SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be **University** *u/s 3 of UGC Act, 1956)*

EXPLAINABLE NEURAL NETWORK

Turning the BlackBox into WhiteBox

A PROJECT REPORT SUBMITTED TO

SRM INSTITUTE OF SCIENCE & TECHNOLOGY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

AWARD OF THE DEGREE OF

MASTER OF SCIENCE

BY

RAJESH K (REG NO. RA2332014010083)

UNDER THE GUIDANCE OF

Dr. ANWAR.R.SHAHEEN PhD, M.Phi, MBA, MCA

DEPARTMENT OF COMPUTER APPLICATIONS

FACULTY OF SCIENCE AND HUMANITIES

SRM INSTITUTE OF SCIENCE & TECHNOLOGY

Kattankulathur – 603 203

Chennai, Tamilnadu

OCTOBER - 2024

BONAFIDE CERTIFICATE

Bonafide

This is to certify that the project report titled "**EXPLAINABLE NEURAL NETWORK - Turning the BlackBox into WhiteBox**" is a bonafide work carried out by **Rajesh K** (RA2332014010083) under my supervision for the Degree of Master of Science award. To my knowledge, the work reported herein is the original work done by this student.

Dr. ANWAR.R.SHAHEEN PhD, M.Phi, MBA, MCA

Assistant Professor,

Department of Computer Applications

(GUIDE)

Dr. R.Jayashree

Associate Professor & Head,

Department of Computer Applications

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

With profound gratitude to the ALMIGHTY, I take this opportunity to thank the people who helped me complete this project.

I would like to take this moment to express my heartfelt thanks to my parents, who have always stood by me with the encouraging words, "YOU CAN."

I am thankful to Dr. T. R. Paarivendhar, Chancellor, and Prof. A. Vinay Kumar, Pro Vice-Chancellor (SBL), SRM Institute of Science & Technology, for providing me with the platform to strive for greater heights.

I sincerely thank Dr. A. Duraisamy, Dean, Faculty of Science and Humanities, SRM Institute of Science & Technology, for continuously encouraging me to pursue new and innovative ideas.

A special note of gratitude to Dr. S. Albert Antony Raj, Deputy Dean, Faculty of Science and Humanities, for his valuable guidance and constant support throughout this project.

I express my sincere thanks to Dr. R. Jayashree, Associate Professor & Head, for her unwavering support in facilitating my learning.

I would also like to extend my heartfelt gratitude to my professor, Dr. Anwar Shaheen, PhD, MPhil, MBA, MCA, Assistant Professor, Department of Computer Applications, Faculty of Science & Humanities, for his guidance and unwavering support throughout this project. His insights and expertise have been invaluable in shaping my understanding of the subject.

I convey my gratitude to all the faculty members of the department who offered their support through valuable comments and suggestions during the reviews.

Finally, my heartfelt thanks to friends and everyone, known and unknown, who helped in the successful completion of this project.

EXPLAINABLE NEURAL NETWORK

Turning the BlackBox into WhiteBox

Table of Contents

ABSTRACT

This project investigates the explainability and interpretability of neural networks by utilizing advanced Explainable AI (XAI) techniques, specifically focusing on saliency maps and activation function visualizations. We employ a fully connected neural network model trained on the MNIST dataset to enhance our understanding of the processes involved in classifying handwritten digits. By implementing saliency maps, we visualize pixel-level influences that significantly contribute to the model's predictions, providing valuable insights into feature importance. Furthermore, we analyze activations from various layers of the network to explore the learned representations at different processing stages. Throughout the training and evaluation phases, we present insights into key performance metrics, including cross-entropy loss and accuracy, while leveraging visual tools to promote transparency in the decision-making processes of neural networks. The outcomes of this study aim to improve model interpretability and foster user trust in AI systems by clarifying the underlying mechanisms of deep learning models.

List of Figures

List of Tables

Project Objectives

1. **Develop a Neural Network Model for Digit Classification**
    - Create and optimize a neural network architecture to accurately classify handwritten digits from input images.

2. **Implement Explainable AI Techniques**
    - Integrate Explainable AI (XAI) methods to improve the interpretability of the model's predictions, allowing users to understand the decision-making process behind each classification.

3. **Visualize Model Predictions and Internal Mechanisms**
    - Utilize saliency maps and activation visualizations to illustrate which parts of the input images are most influential in the model's predictions, providing insights into the model's internal workings.

4. **Evaluate Model Performance and Explainability Features**
    - Conduct comprehensive evaluations of the model's accuracy and performance metrics while assessing the effectiveness of the explainability techniques in enhancing user comprehension of the model's decisions.

1. Introduction

Recent advancements in machine learning have led to significant breakthroughs in a variety of sectors, particularly in image recognition and classification tasks (LeCun et al., 2015; Russell & Norvig, 2016). From autonomous vehicles to biometric identification, machine learning algorithms have showcased their extraordinary abilities to analyze and interpret complex datasets (Goodfellow et al., 2016). Deep learning, which employs artificial neural networks with multiple layers, has emerged as one of the most effective methodologies for identifying patterns and features within high-dimensional information (Krizhevsky et al., 2012). However, as these models grow in complexity, their lack of transparency becomes a pressing concern, particularly in critical fields like healthcare and finance (Doshi-Velez & Kim, 2017). This opacity can hinder their widespread acceptance, especially when understanding the rationale behind predictions is essential for informed decision-making (Gilpin et al., 2018).

In the realm of healthcare, machine learning models are increasingly utilized for disease diagnosis, patient outcome predictions, and treatment customization (Obermeyer et al., 2019). When a model suggests a diagnosis or treatment, healthcare professionals must grasp the basis for these recommendations to ensure both patient safety and ethical compliance (Shen et al., 2017). An erroneous prediction or misclassification can have dire consequences, emphasizing the need for models that not only deliver accurate results but also offer explainable insights into their decision-making processes (Yang et al., 2021).

The financial industry faces similar challenges. Machine learning algorithms play a vital role in areas such as credit scoring, fraud detection, and risk assessment (Friedman, 2001; Bansal et al., 2018). Institutions in this sector have a legal and ethical responsibility to justify their decisions, especially those affecting individuals' financial statuses (Binns, 2018). If a model operates as a "black box," it can erode trust among stakeholders, create regulatory obstacles, and perpetuate biases (O’Neil, 2016). Therefore, enhancing transparency through Explainable AI (XAI) becomes critical to encourage the responsible use of AI, reduce risks, and ensure adherence to legal requirements (Miller, 2019).

