 Neural Network Architectures

NeuralVisUAL and the henceforth described layout algorithm focus on feedforward networks. As section 2.1.1 explains in more detail, in this type of network, the information exclusively flows from layers with a lower index to layers with a higher index while permitting residual connections. Consequently, because the network structure represents a directed acyclic graph, the information flow is representable in a particular direction, which aids in giving the user an intuitive understanding of how the network operates and how the input information travels through it. It is crucial for the ability of the visualization to intuitively convey the network's architecture to

consider these residual connections and let them have a direct influence over the layout of the layers in the virtual world. [Kub99, BOU12, SRK95]

NeuralVisUAL's layout algorithm focuses on feedforward networks for this project's scope. However, it is essential to note that alternative non-feed-forward architectures provide a viable and increasingly competitive alternative to traditional feedforward-CNNs, even in classically CNN-dominated areas like computer vision, especially for large datasets [HWC+20, ZKHB21]. For example, attention-based transformer networks appeared as a novel architecture in 2017, deriving their technology from RNNs [VSP+17]. In June 2021, the Vision Transformer ViT-G/14 [ZKHB21] beat all previous state-of-the-art CNNs with a top 1-accuracy of 90.45% [Cod19]. It is important to note that, as different architectures might supersede feedforward solutions, this layout paradigm might be due for a revision in the future.

3.2.2 Dimensions for the layout

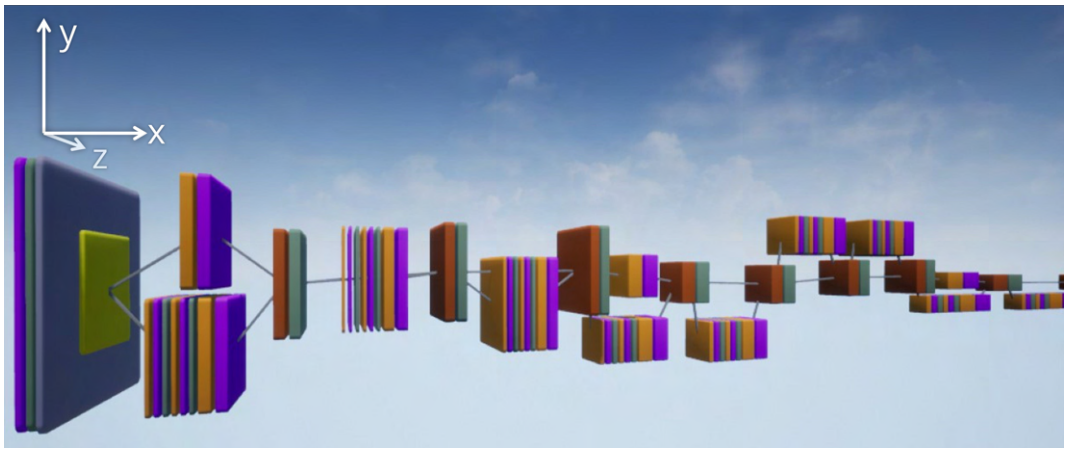

Figure 3.4: Screenshot of the finished layout of Resnet-50 with an overlay of the three axes.

While the visualization takes place for user exploration in three dimensions, the layout algorithm aligns the layer objects along two of those dimensions. The x-dimension facilitates a strict information flow along its axis, usually traversing through the network from the input to the output layer. In contrast, the vertical y-dimension reserves the space to align layers and layer groups in parallel on top of each other, visualizing direct residual relationships between layers and layer groups around skipped layers. The layout algorithm does not utilize the z-dimension. This limitation allows the user to grasp the whole network's architecture from a single perspective without layers occluding each other. Furthermore, it reserves space for other visualizations associated with specific layers and should appear next to their respective layer object.

Layout of a simple MLP

When MLPs, which only fed information from one layer directly to the successive layer, were state of the art, visualizing the order of these layers and their relationship in a meaningful way was trivial. For example, aing at the left, one can place the input layer with its size defined by the shape of parameters in that layer, followed by the subsequent layer to its right, placing all layers linearly along one dimension next to each other. The relationship is intuitively clear: The information gets fed to the network as input on the left and passes through every single subsequent layer, which further processes the information until the last layer on the right present the network's output.

Layout with residual connections

However, when introducing residual connections to this visualization, the layout challenge becomes significantly more complex. One could linearly stack the layers and draw arrows around the skipped layers to note a residual connection. However, this approach could impede the goal of intuitively conveying the architecture to the user if they have to search for such arrows around multiple layers and consciously process the meaning of connections for the flow of information.

To facilitate a more intuitive understanding, layers that directly interact should be closer to each other. Layers that do not have any direct relationship with one another can easily be spread out and do not need to position themselves subsequently next to each other. Therefore, it is sensible to utilize the higher dimensional space for the layers of such a complex network instead of restricting the layout along one axis.

Three-dimensional virtual environment

The visualizations of NeuralVisUAL take place in three dimensions. Although this visual realm usually presents itself on a 2D computer monitor, the capability of interactively and freely moving through this virtual world gives the user a sufficiently good grasp of the make-up of this virtual world so that a 3D representation is not overwhelming. NeuralVisUAL's visualization is meant to be interacted with and moved through similarly to 3D video games.

Time can provide an additional temporal dimension for information. Small animations, color changes, and subtle movement can be implemented in such visualization to convey additional information. The layout only manifests itself in spatial dimensions, keeping itself static in time instead of moving or changing its appearance constantly to keep the virtual world predictable and less overwhelming for the user.

Reserving one dimension

Furthermore, NeuralVisUAL needs to visualize additional information about the layers unrelated to their layout, parameter shape, or information flow in this 3D environment. To easily distinguish between the layout information and the complementary extra layer information, the z-dimension is reserved entirely for such visualizations, restricting the layout of the layers onto two spatial dimensions. This strict distinction between the uses of dimensions helps avoid confusion about the nature of that type of information in users.

This decision also facilitates maintaining an easily comprehensible overview of the whole network's architecture. The user can choose a perspective looking over these two dimensions, where no layers occlude each other. Hence, the user can obtain a clear perspective of the whole network, obtaining an intuitive grasp of its structure.

3.2.3 Goals of the layout algorithm

The layout algorithm maximizes specific priorities to give the user an intuitive overview of the network's architecture with that calculated layout. This section lists these priorities and explains why they are so central to an intelligent, meaningful layout for the user.

Overall, the principal goal of the layout algorithm is to produce a simple, intuitively graspable layout that is only as complex as it needs to be to convey the network structure easily.

Directional information flow

One of these goals is to convey the directional information flow within a network. As discussed above in section 3.2.1, the scope of this layout algorithm is to visualize the architecture of feedforward-style neural networks. Within these networks, information only ever flows from layers with a lower index to layers with a higher index. Since humans prefer some sort of directional indication in directed graph layouts [PAC00], it is helpful to retain this directional information flow along the x-axes of the visualization.

In the generated layout of the neural network, the information between layers should only ever move monotonously to the positive x-direction and never have to go against this direction from one layer to the other. As will be evident later, this property of the layout is highly prioritized and vigorously enforced because any deviation could confuse the user and hinder any easily graspable understanding of the architecture.

Connectedness

Furthermore, the layout algorithm assumes that all layers have at least an indirect connection with each other. Thus, if the layer connections were an undirected graph, it would always be represented as a connected acyclic graph, as is the case for feedforward neural networks.

Other than these two expectations about the connectedness and the direction of information flow, no further assumptions exist about the nature of the network structure. This way, the layout algorithm can handle arbitrarily complex feedforward DNNs, even for future architectures researchers might develop.

Constraint to two dimensions

As explained in section 3.2.2, the algorithm reserves one of the three spatial dimensions for later visualizations; therefore, constraining the layout to a 2D plane.

Negative space between layers

Layer objects naturally should not overlap each other in this space so that the user can easily visually distinguish between the single layers.

Additionally, the layout should utilize negative space between the objects to spread out the layers. This space gives the structure some breathing room; it allows the user to move through the architecture in the visualization without having their vision constantly occluded by layers directly next to each other. The ability to view these layers along the x-axis becomes important when visualizations appear on the planes orthogonal to that axis.

Geographic proximity

Another priority for the layout is that directly connected layers should undoubtedly stay close to each other. In contrast, layers that do not have any direct connection should be kept further from each other, especially along the x-axis.

Furthermore, any group of linearly connected layers should keep together, preferably treated as one unit. Linearly connected layers are each in and of itself only connected to two layers, one as their input and another that receives their output. If the network architecture represented itself as a directed graph, this group would represent any path consisting exclusively of nodes that have precisely two edges each. These groups exclusively contain trivial connections, so they can be kept as a linearly connected unit, ordering their nodes right along the x-axis.

3.2.4 Force-directed algorithms

NeuralVisUAL uses a modified force-directed graph drawing algorithm as the basis for the layout algorithm. Graph drawing algorithms are a flexible method to aesthetically visualize graph layouts without using any domain-specific knowledge [Kob12]. Section 2.3 provides a detailed explanation of the exact mechanics within such FBAs.

Since these force-directed algorithms traditionally work on undirected graphs consisting of nodes without a size, some modifications to the algorithm are necessary in order to satisfy the layout goals defined in section 3.2.3 [Kob12, GFV13].

Reasons for utilizing a FBA

The flexibility of force-directed graph drawing algorithms and their ability to adapt to various input structures make them suitable for NeuralVisUAL's layout algorithm. In contrast, a different type of algorithm that requires significantly more assumptions about the structural composition of neural networks would hinder the ability of the visualization to display any arbitrary architecture that automatically adapts to varying degrees of complexity. [Kob12, GFV13]

Nonetheless, some assumptions and priorities are necessary to be able to produce a viable layout solution. Section 3.2.3 above defines and explains these assumptions in detail.

Through constraints and forces, the force-directed graph drawing algorithm tends to stabilize toward a local optimum that satisfies the parametrically defined goals and priorities in the best possible way [JVHB14, Kob12, GFV13].

User preferences

Moreover, differing subjective approaches exist on what is considered optimal for such a layout algorithm [HL07, Pur98, NL94, JVHB14]. The force-directed graph drawing algorithm has an ingrained tendency to find a suitable solution, with the definition of suitable entirely contingent on the defined rules and priorities for the make-up of the force vectors. [Kob12]

Depending on the user's preferences and the network architecture, it might prove more beneficial to prioritize particular goals of the layout algorithm over others [Pur98, HL07, NL94, JVHB14]. For precisely this reason, the parameters and priorities of the layout algorithm are entirely adjustable to these preferences to give the flexibility to provide the subjectively best result for varying situations. Therefore, instead of calculating the forces continuously in the same manner (like forces in nature), their components' prioritization changes throughout the algorithm's application in a manner specified in the visualization settings.

NeuralVisUAL's open-source code provides suggestions for these settings, carefully chosen by the developer, either for users who do not need to develop optimal parameters on their own or as a starting point for optimizing the parameters based on such a suggestion.

FA2

NeuralVisUAL uses FA2 [JVHB14] for its layout algorithm. Section 2.3.4 discusses the details, advantages, and drawbacks of this specific algorithm.

The continuous adaption of local and global speed makes this algorithm very flexible, performant while at the same time minimizing adverse effects of oscillation on performance [JVHB14]. This feature is very beneficial for this layout algorithm, as users can freely change the priorities and forces in the visualization settings. With FA2, the users do not need to re-calibrate a speed setting to optimize for performance and reduced oscillations, as the automatic speed adaptation adapts and optimizes the force application speed [JVHB14].

Another reason for choosing FA2 is the fact that a well-working implementation of this algorithm is available in python [Chi20]. Using such an implementation reduces development efforts for NeuralVisUAL. Implementation in python is necessary since the layout calculations should take place directly on the python server for performance reasons, as highlighted in section 3.1.3.

Finally, the comparison of FA2 to other FBAs makes it a suitable choice for NeuralVisUAL, as it provides satisfactory results while being performant enough even for graphs with many thousand nodes [JVHB14]. This aids in future-proofing NeuralVisUAL for more complex neural networks with a high layer count.

However, it is essential to note that attributes like usability measures that define the suitability of a layout can vary significantly from person to person and can be challenging to determine [HL07, Pur98, NL94, FHH00]. Therefore, varying users might prefer different layout algorithms for this task, and it is unfeasible to quantify an objectively best algorithm for such a layout [JVHB14]. To counter this issue and provide more flexibility, users have the possibility of changing their preferences and priorities FA2 takes into account, as section 3.2.5 explains. The possible future work of conducting user studies, as mentioned in section 4, could help in this regard to not only compare layout preferences but also analyze the subjective quality of different FBAs.

Python implementation

To calculate the layout of any DNN, NeuralVisUAL utilizes a python implementation of FA2, which Bhargav Chippada developed and published on Github under the GPL-

3.0 license [Chi20]. This implementation applies the FA2 algorithm on undirected graphs defined with the NetworkX or igraph libraries [Chi20].

3.2.5 Modifications

Some modifications and extensions to FA2 are necessary to satisfy the demands of calculating meaningful layouts for arbitrary feedforward neural networks.

One of the main issues is that graph drawing algorithms traditionally operate on nodes that do not have any size or take up any space but instead define single points represented by positions in space [Kob12]. FA2 can consider circular node sizes, given a radius value for each node, but this does not suffice for layers that have a rectangular profile, with often substantially different height and width values. When applying its forces, the algorithm should consider the layers' size and spatial demands, placing high importance on preventing overlapping. Forces between layers should not be calculated based on their geometric centers but the distance between the respective quad faces; otherwise, small layers would have more space, while huge layers could easily overlap.

Additionally, the directional information flow mentioned in subsection 3.2.3 should also be incorporated, as FA2 does not place any importance on the order or indices of the graph's nodes [JVHB14].

