RENN
Retraceable and Explainable Neural Networks

Luca Geiger, Lukas Schmalzer

June 29, 2024

Abstract

The semester project focused on exploring "Explainable AI," particularly within the context of neural networks and the traceability of their results back to the training data. Our research aimed to develop methods to interpret and understand the outputs of complex models such as Large Language Models (LLMs) and image generation AIs.

We propose a pytorch implementation, which is able to store all relevant values of the neurons during the training process. These values help us to create a lookup table. When we now feed the network new and unseen data, this database helps us to find the most similar or closest training sources for each neuron, layer and the whole network.

This information allows us to pinpoint exactly which sources are most responsible for the network's prediction. We are also able to see how much influence different samples had. With this information, we can evaluate the prediction and certainty of the neural networks without the help of an algorithm and even before looking at the output.

Our findings also revealed the potential to pinpoint specific neurons responsible for certain predictions, offering a clear depiction of the network's internal processing. This capability extends to identifying harmful or low-quality training samples, improving model training and performance.

Future applications of our approach include addressing biases in generative models and producing augmented samples to enhance network learning. Our research contributes to the ongoing development of explainable AI, providing tools to trace and understand complex model predictions.

Contents

Chapter 1

Aims and Context

1.0.1 Intentions

The semester project was initially conceived as a research topic where we aimed to delve deeper into “Explainable AI.” The focus was particularly on neural networks and tracing the results back to the training data used. We believe that a neural network is not capable of learning how a certain task or algorithm works, but always references its input data, to get the best approximation for its prediction. We always imagined this example:

A Large Language Model produces the output: “The USA has a population of about 350 million people.”

According to our hypothesis, it should be possible to trace this information back to the documents used to train the model. So we should be capable of giving accurate feedback if this information was found in a wikipedia article, a news article or a reddit post. At the beginning, we spent a significant amount of time getting accustomed to neural networks, as they form the foundation for many modern models like LLMs and image generation AIs. We particularly focused on the mathematical aspect, assuming there must be a way to trace this data flow.

1.0.2 Research

After we got accustomed with the workings and the structure of Neural Networks, we immersed ourselves in numerous articles and papers to understand the state of the art and the existing possibilities. The paper “Why should I trust you?” [2] laid some fundamental groundwork for explainable AI, with the implementation of LIME. This software package can help you to evaluate the networks performance and if the output is actually a true prediction or if there are errors in the training samples. This paper not only showed us what software implementations are currently used to evaluate Neural Networks but actually strengthened our beliefs, that it just does not show the whole picture. Very quickly we also found multiple different papers, which focused on the traceability of features [1] or learning of important features [3] in a neural network.

Even though these papers have a very relevant point, in finding the most important features of an input sample, as it can have a huge impact on the models accuracy, it just does not really trace back the output to its training samples. Until now we have not found a single paper that has tried and successfully implemented a method to retrace the output prediction of a network to its inputs or could give a clear understanding how important a certain sample was for the specific prediction.

After this long time of researching, without any big finding, we started to discuss and implement our own ideas of how we could tackle this problem with the current software solutions available to us.

1.0.3 Initial Ideas

Our ideas involved splitting the output of a neural network into two separate parts. The first part would be the expected normal output, while the second part would be a representation of the input dataset, essentially an identification.

The first idea was to assign the dataset in a vector space as a 2D, 3D, or 4D vector. The network would be trained not only to produce the correct output but also to create a vector representation indicating where in the multidimensional space the new output is located. This would allow us, using an approach like K-nearest Neighbours, to identify which training datasets were closest to this sample.

The second idea was similar. Instead of a vector representation however, we tried to produce a large array as the second output, where each input data sample was marked with one-hot encoding during training. For example, for the fifth training sample, the fifth entry in the array would be marked with a 1, while all other numbers in the array would remain 0. This would allow us to determine which training data contributed to evaluating the current sample. If the 42nd element had a value of 0.6, we could assert that it was particularly important for evaluating this test sample.

Both ideas followed the right approach but ultimately had the issue of significantly reduced accuracy due to the large volume of outputs. We observed this phenomenon by training the neural network without the additional outputs and comparing the results. Additionally, we programmed our own neural network from scratch only using Numpy to prevent obscuring anything by other libraries, which we then used to evaluate our networks. Through testing these two networks, we noticed the deviations in accuracy and started looking for other options.