To address the interpretability challenges associated with machine learning, this paper focuses on incorporating Explainable AI techniques into a neural network-based system designed for digit classification. The main objective is to construct a model that not only demonstrates high accuracy in identifying handwritten digits but also elucidates its decision-making process (Cohen et al., 2020). This is particularly significant in educational contexts, where understanding the model's reasoning can improve learning

experiences by offering students and educators a clearer understanding of how predictions are made (Huang et al., 2020).

By employing various XAI techniques, including saliency maps, Layer-wise Relevance Propagation (LRP), and SHapley Additive exPlanations (SHAP), this study aims to clarify the neural network's internal operations (Sundararajan et al., 2017; Lundberg & Lee, 2017). Saliency maps, for example, visually represent which regions of an input image significantly influence the model's predictions, allowing users to see which features are critical for classification (Springenberg et al., 2014). Similarly, methods like LRP and SHAP provide numerical assessments that highlight the contribution of each feature to the model's output, fostering a deeper comprehension of its logic (Ancona et al., 2017).

This research endeavors to bridge the divide between the intricate nature of deep learning models and the necessity for interpretability in their applications (Miller, 2019). By promoting transparency and enhancing user trust, we aim to facilitate the adoption of machine-learning technologies in essential sectors while empowering stakeholders to engage meaningfully with these systems (Guidotti et al., 2018). Additionally, the insights gained from this integration will pave the way for continuous model enhancement, leading to the development of more robust, fair, and trustworthy AI systems (Karpathy et al., 2015).

In conclusion, this study aims to achieve two primary objectives: to create a highly effective neural network model for digit classification and to enhance its interpretability through the application of Explainable AI techniques (Lakkaraju et al., 2016). By addressing these intertwined challenges, we hope to contribute to the expanding body of knowledge in artificial intelligence, promoting the responsible and transparent use of machine learning technologies across various applications (Binns, 2018).

2. Terms and Definitions

2.1 Neural Network

A **neural network** is a sophisticated computational model inspired by the human brain's architecture, comprising interconnected units known as neurons. Each neuron receives one or more input signals, applies a mathematical transformation—often referred to as an activation function—and produces an output signal. Neural networks have gained prominence due to their remarkable efficacy in pattern recognition and classification tasks, particularly in the realm of image processing (Goodfellow et al., 2016). These

networks learn from vast datasets, enabling them to identify intricate patterns that may not be apparent to traditional algorithms.

2.2 Architecture of the Neural Network

The **architecture** of a neural network describes the layout of its layers and the connections between individual neurons. This structure is pivotal for determining the model's capacity to learn and generalize from data. Several architectural designs exist, including:

- **Feedforward Neural Networks**: Characterized by connections that flow in one direction—from input to output—without forming cycles. This design simplifies the computational process, making it suitable for tasks such as digit recognition (LeCun et al., 2015).
- **Convolutional Neural Networks (CNNs)**: Specifically tailored for processing grid-like data such as images. CNNs utilize convolutional layers to automatically learn spatial hierarchies of features (Krizhevsky et al., 2012).
- **Recurrent Neural Networks (RNNs)**: Designed to handle sequential data by maintaining a form of memory, allowing them to learn temporal dependencies. RNNs are particularly effective in tasks involving language and time series data (Hochreiter & Schmidhuber, 1997).

2.3 Types of Layers

Neural networks typically consist of several types of layers that perform distinct functions:

- **Input Layer**: This layer serves as the entry point for data into the network. Each neuron in this layer corresponds to a feature of the input data.
- **Hidden Layers**: These intermediate layers perform computations and feature extraction. Each hidden layer can contain multiple neurons that transform inputs using activation functions, facilitating complex learning (Bengio et al., 2013).
- **Output Layer**: This final layer produces the network's output, representing the predicted classes or values based on the input data.

2.4 Weights and Bias

Within a neural network, **weights** and **biases** are fundamental parameters that the model adjusts during training to minimize prediction errors.

- **Weights** determine the strength and direction of the influence that one neuron has on another, influencing how inputs are transformed into outputs (Rumelhart et al., 1986).

- **Biases** are additional parameters that allow models to adjust their outputs independently of their inputs, facilitating better fitting of the training data (Goodfellow et al., 2016).

2.5 Feed Forward Neural Network

A **feed-forward neural network** is a foundational type of artificial neural network where connections between nodes do not form cycles. In this architecture, data flows in a single direction—from input nodes, through hidden nodes, to output nodes—enabling clear and efficient computation. The simplicity of this design allows for faster training and straightforward implementation, making it a popular choice for tasks like digit classification (LeCun et al., 2015).

2.6 Activation Functions and Their Types

Activation functions are critical components of neural networks, introducing non-linearities that allow the model to learn complex relationships within the data. Common activation functions include:

- **Sigmoid**: This function maps input values to a range between 0 and 1, making it suitable for binary classification problems. However, it can lead to the vanishing gradient problem, limiting its effectiveness in deep networks (Glorot & Bengio, 2010).
- **Tanh**: The hyperbolic tangent function maps input values to a range between -1 and 1. This function often performs better than sigmoid due to its zero-centered output (LeCun et al., 2015).
- **ReLU (Rectified Linear Unit)**: ReLU outputs the input directly if it is positive; otherwise, it returns zero. This function has become the default activation function for hidden layers in deep networks due to its computational efficiency and ability to alleviate the vanishing gradient problem (Nair & Hinton, 2010).

2.6.1 Types of Activations Used in This Project and Why

In this project, **ReLU** is predominantly utilized in hidden layers due to its computational efficiency and effectiveness in mitigating issues associated with vanishing gradients. In the output layer, the **softmax activation function** is employed to provide a normalized probability distribution across the various classes of digits, facilitating clear predictions in multi-class classification tasks (Goodfellow et al., 2016).

2.7 Backpropagation

Backpropagation is an essential supervised learning algorithm employed to train neural networks. It works by calculating the gradient of the loss function concerning

each weight through the application of the chain rule of calculus. This process enables the model to update its parameters effectively, minimizing the loss over iterations and enhancing the network's ability to make accurate predictions (Rumelhart et al., 1986).

2.8 Loss Functions and Their Definitions

Loss functions are critical in evaluating how well a neural network's predictions align with actual values. Commonly used loss functions include:

- **Cross-Entropy Loss**: Frequently used for multi-class classification problems, this function measures the dissimilarity between the true distribution and the predicted probabilities, thereby penalizing incorrect predictions more significantly (Kullback & Leibler, 1951).
- **Mean Squared Error (MSE)**: This function calculates the average squared difference between predicted and actual values, making it a standard choice for regression tasks. It emphasizes larger errors more than smaller ones, which can be beneficial in certain contexts (Hastie et al., 2009).

