Visualization of Training of Neural Network

Jakub Pícha, Jakub Profota

May 25, 2025

https://nn-visualization-4310e2.pages.fel.cvut.cz
https://gitlab.fel.cvut.cz/profojak/NN-visualization

1 Introduction

In this report, we present an implementation of visualizing the training process of a simple neural network on the MNIST dataset, a well-known benchmark for image classification. We explore techniques such as loss landscape mapping, activation visualization, and embedding projections and discuss their ability to reveal patterns in learning, generalization, and potential model biases. Finally, we choose and implement a concrete approach and present the results.

2 Task

Neural networks (NNs), although a relatively recent development, have rapidly become a cornerstone of modern artificial intelligence. Over the past years, they have undergone a dramatic expansion in both capability and accessibility, driven by advances in hardware, open-source libraries, and simplified model architectures. These developments enabled a surge in their adoption across a wide range of applications.

However, despite their success, a fundamental challenge remains. Neural networks are inherently complex and difficult to interpret. Their decision-making processes are largely opaque, making it hard for developers to debug or improve models efficiently. For end users, this lack of transparency can result in a lack of trust, especially in high-stakes scenarios where decisions made by AI systems may have significant personal or societal consequences[1]. This issue is particularly pressing in regions like the European Union, where AI regulations mandate that companies must be able to explain how algorithmic decisions are made.

Visualization proves to be a powerful tool for understanding and interpreting the behaviour of neural networks. By translating abstract, high-dimensional data into intuitive visual representations, visualization techniques can shed light on various aspects of training dynamics, model architecture, and decision boundaries.

Our objective is to implement a visualization approach that reveals the internal workings of a neural network during training. Specifically, the aim is to understand how the network improves over time by visualizing concrete aspects of its learning progress. By using the MNIST dataset as a controlled, well-understood environment, we can focus on producing clear and informative visual outputs that capture the evolution of the model's performance. These visualizations are intended to offer insight into how the model transitions from random guessing to competent classification.

3 Related Work

Node-link diagrams [2] represent the most common visualization approach for displaying network architectures. These diagrams show neurons as nodes and edge weights as links between them, helping users understand the structure and data flow in neural networks. However, complex models with many connections cn create visual clutter through numerous edge crossings. Researchers have

[1] https://ai-2027.com

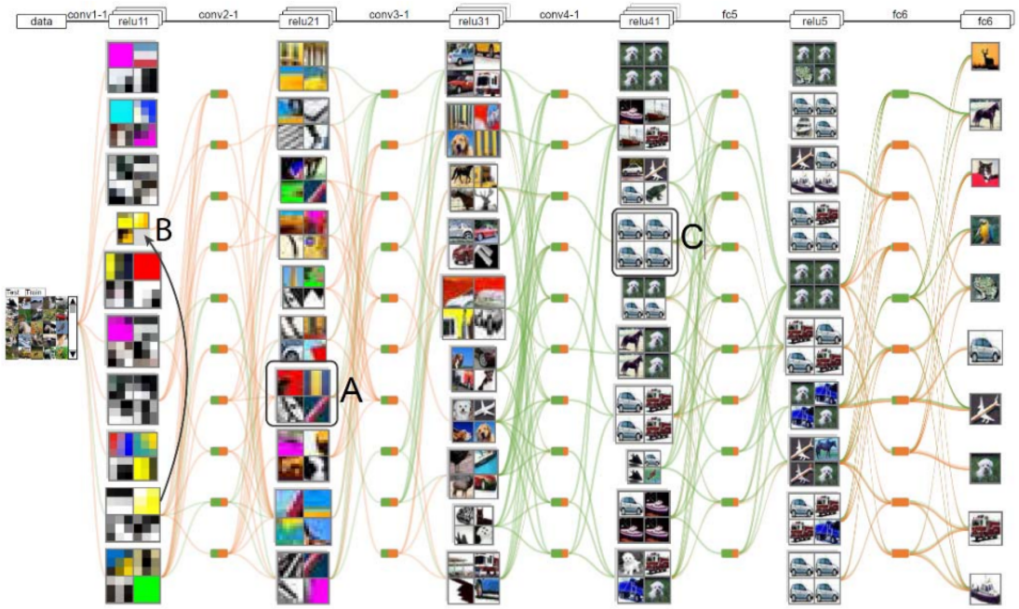

Figure 1: Example of node-link, taken from [1]

addressed this challenge through techniques like extracting high-degree nodes, implementing edge bundling, and embedding additional information within nodes such as showing typical images that activate specific neurons. While effective for smaller networks, these visualizations can become unwieldy with larger models.

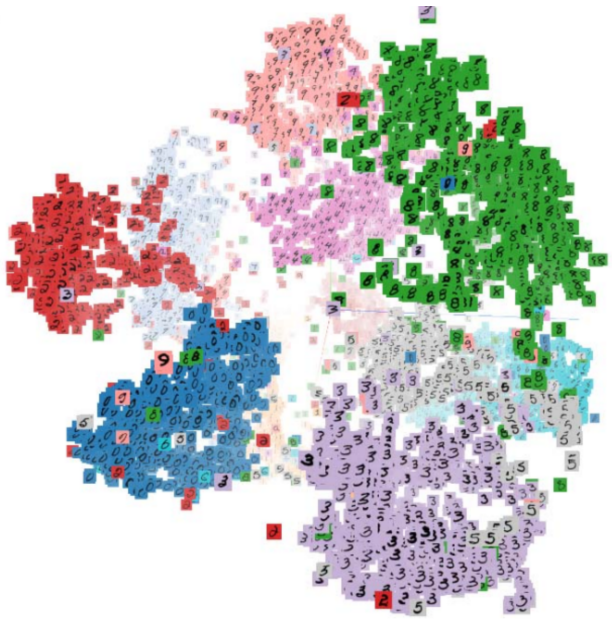

Figure 2: Example of dimensionality reduction, taken from [1]

Dimensionality reduction techniques [8], particularly scatter plots, are widely used to visualize high-dimensional neuron activations in deep networks. This approach involves reducing the dimensionality of activation vectors to two or three dimensions using methods like principal component analysis (PCA) or t-distributed stochastic neighbor embedding (t-SNE). Each point in the resulting scatter plot represents a data instance, with proximity indicating similarity in activation patterns. For 3D visualizations, interactive tools allow users to pan, rotate, and navigate the space.