Chapter 2

Project Details

2.0.1 Training Data

To test and visualize the models in different scenarios, we considered two datasets. The first dataset is the well-known MNIST dataset, widely used in the machine learning community. This dataset consists of thousands of small images, each composed of 28x28 black-and-white pixels, depicting handwritten digits from 0 to 9.

For the second problem statement, we designed our own dataset. It involves a three-dimensional representation of the conversion from the HSV to the RGB color model. We transformed the values of individual colors into a coordinate system, thus creating a spatial representation of the colors. This approach particularly relates to our first idea, where the vector representation of the second output in space showed the colors most frequently used in the calculations.

However, this logic presented some additional challenges. The vector representation only showed the sum of the used data rather than the actual nearest color from the training dataset. We incorporated these insights from our tests into the final implementation.

2.0.2 Final Implementation

The current and final implementation for this semesterproject focuses heavily on the MNIST dataset. We define the number of layers, the number of neurons, and the activation functions ourselves before training. Typically, we train the simple model for about 10 epochs to minimize the loss function as much as possible.

After these initial 10 epochs, we use an 11th epoch, which we do not use for training. In this iteration, we examine all the values used in the network for each specific sample of the training set using Pytorch´s hook function. For instance, when we input the first training sample, we go through each individual neuron in the entire network and save the output values of these specific neurons. We then store all of these values in a separate data structure, which we later use as a lookup table. After completing the 11th pass through all our training samples, we are able to evaluate the test or validation samples.

Chapter 3

System Documentation

Our current implementation is designed to provide a robust and user-friendly platform for researchers and developers. We opted to create a Google Colab “Playground” to ensure that no additional setup is required to test our approach. The implementation includes three datasets: the widely used MNIST dataset, our custom HSV to RGB conversion dataset, and a “Small 1x1” dataset to illustrate the potential application of our methods to Large Language Models (LLMs). To maintain clarity and prevent the notebook from becoming cluttered, the core code has been offloaded into separate Python scripts.

3.0.1 Library Intitialization

The first step involves downloading and importing all necessary libraries into the notebook. The Python scripts are automatically imported from GitHub to ensure that users always access the most up-to-date version without needing additional integration. We realized that downloading libraries separately for each file created unnecessary bottlenecks, so we optimized the process by parametrizing the library imports and setting them in a separate initialization step for each script. This approach reduces setup time significantly, bringing it down to under 10 seconds from several minutes.

3.0.2 “Playground” Intitialization

After the libraries are initialized, users are introduced to the interactive “Playground,” built with Jupyter widgets. This playground provides an intuitive interface with four main tabs: Dataset, Network, Training, and Visualization. Each tab allows for extensive customization to facilitate the creation and testing of neural networks.

Network Tab:

- Users can set a seed to ensure consistent results, applying it internally to both Numpy and Torch.
- Adjust the number of network layers using a slider, which dynamically updates nested tabs for each layer.
- Specify the number of neurons and activation functions for each layer. Up to 16 layers can be added, considering visualization constraints.

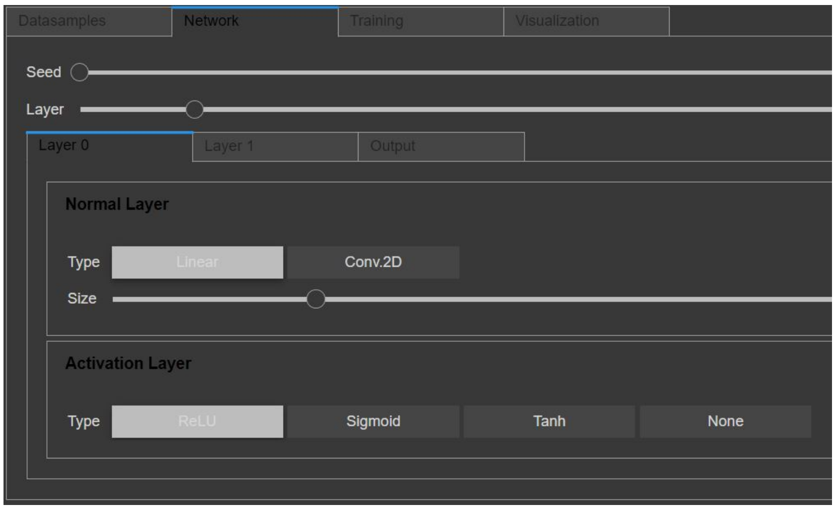


Figure 3.1: The interactive playground. This gui gives the user a simple way to create their own models and change the parameters without touching any code.