2.9 Optimization Algorithms (Gradient Descent)

Optimization algorithms are essential for minimizing loss functions by adjusting the parameters of the model. **Gradient descent** is the most widely utilized optimization algorithm, which updates weights iteratively in the opposite direction of the gradient of the loss function concerning the parameters. Variants of gradient descent, such as stochastic gradient descent (SGD), introduce randomness to the updates, often leading to faster convergence (Bottou, 2010).

2.10 Explainable AI

Explainable AI (XAI) encompasses various methods and techniques aimed at making AI systems transparent and interpretable to humans. The goal of XAI is to elucidate how models arrive at specific decisions, thereby improving user trust and facilitating model validation. Techniques such as saliency maps and feature importance analyses are integral to enhancing the interpretability of complex models (Miller, 2019).

2.11 Visualization Techniques

Visualization techniques in artificial intelligence are essential for interpreting model predictions and understanding underlying patterns in data. Techniques include:

- **Saliency Maps**: These maps highlight regions within an input image that significantly impact the model's predictions. By computing the gradients of the output concerning input pixels, saliency maps illuminate which areas of an image the model focuses on during prediction (Simonyan et al., 2014).
- **Activation Visualizations**: These techniques provide insights into the internal workings of the model by visualizing the activations of different layers, helping researchers understand what features are being learned at various stages (Zeiler & Fergus, 2014).
- **Model Distillation**: This technique involves simplifying a complex model into a smaller, more interpretable one while retaining essential performance characteristics, enhancing understandability without sacrificing accuracy (Hinton et al., 2015).

2.12 Saliency Map

A **saliency map** serves as a powerful visualization tool that identifies the regions of an input image that significantly influence the model's output. By calculating the gradients of the model's output concerning input pixels, saliency maps reveal which parts of the image the model focuses on when making predictions. This visualization not only aids in understanding model behavior but also highlights potential biases in decision-making (Fong & Vedaldi, 2017).

2.13 Interpretability of the Hidden Layers

Interpretability of hidden layers involves understanding the features and patterns that a model learns during different stages of processing. Analyzing the activations of various layers can provide researchers with insights into the hierarchical feature extraction performed by the neural network. Techniques such as layer-wise relevance propagation and t-SNE visualizations are often employed to interpret these complex interactions, contributing to a deeper understanding of model behavior and enhancing trust in AI systems (Sundararajan et al., 2017).

3. Dataset Overview

The MNIST (Modified National Institute of Standards and Technology) dataset is one of the most important resources in machine learning and computer vision. Widely used for image classification tasks, MNIST comprises 70,000 grayscale images of handwritten digits ranging from 0 to 9. Each image has a resolution of 28x28 pixels, making it compact and manageable for various computational tasks. The dataset is systematically organized into 60,000 training samples and 10,000 test samples, enabling effective training and evaluation of different machine learning models.

3.1 Origin and Characteristics

The MNIST dataset originated from the NIST (National Institute of Standards and Technology) dataset, which initially contained scanned images of handwritten digits created by U.S. Census Bureau employees and high school students. The dataset was meticulously processed by researchers Yann LeCun, Corinna Cortes, and Chris Burges, who focused on cleaning and normalizing the images. Their preprocessing steps included centering each image and resizing them to a consistent 28x28 pixel format, ensuring uniformity across the dataset.

Each image in the MNIST dataset is represented as a flattened vector of 784 pixels (28 multiplied by 28). The pixel values are in the range of 0 (black) to 255 (white), representing different grayscale intensities. This standardization not only facilitates ease of use in various machine learning algorithms but also enhances the efficiency of image classification tasks.

3.2 Dataset Characteristics

- **Size:** 70,000 images
- **Training Samples:** 60,000 images
- **Test Samples:** 10,000 images
- **Image Size:** 28x28 pixels
- **Pixel Values:** 0 to 255 (grayscale)

3.3 Accessibility and Open-Source Nature

One of the remarkable features of the MNIST dataset is its open-source nature and widespread availability. It can be easily accessed through several platforms, including the official MNIST website, Scikit-learn (OpenML), and Keras. This accessibility has played a significant role in establishing MNIST as one of the most thoroughly documented and supported datasets in the field of machine learning.

Numerous resources, including research papers, tutorials, and code implementations, accompany the MNIST dataset, making it an ideal starting point for beginners. The abundance of documentation allows newcomers to quickly grasp fundamental concepts related to machine learning, data preprocessing, model training, and evaluation, all without the complexities that typically accompany real-world datasets.

3.4 Rationale for Selecting MNIST

There are several compelling reasons for choosing the MNIST dataset for projects and educational purposes:

1. **Simplicity of the Task**
   The MNIST dataset presents a straightforward image classification problem centered around recognizing handwritten digits. This simplicity makes it particularly attractive to those new to machine learning and computer vision. By working with MNIST, learners can acquire essential knowledge about data preprocessing, model training, and evaluation without being overwhelmed by more complex datasets.
2. **Low Computational Requirements**
   Given the relatively small size of each image and the overall dataset, MNIST allows for rapid processing and modeling on standard hardware. Unlike larger datasets such as CIFAR-10 or ImageNet, MNIST does not require high-performance GPUs or extensive cloud computing resources. This feature makes it a suitable choice for researchers and practitioners who may be operating within constrained computational environments.
3. **Benchmarking Capabilities**
   Historically, MNIST has served as a benchmark dataset for evaluating and comparing the performance of various machine learning models, especially Convolutional Neural Networks (CNNs). The dataset’s controlled environment enables researchers to assess algorithm performance accurately and provides a solid foundation for subsequent research involving more complex datasets. This benchmarking aspect has solidified MNIST's importance in the development of new algorithms and techniques in machine learning.
4. **Extensive Documentation and Open Access**
   The MNIST dataset is accompanied by comprehensive documentation, tutorials, and example code available across multiple platforms such as TensorFlow, PyTorch, and Scikit-learn. This extensive support promotes easy learning and troubleshooting, making MNIST a popular choice for initial experimentation. Learners can quickly implement algorithms and evaluate their performance, fostering a hands-on understanding of machine learning concepts.

5. **Support for Deep Learning**
   The simplicity and clear classification objectives associated with the MNIST dataset make it an excellent platform for exploring deep learning techniques, particularly CNNs. It provides practitioners with the opportunity to gain practical experience in developing and training neural networks, laying the groundwork for future work with more complex datasets.

3.5 Comparison with Other Datasets

While alternative datasets, including CIFAR-10, Fashion MNIST, and ImageNet, are available, MNIST remains the optimal choice for foundational projects due to its simplicity and minimal computational demands.