Some systems place the original images at their corresponding coordinates in the reduced space, despite potential overlap issues. The effectiveness of these visualizations depends on the quality of the dimensionality reduction algorithm used, with different techniques capturing different aspects of the data relationships.

Figure 3: Example of line chart, taken from [4]

Line charts [2] are the predominant method for tracking temporal metrics during model training. These charts visualize crucial performance indicators like loss, accuracy, and error measurements over time, helping users identify convergence patterns, potential over-fitting, and other training dynamics. Systems like TensorBoard[2] prominently feature these visualizations, enabling users to compare multiple metrics, overlay results from different models, filter specific models, and interact with the data through tooltips and resizing options. These charts are essential for model training, comparison, and selection decisions.

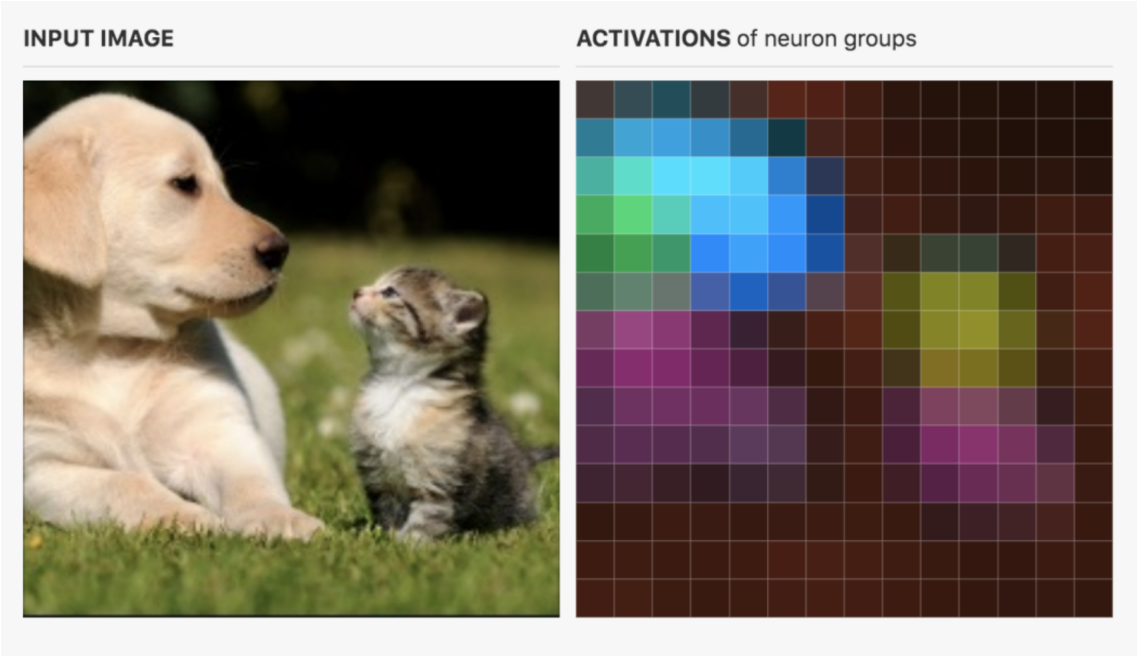

Figure 4: Example of a heatmap visualization, taken from [2]

Feature visualization [1] takes a different approach by generating entirely new synthetic images that maximally activate specific neurons or classes. These methods create heatmaps or similar

[2]https://www.tensorflow.org/tensorboard

visualizations that highlight important regions of the input data, particularly images, that contribute to a specific classification. For example, when a model classifies an image, these techniques can generate a new overlay image showing which pixels most strongly influenced the classification decision. They do not directly visualize existing model components, like weights or activations.

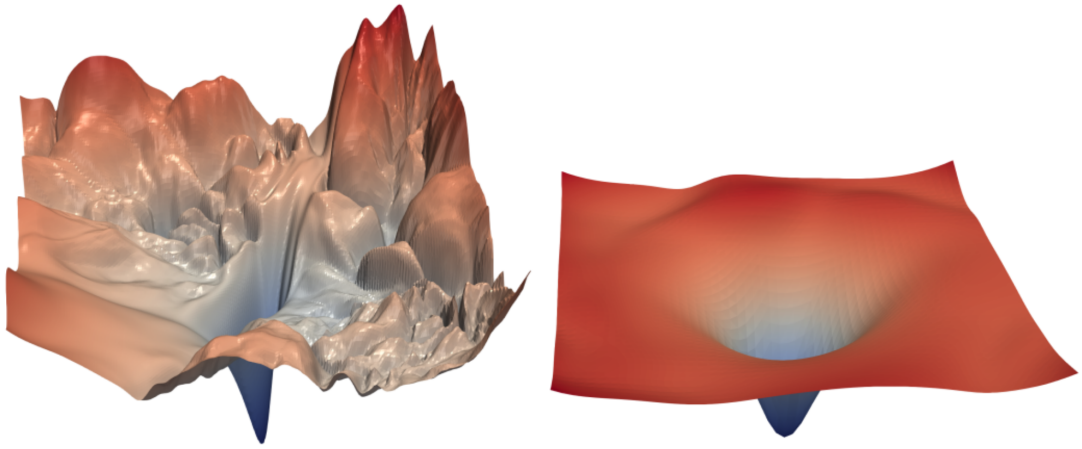

Figure 5: Loss landscape without and with skip connections, taken from [5]