Nodes with size parameters

First of all, the graph nodes within the FBA need to have width and height parameters, as figure 3.5 illustrates. The neural network visualization draws the layers as quads that take up a definite volume in their virtual representation. While the quad's color is chosen based on the layer type according to visualization settings, the measurements of that volume directly depend on the dimensions of the layer parameters. The RGB color values for each layer type are accessible in the visualization settings. For layer connections, the user can specify their color, visibility, thickness, and visibility between grouped layers.

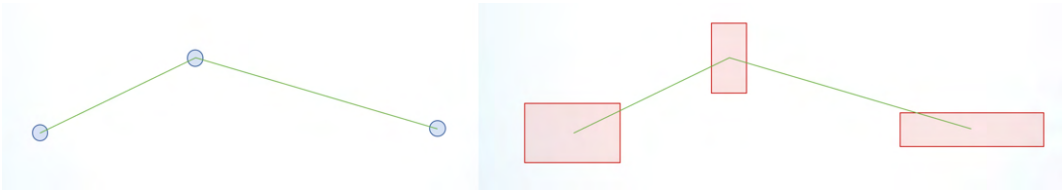

Figure 3.5: Illustration of this modification. Left: Nodes without size parameters. Right: Nodes with width and height, their position remains in the center of the node.

The algorithm retrieves the output shape of that layer from the TensorFlow interface as a list to obtain a size along three axes for a layer object. Since often this shape list as with *None* due to batch normalization, the algorithm ignores any non-numerical values or values smaller than 1 in that list. Then, if the length of that list exceeds three values, the list is shortened to a length of three. If it contains less than three values, the algorithm repeatedly prepends the value one until the list contains three values. These values are scaled and incremented according to values defined in the visualization settings. Optionally, minimum and maximum dimension limits can be specified to constrain the layer object sizes in the virtual world. After these operations, these three values represent the size of the layer object in the virtual environment.

Additionally, the visualization settings specify a horizontal space between layers, vertical space between layers, and a buffer zone. These three values are used in the force calculations to maintain an optimal space between the layers.

These dimensions are stored for each layer, whereas the positions of each node stay unchanged, representing the position of the node center. Therefore, to obtain the coordinates of any corner of that rectangle, one would add or subtract half of these size values.

Attraction towards sides

Since the force-directed graph drawing algorithm traditionally does not consider size parameters, it would simply apply the forces to the centers of the nodes. However, this is not optimal. As explained in section 3.2.3, one of the goals of this layout algorithm is to represent the network's information flow. Therefore, earlier layers with a lower index should always position themselves left of directly connected later layers with a higher index. The attracting forces between directly connected layers should be applied to the sides of the layer objects to optimize the algorithm towards this goal. Thus, the force acts between the right side of the bounding box of the lower-index layer and the left side of the higher-indexed layer's bounding box, including horizontal spacing specified in the visualization settings, as figure 3.6 demonstrates.

That results in two directly connected layers reaching their optimal relative position when the layer with the higher index is to the right of the other layer, vertically centered and separated by the desired horizontal spacing. When the two nodes reach this optimal position, no more attraction forces need to be applied. This state exactly satisfies the goals for directional information flow and a visual separation of the layers, which section 3.2.3 describes. Moreover, it makes the optimal layout, which the FBA attempts to find, attainable without overlapping layers, instead of conflicting with the measures preventing this overlapping.

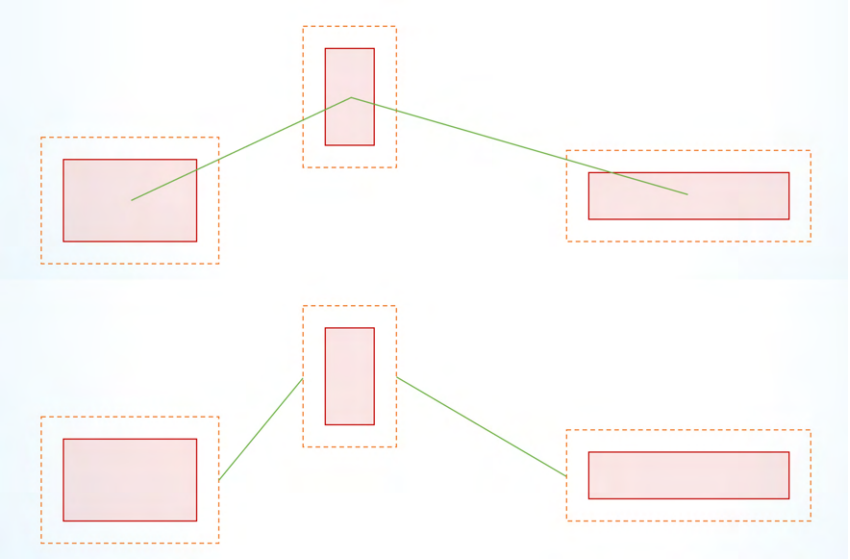

Figure 3.6: Illustration of this modification. Top: Nodes attract each other towards their center point, which would create overlapping. Bottom: Nodes attract each other at their sides, depending on their index. The orange area represents the desired spacing and buffer zone.

Horizontal order

An additional force further aids the goal of visualizing directional information flow. This force only operates along the x-axis on directly connected layers if the information between these layers would flow backward in the x-direction. That means if the right side of the earlier layer plus horizontal spacing has a higher x-value than the left side of the later layer. In that case, the higher-index layer moves to the left, while the other layer shifts to the right, as figure 3.7 visualizes.

This rule only applies when the layout does not satisfy the preference for directional information flow. It directly fixes a reverse information flow without disturbing the other forces of the algorithm when the layout satisfies the directional information flow. Therefore, it can act with quite a high strength.

Overlap repulsion

Along with the attraction force, one of the main driving forces behind FA2 is repulsion. FA2 applies a linear repulsion on any pair of nodes within the network. This force is inversely proportional to the squared distance between the nodes so that the repulsion force is more potent between two geographically close nodes.

With the introduction of rectangular sizes of the nodes and NeuralVisUAL's high priority on producing an intuitively understandable layout, this repulsion force

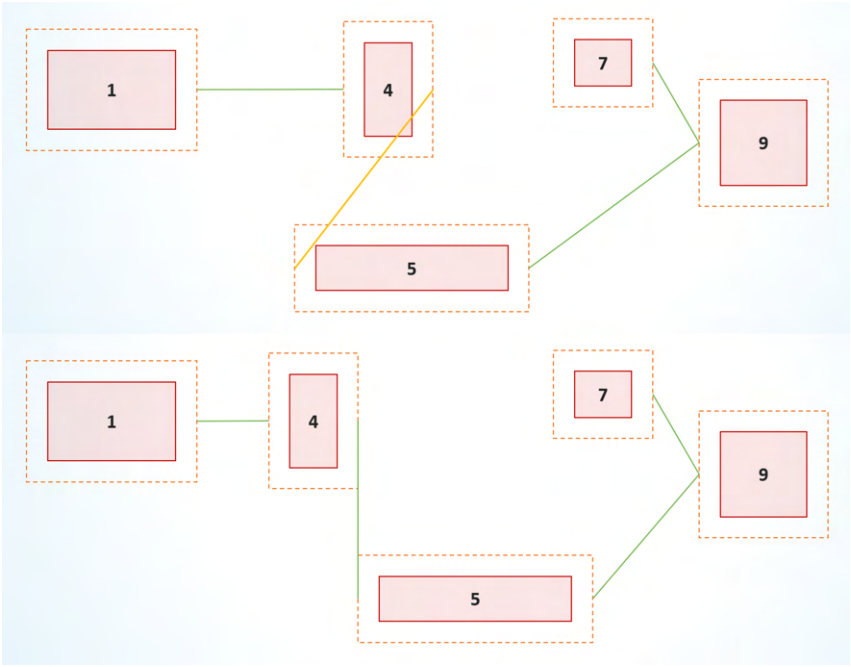

Figure 3.7: Illustration of this modification. Top: Nodes 4 and 5 are aligned in a way that the information would have to flow backwards along the x-axis. Bottom: After correction, the information constantly moves monotonously to the right through the network.

is relatively weak. It tapers off quickly with increasing distance from the node center. NeuralVisUAL complements this repulsion with another repulsion force that considers the layer sizes and only applies when two layers are overlapping or close to overlapping. For this task, the user can specify the size of a buffer zone around the objects, which acts as a soft border for this overlap repulsion. The buffer zone reduces oscillating in the algorithm, which would happen if a strong force suddenly appears by only shifting an object by a minuscule distance.

The algorithm applies the overlap repulsion between each pair of nodes within the network. A quick conservative estimation for computational efficiency disregards most node pairs due to a significant geographical distance, where overlapping is not an issue.

The repulsion is only applied if the nodes have a distance smaller than the specified spacing plus buffer zone between their closest edges. Next, the algorithm applies a sigmoid function to the closer distance between the nodes along either axis while considering the buffer zone. Finally, the overlap repulsion function pushes the two nodes away from each other. However, as the distances between the closest edges are utilized for this calculation, directly using these distances for the force would result in the nodes aligning diagonally at their corners.

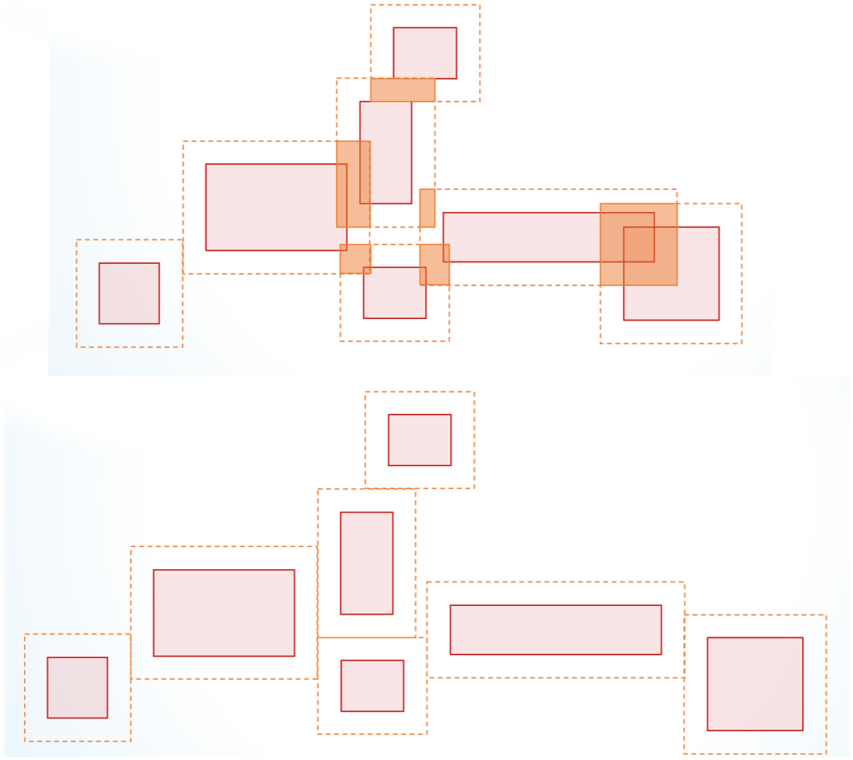

Figure 3.8: Illustration of this modification. Top: Collection of nodes that profoundly overlap each other or each other's spacing and buffer zones. Bottom: After correcting the layout, the nodes shift away from each other. Such a correction for significant overlap does not happen in a single step, but over multiple iterations. It is notable how the nodes tend to take the shorter path away from each other instead of pushing towards each other's corners; this is especially visible between the top two nodes.

Such a diagonal layout is neither the most space-efficient layout and nor the quickest way to separate the nodes visually. For example, if nodes only need a little more vertical separation but significantly more horizontal movement to undo overlapping, it would naturally be more efficient to prioritize vertical movement instead of repelling the nodes to separate horizontally and vertically simultaneously.

The overlap repulsion function of the modified FA2 algorithm achieves this by switching the numerical absolute values of the x and y forces while retaining their sign. Retaining the sign guarantees that the algorithm still improves the overlap situation as the nodes move away from each other on either axis. This way, the algorithm favors moving the nodes along the axis that achieves the desired separation quicker. If the nodes would have to move the same distance on either axis, they are moved along both axes equally. Figure 3.8 demonstrates this advantage.

The overlap repulsion function is crucial to retain enough space between the elements to keep a visual separation, allowing the user to effortlessly distinguish the various layers within the network.

Furthermore, after the FBA finishes all its iterations, one last check over all layers validates the layer positions. If it finds any overlaps, it shifts both affected layers away from each other to create enough space between them to satisfy the specified space between layers. This overlap check benefits from the fact that the FBA additionally works with a buffer zone, therefore utilizing extra space between other layers to shift overlapping layers towards empty space.

Initialization along x-axis

Another aspect of the FA2 algorithm that can be improved is the initial layout utilized before the FBA is applied. As FA2 works on undirected graphs, it distributes the nodes randomly in space as an initial layout to find a local optimum from that layout via force applications.

In contrast, the neural network layers already have some information about their place in the layout due to the strict information flow from layers with a lower index to layers with a higher one, as described above in section 3.2.3. This information can help initialize the layout algorithm with a more helpful layout than random positioning of the nodes. It helps the FBA ease into an optimal solution, as the information flow rule is satisfied right from the beginning and instead of fighting against the other forces in generating a meaningful layout.

Figure 3.9: Illustration of this modification. The layers are aligned consecutively along the x-axis with generous constant spacing.