Table 1: Comparison of the Datasets

Dataset	Image Size	Task Complexity	Hardware Requirements	Use Case
MNIST	28x28	Simple (Digit Classification)	Low	Beginner-friendly, Benchmarking
CIFAR-10	32x32	Moderate (Object Classification)	Moderate to High	Learning CNNs, Intermediate ML
Fashion MNIST	28x28	Simple (Fashion Item Classification)	Low	More realistic than MNIST, Intermediate ML
ImageNet	Variable	Complex (1000 Classes)	High	Advanced CV Research, Deep Learning

- **CIFAR-10**
  The CIFAR-10 dataset contains color images categorized into 10 distinct classes. This increased complexity not only requires higher computational resources but also presents greater challenges in model training and evaluation. Consequently, CIFAR-10 may not be the best option for individuals seeking to build foundational skills in machine learning.
- **Fashion MNIST**
  The Fashion MNIST dataset serves as a more challenging variant of the original MNIST, featuring images of various clothing items. Although it introduces an interesting twist, the added complexity may not align with the goals of foundational learning projects focused on accessibility and simplicity.

- **ImageNet**
  ImageNet is a large-scale dataset that includes millions of high-resolution images categorized into thousands of classes. The substantial processing and training requirements associated with ImageNet demand significant hardware capabilities, rendering it less suitable for early-stage experimentation. Beginners looking to establish foundational skills in machine learning would benefit more from the straightforward classification task offered by MNIST.

4. Hardware Requirements

The hardware requirements for implementing and running the neural network model for digit classification, utilizing Explainable AI techniques, may vary depending on the dataset size, model complexity, and expected performance. Below are the recommended hardware specifications for an optimal experience:

4.1 Processor (CPU)

- **Minimum:** Dual-core processor (e.g., Intel i3 or AMD Ryzen 3)
- **Recommended:** Quad-core or higher (e.g., Intel i5/i7 or AMD Ryzen 5/7)

4.2 Graphics Processing Unit (GPU)

- **Minimum:** Integrated graphics (e.g., Intel HD Graphics)
- **Recommended:** Dedicated GPU with at least 4 GB VRAM (e.g., NVIDIA GeForce GTX 1050 or higher) for accelerated training and inference of neural networks

4.3 Memory (RAM)

- **Minimum:** 4 GB
- **Recommended:** 8 GB or more for smoother multitasking and handling larger datasets

4.4 Storage

- **Minimum:** 100 GB of available storage
- **Recommended:** Solid State Drive (SSD) with 256 GB or more for faster data access and model loading times

4.5 Display

- A monitor is capable of 1920x1080 resolution or higher for better visualization of results and code.

4.6 Network Connection

- A stable internet connection is recommended for downloading datasets, packages, and libraries during development.

4.7 Optional Hardware

- **External Storage:** For backup of datasets and models.

5. Software Requirements

To successfully implement and run the digit classification model using a neural network with Explainable AI features, the following software and tools are necessary:

5.1 Operating System

Recommended:

- **Windows:** Windows 10 or higher
- **macOS:** macOS Mojave or higher
- **Linux:** Ubuntu 18.04 or higher

5.2 Integrated Development Environments (IDEs)

- **Visual Studio Code:** A popular code editor with extensive support for Python development through various extensions.
- **Jupyter Notebook:** Ideal for interactive coding, data analysis, and visualization. Allows you to create and share documents that contain live code, equations, and visualizations.
- **Google Colab:** A cloud-based Jupyter Notebook environment that provides free access to GPUs, which is particularly beneficial for training deep learning models without local hardware constraints.

5.3 Python

- **Recommended Version:** Python 3.8 or higher
  Ensure Python is installed and included in the system's PATH.

5.4 Required Python Packages

The following libraries are essential for building and training your neural network, as well as for implementing Explainable AI techniques:

- **NumPy:** For numerical operations and handling multi-dimensional arrays.
- **Pandas:** For data manipulation and analysis.
- **Matplotlib:** For creating static, animated, and interactive visualizations in Python.
- **Scikit-learn:** For machine learning utilities, including metrics and model evaluation functions.

5.5 Virtual Environment

It is highly recommended to use a virtual environment to manage project dependencies effectively.

5.6 Version Control

- **Git:** For version control and collaboration. Install Git to manage your code repository and track changes.

5.7 Documentation

- **Markdown Editor:** Any Markdown editor (e.g., Typora, Dillinger) can be used for documenting the project.

6. Setup and Installation

This section outlines the steps required to set up and install the necessary software for implementing the digit classification model using a neural network with Explainable AI features.

6.1 Operating System Installation

1. **Windows:**
    - Download the Windows 10 or higher ISO from the Microsoft website.
    - Create a bootable USB drive using tools like Rufus or the Windows Media Creation Tool.
    - Follow the installation prompts to install Windows.
2. **macOS:**
    - Download the latest macOS version from the Mac App Store.
    - Follow the installation instructions provided by Apple to update or install the operating system.

3. **Linux (Ubuntu):**
    - Download the latest Ubuntu ISO from the official Ubuntu website.
    - Create a bootable USB drive using software like Balena Etcher.
    - Boot from the USB drive and follow the installation prompts to install Ubuntu.

6.2 IDE Installation

1. **Visual Studio Code:**
    - Download the installer from the Visual Studio Code website.
    - Run the installer and follow the setup wizard to complete the installation.
    - Optionally, install Python extensions for enhanced support.
2. **Jupyter Notebook:**
    - Jupyter Notebook is typically installed via Anaconda or Pip.
    - **Using Anaconda:**
        - Download and install Anaconda (which includes Jupyter).

figure 1:

- Using pip:

```bash
pip install notebook
```

- Launch Jupyter Notebook by running:

```bash
jupyter notebook
```

3. **Google Colab:**
    - No installation is required. Simply navigate to Google Colab in a web browser. You can sign in with your Google account to save and share notebooks.

6.3 Python Installation

- **Windows and macOS:**
    - Download the latest Python installer from the Python website.
    - Run the installer and ensure you check the option to add Python to your system PATH during installation.

figure 2:

- Linux (Ubuntu):

```bash
sudo apt update
sudo apt install python3 python3-pip
```

6.4 Required Python Packages Installation

Once Python is installed, you can use pip to install the necessary libraries:

figure 3:

```bash
pip install numpy pandas matplotlib scikit-learn
```

6.5 Documentation Tools

- **Markdown Editor:**
  - Download and install any preferred Markdown editor like Typora or use online editors like Dillinger.