Another technique is to focus on the loss landscape [5], and how the shape of loss function minimizers impacts network's performance. The 2D contour plot method reveals the geometry of the landscape via contours, showing whether they're sharp, flat, wide, or surrounded by complex terrain. The trajectory visualization technique addresses the challenge of showing how optimization algorithms traverse the loss landscape. The paper also introduces a method for visualizing negative curvature in loss landscapes. Particularly impressive is the visualization of the landscape in three dimensions.

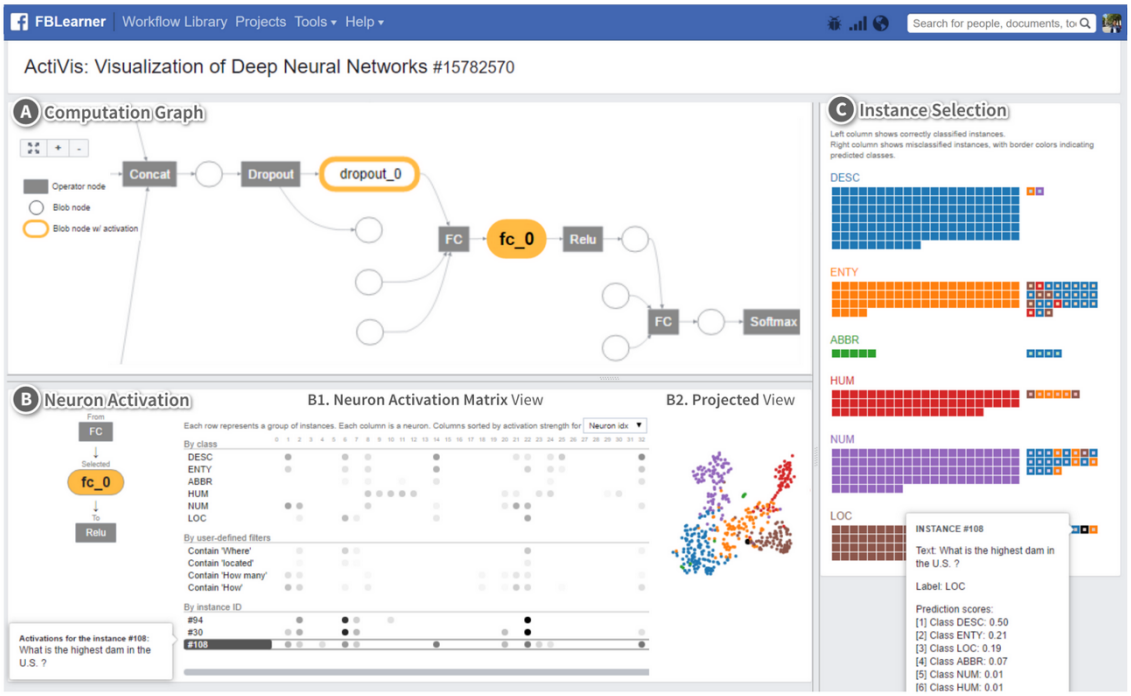

Figure 6: Picture of acti-vis, taken from [3]

ActiVis [3] integrates four key visualization techniques to help explore and interpret industry-

scale deep neural networks. The computation graph visualization displays the model architecture as a directed graph, showing operators and tensors with data flowing left to right, giving users an interactive overview of complex model structures. Neuron activations are seen in a matrix where rows are instances/subsets, columns are neurons, and cell darkness indicates activation strength, revealing relationships between neurons and data classes. The instance selection panel displays classification results as squares organized by true labels, with correct/incorrect predictions separated and color-coded, enabling users to select specific examples for detailed analysis.

4 Implementation

In our implementation, we create a visualization tool that allows users to explore the activation patterns of a neural network trained on the MNIST handwritten digit dataset using the embedding projections mapping techniques. We aim to hack our own novel solutions, try to implement some of the generally used methods, and compare our results with references from open-source libraries. This approach will help us understand how the network organizes different digit classes in its internal representation space, providing insights into the model's decision-making process, which we consider to be the most valuable visualization to focus on given the time constraints.

We begin by training a straightforward neural network using a tutorial[3] on the MNIST dataset. Our model architecture will include several convolutional layers followed by dense layers, culminating in a final layer with 10 neurons, one for each digit class. This architecture represents a standard approach for digit classification tasks while remaining simple enough for clear visualization and interpretation.

Once our model is trained, we extract the activations from the final layer for all 10,000 images in the MNIST test set. These activations represent high-dimensional embeddings that capture probabilities of the image to be considered as one of the digits just before the final classification. By examining these activation patterns, we can gain insights into which digits the model finds similar or confusing, and we expect to see obvious grouping of test data.

The core of our visualization is an embedding projection that reduces these high-dimensional activation vectors to a 2D representation that can be displayed on a screen. We utilize t-SNE, UMAP, PAC, and star coordinates dimensionality reduction techniques for comparison, as these approaches are particularly effective at preserving local relationships between data points and are widely used. The resulting 2D projection places similar activation patterns close together, revealing the model's internal organization of the digit space. We add interactive features that allow users to visualize different training epochs, hover over points to see the corresponding digit image, and choose a point to trace its learning process throughout learning epochs.