The layers are placed along the x-axis, sorted by their index to initialize this layout. The first layer moves to a position where its left edge is at the coordinate 0. Every successive layer then places itself to the right so that the space between it and the previous layer equals three times the horizontal spacing plus twice the buffer zone, as figure 3.9 shows. This approach gives the layout algorithm enough room to move the layers around and contract the layout with attraction forces without immediately needing to fix overlapping layers or reverse information flow in the layout.

Consequently, after initialization, the whole layout has a width equal to $n(3 \cdot \text{horizontalspacing} + 2 \cdot \text{bufferzone}) \sum_{l \in \text{layers}} \text{width}_l$, with n being the number of layers in the network. When iteratively applying the forces during the layout process, the

width shrinks as the layout contracts due to attraction forces and arrangement of layers in parallel to other layers.

Group linearly connected layers

Another mechanism added to FA2 is the grouping of linearly connected layers, intending to improve the clarity of a layout more directly related to the network architecture. Such grouping facilitates a result that is easier to understand intuitively, as described in the algorithm's goals in section 3.2.3. As there are no branching connections from the layers within any linearly connected group, calculating the optimal layout of this subgraph becomes trivial.

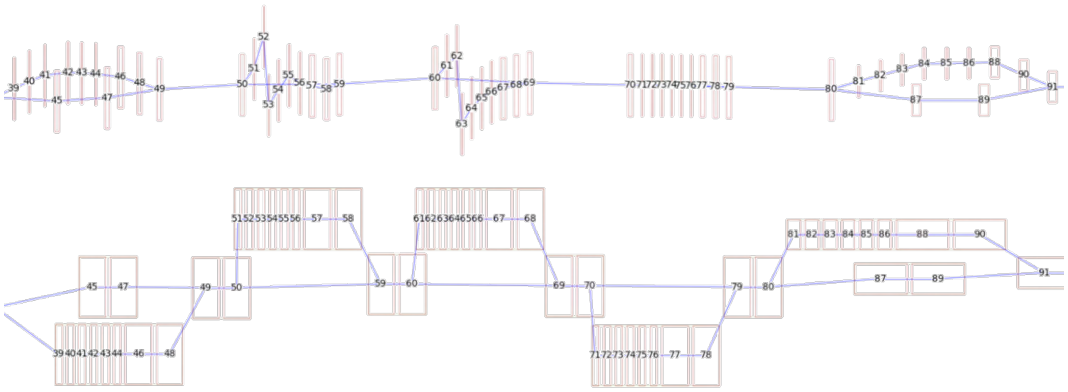

Figure 3.10: Illustration of this modification. Top: Cropped Resnet-50 layout after applying the FBA algorithm without grouping. It is clearly visible how the linearly connected layer groups interweave or split due to pressure from attracting forces to either side of the group. Bottom: Cropped Resnet-50 layout after applying the algorithm with grouping enabled. The linearly connected layers within each group are well aligned and treated as a solid object.

Every successive layer is aligned vertically centered and to the right of the previous layer, separated by a user-defined distance, aing with the position of the earliest layer within that group. The user can set this distance in the visualization settings under the name horizontalSpacingWithinGroup. This way, the user has complete control over the geographic closeness of these grouped layers and can set it to a value smaller than the default horizontal spacing. Thus, it is possible to easily visually communicate the strictly successive nature of that part of the processing pipeline within the architecture.

Furthermore, the layout process is simplified, increasing the speed of the entire layout algorithm. It removes unnecessary complexity in parts of the graph that do not need this level of complication, producing a visual result that is quickly comprehensible. Also, this grouping mechanic minimizes possible edge crossings

and bends in the layout, thus improving aesthetics and usability, since a predominant majority of users dislike these attributes in graph layouts [PAC00, Pur97, PCJ97].

The optimal solution for structuring the network's subgraph that correlates with any such group would always gravitate towards this layout. This tendency is due to the forces reaching a minimal equilibrium when linearly connected layers order themselves consecutively and vertically centered along the x-axis. The ability to specify an exact distance between layers within this group does change this solution towards more user control. However, if the user wishes, they can always set the grouped layer distance to the median distance between ungrouped layers that organically arises with horizontal spacing and buffer zones.

After positioning all individual layers in this group along the x-axis, the layer group then combines its nodes into a singular node for the FBA. The combined node of that group obtains the group's geographic center, the highest layer's height, and the width of the whole group as its own position, height, and width, respectively. The individual layers of that group stay rigidly connected during the layout process and have no possibility of shifting away from each other, keeping their relative position. The overlap repulsion force interprets empty space between linearly connected layers as an overlap with any other layer. This handling prevents other layers from squeezing into the space between individual layers of such a group, as figure 3.10 demonstrates.

A possible drawback of this grouping becomes apparent if the groups are relatively large due to an architecture that employs many successively directly connected layers without branches. In this case, the large groups take up significant areas. They could limit the algorithm's flexibility in finding a suitable layout, as these groups prohibit other layers from permeating them and switching sides across the group or interweaving with the layers of the group.

Because of this drawback of the grouping approach, the feature of grouping linearly connected layers can be disabled in the visualization settings. The user should only disable this flag if the algorithm encounters such an architecture with which this grouping is counter-productive to producing a subjectively good and easily understandable layout.

Force strength variation over the algorithms execution cycle

The ultimate modification to FA2 is introducing variation within the forces during the algorithm's application cycle. The user can specify which features to enable, in which iteration cycles of the algorithm each force is active, with which strength. Furthermore, this strength can be constant, increasing or decreasing at varying exponential rates. These customizations allow the user to tune the algorithm towards prioritizing varying goals at different periods of the layout process and even separate the process into discrete steps applied one after the other.

Section 3.2.7 presents the exact mechanisms of how these forces are prioritized and with what settings the user can calibrate this combination.

3.2.6 Forces in the layout algorithm

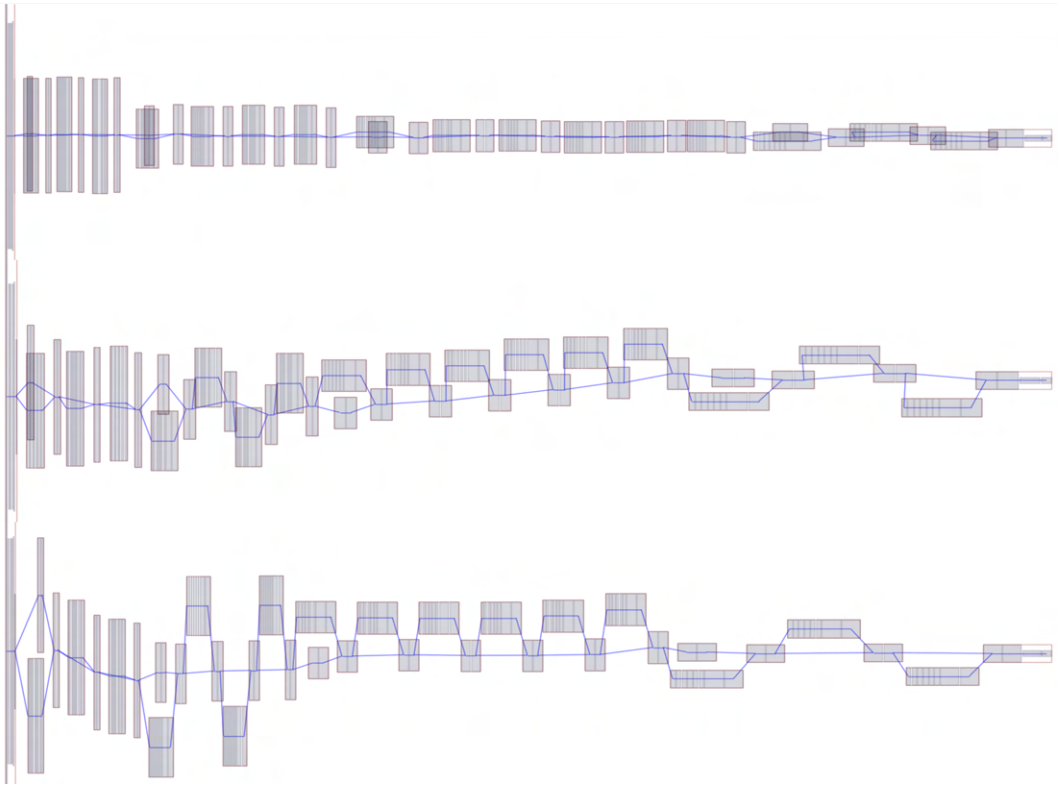

Figure 3.11: Progress of the FBA on a Resnet-50-architecture, snapshots at iterations 240, 720, and 1200.

Figure 3.11 shows how the FBA operates to generate a layout for the residual neural network Resnet-50. The appendix contains screenshots from ablation experiments with differnt force combinations, showing what happens with the layout when a certain force is deactivated. It also shows how much node movement each single force generates per layout iteration.

Mathematical definitions

The modified FBA contains five distinct forces.

Let G be a directed graph representing the network structure with vertices V for all layers and edges E representing the direct connections between layers.

Additionally, for the layout let each vertex $v \in V$ have a position $\mathbf{p}_v = (x_v, y_v)$, width w_v, height h_v, and index n_v that determines the network's information flow.

A position change on vertex $v \in V$ over layout iteration i by force f is defined as $\Delta\mathbf{p}_{v,f}(i)$

The distance between two vertices u, v is defined as
$\delta(u,v) = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$

The visualization settings specify horizontal spacing h, vertical spacing g, and buffer zone b.

The specific strength factor γ for force f at iteration i specifies how strong the force is at that iteration, according to the visualization settings. The mathematical definition of the specific strength factor appears at the end of this section.

Connected attraction

Connected attraction is the main attractive force within the algorithm and is responsible for bringing together connected nodes. It also originates from ForceAtlas 2 and is only applied on any combination of two directly connected nodes. In each iteration, the force applies to every node pair, pushing the two nodes towards each other. The strength of that force is proportional to the distance between the two nodes, meaning that connected nodes further away from each other experience a stronger push. The force does not apply to the nodes' geometric centers but rather to the layers' sides, depending on the order of their indices to improve directional information flow. This calculation includes a user-defined horizontal spacing value, as section 3.2.5 explains in more detail.

Formula for connected attraction: $f = connectedAttraction$

$$\Delta\mathbf{p}_{v,f}(i) = \sum_{u \in V | (u,v) \in E} \left(x_u - x_v + h + b + \frac{w_u + w_v}{2}, y_u - y_v\right) \cdot sgn(n_v - n_u) \cdot 100 \cdot \gamma_f(i)$$

Shift on axis to order by index

This force is not present as a force in the original FA2 but is added as a modification to achieve the goal of directional information flow. It exclusively applies to any two directly connected nodes that do not satisfy the goals for directional information flow. In this case, the two nodes move along the x-axis with force proportional to the horizontal component of their distance vector. Section 3.2.5 describes this force in more detail.

Formula for shift on axis to order by index: $f = shiftOnAxis$

$$\Delta\mathbf{p}_{v,f}(i) = \sum_{u \in V} \left(x_u - x_v + h + \frac{w_u + w_v}{2}, 0\right) \cdot 100\gamma_f(i) \cdot \max\{0, sgn(n_v - n_u) \cdot sgn(x_u - x_v)\}$$

Overlap repulsion

Overlap repulsion is another force added as part of the modifications to FA2 for this layout algorithm. It assists the traditional repulsion force by more directly preventing layer overlapping due to consideration of the layer sizes. When layers come too close to each other due to their size and user-defined spacing and buffer zone, this force pushes the nodes apart. As it only is active when layers are overlapping or close to overlapping, this force can act much stronger than the classic repulsion force, which is consistently active. Section 3.2.5 explains the overlap repulsion and its strength calculation in more detail.

Formula for overlap repulsion: $f = overlapRepulsion$
$\chi(u,v) = \max\{0, h + b + w_v/2 + w_u/2 - |x_v - x_u|\}$
$\psi(u,v) = \max\{0, g + b + h_v/2 + h_u/2 - |y_v - y_u|\}$

$$\Delta\mathbf{p}_{v,f}(i) = \sum_{u \in V} (sgn(x_v - x_u)\psi(u,v), sgn(y_v - y_u) \cdot \chi(u,v)) \cdot$$
$$\gamma_f(i) \cdot sigmoid\left(\max\{\chi(u,v), \psi(u,v)\}\frac{6}{b} + 1\right)$$

Classic repulsion

Classic repulsion is a repulsive force that directly originates from the FA2 algorithm. It affects any combination of two nodes with a strength inversely proportional to the squared distance between them, similar to the repulsion of particles of the same electric charge in the physical world. This force is disabled by default, as the overlap repulsion force better fits the rectangular shape of the nodes.

Formula for classic repulsion: $f = classicRepulsion$
$$\Delta\mathbf{p}_{v,f}(i) = \sum_{u \in V} (x_v - x_u, y_v - y_u) \cdot \delta(u,v)^2 \cdot \gamma_f(i)$$

Gravity

Gravity is an attractive force directly adopted from the FA2 algorithm, which also applies itself between any combination of two nodes. The strength of this force is inversely proportional to the linear distance between the nodes. This approach is in stark contrast to actual gravity in the real world, which operates based on the squared distance. However, the goal of the gravity force in FA2 is not to stick geographically close nodes firmly together, which could create gravity wells comparable to heavy objects in the universe. Instead, the goal of this force is to keep all nodes in general slightly closer together, preventing them from drifting apart or spreading out more than would be beneficial [JVHB14].

This force is disabled by default as well, as any feed-forward network can convert into a connected graph. Therefore, no force other than the connected attraction force

is needed to keep all elements together.

Formula for gravity: $f = gravity$
$\Delta \mathbf{p}_{v,f}(i) = -\mathbf{p}_v \frac{\gamma_f(i)}{\sqrt{x_v^2+y_v^2}}$

3.2.7 Combining forces

These five forces are not applied in the same manner constantly in every iteration. Instead, they vary throughout the application cycle of the layout algorithm. They can incorporate decreasing or increasing levels of strength or be deactivated completely over some iteration cycles.