6.6 Setting Up a Virtual Environment

figure: 4

1. Create a Virtual Environment:

```bash
python -m venv myenv
```

2. Activate the Virtual Environment:
   - Windows:

```bash
myenv\Scripts\activate
```

   - macOS/Linux:

```bash
source myenv/bin/activate
```

6.7 Version Control Installation

- **Git:**
    1. **Windows:**
        - Download the Git installer from the Git website.
        - Run the installer and follow the prompts to complete the installation.
    2. **macOS:**
        - Install Git using Homebrew:

figure: 5

```bash
brew install git
```

3. Linux (Ubuntu):

```bash
sudo apt install git
```

7. Packages and Libraries

In developing a digit classification model using neural networks, several Python libraries play a crucial role in facilitating data manipulation, visualization, and machine learning tasks. This section discusses key libraries, including NumPy, Matplotlib, and Scikit-learn, highlighting their functionalities.

7.1 NumPy

NumPy (Numerical Python) is a foundational library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions designed to operate on these arrays efficiently.

Key Functions:

- **Array Creation**:

    - **`np.array()`**: Creates a NumPy array from a list or tuple, allowing for the conversion of nested lists into multi-dimensional arrays.
    - **`np.zeros()`**: Generates an array filled with zeros, with customizable shape for initializing data structures.
    - **`np.ones()`**: Similar to `np.zeros()`, but fills the array with ones, often used for initializing algorithms.
    - **`np.arange()`**: Produces an array with evenly spaced values within a specified range, akin to Python's built-in `range()` function.
    - **`np.linspace()`**: Creates an array with a defined number of evenly spaced values over a specified interval, useful for generating sample data.
- **Array Manipulation**:
    - **Reshaping Arrays**: **`np.reshape()`** allows for changing the shape of an array without altering its data, which is essential for preparing data for various algorithms.
    - **Concatenation**: Functions such as **`np.concatenate()`** and **`np.vstack()`** enable users to merge two or more arrays along a specified axis, facilitating the addition of new data points.
    - **Splitting Arrays**: The **`np.split()`** function divides an array into multiple sub-arrays, often utilized for creating training and testing datasets in machine learning.
- **Mathematical Operations**:
    - **Element-wise Operations**: NumPy supports direct arithmetic operations (addition, subtraction, multiplication, division) on arrays, applying the operation to each element.
    - **Universal Functions (ufuncs)**: Functions like **`np.add()`**, **`np.subtract()`**, **`np.multiply()`**, and **`np.divide()`** perform element-wise operations optimized for performance across arrays of different shapes.
    - **Aggregate Functions**: Functions such as **`np.sum()`**, **`np.mean()`**, **`np.median()`**, and **`np.std()`** compute statistical properties of arrays, with the capability to operate along specified axes for multi-dimensional data analysis.
- **Linear Algebra Functions**:
    - **Matrix Operations**: Functions such as **`np.dot()`** and **`np.matmul()`** facilitate matrix multiplication, a fundamental operation in various applications, including neural networks.

    - **Determinants and Inverses**: `np.linalg.det()` and `np.linalg.inv()` computes the determinant and inverse of a matrix, crucial for solving systems of linear equations.
    - **Eigenvalues and Eigenvectors**: The `np.linalg.eig()` function calculates the eigenvalues and eigenvectors of a square matrix, essential in many areas, including principal component analysis (PCA).
- **Broadcasting**: Broadcasting is a powerful feature that enables arithmetic operations on arrays of different shapes, automatically expanding the smaller array across the larger one. This capability simplifies code and enhances performance.
- **Random Number Generation**: NumPy's random module is robust for generating random numbers, useful for simulations and stochastic processes.
    - `np.random.rand()`: Generates an array of random numbers drawn from a uniform distribution over [0,1)[0, 1)[0,1).
    - `np.random.randn()`: Produces an array of random numbers from a standard normal distribution (mean 0, variance 1).
    - `np.random.randint()`: Creates random integers within a specified range.

By leveraging these features, NumPy becomes an invaluable tool for managing and processing the data used in training neural networks.

7.2 Matplotlib

Matplotlib is a powerful plotting library for Python that enables users to create static, animated, and interactive visualizations. It is particularly beneficial for data visualization in the context of machine learning, as it provides various plotting options and customization features.

Key Functions:

- **Plotting Capabilities**: Matplotlib allows the creation of diverse plot types, including line plots, scatter plots, and histograms. These visualizations help in understanding data distributions and trends.
- **Customization Options**: Users can customize the appearance of their plots extensively, including adding titles, labels, legends, and adjusting colors and styles to enhance clarity and presentation.

- **Subplot Functionality**: Matplotlib supports the creation of multiple plots within a single figure, making it easier to compare different datasets or visualizations side by side.

Through these capabilities, Matplotlib aids in effectively communicating insights derived from data analyses and model predictions.

7.3 Scikit-learn

Scikit-learn is a robust machine learning library for Python that provides simple and efficient tools for data mining and analysis. Built on top of NumPy, SciPy, and Matplotlib, it offers a comprehensive solution for machine learning tasks, making it a popular choice among data scientists.

Key Functions:

- **Data Loading**: Scikit-learn facilitates loading datasets through various methods, including direct downloads from online repositories, streamlining the process of obtaining datasets for experimentation.
- **Preprocessing Tools**: It provides a range of tools for data preprocessing, such as normalization, encoding categorical variables, and splitting datasets into training and testing sets. These steps are crucial for preparing data before feeding it into machine learning models.
- **Model Evaluation Metrics**: Scikit-learn includes a variety of metrics for evaluating model performance, such as accuracy, precision, recall, and F1 score. These metrics enable users to assess the effectiveness of their models and make informed decisions about improvements.

By harnessing the power of Scikit-learn, developers can implement machine learning models efficiently and evaluate their performance systematically.

8. Data Preprocessing

Effective data preprocessing is vital for building robust and accurate machine-learning models. This section outlines the data loading, normalization, encoding, and splitting processes employed in developing a digit classification model using the MNIST dataset. The MNIST dataset is widely recognized as a benchmark in machine learning, particularly in the area of image classification, and consists of 70,000 grayscale images of handwritten digits.

8.1 Data Loading

The first step in any machine learning project is to load the dataset into the programming environment. In this project, we utilize the MNIST dataset, which is readily accessible via the `fetch_openml` function from the `sklearn.datasets` module. This function simplifies the process of downloading and importing datasets directly into Python. The MNIST dataset comprises 70,000 images, each of size 28x28 pixels, representing digits from 0 to 9. This comprehensive dataset not only allows for the testing of machine learning algorithms but also serves as a reliable standard for model evaluation.