4.1 Technology Stack

On the backend, we utilize Python as our primary language, with TensorFlow and its high-level Keras API for neural network implementation. For dimensionality reduction, we are on a search for established open-source libraries that implement UMAP, t-SNE, PCA, and star coordinates algorithms. These techniques represent the current state-of-the-art for visualizing high-dimensional data in lower dimensions while preserving important structural relationships.

For the frontend, we implement an interactive web interface using JavaScript with the D3.js[4], a powerful data visualization library that gives us precise control over how we represent our embedding projections and interactive user control in real-time. To connect our Python backend with the JavaScript frontend, we serialize neural network weights and activation data using both JSON for human readable output, and MessagePack[5] for efficient binary representation of the same data. We host the final visualization website on GitLab Pages. [7]

[3] https://keras.io/examples/vision/mnist_convnet/
[4] https://d3js.org
[5] https://msgpack.org

4.2 Online and Offline

The visualization of the training of the neural network can be approached through either online processing in real-time or offline. In our implementation, we deliberately choose the offline approach, where we first train the model while saving the necessary activation data throughout training epochs, and then load this information into our visualization system afterward. This decision brings several practical advantages for our specific use case.

Offline processing eliminates the need to establish complex communication pipelines between the training environment and the JavaScript frontend. While online visualization would require setting up WebSockets or similar technology to stream data continuously, our offline approach allows for a clean separation between the computationally intensive training phase and the interactive visualization phase. This architectural simplicity reduces development complexity and potential points of failure in the system.

Our approach also decouples the visualization experience from the training process. Neural network training, particularly for complex models, can be slow and computationally intensive. With offline processing, users do not need to wait through lengthy training periods before interacting with visualizations. Instead, they can immediately begin exploring pre-computed activation patterns as soon as they load the visualization interface. Additionally, we can more easily compare activation patterns across different epochs of training by pre-saving multiple checkpoints, enabling users to observe how the network's internal representations evolve throughout the learning process.

Finally, offline processing creates better resource efficiency through reusability. Once we have extracted and saved the activation data, it can be reused across multiple visualization sessions without repeating the computationally expensive training and inference steps. This approach conserves computing resources and provides a more responsive user experience, particularly beneficial when working with limited computational capacity or when sharing the visualization tool with others who may not have access to the original training environment.

In our implementation, we specifically slowed down the training process to enhance visualization quality by using large batch sizes for training, which naturally slows down the convergence rate. While this approach necessitated increasing the number of training epochs to compensate, resulting in longer overall training time, this deliberate slowing of the training rate prevented the model from reaching near-perfect accuracy within just a few epochs, which would have limited our ability to observe meaningful changes in the embedding space. Instead, we can now visualize a more gradual evolution of the network's learning process, capturing subtle shifts in how the model organizes and separates different digit classes across multiple stages of training. In online processing, we would not agree to invest significantly more time than the original training period, solely for the purpose of achieving visually more appealing results.

4.3 Data Compression

Offline processing, while offering many advantages, requires storing substantial amounts of neural network activation data. In our case, we needed to save activation values for 10,000 MNIST test images across 45 training epochs. Efficient storage became a critical consideration to ensure reasonable file sizes and loading times for our visualization tool.

Initially, our naive approach stored data in a JSON format where each image had its own array object containing 10 activation weights. This organization, though intuitive, proved highly inefficient due to the redundant structure and metadata overhead. We restructured our data format to instead organize by activation node, where each of the 10 output neurons became a key with an array of 10,000 values, one per image. This transposed representation eliminated the need to store indices explicitly, as they could be implicitly inferred from array positions. This reorganization dramatically reduced our JSON file size from over 200 MB to approximately 95 MB after also removing all unnecessary whitespace and formatting.

To further optimize storage efficiency, we utilized MessagePack, a binary serialization format that offers the structured data capabilities of JSON. MessagePack encodes numeric values more efficiently and eliminates the text-based nature of JSON, resulting in our final data payload shrinking to just 40 MB.

4.4 Embedding Projections

Dimensionality reduction techniques we choose offer different approaches to visualizing data of high dimensions, and we incorporate and compare them.

Principal component analysis (PCA) projects data along directions of maximum variance, providing a mathematically straightforward and deterministic method that preserves global structure but often fails to reveal local relationships. The t-distributed stochastic neighbour embedding (t-SNE) works by minimizing sum across all pairs of points marked i and j of $p_{ij}log(\frac{p_{ij}}{q_{ij}})$ where p_{ij} is distance in the original space and q_{ij} is new distance in the projection. Due to properties of logarithm, it does not care much about putting distant points further from each other but not closer and preserves well close points, which leads to nice dimensional reduction for clearly clustered data, although it can be computationally expensive. Uniform manifold approximation projection (UMAP) similarly emphasizes local structure while better preserving global relationships than t-SNE and offering faster computation, though its mathematical foundations are more complex. Star coordinates takes a different approach by mapping each dimension to an axis radiating from a central point, allowing interactive manipulation of the projection through axis adjustment, providing intuitive control but potentially introducing distortions that can be difficult to interpret without careful axis placement.

Data from the neural network are stored in 3D $E \times N \times 10$ matrix, where E is the number of epochs and N is the number of classified points. Classification of one digit in one epoch is represented by a vector of ten floats, which correspond to normalized likelihood of digit belonging to each of the ten classes. This in effect means that we need to transform 10 dimensional points to space which we can visualize via dimension reduction. In our case, we choose to visualize them in 2D.