This way, the prioritization of different layout goals can change over time, which allows for a coarse structuring of a rough layout, moving towards a more precise approach for fine-tuning the layer positions in the end. The user can change these priorities and discretize the algorithm into completely separated layout steps by only activating the forces for specific iteration ranges.

Visualization settings for force combination

For the layout process, the user can define how many iterations the FBA should execute. By default, visualization settings suggest 1200 iteration cycles.

For each of the five forces defined above, the user can specify the following parameters:

strength acts as a constant factor on the force, allowing the user to scale the force's influence on node movement. Default value: 1

withinIterations specifies the range of iterations in which the force is active. The force is enabled while the iteration index is between the tuple's two numbers, including the start and excluding the end value. Floating-point numbers in the tuple define a fraction of the iteration range. The algorithm multiplies them with the total number of iterations. An optional third parameter defines the step size within that range. If the user omits this parameter, the force stays active throughout all iterations by default.

importance is given as a string and defines how the strength changes over the previously specified *withinIterations*-range:
'disabled' completely deactivates the force for the layout algorithm.
'constant' is the default, which results in the force being constantly at full strength.
'increasing' means that the force as at 0 and increases to full strength.
'decreasing' means that the force as at full strength and decreases to 0.
'middle' means that the force as at 0, increases to full strength, and decreases back to 0.

'outsides' means that the force as at full strength, decreases to 0, and increases back to full strength.

exponentialCurveFactor defines how strongly the varying importance curves over the course of the *withinIterations*-range. A zero value denotes a linear increase or decrease; above zero signifies a downward curve, and negative values an upward curve. This curve becomes more strongly pronounced with increasingly extreme values farther away from zero.

Additional settings related to gif rendering allow the user to generate an animation of the layout process.

NeuralVisUAL's algorithm adds up all of the movements $\Delta p_{v,f}$ for each node v, that arise from applying the forces f to all nodes or a combination of nodes. Before applying this movement, the function *adjust speed and apply forces* from FA2 adjusts the speed to optimize jitter tolerance, avoiding erratic behavior from oscillation, which would make the algorithm less effective [JVHB14, Chi20].

Calculating the specific strength factor per force

The specific strength factor for each force based on these settings and the iteration index is defined and calculated as follows:

Let f be the force with the iteration range $\mathbf{w}_f$ as specified in the visualization settings $\mathbf{w}_f \in \{\varnothing, \langle a_f \rangle, \langle a_f, k_f \rangle, \langle a_f, k_f, t_f \rangle\}$

a denotes the start, k the end, and t the step of the iteration range. The total number of iterations is defined as z

$$a_f = \begin{cases} 0 & \text{if } \mathbf{w}_f = \varnothing \\ (\mathbf{w}_f)_1 & \text{otherwise} \end{cases}$$

$$k_f = \begin{cases} z & \text{if } \mathbf{w}_f = \varnothing \\ a_f + 1 & \text{if } \mathbf{w}_f = \langle a_f \rangle \\ (\mathbf{w}_f)_2 & \text{otherwise} \end{cases}$$

$$t_f = \begin{cases} (\mathbf{w}_f)_3 & \text{if } \mathbf{w}_f = \langle a_f, k_f, t_f \rangle \\ 1 & \text{otherwise} \end{cases}$$

The range of iterations in which the force is active is therefore defined as

$$\mathrm{r}_f = \{q \in \mathbb{Z} \mid a_f \leq q < k_f \ \wedge \ \exists\, o \in \mathbb{Z} : a_f + o * t_f = q\}$$

The progress of an iteration i within the defined range of iterations is defined as α:

$$\alpha_f(i) = \frac{i - a_f}{k_f - a_f}$$

The exponential curve ϵ is calculated as follows, with c_f being the exponential curve factor for the force f:

$$\epsilon_f(\alpha) = \begin{cases} \alpha & \text{if } c_f = 0 \\ (e^{c_f \alpha} - 1)/(e^{c_f} - 1) & \text{otherwise} \end{cases}$$

Finally, the specific strength factor γ for force f at iteration i is calculated with the following formula, m_f being the importance and s_f being the general strength of that force as specified in the visualization settings:

$$\gamma_f(i) = s_f * \begin{cases} 0 & \text{if } i \notin \text{range}_f \\ 0 & \text{if } m_f = \text{"disabled"} \\ 1 & \text{if } m_f = \text{"constant"} \\ \epsilon_f(\alpha_f(i)) & \text{if } m_f = \text{"increasing"} \\ \epsilon_f(1 - \alpha_f(i)) & \text{if } m_f = \text{"decreasing"} \\ \epsilon_f(2 * \min\{\alpha_f(i), 1 - \alpha_f(i)\}) & \text{if } m_f = \text{"middle"} \\ \epsilon_f(2 * \max\{\alpha_f(i) - 0.5, 0.5 - \alpha_f(i)\}) & \text{if } m_f = \text{"outsides"} \end{cases}$$

Recommended force variation settings

The recommended settings for all five forces are as follows:

connectedAttraction:
$s = 1$
$\mathbf{w} = \langle 0, 0.9 \rangle$
m = "decreasing"
$c = 1$

overlapRepulsion:
$s = 3$
$\mathbf{w} = \langle 0, 1 \rangle$
m = "increasing"
$c = 3$

shiftOnAxis:
$s = 1.4$
$\mathbf{w} = \langle 0, 1 \rangle$
m = "increasing"
$c = -3$

classicRepulsion:
$s = 1$
$\mathbf{w} = \langle 0, 0.9 \rangle$
m = "disabled"
$c = 0$

gravity:
$s = 1$
$\mathbf{w} = \langle 0, 0.9 \rangle$
m = "disabled"
$c = 1$

The algorithm as with the initialization of layers on the x-axis, according to their index. Right at the beginning, the algorithm puts a high priority on connected attraction. In combination with the initialization along the x-axis, this force quickly produce a coarse blueprint for a structure in which directly connected layers are close to each other while increasingly repelling layers that come too close. Apart from the attraction force reaching for the sides of the layers, the early iterations of this algorithm place no importance on strict overlap repulsion nor on shifting the layers along the x-axis if they prevent directional information flow.

In the following 1080 iterations, attraction becomes less relevant, while overlap repulsion and shifting along the x-axis become increasingly important, as figure 3.12

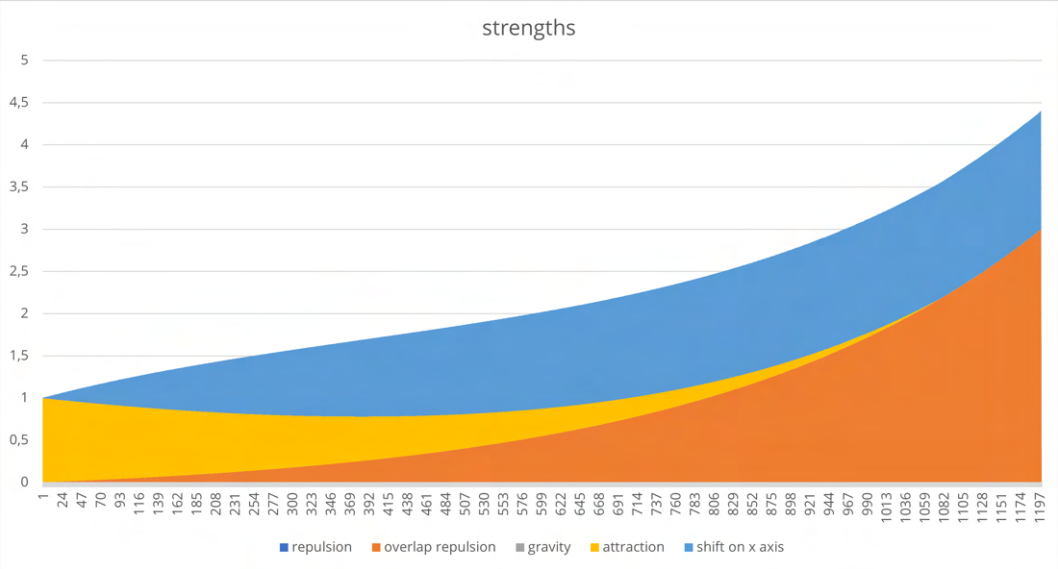

Figure 3.12: Diagram showing the varying strength of each force over the 1200, according to the default settings.

shows. The job of these two forces is to enhance this layout further, ensuring that every layer has enough space and that the information flow criterion is satisfied. After passing iteration 1080, for the remaining 120 iterations, only these two forces remain; classic repulsion, gravity, and attraction are inactive. This final step aims to fine-tune the layout, finally settling down at a locally optimal solution.

3.3 Visualizations

NeuralVisUAL provides several visualizations for the CNNs that it analyzes. These visualizations include rendering the kernels, showing activation maps for each kernel, visualizing saliency maps, and displaying integrated gradients. All of these visualizations take place on a per-layer basis. Kernels and activation maps are only available for convolutional layers.

NeuralVisUAL's client positions these visualizations in its virtual environment on the same plane as the corresponding layer. This way, the user always knows to what layer these visualizations belong.

The user can highly customize each visualization in the visualization settings, which are accessed by the server whenever it renders a new request. These settings allow the user to fine-tune how the visualization conveys the information in the subjectively best manner.

Furthermore, the server caches all generated images to send them as a texture and

to reuse them for later visualizations. Each of these cached files contains the relevant settings in the filenames. Therefore, if the user changes any setting, the server renders and caches files with the new preferences without deleting the old ones.

3.3.1 Kernels and activation maps

The kernels themselves are relatively trivial to retrieve, as the server can access the kernel weights directly from the loaded neural network's trainable variables as a NumPy-array.

The server then normalizes these values in the range from zero to one, ensuring that all kernels of the same layer use the same scale so that one brightness level later always corresponds with the same underlying numerical value.

If the kernel has at least three dimensions and the third dimension, depth, has the length three, then the kernel is interpreted as an RGB kernel, being automatically rendered to the corresponding color channels, as figure 3.13 demonstrates. Otherwise, the kernel weights translate directly to grayscale colors.

When calculating the colors and layout of the kernels, the server considers visualization preferences. Color settings include contrast, brightness, and texture opacity. Brightness can be set from zero, black, via fifty, default, up to hundred, white, applying an exponential function to the normalized weight values. Given a weight w between 0 and 1 and a brightness level b, the new weight w' is calculated as follows: $w' = w^{(100-b)/50}$ for $b > 50$, $w' = w$ for $b = 50$ and $w' = 1 - (1 - w)^{b/50}$ for $b < 50$ The contrast value is applied before brightness with the same formula, only that the server normalizes the weight values between -1 and 1 beforehand. Any negative values on that scale are negated, transformed with contrast, and negated again. Contrast also modifies the saturation for RGB kernels, as the server transforms each color channel individually.

The layout settings include default, minimum, and maximum pixel dimensions for each weight, spacing between kernels, distance to the network layer, wrapping kernels in multiple rows, removing spacing between 1×1-kernels, texture resolution, and anti-aliasing.

The server could then send instructions to the client to spawn every single kernel weight as an individual cuboid. However, as this has several performance drawbacks and the virtual world has problems dealing with over 2000 game objects, the setting to render the kernels as a singular texture image should stay enabled. This texture can then be spawned onto a plane in the game environment, resulting in significantly better performance.

The activation maps are created and combined in a layout similar to the kernels. The data is read out as output of a partial copy of the CNN, transformed and rendered into a texture. Furthermore, the server spawns the image used for the network's

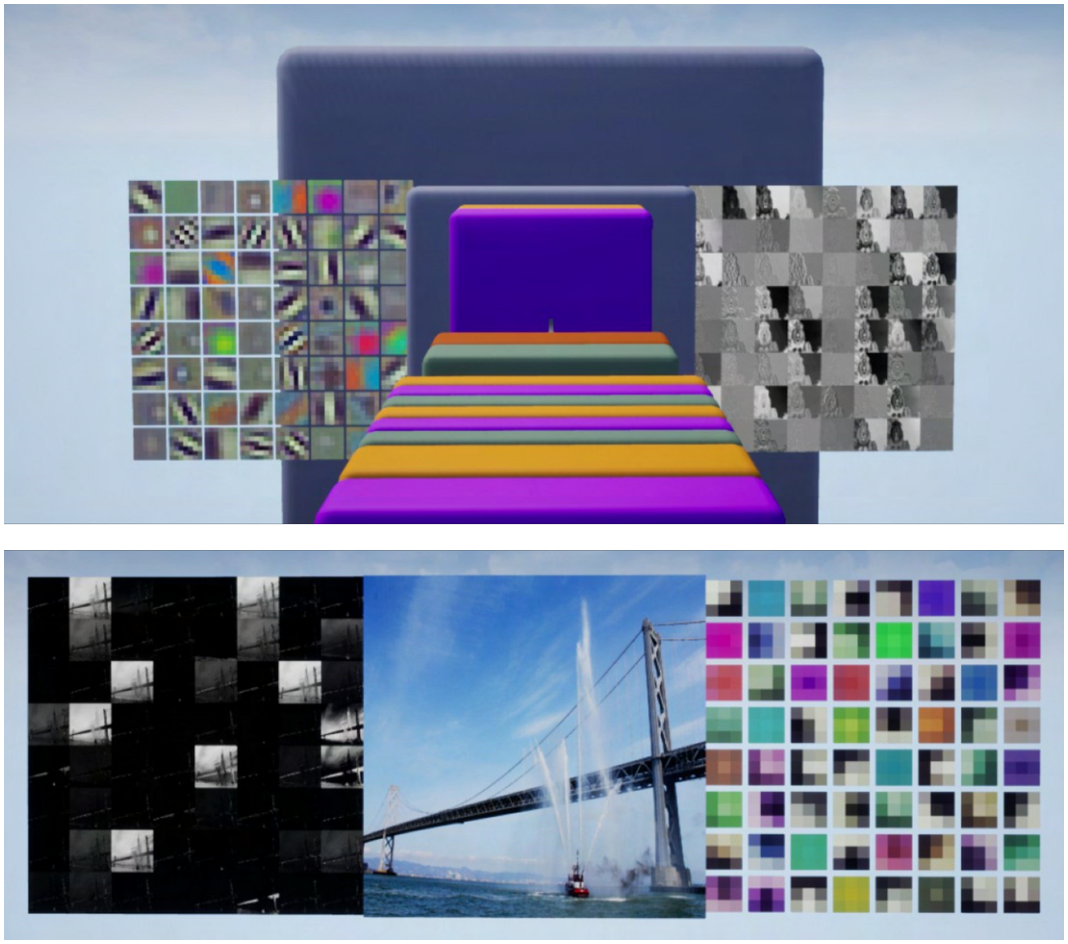

Figure 3.13: Kernels and activation maps for the first convolutional layer of Resnet-50 (top) and VGG-16 (bottom, viewed from the other side), screenshots from NeuralVisUAL.

prediction right in front of the first layer to visualize the input data, as figure 3.13 shows. Additionally, the predicted output class and its certainty appear in the server output and the UE4 console.