The following code snippet illustrates how to load the dataset:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"], mnist["target"].astype(int)

X = np.array(X)
y = np.array(y)
```

In this code, X contains the pixel values of the images, while y holds the corresponding labels (digits). The labels are converted to integers to facilitate further processing. Loading the data in this manner allows us to efficiently handle the dataset as an array of pixel values and a separate array of corresponding labels.

8.2 Normalization

Normalization is a critical step in the preprocessing phase, as it scales the pixel values of the input images to a range between 0 and 1. Each pixel in an MNIST image has a value between 0 and 255. Normalizing these values improves convergence during training and ensures that all features contribute equally to the model's learning process. This is particularly important for neural networks, which can be sensitive to the scale of input data.

To normalize the data, we divide each pixel value by 255.0, as shown below:

```
# Normalize the data
X = X / 255.0
```

By performing this operation, we transform the pixel values, allowing for a more stable training process and enhanced performance of the neural network. Normalization is a widely used technique that not only aids in achieving faster convergence but also helps prevent issues related to numerical instability during training (Bengio et al., 2013).

Why Normalization for This Project?

In the context of the MNIST digit classification task, normalization plays a crucial role. The pixel values of the images can vary significantly, which can lead to uneven gradients during the training process. If the pixel values are not normalized, some features may dominate the learning process, causing the model to converge poorly or even diverge. By scaling the pixel values to a range of [0, 1], we ensure that the neural network learns more effectively, thereby improving classification accuracy.

8.3 Encoding the Output

Output encoding is essential for converting class labels (digits 0-9) into a format that a machine-learning model can work with. In this project, we use one-hot encoding to represent the class labels. One-hot encoding transforms each digit into a binary vector where only the index corresponding to the digit is set to 1, and all other indices are set to 0. For example, the digit '3' is represented as `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`.

This representation allows the model to produce probabilities for each digit class during training, facilitating better interpretation and optimization of the output layer during the learning process. By encoding the labels in this manner, we ensure that the model learns to classify each digit independently without implying any ordinal relationships among the digits.

The following code demonstrates how to implement one-hot encoding using the `OneHotEncoder` from the `sklearn.preprocessing` module:

```
# Convert labels to one-hot encoding
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y.reshape(-1, 1))
```

The `OneHotEncoder` initializes an encoder that converts the labels into one-hot encoded format, ensuring that the model learns to predict the probability of each digit class. One-hot encoding is a standard practice in multi-class classification problems, allowing models to output class probabilities effectively (Goodfellow et al., 2016).

Why One-Hot Encoding for This Project?

For the MNIST classification task, one-hot encoding is particularly beneficial because it allows the model to interpret each digit as an independent class without any ordinal implications. Unlike using a single integer to represent the digit, one-hot encoding prevents the model from incorrectly assuming a numerical relationship between classes (e.g., treating '3' as closer to '2' than to '7'). This distinction is critical for improving the accuracy of the model's predictions, as it helps in optimizing the output layer's weights more effectively.

8.4 Data Splitting

After loading and preprocessing the data, the next step is to split it into training and testing sets. This division allows the model to be trained on one subset of the data while being evaluated on another, ensuring that it generalizes well to unseen data. Proper data splitting is crucial for obtaining an unbiased evaluation of the model's performance and for preventing overfitting.

We utilize the `train_test_split` function from the `sklearn.model_selection` module to achieve this. This function randomly divides the dataset into training and testing subsets based on a specified proportion.

The code below illustrates how to split the data:

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

In this code, we allocate 80% of the data for training and 20% for testing. The `random_state` parameter ensures reproducibility, allowing the same split to be obtained in subsequent runs. The shapes of the training and testing sets are printed to verify the successful splitting of the dataset. By maintaining a consistent random state, we can ensure that our model's performance can be reliably compared across different training sessions.

9. Neural Network Architecture

In this section, we explore the structure and underlying mechanics of the neural network model used for classifying handwritten digits from the MNIST dataset. Neural networks are composed of interconnected layers of nodes (neurons) that process and transform input data, learning patterns through backpropagation and gradient descent.

The architecture of a neural network plays a crucial role in its ability to learn and generalize from data. For our digit classification project, the model comprises three primary layers: an input layer, hidden layers, and an output layer. These layers work together to transform the 28x28 pixel images of handwritten digits into a probability distribution over the 10 possible digit classes (0–9).

9.1 Overview of the Architecture

The neural network used in this project has a straightforward feedforward architecture with two hidden layers, each designed to capture increasingly complex patterns from the input data. Here's an overview of the key components:

- **Input Layer:**
    - The input layer takes the raw pixel values of the 28x28 grayscale images, which are flattened into a 784-dimensional vector (`28 * 28 = 784`). Each pixel value represents the grayscale intensity of the image, and normalization (as explained in earlier sections) ensures that the input values range between 0 and 1.
    - The input layer doesn't perform any computations but simply serves as the starting point for the model's forward pass, passing data into the first hidden layer.
- **Hidden Layers:**
    - The neural network has two hidden layers, which serve to learn intermediate representations of the input data. The number of neurons in each hidden layer can vary, but common choices might include 128 or 64 neurons for the first hidden layer and a slightly smaller number for the second hidden layer.
    - Each neuron in the hidden layers applies a linear transformation to the input data, followed by a non-linear activation function. In our model, we typically use the **ReLU (Rectified Linear Unit)** activation function, which

introduces non-linearity by outputting the input directly if it is positive, and zero otherwise (`f(x) = max(0, x)`).
    - The purpose of the hidden layers is to gradually abstract the input image data, identifying edges, corners, textures, and eventually more complex structures, such as digit shapes.
- **Output Layer:**
    - The output layer consists of 10 neurons, one for each digit class (0–9). The output from each neuron represents the likelihood (or probability) that the input image corresponds to a particular digit.
    - The activation function used in the output layer is **softmax**, which converts the raw scores (logits) from the neurons into probabilities that sum to 1. The softmax function is ideal for multi-class classification problems because it allows the model to express confidence in its predictions by outputting a probability distribution.

Table 2: Model Architecture Summary

Layer Type	Output Shape	Activation Function	Parameters
Input Layer	(batch_size, 784)	-	-
Hidden Layer 1	(batch_size, 128)	ReLU	100480 (784 * 128 + 128)
Hidden Layer 2	(batch_size, 64)	ReLU	8256 (128 * 64 + 64)
Output Layer	(batch_size, 10)	Softmax	650 (64 * 10 + 10)

The overall goal of this architecture is to allow the network to learn, through repeated training, how to map the pixel intensities of an input image to the corresponding digit class.

9.2 Initializing Weights and Biases

The success of training a neural network largely depends on how the model's weights and biases are initialized. Proper initialization ensures faster convergence during training and prevents issues such as vanishing or exploding gradients, especially in deeper networks. We employ **Xavier initialization** (also known as **Glorot initialization**) for the weights in this project, which has become a standard method for initializing weights in deep learning models.

9.2.1 Weight Initialization

Weights in a neural network define the strength of the connections between neurons. During the forward pass, each neuron computes a weighted sum of its inputs and passes the result through an activation function. Initially setting the weights to appropriate values is essential to prevent the network from starting with poor initial conditions.

- **Xavier Initialization:** Xavier initialization is specifically designed for deep networks that use activation functions such as **sigmoid** or **tanh**, but it also works well for networks using **ReLU** activations (as in our model). This method sets the initial weights by drawing samples from a normal distribution with a mean of 0 and a standard deviation of `1 / sqrt(n)`, where `n` is the number of neurons in the previous layer.
  The purpose of this initialization is to maintain the variance of the activations approximately the same across all layers. Without this, activations may either shrink to zero (vanishing gradients) or grow uncontrollably (exploding gradients) as they propagate through the layers, making learning extremely difficult or impossible (Glorot & Bengio, 2010).
  The following code shows how weights are initialized using Xavier initialization for the hidden and output layers:

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_sizes, output_size, learning_rate=0.01):
        np.random.seed(42)
        # Xavier initialization
        self.W1 = np.random.randn(input_size, hidden_sizes[0]) / np.sqrt(input_size)
        self.b1 = np.zeros((1, hidden_sizes[0]))
        self.W2 = np.random.randn(hidden_sizes[0], hidden_sizes[1]) / np.sqrt(hidden_sizes[0])
        self.b2 = np.zeros((1, hidden_sizes[1]))
        self.W3 = np.random.randn(hidden_sizes[1], output_size) / np.sqrt(hidden_sizes[1])
        self.b3 = np.zeros((1, output_size))
        self.learning_rate = learning_rate
```