4.4.1 STAR coordinates

First, we implemented STAR coordinates embedding for which we simply generate 10 evenly spaced points on a circle with diameter of 1. Then from these points, we calculate sine and cosine values into 10×2 matrix, called *angles*. Then, for each epoch, our implementation takes the $N \times 10$ array representing the points and perform matrix multiplication with the *angles* matrix. Resulting from this is $N \times 2$ matrix of 2D positions we plot. This multiplication is in effect sum of x and y coordinates of each classification weighted by the likelihood assigned by the neural network.

The projections were first tried in python, due to ease of implimentation, with libraries such as numpy or Sklearn. Working implementation would then be translated into JavaScript, which was the language chosen for our web application.

4.4.2 PCA

We implemented and modified PCA to work better for our intended purpose. In general, PCA works by multiplying data with transformation matrix. Normally, transformation matrix is calculated for the data, but to keep it from spinning and more easy to evaluate, we calculate it only once from one of the more trained state. This in effect makes our implementation of PCA very similar to STAR coordinates, but with points not evenly on a circle but spaced around to better represent the evaluation by model. The data in PCA need to have mean of zero across dimensions, and for creation of the transformation matrix, they also need to be divided by standard deviation. Once we have processed the data from which we will form the transformation matrix, we calculate their covariance matrix and then we perform eigen decomposition. Last step is ordering eigenvectors by corresponding eigenvalues, in descending order, and picking first X to form the transformation matrix where X is the dimension we want to transform to, and the vectors indicate which directions preserve most variance. For every epoch then, as mentioned above, we just need to multiply the weights with the matrix.

4.4.3 UMAP and t-SNE

The UMAP and t-SNE were included in the visualization via library functions, mainly due to their complexity. There was an attempt to implement t-SNE, but resulting runtime was unbearable.

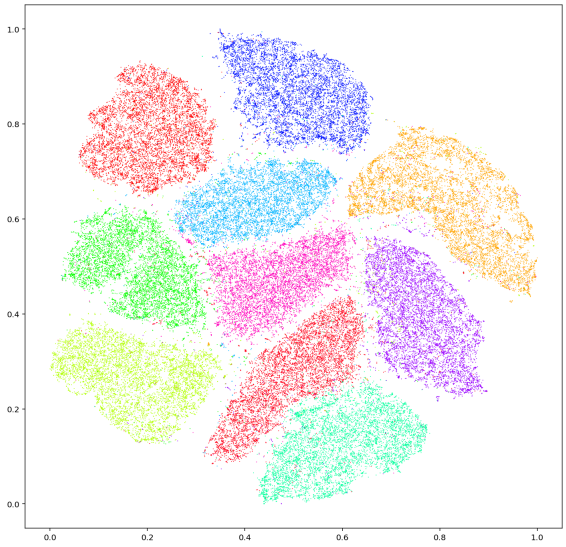

Figure 7: MNIST visualization via t-SNE from Wikipedia

The problem with both approaches is that they are quite slow to compute and do not run in real-time in our web application as opposed to our implementations of star coordinates and PCA. While this was bearable for offline processing in Python, the resulting unresponsiveness of the JavaScript application would be unbearable. For t-SNE embedding, we sampled only a fraction (every 25-th point) of the 10,000 samples to not wait ages for the result.

5 Results

First issue we had to deal with was too fast learning rate of our neural network. Just first batch got it to around 80% accuracy, which meant that we could not get data from early classification. Two solutions were possible, either somehow split training epoch and launch evaluations in these splits, or slow down the training speed of the model, which, due to ease of implementation, we eventually went with.

As mentioned in the implementation section, there was an unpleasant surprise of the long runtimes of UMAP and t-SNE, especially in Javascript, where we could not rely on efficient libraries. It could have been anticipated, these methods are far more complex than STAR or PCA to calculate and they must run multiple iterations before getting result. This led to limiting ambitions with regard to these methods. As for results of visualization, we have recieved interesting results for t-SNE. The expected result of visualizing clasification of MNIST dataset is to have clouds or blobs as seen in picture 7. Whereas our visualization produced kind of tentacles and snaking lines, as can be seen in picture 8. We have two theories for this. Either it is due to overfitting by the neural network, accidentaly caused by our slowing down of learning rate, which could be putting extreme differences where they should not really be, in turn making t-SNE averse to putting some points close to each other. Or it can be caused by relatively low sample size in our plot. Since in the original dimension, data has kind of star/ball shape and most of data of one type are concentrated around the intersections with axes, leaving most of the star relatively empty. Maybe due to small sample size, this could lead t-SNE to disregard most of the shape and visualize only the cones of data not from the *top* but from *side*.

5.1 Web Application

The repository is available on FEE CTU GitLab instance [6] (CTU login is needed) and is being hosted on the GitLab Pages for user to experience right from their web browser. [7]

After a moment of loading a website when the MessagePack is being decompressed and the data representing the weights of all 10,000 test images across all 45 training epochs are loaded into

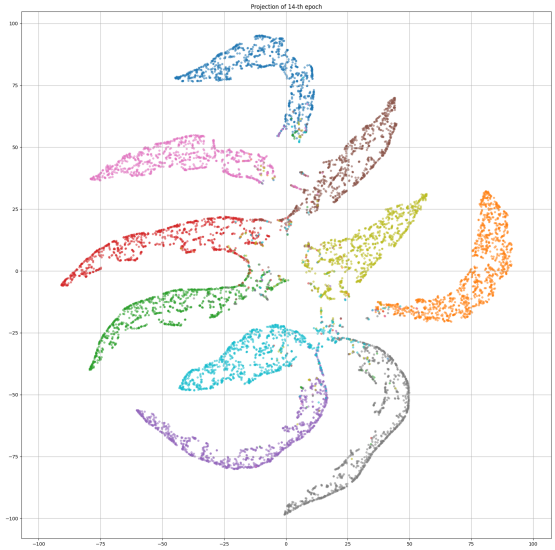

Figure 8: Our results of MNIST visualization via t-SNE (from Python)

the memory, the user is greeted with our visualization web application interface as seen in picture 9. The majority of the screen is occupied by the 2D projection window where 10,000 points are plotted. In the lower right corner, there is a small star graph of weights distribution which we explore later, and an accuracy of the neural network in the current epoch, here being the 1st epoch. Below is a slider used for choosing the epoch. Finally, at the bottom the user can choose between four embeddings and use the colour labels to interpret the point colours as digit classes.