Activation maps appear in many neural network visualizations, as they provided a relatively straightforward strategy to display the calculations happening inside of the CNN. For example, NeuralVis [ZYF+19], TensorSpace.js [Ten19], the deep learning development environment by VanHorn et al. [VZC19], and the CNN node-link visualization by Harley [Har15] all utilize activation maps in their visualization environments. Section 1.2 provides more details about each of these applications.

3.3.2 Saliency Maps

The server retrieves the saliency map over a specified prediction class from the TensorFlow/Keras network in the manner described in section 2.2.3. The server then applies a gaussian blur over the whole matrix with a radius defined in the visualization settings. This blur helps to distribute extraordinarily high singular values over a larger area to make the result visually easier to grasp. The server then divides this matrix by a normalization factor that the user can specify in the visualization settings. If no factor is specified or the value range exceeds this factor, the maximum value of the matrix serves as a factor. Additionally, the server applies a brightness with the same formula as described for the kernels.

Figure 3.14: Saliency maps for different input images on their top predicted class, generated by NeuralVisUAL on a pre-trained VGG-16 model.
Top row: Saliency maps of three different images with opacity mixing, top right: additive mixing.
Bottom row (on the same input image): brightness 50 without blur, brightness 70 without blur, brightness 70 with blur radius 5, brightness 70 with blur radius 1.5 on the colormap *hsv* instead of *magma*. The first three images of the bottom row have full opacity for all saliency values. Images generated with NeuralVisUAL, using a pre-trained VGG-16 network.

Additionally, the user can define a colormap for the saliency, and opacity values for the minimum and maximum saliency, letting the original input image with its individual opacity setting show through for specific value ranges. This opacity gradient also has a brightness value which determines how quickly the opacity changes in between. Additive mixing between colormap and input image is also available as an option. Figure 3.14 demonstrates these different settings.

3.3.3 Integrated gradients

The server is also able to obtain the integrated gradient for any layer, utilizing the calculations described in section 2.2.4 based on a zero-baseline. Figure 3.15 shows an example of an integrated gradient generated this way.

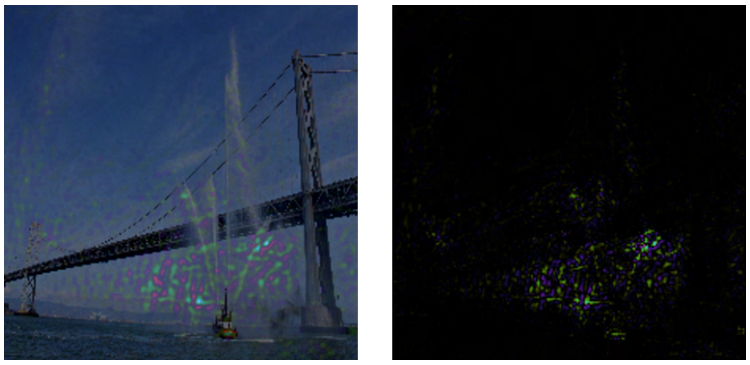

Figure 3.15: Integrated gradient for the same image over prediction fireboat (predicted label with 98.99%). In the dark image, the integrated gradient has full opacity. Both images use the colormap magma with inverted hues for negative values and were generated with NeuralVisUAL, using a pre-trained VGG-16 network.

Similar to the visualization settings for saliency maps, the preferences include a gaussian blur radius, normalization factor, brightness, colormap, opacity at strongest and weakest values, opacity brightness, image opacity, and additive mixing.

Additionally, the integrated gradient initially possesses three dimensions, including RGB channels. The user can select whether to keep this information, translating it to dark and bright values, or to average these color channels and apply a custom colormap to the values.

Since the integrated gradient also contains values in the negative direction, the user can use a second colormap, apply the positive colormap with inverted hues or normalize all values into one colormap, leaving the middle color for neutral values.

4 Future Work

Current status

NeuralVisUAL has been developed and tested on a 64-bit Windows machine, with the WebSocket connection running through localhost. It works on feed-forward DNNs that utilize the TensorFlow and Keras libraries.

As section 3 explains in detail, NeuralVisUAL can calculate a meaningful layout for the structure of these networks and is able to visualize kernels, activation maps, saliency maps, and integrated gradients within the virtual environment.

Modularity

NeuralVisUAL is the basis for an expandable solution designated for interactively visualizing DNNs. Due to the modular nature of NeuralVisUAL, as explained in section 3.1.3, it is uncomplicated for developers to expand NeuralVisUAL for more use cases or add new visualizations to it.

Therefore, countless possible extensions can increase the capabilities of this framework, providing even better interaction opportunities to users. This section discusses a selection of suggestions describing how developers can improve and expand NeuralVisUAL in the future.

User guidance

Interactivity and user guidance could have room for improvement in future work. At the moment, the user has to interact with a specifically created console within the UE4 client to interact with the server, load networks, and request visualizations. A comprehensive graphical user interface in the game environment could guide the users through this process, which becomes essential for user groups outside the AI research sector, like for educational purposes.

Performance and compatibility

Furthermore, NeuralVisUAL would benefit from increased compatibility with various existing technologies. For example, future work can extend the visualizations to work with Tensorflow projects that do not utilize Keras. It could also expand

the scope of NeuralVisUAL to interact with neural networks developed in PyTorch, another open-source python library for ML [Ket17]. Moreover, the UE4 application could be built and tested for other platforms, including Linux. This way, the developers can directly open the UE4 visualization on the Linux machine hosting the network if it permits it.

Finally, a proper multithreading approach could replace the asynchronous solution explained in section 3.1.3. This approach would eliminate rare occasions when a network import or data retrieval blocks the server. It can also lead to parallelizing some computationally expensive calculations that are generating the visualizations. Depending on the client's processor hardware, utilizing multiple cores could bring a performance boost to NeuralVisUAL's visualizations.

Layouting recurrent networks

In the future, with the advent of network architectures differing from feed-forward approaches, such as recurrent and transformer networks, it could become helpful to conceptualize how to layout their structure in the virtual world. For these networks, it might be necessary to relinquish the layout goal of preserving directional information flow along a spatial axis. Instead, it could prove helpful to utilize all three spatial dimensions for the layout of such networks. In this case, the visualizations would need to find another space; they could be projected right onto the layer, squeeze into the empty space between layers, use transparency to fade out nearby layers, or directly display themselves on a GUI overlay.

Moreover, the layout algorithm of NeuralVisUAL only provides a locally optimal solution for the goals it pursues. Developers might choose to implement a more sophisticated algorithm, that better approaches the globally optimal solution for the layout demands, if they feel that the FBA does provide the best results.

Visualizations

Furthermore, to facilitate more interactivity, the objects spawned for visualizations should have a reference and be directly accessible by the server in order to change their properties later. This way, the server could remotely manipulate the opacity, color, position, or size of already created shapes, having more influence over the virtual world and increasing interactivity.

It could also be helpful for specific contexts to display individual neurons, visualizing their weight data, neuron activations, and possibly even the backpropagation process. Such a visualization could prove helpful for educational applications when explaining the fine details of calculations within neural networks. NeuralVisUAL could even show the influence of specific layers, kernels, or neurons on the overall prediction result.

Advanced feature proposals

Additionally, future features could allow the user to select training checkpoints of the analyzed network, displaying the difference in the network between various checkpoints. This comparison allows AI developers to assess the progress of the network along its training process. This feature could enable users to determine which parts of the networks significantly change their weights during training and which parts they could conceivably leave out of the network's architecture.

In CNNs designed for computer vision, the kernels and neurons recognize edges, textures, patterns, parts, and even objects, depending on their location in the network's architecture [OMS17]. To demonstrate the specific features that these parts of the network are looking for, feature visualization generates inputs explicitly optimized to trigger the analyzed network element using gradient-based methods [OMS17, ZZ18]. If implemented reliably, this kind of visualization could be a beneficial addition to NeuralVisUAL, providing more insight into the detection and activations mechanisms of the neural network.

Moreover, visualizing embeddings and providing the user with a method to interactively explore the network's latent space could be a fascinating addition to NeuralVisUAL [LNH+18, STN+16], as the 3D environment already exists in the game engine. Applications for this feature reach from debugging in AI research to educational purposes.

Alternatively, developers could extend NeuralVisUAL to visualize spatial activations. These activations generate examples to demonstrate which features the hidden layers of the network have learned during training [CAS+19]. Millions of these activations can then be united into a complex activation atlas of the network, providing a global overview over the features that the hidden layers recognize [CAS+19]. NeuralVisUAL could implement this visualization in future work and provide an interface to interactively explore this atlas in the virtual game environment, similar to latent space exploration.

An interactive VR visualization would also be a compelling expansion to NeuralVisUAL, especially if it visualizes processes like neuron activations, slowly showing the training and prediction process.

Including sound design into the project could aid in immersion [Gri12], especially to figuratively feel the activations and flow of data. Such addition of sound can be beneficial for educational applications, making the application more engaging [Gri12], ultimately increasing excitement over this technology among users unfamiliar with AI research.

User study

A user study could help test the users' reception and interpretation of the visualizations and find out what visualization features are helpful to users [Tor14]. These studies can help verify if the visualization settings and layout goals are reasonable or should be modified to facilitate a better intuitive understanding of the networks [Tor14, CM07]. With such a study, researchers can work out which visualization details work best on which network architectures. Naturally, this highly depends on the intended target group [Tor14, Pla04]. Software developers or AI researchers would predominantly prefer a different visualization environment than students learning of AI the first time.

Independent of such survey results, it is beneficial to leave the options to specify individual preferences in the visualization settings, giving users the freedom and flexibility to change NeuralVisUAL's properties if they so desire.

5 Conclusion

This thesis presents NeuralVisUAL, a modular open-source application designated to visualizing feed-forward DNNs using UE4 [Inc21]. It utilizes a modified FBA based on ForceAtlas 2 [JVHB14] to calculate an easily comprehensible 2D layout that aims to visualize the directional information flow, keeping connected layers close while maintaining space between unrelated ones. After calculating and rendering this layout in the virtual environment, the application can visualize convolution kernels, activation maps, saliency maps, and integrated gradients within the network.

NeuralVisUAL contains several distinct modules connected by well-defined interfaces. Most crucial is the separation between server and client, which allows the visualization to run on a different machine than the neural network to be analyzed. The python server directly loads the neural network, interacts with it, and processes its data. Furthermore, it is responsible for calculating visualizations from this obtained data, according to visualization settings that the user can adapt to each project. To render these visualizations, it generates instructions for manipulating the virtual world and sends them to the client via a WebSocket connection [FM11], serialized with msgpack [Fur19]. Within the NeuralVisUAL's client module, a UE4 C++ plugin receives these instructions and calls blueprint functions [Inc20b] during this unpacking process. These methods, designed in Unreal's blueprint visual scripting system [Inc20b], are responsible for caching the relevant data from unpacking, interpreting the instructions, and spawning objects in the virtual world. Finally, UE4 renders this world, providing an interactive game rendering environment and allowing the user to freely explore the visualization as a 3D representation [Inc21]. The user exclusively interacts with the neural network through this application and can send commands to the server through a custom console within the game environment. The client relays those commands back through blueprints, the C++ plugin, and the WebSocket connection to the server. The python server finally asynchronously receives and processes these commands, calling the corresponding methods to fulfill the user's wishes, such as requests for new visualizations.

This modularity provides several key advantages. It relieves development efforts due to circumventing relatively long C++ compilation times, yet still providing a high performance for computationally expensive calculations, especially when the server runs on a high-performance machine. Furthermore, the separation between server and client facilitates analyzing networks on servers dedicated to ML research while executing the UE4 rendering application on a client machine. Finally, this modularity permits quick adaptation of NeuralVisUAL to other use cases, extending

its functionality or even modifying modules to work for completely different use cases.

The primary purpose of NeuralVisUAL is to further the research in the field of neural network visualizations. DNNs, especially for computer vision, possess a high level of complexity, often having hundreds of millions, or even billions of trainable weights [SZ14, BMR+20, SMM+17]. This complexity makes it relatively difficult for researchers to comprehensively grasp these networks' precise mechanisms to generate super-human results [ZZ18, ZGCH21]. The fields of neural network interpretability and visualization help answer this question, aiming to give researchers and developers more profound insight into the inner workings of AI [ZZ18].