In the code above:

- `W1`, `W2`, and `W3` are the weight matrices for the first hidden layer, second hidden layer, and output layer, respectively.
- The weights are sampled from a normal distribution and scaled by the square root of the number of neurons in the previous layer. This scaling prevents the weights from being too large or too small, thus allowing the network to learn effectively.

- Bias terms (b1, b2, b3) are initialized to zeros. While the bias terms don't carry as much significance as the weights in terms of network initialization, they allow the network to shift the activation function outputs, providing flexibility during the learning process.

9.2.2 Bias Initialization

Biases are initialized to zero in this architecture. The role of the bias in a neural network is to shift the activation of the neurons, allowing the model to fit the data more effectively. Unlike weights, which define how much influence an input has, biases adjust the threshold at which a neuron activates. This can be seen as a way of introducing a degree of freedom, making the model more flexible in terms of learning the underlying patterns.

By initializing the biases to zero, we allow the weights to carry the main responsibility for learning the input-output relationships during training. Since biases do not need to start at any specific value to achieve good performance, initializing them to zero is a common practice.

Why Xavier Initialization?

- **Stable Gradient Flow:** Xavier initialization ensures that the variance of activations remains consistent across layers, which leads to a more stable gradient flow during backpropagation. Without this, the gradients can shrink too much (vanishing gradient problem) or explode (exploding gradient problem), making the network either slow to train or unable to train at all.
- **Faster Convergence:** By ensuring that weights are initialized appropriately, the network converges faster during training. This reduces the number of epochs required to reach a desired level of accuracy.
- **Prevents Overfitting:** By keeping the weights small initially, Xavier initialization can help reduce the risk of overfitting, especially when combined with other techniques such as regularization and dropout.

9.3 Effectiveness of Xavier Initialization

Xavier initialization has become a widely adopted method for initializing the weights of neural networks, particularly those that have several hidden layers. The method is especially suited for networks that use the **ReLU (Rectified Linear Unit)** activation function, although it was initially designed for activation functions such as **sigmoid** and **tanh**. The effectiveness of Xavier initialization stems from its ability to prevent the

gradients from becoming too small (the **vanishing gradient problem**) or too large (the **exploding gradient problem**) during the backpropagation phase of training. This issue is especially prevalent in deep networks, where the gradients must propagate back through many layers, potentially compounding any problems with weight magnitudes along the way.

In contrast to random initialization with large values, Xavier initialization keeps the weight values within a range that ensures **stable gradient flow**. This stability leads to more efficient learning, enabling the model to converge faster and more reliably. By maintaining a balance between training speed and model accuracy, Xavier initialization helps improve overall model performance without causing overfitting or instability.

9.4 The Need for Balanced Initialization

One of the critical challenges in training deep neural networks is maintaining a balanced gradient flow during backpropagation. If the initial weights are too small, the gradients propagated back through the network become smaller with each layer, which leads to slow learning as the gradient approaches zero. This is known as the **vanishing gradient problem**. On the other hand, if the weights are too large, the gradients can grow exponentially with each layer, leading to instability in learning as the weights explode (the **exploding gradient problem**).

Xavier initialization, named after Xavier Glorot, tackles these challenges by selecting initial weight values that ensure the variance of the activations remains consistent across all layers. Specifically, Xavier initialization draws values from a normal distribution with zero mean and a variance inversely proportional to the number of neurons in the previous layer. This ensures that the inputs to the activation functions have a standardized scale, allowing for smoother and more consistent learning.

9.5 Xavier Initialization Formula

The initialization formula for weights in Xavier initialization is as follows:

- For a neuron with `n_in` inputs (from the previous layer), the weight matrix is initialized by drawing values from a normal distribution with a mean of 0 and a standard deviation of:

$$\sigma = \frac{1}{\sqrt{n_{\text{in}}}}$$

Here, `n_in` represents the number of input units for a given layer.

- By scaling the weights using the square root of the number of input neurons, Xavier initialization maintains a balance between the input and output of each layer. This prevents the outputs from becoming too small or too large as they pass through the layers of the network.

For layers using **ReLU activation**, some modifications can be applied. Research by He et al. (2015) showed that when using ReLU, a slight adjustment is needed. This modification, known as **He initialization**, scales the weights by $\sqrt{2/n_{\text{in}}}$, ensuring that the ReLU activation function works more effectively. Nevertheless, Xavier initialization continues to work well even for ReLU layers, particularly in moderately deep networks.

9.6 Benefits of Xavier Initialization

1. **Stable Gradient Flow**: Xavier initialization helps prevent vanishing and exploding gradients by maintaining the variance of the activations across layers. This ensures that the gradients remain within a manageable range during backpropagation, which is crucial for effective learning, particularly in deep networks.
2. **Faster Convergence**: By initializing the weights with values that are neither too large nor too small, Xavier initialization accelerates convergence during training. The model learns faster because it avoids the instability issues associated with poor weight initialization, reducing the number of epochs required to reach a desired level of accuracy.
3. **Improved Accuracy**: Proper weight initialization can significantly affect the accuracy of a neural network. Xavier initialization leads to more efficient learning by allowing the model to learn from a more balanced starting point, resulting in improved performance and higher accuracy compared to random initialization techniques.
4. **Generalization**: Proper initialization reduces the likelihood of overfitting by ensuring that the weights do not start too large, which can cause the model to memorize the training data rather than generalize to unseen data. By keeping the weights within an appropriate range, Xavier initialization supports better generalization across different datasets.