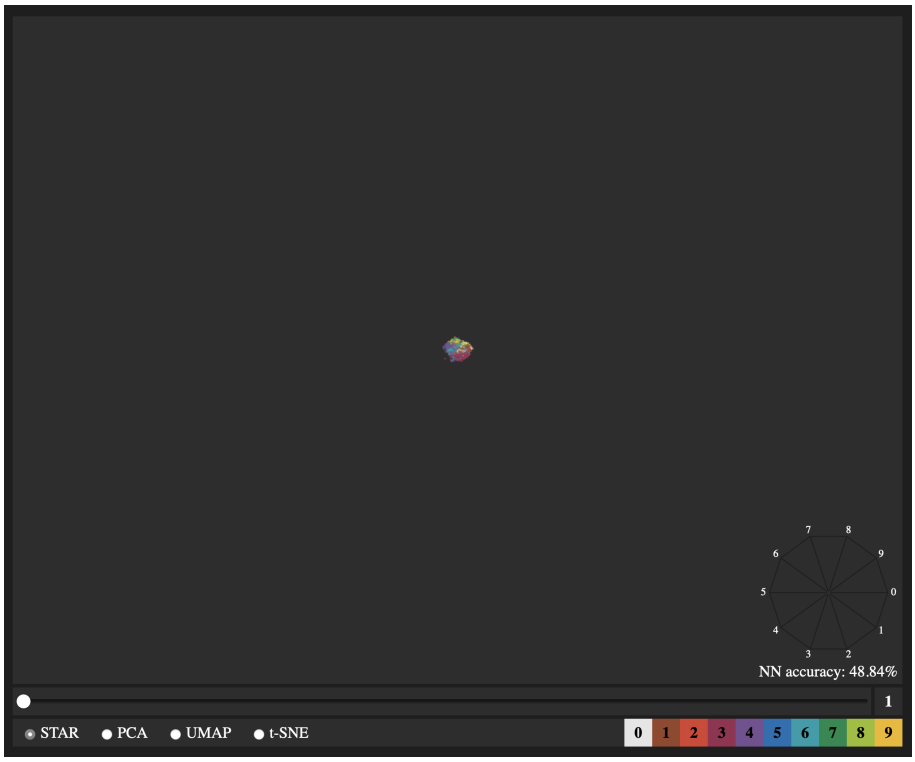

Figure 9: Visualization web application in JavaScript

Upon moving the slider and choosing later training epoch, the initial cloud of points explodes to its projected positions. As discussed in the implementation section, star coordinates and PCA

embeddings are fast enough to be able to manipulate the slider and see the results in real-time, but UMAP and t-SNE both introduce a noticeable stutter when selected. User is advised to remain patient and wait for the result, it will eventually appear as in picture 10.

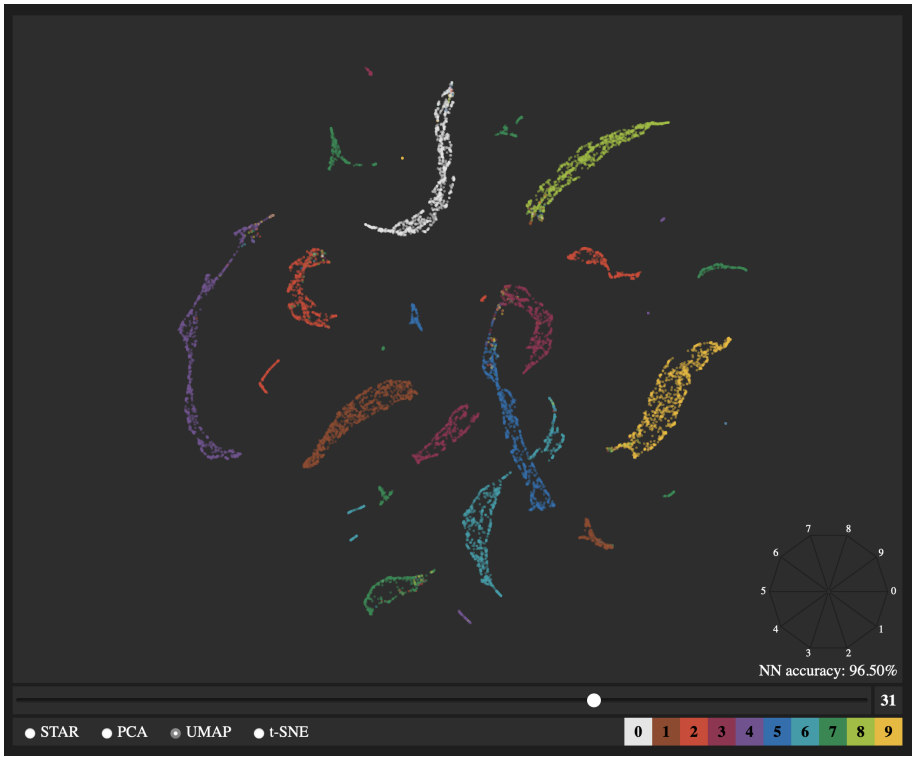

Figure 10: Projections of 10,000 points in epoch 31 using the UMAP embedding.