NeuralVisUAL currently works exclusively on feed-forward DNNs developed with TensorFlow and Keras, has been tested on a 64-bit Windows computer with the WebSocket connecting through localhost. Due to its adaptability, the provided visualization can be extended, depending on the project-specific requirements of such a visualization. For example, it could be helpful to increase interactivity, implement more user guidance, display individual neurons or show backpropagation. It also could be useful to implement feature visualization [OMS17], training progress comparison, VR visualization, or interactive exploration of latent space [LNH+18, STN+16] and spatial activations [CAS+19].

NeuralVisUAL is a modular CNN visualization framework that helps researchers obtain a more exhaustive understanding of neural networks. Its settings, modularity, and open-source status make it easily adaptable to personal preferences and extendable to individual visualization requirements. NeuralVisUAL contains helpful features to visualize architecture, kernels, activations, saliency, and gradients and allows users to intuitively comprehend how a CNN processes the input information, furthering research into AI visualization and explainability.

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[AGQZ13] Michael Auli, Michel Galley, Chris Quirk, and Geoffrey Zweig. Joint language and translation modeling with recurrent neural networks. 2013.

[Ama93] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[AOJJ89] Stig K Andersen, Kristian G Olesen, Finn Verner Jensen, and Frank Jensen. Hugin-a shell for building bayesian belief universes for expert systems. In *IJCAI*, volume 89, pages 1080–1085, 1989.

[ASWDF16] Yannis M Assael, Brendan Shillingford, Shimon Whiteson, and Nando De Freitas. Lipnet: End-to-end sentence-level lipreading. *arXiv preprint arXiv:1611.01599*, 2016.

[BBS96] Jürgen Branke, Frank Bucher, and Hartmut Schmeck. Using genetic algorithms for drawing undirected graphs. In *The Third Nordic Workshop on Genetic Algorithms and their Applications*. Citeseer, 1996.

[Ben09] Yoshua Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009.

[BETT98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR, 1998.

[BH86] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[BH19] Andrew Burt and Patrick Hall. Why you should care about debugging machine learning models, Dec 2019. Accessed 20 Sep 2021. URL: `https://www.oreilly.com/radar/why-you-should-care-about-debugging-machine-learning-models/`.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: an open source software for exploring and manipulating networks. In *Third international AAAI conference on weblogs and social media*, 2009.

[BMR+20] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[Boo17] Boostorg. Boostorg/beast: Http and websocket built on boost.asio in c++11, 2017. Accessed 27 May 2021. URL: `https://github.com/boostorg/beast`.

[BOU12] Hassen BOUZGOU. *Advanced Methods for the Processing and Analysis of Multidimensional Signals: Application toWind Speed*. PhD thesis, Université de Batna 2, 2012.

[BP06] Ulrik Brandes and Christian Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *International Symposium on Graph Drawing*, pages 42–53. Springer, 2006.

[BS18] Marcel Bock and Andreas Schreiber. Visualization of neural networks in virtual reality using unreal engine. In *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*, pages 1–2, 2018.

[B.V15] Elasticsearch B.V. Kibana: Explore, visualize, discover data, 2015. Accessed 3 Sep 2021. URL: `https://www.elastic.co/kibana/`.

[CAS+19] Shan Carter, Zan Armstrong, Ludwig Schubert, Ian Johnson, and Chris Olah. Activation atlas. *Distill*, 4(3):e15, 2019.

[Chi20] Bhargav Chippada. Bhargavchippada/forceatlas2: Fastest gephi's forceatlas2 graph layout algorithm implemented for python and networkx, Dec 2020. Accessed 12 Jul 2021. URL: `https://github.com/bhargavchippada/forceatlas2`.

[CHW16] Luis Ceze, Mark D Hill, and Thomas F Wenisch. Arch2030: A vision of computer architecture research over the next 15 years. *arXiv preprint arXiv:1612.03182*, 2016.

[CLT+18] Chensi Cao, Feng Liu, Hai Tan, Deshou Song, Wenjie Shu, Weizhong Li, Yiming Zhou, Xiaochen Bo, and Zhi Xie. Deep learning and its applications in biomedicine. *Genomics, proteomics & bioinformatics*, 16(1):17–32, 2018.

[CM07] Nick Cawthon and Andrew Vande Moere. The effect of aesthetic on the usability of data visualization. In *2007 11th International Conference Information Visualization (IV'07)*, pages 637–648. IEEE, 2007.

[Cod19] Papers With Code. Image classification on imagenet, 2019. Accessed 22 Sep 2021. URL: https://paperswithcode.com/sota/image-classification-on-imagenet.

[CPC16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[CS20] Se-Hang Cheong and Yain-Whar Si. Force-directed algorithms for schematic drawings and placement: A survey. *Information Visualization*, 19(1):65–91, 2020.

[CWX+20] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. Addernet: Do we really need multiplications in deep learning? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1468–1477, 2020.

[DAR21] Beman Dawes, David Abrahams, and Rene Rivera, 2021. Accessed 26 May 2021. URL: https://www.boost.org/.

[Das20] Kaushik Das. How recurrent neural network (rnn) works, Dec 2020. Accessed 7 Sep 2021. URL: https://dataaspirant.com/how-recurrent-neural-network-rnn-works/.

[DB16] Alexey Dosovitskiy and Thomas Brox. Inverting visual representations with convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4829–4837, 2016.

[DBH92] Howard Demuth, Mark Beale, and Martin Hagan. Neural network toolbox. *For Use with MATLAB. The MathWorks Inc*, 2000, 1992.

[DBK+20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[DH96] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics (TOG)*, 15(4):301–331, 1996.

[Dob12] Dimiter Dobrev. A definition of artificial intelligence. *arXiv preprint arXiv:1210.1568*, 2012.

[DV16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[DW20] Minal Dhankar and Nipun Walia. *Emerging Trends in Big Data, IoT and Cyber Security*, page 105–108. Excellent Publishing House, 2020.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.

[ECS+21] Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, and Joey Yiwei Li. Deepshift: Towards multiplication-less neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2359–2368, 2021.

[Epp03] David Eppstein. Hyperbolic geometry, möbius transformations, and geometric optimization. In *MSRI Introductory Workshop on Discrete and Computational Geometry*, 2003.

[ESYF+20] Frank Emmert-Streib, Zhen Yang, Han Feng, Shailesh Tripathi, and Matthias Dehmer. An introductory review of deep learning for prediction models with big data. *Frontiers in Artificial Intelligence*, 3:4, 2020.

[Fal19] Vinnie Falco, 2019. Accessed 26 May 2021. URL: `https://www.boost.org/doc/libs/1_75_0/libs/beast/example/websocket/client/async/websocket_client_async.cpp`.

[FALL17] Max Ferguson, Ronay Ak, Yung-Tsun Tina Lee, and Kincho H Law. Automatic localization of casting defects with convolutional neural networks. In *2017 IEEE international conference on big data (big data)*, pages 1726–1735. IEEE, 2017.

[Fet90] James H Fetzer. What is artificial intelligence? In *Artificial Intelligence: Its Scope and Limits*, pages 3–27. Springer, 1990.

[FG18] Alban Flachot and Karl R Gegenfurtner. Processing of chromatic information in a deep convolutional neural network. *JOSA A*, 35(4):B334–B346, 2018.

[FHH00] Erik Frøkjær, Morten Hertzum, and Kasper Hornbæk. Measuring usability: are effectiveness, efficiency, and satisfaction really correlated? In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 345–352, 2000.

[FLM94] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In *International Symposium on Graph Drawing*, pages 388–403. Springer, 1994.

[FM11] Ian Fette and Alexey Melnikov. The websocket protocol, 2011.

[Fou17] The Apache Software Foundation. Apache zeppelin, 2017. Accessed 3 Sep 2021. URL: `https://zeppelin.apache.org/`.

[Fou18] Python Software Foundation. asyncio — asynchronous i/o, 2018.

Accessed 24 May 2021. URL: https://docs.python.org/3/library/asyncio.html.

[Fou21] Python Software Foundation. Coroutines and tasks, 2021. Accessed 24 May 2021. URL: https://docs.python.org/3/library/asyncio-task.html#running-in-threads.

[FR91] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[Fri17] Jerome H Friedman. *The elements of statistical learning: Data mining, inference, and prediction*. springer open, 2017.

[Fuk07] Kunihiko Fukushima. Neocognitron. *Scholarpedia*, 2(1):1717, 2007.

[Fur19] Sadayuki Furuhashi. Messagepack, 2019. Accessed 14 Apr 2021. URL: https://msgpack.org/.

[FV17] Ruth C Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *Proceedings of the IEEE international conference on computer vision*, pages 3429–3437, 2017.

[Gar17] Alan Garnham. *Artificial intelligence: An introduction*. Routledge, 2017.

[GEH19] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[GFV13] Helen Gibson, Joe Faith, and Paul Vickers. A survey of two-dimensional graph layout techniques for information visualisation. *Information visualization*, 12(3-4):324–357, 2013.

[GGK00] Pawel Gajer, Michael T Goodrich, and Stephen G Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In *Graph Drawing'00 Conference Proceedings*, pages 211–221, 2000.

[GGS03] Craig Gotsman, Xianfeng Gu, and Alla Sheffer. Fundamentals of spherical parameterization for 3d meshes. In *ACM SIGGRAPH 2003 Papers*, pages 358–363. 2003.

[GJ14] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International conference on machine learning*, pages 1764–1772. PMLR, 2014.

[GK04] Pawel Gajer and Stephen G Kobourov. Grip: Graph drawing with intelligent placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2004.

[GKN04] Emden R Gansner, Yehuda Koren, and Stephen North. Graph

drawing by stress majorization. In *International Symposium on Graph Drawing*, pages 239–250. Springer, 2004.

[GMH13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.

[Goe91] Philip W Goetz. *Encyclopedia Britannica*. Encyclopedia Britannica Incorporated, 1991.

[Gol17] Yoav Goldberg. Neural network methods for natural language processing. *Synthesis lectures on human language technologies*, 10(1):1–309, 2017.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[Gri12] Mark Grimshaw. Sound and player immersion in digital games. In *The Oxford handbook of sound studies*. 2012.

[GTCM20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[GWFM+13] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. In *International conference on machine learning*, pages 1319–1327. PMLR, 2013.

[H+96] Thomas S Huang et al. Computer vision: Evolution and promise. *CERN European Organization for Nuclear Research-Reports-CERN*, pages 21–26, 1996.

[Har15] Adam W Harley. An interactive node-link visualization of convolutional neural networks. In *International Symposium on Visual Computing*, pages 867–877. Springer, 2015.

[HBA+19] Liming Hu, David Bell, Sameer Antani, Zhiyun Xue, Kai Yu, Matthew P Horning, Noni Gachuhi, Benjamin Wilson, Mayoore S Jaiswal, Brian Befano, et al. An observational study of deep learning and automated evaluation of cervical images for cancer screening. *JNCI: Journal of the National Cancer Institute*, 111(9):923–932, 2019.

[HH01] Ronny Hadany and David Harel. A multi-scale algorithm for drawing graphs nicely. *Discrete Applied Mathematics*, 113(1):3–21, 2001.

[HHW18] Jie Hua, Mao Lin Huang, and Guohua Wang. Graph layout performance comparisons of force-directed algorithms. *International Journal of Performability Engineering*, 2018.

[HJ04] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *International Symposium on Graph Drawing*, pages 285–295. Springer, 2004.

[HK19] Michael Haenlein and Andreas Kaplan. A brief history of artificial intelligence: On the past, present, and future of artificial intelligence. *California management review*, 61(4):5–14, 2019.

[HL07] Kasper Hornbæk and Effie Lai-Chong Law. Meta-analysis of correlations among usability measures. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 617–626, 2007.

[HLVDMW17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[HN92] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[HND19] Anton Hristov, Maria Nisheva, and Dimo Dimov. Filters in convolutional neural networks as independent detectors of visual concepts. In *Proceedings of the 20th International Conference on Computer Systems and Technologies*, pages 110–117, 2019.

[HSL+16] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016.

[Hu05] Yifan Hu. Efficient, high-quality force-directed graph drawing. *Mathematica journal*, 10(1):37–71, 2005.

[HWC+20] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. A survey on visual transformer. *arXiv preprint arXiv:2012.12556*, 2020.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Inc20a] Epic Games Inc. Balancing blueprint and c++, 2020. Accessed 17 Jul 2021. URL: `https://docs.unrealengine.com/4.26/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/`.

[Inc20b] Epic Games Inc. Introduction to blueprints, 2020. Accessed 17 Jul 2021. URL: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/.

[Inc21] Epic Games Inc. Make something unreal, 2021. Accessed 17 Jul 2021. URL: https://www.unrealengine.com/en-US/unreal.

[Iva71] Alexey Grigorevich Ivakhnenko. Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, (4):364–378, 1971.

[Jad18] Shruti Jadon. Introduction to different activation functions for deep learning. *Medium, Augmenting Humanity*, 16, 2018.

[JEP+21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[Jup17] Project Jupyter, 2017. Accessed 3 Sep 2021. URL: https://jupyter.org/.

[JVHB14] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9(6):e98679, 2014.

[Ket17] Nikhil Ketkar. Introduction to pytorch. In *Deep learning with python*, pages 195–208. Springer, 2017.

[KGV83] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[Kha20] Renu Khandelwal. Convolutional neural network: Feature map and filter visualization, May 2020. Accessed 24 Aug 2021. URL: https://towardsdatascience.com/convolutional-neural-network-feature-map-and-filter-visualization-f75012a5a49c.