- The input layer is fixed to prevent issues like tensor dimension mismatches, while the output layer allows adjustments to the activation function.

Training Tab:

- Users can configure training parameters, including the number of epochs, learning rate, batch sizes for training and testing, preferred optimizer, and loss function.
- These values are adjustable via sliders and buttons, with direct input allowed for precise control.
- Batch sizes are limited by the number of samples set, ensuring no exceedance.
- Jupyter widgets' "jsLinks" functionality creates dependencies between widgets, ensuring smooth and intuitive interaction.

Visualization Tab:

- Users can select the number of evaluations (up to 100) and the number of closest sources to display (up to 20).
- Choose the visualization type: weighted view, layer and neuron view, only layers, or full customization.
- Customization options allow selecting specific neuron ranges for each layer individually.
- Activation layers can be visualized for each layer, offering maximum flexibility.
- Customizable visualization ensures detailed analysis without overwhelming the interface, making it easier to understand the network's behavior.

3.0.3 Functionality

The remainder of the notebook is structured to ensure ease of use and comprehensive functionality. Here's a detailed breakdown of the steps involved:

Hook Initialization:

- Dictionaries are created to store activation values for each neuron and layer.
- A hook is initialized to capture these values during the training process.
- The Customizable RENN ensures consistent data handling and storage across different datasets.
- This process allows for detailed tracking and analysis of neuron activations, making it easier to understand the inner workings of the network.

Visualization:

- The most used sources are identified by comparing neuron activations with current evaluation samples.
- The frequency of each source is determined by counting occurrences within the closest sources.
- Blended visualizations are displayed based on user-selected options, providing insights into the network's behavior.
- Users can see how different samples influence network predictions, aiding in understanding the decision-making process and the certainty of the predictions.

HSV-RGB Special Section:

- A dedicated section for the HSV-RGB dataset allows users to pick a layer and neuron to visualize the closest sources in a 3D cube.
- This section uses Plotly for interactive 3D visualizations, providing a clear and intuitive representation of color space transformations and neuron activations.
- This feature requires the HSV-RGB conversion selection and prior visualization to ensure vectors are created and stored.
- The 3D visualization offers unique insights into how the network processes and interprets color data.

Chapter 4

Evidence and Results

When we introduce a new sample into the network—one that it has not seen during training—we can identify the most similar samples from the training set. Using our created database, we can find the closest matches for each neuron, each layer, and the entire network.

We analyze the values of each neuron, counting and evaluating the closest matches in the database based on their frequency. The samples that occur most frequently have the highest similarity to the input. We can specify how many of the closest matches we want to see, rather than just the single most similar sample. While each sample could theoretically be checked for similarity due to the network's size and the possibility of random values, we found that only the top 50 most similar samples provide truly meaningful insights.

This information allows us to understand what the network references at each step of its prediction. For example, with the MNIST dataset, we can overlay the most similar samples, weighting the images based on their frequency. The resulting image accurately represents what the network perceives when making a prediction. We observed instances where the network misclassified an image, and through the network's reference image, we could trace the reasoning behind the prediction.

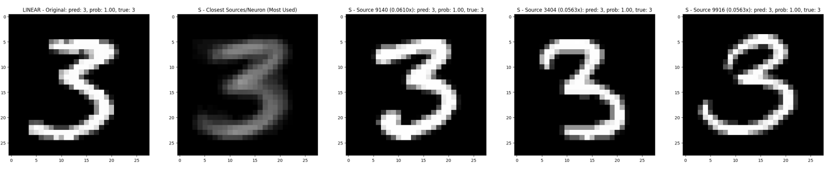


Figure 4.1: Showcase of the final output. From left to right: Input sample, Most frequent training samples blended by frequency, The three most important training samples

4.0.1 Bonus findings

In addition to our previous findings, we discovered a method to precisely identify which neurons or regions in a neural network are responsible for specific outputs. In the context of the MNIST dataset, we can accurately determine which neurons are most relevant for predicting certain numbers. This is possible because we measure the activation for each individual training sample.

After storing all these values, we aggregate the activations for a specific number—let's say the number 5—and divide by the number of samples used. This provides us with a representation of our neural network with the different activations. The evaluation depends on the activation function used in your layers, as results can vary if the activations are negative, range from -1 to +1, or are entirely above 0.