Each image point is coloured by its real image class. The user can hover their mouse above some of the points or above the digit labels at the lower right corner of the window to highlight the corresponding digit class. The class remains coloured while others are drawn black. This behaviour can be used to study the distribution of the points in the projection space, with lonely points appearing away from the corresponding clusters being miss-classified by the neural network, see picture 11.

Any point can be clicked on. This action fills the star graph at the lower right corner of the projection window with the predictions of the neural network for that specific test image. The user can then compare these predictions with the actual real digit class of the selected point represented as its colour. Additionally, the user can see the trace of the point throughout the training epochs. The current point for the current epoch in the highlighted trace is represented by a slightly larger circle. It may happen that the trace does not contain all the points of all the epochs, as seen in picture 12. That is because the projections are computed on demand and are cached, so the user must first move the slider from the very first epoch to the last, computing all the embeddings. Then, the trace of the point shows its path across all epochs, see picture 13.

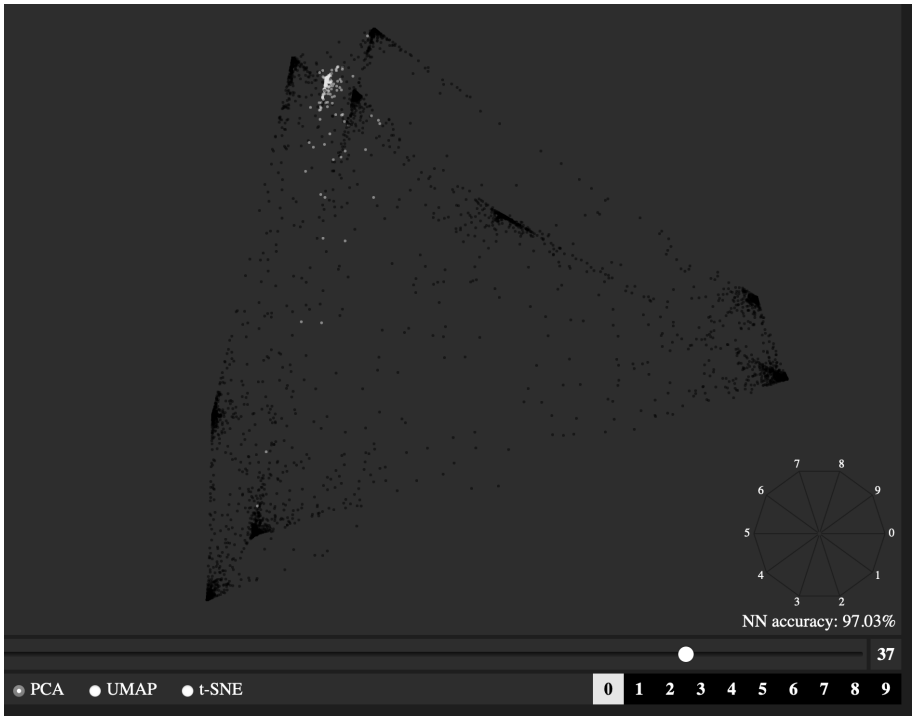

Figure 11: Highlighted points of digit 0 show some points that are scattered away from the primary class point cloud. These lonely points represent a miss-classified test image.

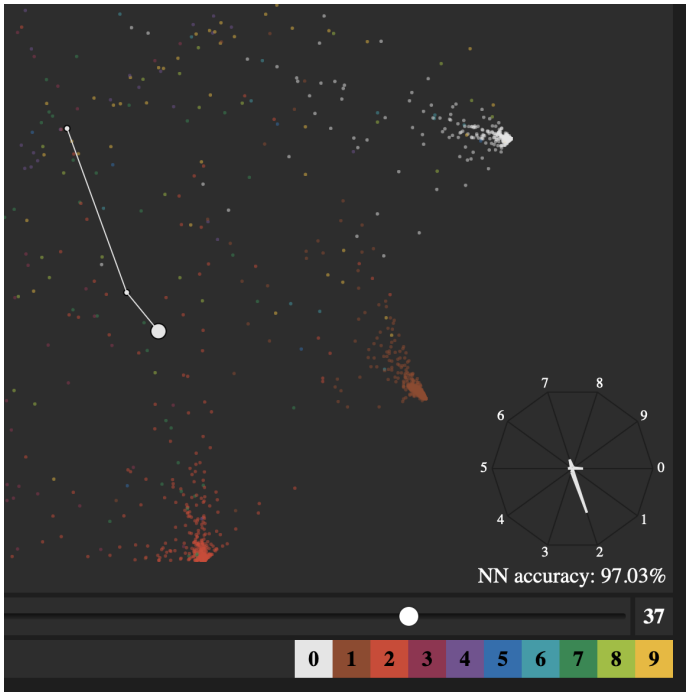

Figure 12: Selecting a point far away from the main cluster for its digit class, in this case digit 0, reveals the wrong predictions of the neural network. Here, the test image of the digit 0 is incorrectly assumed to be a digit 2.