[KK+89] Tomihisa Kamada, Satoru Kawai, et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[KNBW11] Jon M Kleinberg, Mark Newman, Albert-László Barabási, and Duncan J Watts. *Authoritative sources in a hyperlinked environment*. Princeton University Press, 2011.

[Kob12] Stephen G Kobourov. Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*, 2012.

[Kok18] Apurva Kokate. *A study of interpretability mechanisms for deep networks*. PhD thesis, Iowa State University, 2018.

[Kor04] David Harel Yehuda Koren. A fast multi-scale method for drawing large graphs. *Graph Algorithms and Applications 3*, 6:179, 2004.

[Kos91] Corey Kosak. A parallel genetic algorithm for network-diagram layout. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 458–465, 1991.

[Kra91] Mark A Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991.

[KRSB18] Salman Khan, Hossein Rahmani, Syed Afaq Ali Shah, and Mohammed Bennamoun. A guide to convolutional neural networks for computer vision. *Synthesis Lectures on Computer Vision*, 8(1):1–207, 2018.

[Kru64] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[Kub99] Miroslav Kubat. Neural networks: a comprehensive foundation by simon haykin, macmillan, 1994, isbn 0-02-352781-7. *The Knowledge Engineering Review*, 13(4):409–412, 1999.

[Kum19] Niranjan Kumar. Visualizing convolution neural networks using pytorch, Dec 2019. Accessed 24 Aug 2021. URL: `https://towardsdatascience.com/visualizing-convolution-neural-networks-using-pytorch-3dfa8443e74e`.

[KW05] Stephen G Kobourov and Kevin Wampler. Non-eeuclidean spring embedders. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):757–767, 2005.

[Lab20] Grafana Labs. Grafana: The open observability platform, 2020. Accessed 3 Sep 2021. URL: `https://grafana.com/`.

[Lah16] Sampo Jukka Tapio Lahtinen. Utilization of game engine in simulation visualization. Master's thesis, 2016.

[LB+95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner.

Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[Li17] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[LNH+18] Quan Li, Kristanto Sean Njotoprawiro, Hammad Haleem, Qiaoan Chen, Chris Yi, and Xiaojuan Ma. Embeddingvis: A visual analytics approach to comparative network embedding inspection. In *2018 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 48–59. IEEE, 2018.

[LS18] Zachary C Lipton and Jacob Steinhardt. Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*, 2018.

[LSL+16] Mengchen Liu, Jiaxin Shi, Zhen Li, Chongxuan Li, Jun Zhu, and Shixia Liu. Towards better analysis of deep convolutional neural networks. *IEEE transactions on visualization and computer graphics*, 23(1):91–100, 2016.

[LWL+17] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[Mac16] Bohdan Macukow. Neural networks–state of art, brief history, basic models and architecture. In *IFIP international conference on computer information systems and industrial management*, pages 3–14. Springer, 2016.

[MAK17] Mustafa Alghali Elsaid Muhammed, Ahmed Abdalazeem Ahmed, and Tarig Ahmed Khalid. Benchmark analysis of popular imagenet classification deep cnn architectures. In *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, pages 902–907. IEEE, 2017.

[Mar15] John Markoff. A learning advance in artificial intelligence rivals human abilities. *The New York Times*, 10, 2015.

[MB95] Tamara Munzner and Paul Burchard. Visualizing the structure of the world wide web in 3d hyperbolic space. In *Proceedings of the first symposium on Virtual reality modeling language*, pages 33–38, 1995.

[MBKB11] Shawn Martin, W Michael Brown, Richard Klavans, and Kevin W Boyack. Openord: an open-source toolbox for large graph layout. In *Visualization and Data Analysis 2011*, volume 7868, page 786806. International Society for Optics and Photonics, 2011.

[MHN+13] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier

nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.

[Mit97] Tom Mitchell. Machine learning. 1997.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[Mog20] Tsuguo Mogami. Deep neural network training without multiplications. *arXiv preprint arXiv:2012.03458*, 2020.

[Mun97] Tamara Munzner. H3: Laying out large directed graphs in 3d hyperbolic space. In *Proceedings of VIZ'97: Visualization Conference, Information Visualization Symposium and Parallel Rendering Symposium*, pages 2–10. IEEE, 1997.

[Mun98] Tamara Munzner. Drawing large graphs with h3viewer and site manager. In *International Symposium on Graph Drawing*, pages 384–393. Springer, 1998.

[MV15] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5188–5196, 2015.

[NCB+17] Anh Nguyen, Jeff Clune, Yoshua Bengio, Alexey Dosovitskiy, and Jason Yosinski. Plug & play generative networks: Conditional iterative generation of images in latent space. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4467–4477, 2017.

[NDB+19] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[NHH15] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[Nil91] Nils J Nilsson. Logic and artificial intelligence. *Artificial intelligence*, 47(1-3):31–56, 1991.

[nip17] Nips test of time award presentation, Dec 2017. Accessed 17 Sep 2021. URL: https://www.youtube.com/watch?v=Qi1Yry33TQE.

[NL94] Jakob Nielsen and Jonathan Levy. Measuring usability: preference vs. performance. *Communications of the ACM*, 37(4):66–75, 1994.

[nnh16] In *IFIP international conference on computer information systems and industrial management*, pages 3–14. Springer, 2016.

[Noa07] Andreas Noack. Energy models for graph clustering. *J. Graph Algorithms Appl.*, 11(2):453–480, 2007.

[Num20] NumPy. Numpy, 2020. Accessed 15 Aug 2021. URL: https://numpy.org/.

[NYC19] Anh Nguyen, Jason Yosinski, and Jeff Clune. Understanding neural networks via feature visualization: A survey. In *Explainable AI: interpreting, explaining and visualizing deep learning*, pages 55–76. Springer, 2019.

[OMK20] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624, 2020.

[OMS17] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2(11):e7, 2017.

[Ost96] Diethelm Ironi Ostry. Some three-dimensional graph drawing algorithms. 1996.

[PAC00] Helen C Purchase, Jo-Anne Allder, and David Carrington. User preference of graph layout aesthetics: A uml study. In *International Symposium on Graph Drawing*, pages 5–18. Springer, 2000.

[PCJ97] Helen C. Purchase, Robert F. Cohen, and Murray I James. An experimental study of the basis for graph drawing algorithms. *Journal of Experimental Algorithmics (JEA)*, 2:4–es, 1997.

[Pla04] Catherine Plaisant. The challenge of information visualization evaluation. In *Proceedings of the working conference on Advanced visual interfaces*, pages 109–116, 2004.

[PLWZ19] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Gaugan: semantic image synthesis with spatially adaptive normalization. In *ACM SIGGRAPH 2019 Real-Time Live!*, pages 1–1. 2019.

[PMG98] David Poole, Alan Mackworth, and Randy Goebel. Computational intelligence. 1998.

[Pur97] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[Pur98] Helen C Purchase. Performance of layout algorithms: Comprehension, not computation. *Journal of Visual Languages & Computing*, 9(6):647–657, 1998.

[QE00] Aaron Quigley and Peter Eades. Fade: Graph drawing, clustering, and visual abstraction. In *International Symposium on Graph Drawing*, pages 197–210. Springer, 2000.

[QKL19] Zhongang Qi, Saeed Khorram, and Fuxin Li. Visualizing deep networks by optimizing with integrated gradients. In *CVPR Workshops*, volume 2, 2019.

[RDGC95] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[RORF16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[Ros57] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[Ros61] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[RPG+21] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[SHD14] Ruhi Sarikaya, Geoffrey E Hinton, and Anoop Deoras. Application of deep belief networks for natural language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(4):778–784, 2014.

[She18] Alex Sherstinsky. Deriving the recurrent neural network definition and rnn unrolling using signal processing. In *Critiquing and Correcting Trends in Machine Learning Workshop at Neural Information Processing Systems*, volume 31, 2018.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[SIVA17] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[SLL20] Pascal Sturmfels, Scott Lundberg, and Su-In Lee. Visualizing the impact of feature attribution baselines. *Distill*, 5(1):e22, 2020.

[Šmí17] Antonín Šmíd. Comparison of unity and unreal engine. *Czech Technical University in Prague*, pages 41–61, 2017.

[SMM+17] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[SMV+19] Wojciech Samek, Grégoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Müller. *Explainable AI: interpreting, explaining and visualizing deep learning*, volume 11700. Springer Nature, 2019.

[Sof03] Tableau Software. Business intelligence and analytics software, 2003. Accessed 3 Sep 2021. URL: `https://www.tableau.com/`.

[SRK95] Kai-Yeung Siu, Vwani Roychowdhury, and Thomas Kailath. *Discrete neural computation: A theoretical foundation*. Prentice-Hall, Inc., 1995.

[SS17] Sagar Sharma and Simone Sharma. Activation functions in neural networks. *Towards Data Science*, 6(12):310–316, 2017.

[SS18] Pramila P Shinde and Seema Shah. A review of machine learning and deep learning applications. In *2018 Fourth international conference on computing communication control and automation (ICCUBEA)*, pages 1–6. IEEE, 2018.

[SSJ+17] Manli Sun, Zhanjie Song, Xiaoheng Jiang, Jing Pan, and Yanwei Pang. Learning pooling for convolutional neural network. *Neurocomputing*, 224:96–104, 2017.

[SSKS17] Supasorn Suwajanakorn, Steven M Seitz, and Ira Kemelmacher-Shlizerman. Synthesizing obama: learning lip sync from audio. *ACM Transactions on Graphics (ToG)*, 36(4):1–13, 2017.

[SSSEA19] Thilo Spinner, Udo Schlegel, Hanna Schäfer, and Mennatallah El-Assady. explainer: A visual analytics framework for interactive and explainable machine learning. *IEEE transactions on visualization and computer graphics*, 26(1):1064–1074, 2019.

[SSWR18] David Sculley, Jasper Snoek, Alex Wiltschko, and Ali Rahimi. Winner's curse? on pace, progress, and empirical rigor. 2018.

[STK+17] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.

[STN+16] Daniel Smilkov, Nikhil Thorat, Charles Nicholson, Emily Reif, Fernanda B Viégas, and Martin Wattenberg. Embedding projector: Interactive visualization and interpretation of embeddings. *arXiv preprint arXiv:1611.05469*, 2016.

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*, pages 3319–3328. PMLR, 2017.

[SVZ13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[SWM17] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and in-

terpreting deep learning models. *arXiv preprint arXiv:1708.08296*, 2017.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[Tea20] Keras Team. Keras documentation: About keras, 2020. Accessed 16 Aug 2021. URL: `https://keras.io/about/`.

[Ten19] TensorSpace.js. Tensorspace.js, 2019. Accessed 14 Jul 2021. URL: `https://tensorspace.org/`.

[Tor14] Melanie Tory. User studies in visualization: A reflection on methods. In *Handbook of Human Centric Visualization*, pages 411–426. Springer, 2014.

[Tut63] William Thomas Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 3(1):743–767, 1963.

[VJKH20] Raju Vaishya, Mohd Javaid, Ibrahim Haleem Khan, and Abid Haleem. Artificial intelligence (ai) applications for covid-19 pandemic. *Diabetes & Metabolic Syndrome: Clinical Research & Reviews*, 14(4):337–339, 2020.

[VS15] Kritika Verma and Pradeep Kumar Singh. An insight to soft computing based defect prediction techniques in software. *International Journal of Modern Education and Computer Science*, 7(9):52, 2015.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[VZC19] Kevin C VanHorn, Meyer Zinn, and Murat Can Cobanoglu. Deep learning development environment in virtual reality. *arXiv preprint arXiv:1906.05925*, 2019.

[Wal00] Chris Walshaw. A multilevel algorithm for force-directed graph drawing. In *International Symposium on Graph Drawing*, pages 171–182. Springer, 2000.

[Wan19] Pei Wang. On defining artificial intelligence. *Journal of Artificial General Intelligence*, 10(2):1–37, 2019.

[WCD+18] Meredith Whittaker, Kate Crawford, Roel Dobbe, Genevieve Fried, Elizabeth Kaziunas, Varoon Mathur, Sarah Mysers West, Rashida Richardson, Jason Schultz, and Oscar Schwartz. *AI now report 2018*. AI Now Institute at New York University New York, 2018.

[Wei76] Joseph Weizenbaum. Computer power and human reason: From judgment to calculation. 1976.

[YCC95] Xiao-Hu Yu, Guo-An Chen, and Shi-Xin Cheng. Dynamic learning rate optimization of the backpropagation algorithm. *IEEE Transactions on Neural Networks*, 6(3):669–677, 1995.

[YCN+15] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[YEK02] Xinghuo Yu, M Onder Efe, and Okyay Kaynak. A general backpropagation algorithm for feedforward neural networks learning. *IEEE transactions on neural networks*, 13(1):251–254, 2002.

[YHPC18] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[Yin19] Li Yin. A summary of neural network layers, Jan 2019. Accessed 6 Sep 2021. URL: `https://medium.com/machine-learning-for-li/different-convolutional-layers-43dc146f4d0e`.

[YK15] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[YL17] Dong Yu and Jinyu Li. Recent progresses in deep learning based acoustic models. *IEEE/CAA Journal of automatica sinica*, 4(3):396–409, 2017.

[ZCS+18] Quanshi Zhang, Ruiming Cao, Feng Shi, Ying Nian Wu, and Song-Chun Zhu. Interpreting cnn knowledge via an explanatory graph. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[ZGCH21] Jianlong Zhou, Amir H Gandomi, Fang Chen, and Andreas Holzinger. Evaluating the quality of machine learning explanations: A survey on methods and metrics. *Electronics*, 10(5):593, 2021.

[ZKHB21] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. *arXiv preprint arXiv:2106.04560*, 2021.

[ZKL+14] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Object detectors emerge in deep scene cnns. *arXiv preprint arXiv:1412.6856*, 2014.