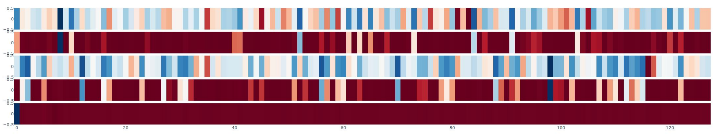
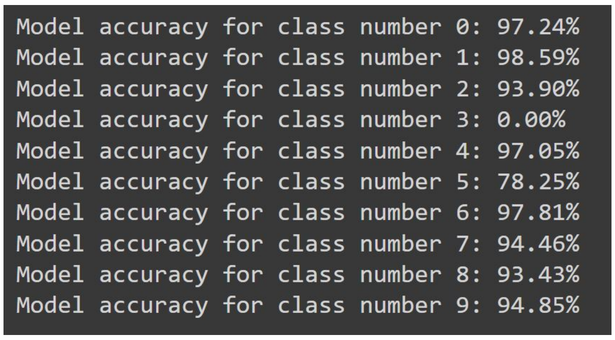


Figure 4.2: Each row represents one layer in a neural network, while each column represents a neuron, which are individually color coded. The red color depicts very low values or the number 0. The more saturated the blue color, the higher the value or the activation of the neuron in this layer is.

Using this information, we can create a matrix showing which neurons are most or least activated when certain numbers are recognized. This allows us to demonstrate that specific neurons can be deactivated without significantly impacting the overall network performance. It reveals where in the neural network data is processed and the responsibilities within the network.



```
Model accuracy for class number 0: 97.24%
Model accuracy for class number 1: 98.59%
Model accuracy for class number 2: 93.90%
Model accuracy for class number 3: 0.00%
Model accuracy for class number 4: 97.05%
Model accuracy for class number 5: 78.25%
Model accuracy for class number 6: 97.81%
Model accuracy for class number 7: 94.46%
Model accuracy for class number 8: 93.43%
Model accuracy for class number 9: 94.85%
```

Figure 4.3: This image showcases the accuracy of all the different classes individually. All necessary neurons for classifying the number 3 have been deactivated, therefore completly removing the networks possibility to classify the number. All other values still achieve a very good accuracy.

This approach is applicable to various neural network use cases. It is easier to implement in classification problems since the classes into which the training data can be divided are already provided. In other applications, the input data would need to be divided by the developer to identify the relevant neurons and regions.

Chapter 5

Summary

Our semester project focused on “Explainable AI,” specifically investigating how neural networks’ outputs can be traced back to their training data. We proposed methods using vector space representations and one-hot encoding to track training samples’ influence on neural network outputs. These methods, tested on the MNIST dataset and a custom HSV to RGB conversion dataset, provided valuable insights despite some challenges with accuracy.

We ended up developing a PyTorch implementation to store all relevant neuron values during training, creating a lookup table. This table helps to identify the most similar training samples for any new input, allowing us to understand the network’s decision-making process. By analyzing neuron activations, we can pinpoint which training sources most influence the network’s predictions and measure each sample’s impact. This method also enables us to evaluate the network’s predictions and certainty without directly looking at the output.

5.0.1 Future possibilities

This method holds great promise for the future, as it can be used to identify which samples in a dataset are harmful or particularly effective for recognizing unknown data. In Generative Models, for example, this could help combat harmful issues like racism and sexism more effectively. Additionally, it could help identify samples that are unusable for training due to their poor quality.

Furthermore, this approach can assist in producing augmented samples that do not exist in the original dataset. Based on the quality of the network’s references, it can help the network learn different behaviors and arrangements of samples that are not present in the training set. This capability enhances the network’s ability to generalize and improve its performance on unseen data.

References

[1] Vincent Aravantinos and Frederik Diehl. *Traceability of Deep Neural Networks*. 2019. arXiv: 1812.06744 [cs.LG]. URL: https://arxiv.org/abs/1812.06744 (cit. on p. 3).

[2] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. 2016. arXiv: 1602.04938 [cs.LG]. URL: https://arxiv.org/abs/1602.04938 (cit. on p. 3).

[3] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. *Learning Important Features Through Propagating Activation Differences*. 2019. arXiv: 1704.02685 [cs.CV]. URL: https://arxiv.org/abs/1704.02685 (cit. on p. 3).