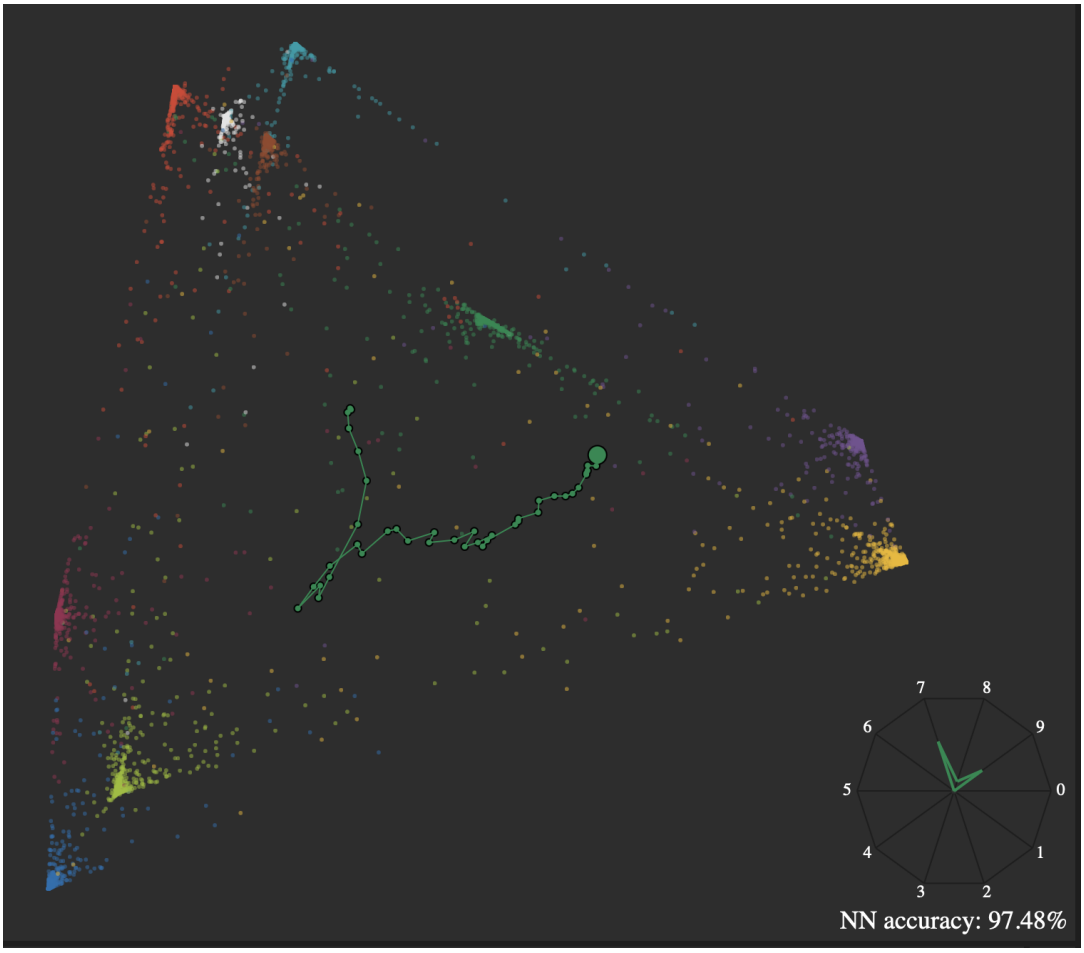

Figure 13: Tracing an image point enables the user to see how the predictions of the neural network changed throughout the 45 training epochs. The highlighted point was first interpreted as a digit 8, here represented as the lime coloured cluster. It then started moving towards the digit 9, the yellow right-most cluster, after finally turning towards its correct digit class 7. The star graph at the lower right corner shows the predictions for this concrete image point after the last training epoch 45. The user can see the image is correctly classified as digit 7, but the neural network is not really confident, giving high probability also to the digit 9 and still some probability to the initial incorrect classification as digit 8.

6 Conclusion

In this report, we presented an analysis of neural network training visualization techniques with a focus on embedding projections. By implementing a visualization system for MNIST digit classification, we demonstrated how dimensionality reduction approaches like t-SNE, UMAP, PCA, and star coordinates reveal the internal organization and learning process of neural networks. Our offline processing approach offers practical advantages in terms of implementation simplicity, user experience, and resource efficiency. Through data compression techniques, we successfully reduced our storage requirements from 200 MB to just 40 MB using MessagePack serialization. While our current implementation provides valuable insights into model behavior through 2D projections, there are several promising directions for future work. These include extending the visualization to 3D representations for richer structural understanding, incorporating feature visualization techniques to showcase what specific neurons are detecting, and creating more interactive elements to allow users to manipulate projections in real-time.

References

[1] Jaegul Choo and Shixia Liu. "Visual Analytics for Explainable Deep Learning". In: *IEEE computer graphics and applications* (2018).

[2] Fred Hohman et al. "Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers". In: *IEEE transactions on visualization and computer graphics* (2018).

[3] Minsuk Kahng et al. "ACTIVIS: Visual Exploration of Industry-Scale Deep Neural Network Models". In: *IEEE transactions on visualization and computer graphics* (2018).

[4] Birant D. Kosemen C. "Multi-label classification of line chart images using convolutional neural networks". In: *SN Applied Sciences* 2 (2020). ISSN: 2523-3971. DOI: 10.1007/s42452-020-3055-y. URL: https://doi.org/10.1007/s42452-020-3055-y.

[5] Hao Li et al. "Visualizing the Loss Landscape of Neural Nets". In: *Advances in Neural Information Processing Systems* (2018).

[6] Jakub Pícha and Jakub Profota. *NN Visualization Repository (CTU login needed)*. 2025. URL: https://gitlab.fel.cvut.cz/profojak/NN-visualization.

[7] Jakub Pícha and Jakub Profota. *Visualization of Training of Neural Network*. 2025. URL: https://nn-visualization-4310e2.pages.fel.cvut.cz.

[8] Fan-Yin Tzeng and Kwan-Liu Ma. "Opening the Black Box – Data Driven Visualization of Neural Networks". In: *VIS 05* (2005).