[ZYF+19] Xufan Zhang, Ziyue Yin, Yang Feng, Qingkai Shi, Jia Liu, and Zhenyu Chen. Neuralvis: visualizing and interpreting deep learning models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1106–1109. IEEE, 2019.

[ZYMW19] Quanshi Zhang, Yu Yang, Haotian Ma, and Ying Nian Wu. Interpreting cnns via decision trees. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6261–6270, 2019.

[ZZ18] Quanshi Zhang and Song-Chun Zhu. Visual interpretability for deep learning: a survey. *arXiv preprint arXiv:1802.00614*, 2018.

Appendix

Node movement created by each force during layout calculations

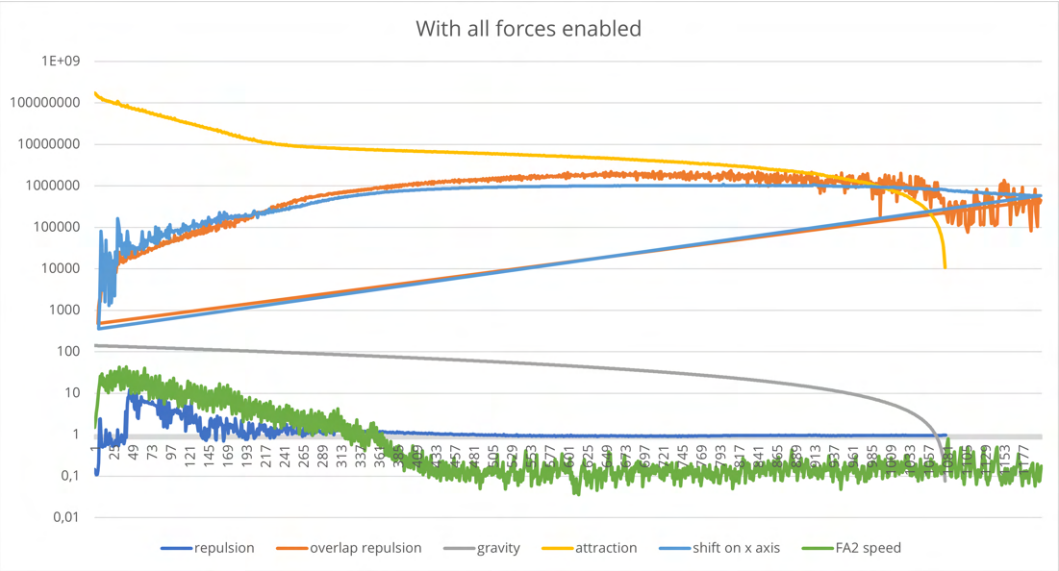

Figure 5.1: Node movement triggered by each force per iteration with all five forces enabled, when calculating the layout of Resnet-50 over 1200 iterations. With default settings, with repulsion constantly active in iterations 1 to 1080 and gravity in the same range with decreasing importance. FA2 speed (green) displays the dynamically adapting speed factor from FA2 to reduce jittering [JVHB14].

Ablation experiments for the forces in NeuralVisUAL's FBA

Figures 5.2, 5.3, and 5.4 show the first part of same Resnet-50 architecture in iterations 480, 720, 960, and 1200 during application of NeuralVisUAL's layout algorithm with different combinations of forces.

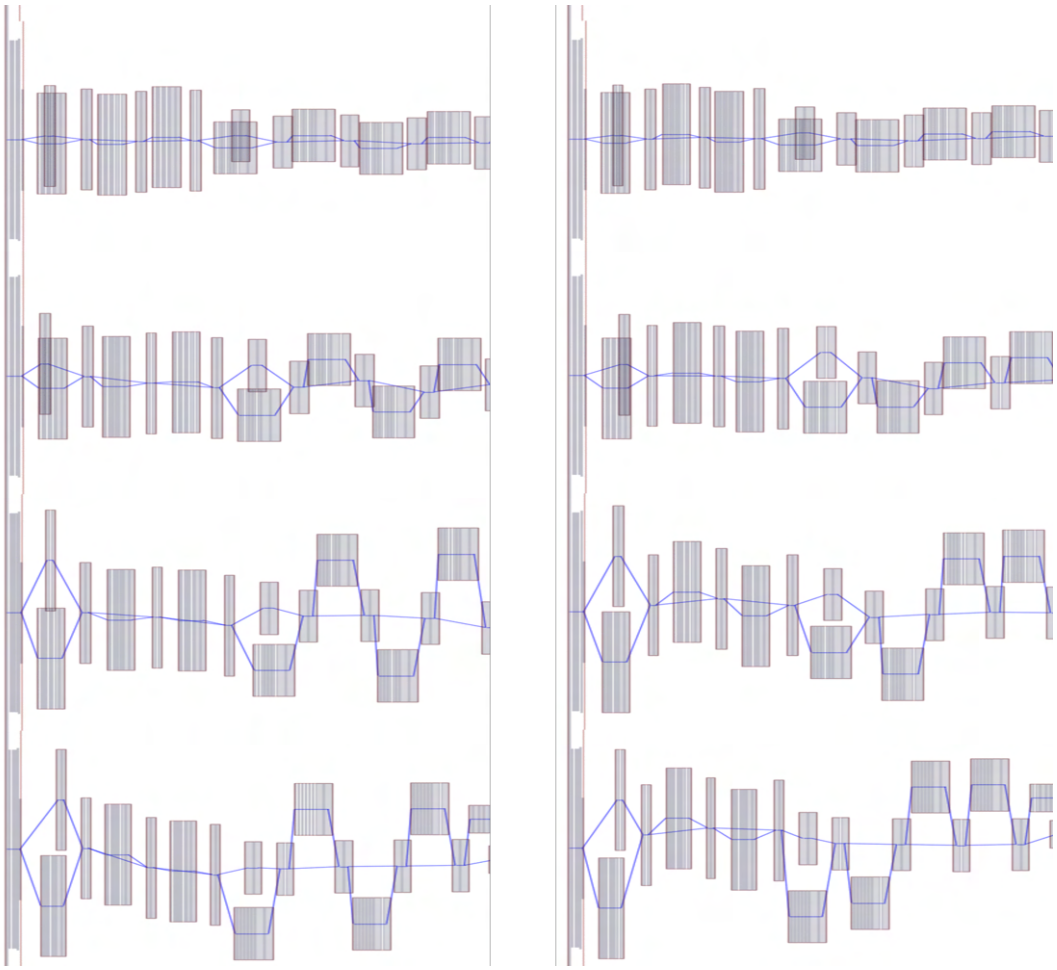

Figure 5.2: Left: all five forces enabled. Right: all forces except repulsion enabled. No difference in layout progress is noticable, only the exact positions, which depend on random processes.

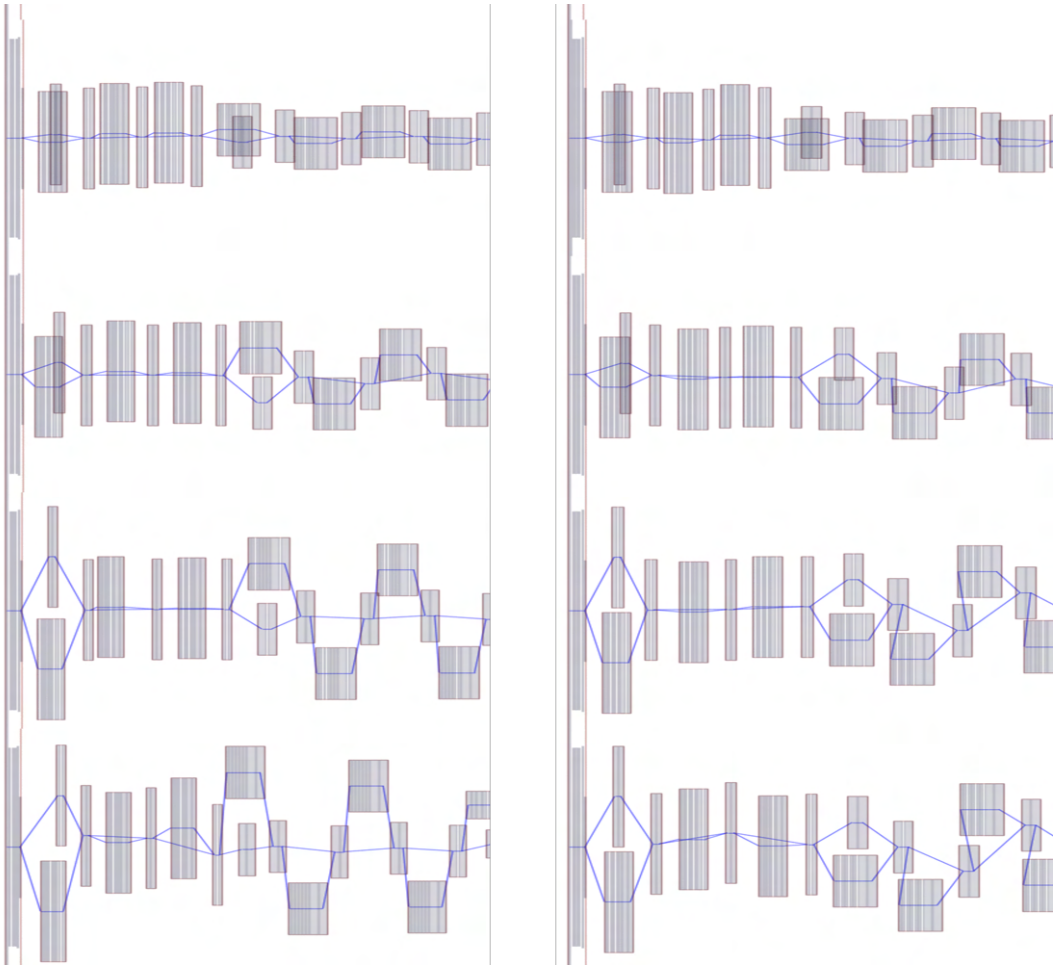

Figure 5.3: Left: all forces except gravity enabled, again no difference in layout quality is noticable compared to all forces enabled.
Right: all forces except order along x axis enabled, the information flow criterion is not satisfied anymore, even after all 1200 iterations.

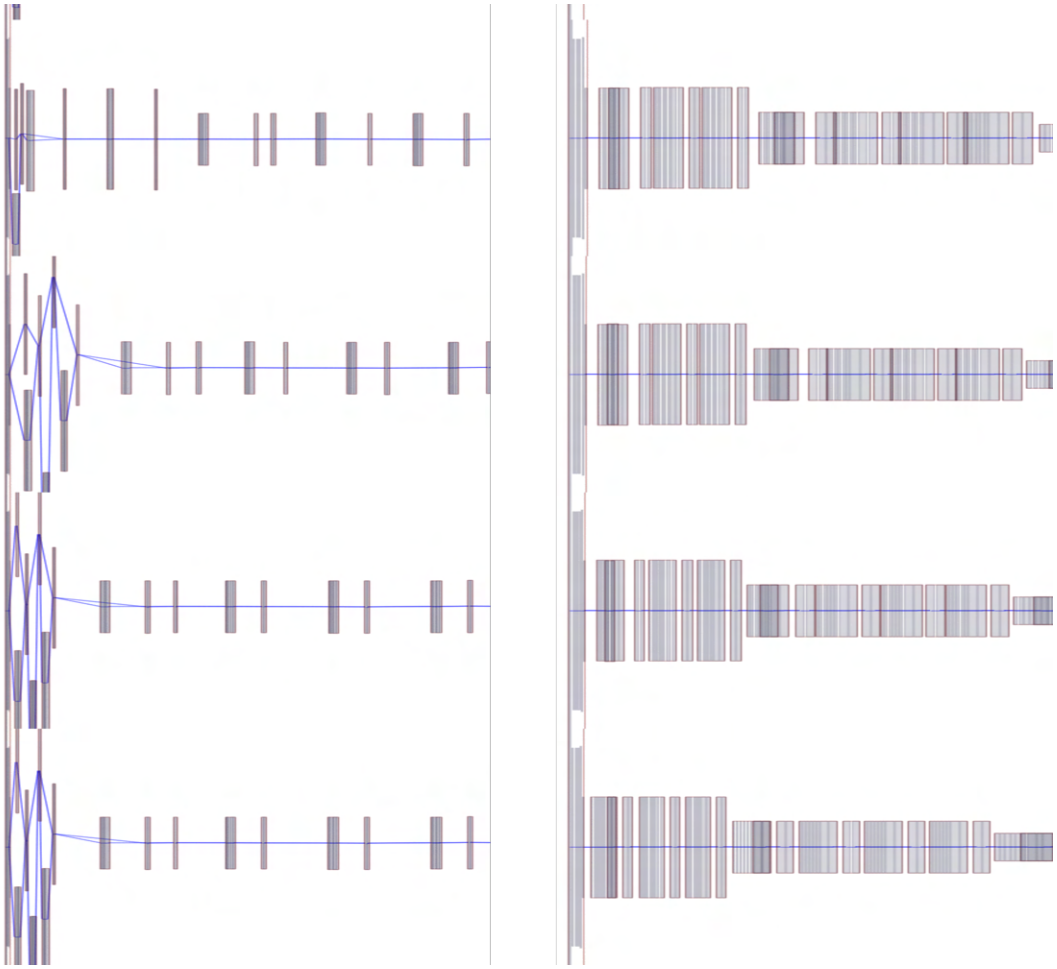

Figure 5.4: Left: all forces except for attraction enabled, some repulsion takes place in the first few layers, but other than that the network barely moves Right: all forces except overlap repulsion enabled, the network shrinks towards a local minimum that satisfies the attraction forces, layer groups completely overlap.