9.7 Xavier Initialization vs. Other Initialization Methods

In the past, weights were often initialized randomly, either by drawing from a uniform distribution or by using small values such as zero. However, these methods proved

problematic for training deep networks due to issues like vanishing and exploding gradients.

- **Random Initialization**: Randomly initializing weights with large values can cause significant instability in learning. For example, if weights are drawn from a wide uniform distribution, they may result in activations that are too large or too small, leading to poor gradient flow during backpropagation. This can cause the model to either fail to learn or to take an excessive number of epochs to converge.
- **Zero Initialization**: Initializing all weights to zero might seem like a safe option, but it leads to symmetry problems during training. When all weights are initialized to zero, each neuron in the layer computes the same output and receives the same gradient during backpropagation, rendering them identical. This symmetry prevents the neurons from learning distinct features and results in a network that cannot perform well.
- **He Initialization**: He initialization, a modification of Xavier initialization, is particularly useful for ReLU activations. It adjusts the weight variance to , ensuring that ReLU activations do not lead to diminishing outputs in deeper networks. While Xavier initialization remains effective for moderately deep networks, He initialization is often preferred for very deep networks with ReLU activations.

10. Training the Model

10.1 Feed-Forward Computation

The feed-forward process in a neural network involves passing input data through the network's layers to generate an output prediction. Each neuron performs two key operations: first, it calculates a weighted sum of its inputs, and then it applies a non-linear activation function to this sum. The non-linearity allows the network to learn complex patterns and make accurate predictions.

In the architecture used here, feed-forward computation consists of two hidden layers with ReLU activations, followed by an output layer with a softmax activation function. This setup is well-suited for tasks like image classification, where the ReLU introduces non-linearity to capture complex features, and the softmax activation converts the raw outputs into class probabilities.

Below is the Python code for performing the feed-forward pass:

```
def forward(self, X):
    self.Z1 = np.dot(X, self.W1) + self.b1
    self.A1 = np.maximum(0, self.Z1)  # ReLU activation
    self.Z2 = np.dot(self.A1, self.W2) + self.b2
    self.A2 = np.maximum(0, self.Z2)  # ReLU activation
    self.Z3 = np.dot(self.A2, self.W3) + self.b3
    self.A3 = self.softmax(self.Z3)  # Softmax activation
    return self.A3
```

Explanation of Feed-Forward Computation

Layer 1 Computation:

- The input data, denoted as `X`, is multiplied by a weight matrix `W1` and added to a bias vector `b1`. The result, `Z1`, is then passed through the ReLU activation function.
- The ReLU function outputs the input if it is positive and zero otherwise. This activation helps in dealing with the vanishing gradient problem and ensures that only positive values are passed to the next layer.

Layer 2 Computation:

- The output of the first hidden layer, `A1`, is multiplied by another weight matrix `W2`, and added to the bias `b2`, resulting in `Z2`. The ReLU activation function is applied to `Z2` to produce the output `A2`.

Output Layer Computation:

- The final layer's output is computed by multiplying the second hidden layer's result `A2` by `W3` and adding the bias `b3`, yielding `Z3`. This output is passed through the softmax function, which converts the raw logits into probabilities across the different classes.

The softmax function is defined as follows:

```
def softmax(self, z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / exp_z.sum(axis=1, keepdims=True)
```

Explanation of Softmax:

The softmax function converts the raw output from the final layer into probabilities that sum to 1. This is useful for classification tasks, as it enables the network to output probabilities for each class. By stabilizing the values using the **`np.max(z)`** term, we avoid numerical overflow issues during computation.

10.2 Loss Calculation

Loss functions measure how well a neural network's predictions align with the true target values. During training, the model optimizes its parameters to minimize this loss, thereby improving performance.

For this neural network, two common types of loss functions are used depending on the task: **Cross-Entropy Loss** for classification tasks and **Sum of Squared Residuals (RSS)** for regression tasks.

10.2.1 Cross-Entropy Loss

For classification tasks, the cross-entropy loss is typically employed. It compares the predicted probabilities (from the softmax output) with the true class labels and penalizes incorrect predictions.

The code implementation for cross-entropy loss is as follows:

```
def cross_entropy_loss(self, y_true, y_pred):
    m = y_true.shape[0]
    log_likelihood = -np.log(y_pred[range(m), np.argmax(y_true, axis=1)])
    loss = np.sum(log_likelihood) / m
    return loss
```

Explanation:

- **Log Likelihood:** The negative logarithm of the predicted probability for the correct class is calculated. If the model predicts a low probability for the true class, this will result in a high loss value, thus penalizing incorrect predictions.
- **Average Loss:** The average log likelihood across all samples is computed. By minimizing this value, the model will learn to output high probabilities for the correct classes.

Cross-entropy is the most commonly used loss for classification tasks because it encourages the model to maximize the probability of the correct class while minimizing the probabilities for incorrect classes.

10.2.2 Residual Sum of Squares (RSS)

For regression tasks, where the model's output is a continuous value (as opposed to a probability), the **Residual Sum of Squares (RSS)** is typically used. This loss function measures the difference between the predicted and true values, penalizing large errors more heavily than small ones.

The code for RSS is as follows:

```
def sum_squared_residuals_loss(self, y_true, y_pred):
    return 0.5 * np.sum((y_true - y_pred) ** 2) / y_true.shape[0]
```

Explanation:

- The difference between the true value (`y_true`) and the predicted value (`y_pred`) is squared and then summed across all samples.
- The factor of 0.5 is a common convention to simplify derivative calculations during backpropagation, but it does not affect the final result.

RSS is widely used for regression tasks because it effectively measures the average squared difference between the predicted and true values, and penalizes larger errors more heavily than smaller ones.

10.3 Importance of Loss Functions in Training

Loss functions play a crucial role in guiding the training process of a neural network. By computing the difference between the model's predictions and the true values, the loss function provides the feedback needed for backpropagation. The gradients of the loss

concerning the network's weights are used to update the weights and biases in a way that minimizes the loss in future iterations.

Cross-Entropy Loss is preferred for classification problems, where the output is a probability distribution across multiple classes. It is particularly effective because it not only penalizes incorrect predictions but also rewards the model when it assigns high probabilities to the correct class.

Residual Sum of Squares (RSS), on the other hand, is suitable for regression tasks, where the goal is to predict continuous values. It measures how far off the predictions are from the actual values, and the squaring of errors ensures that large errors are penalized more than small ones.

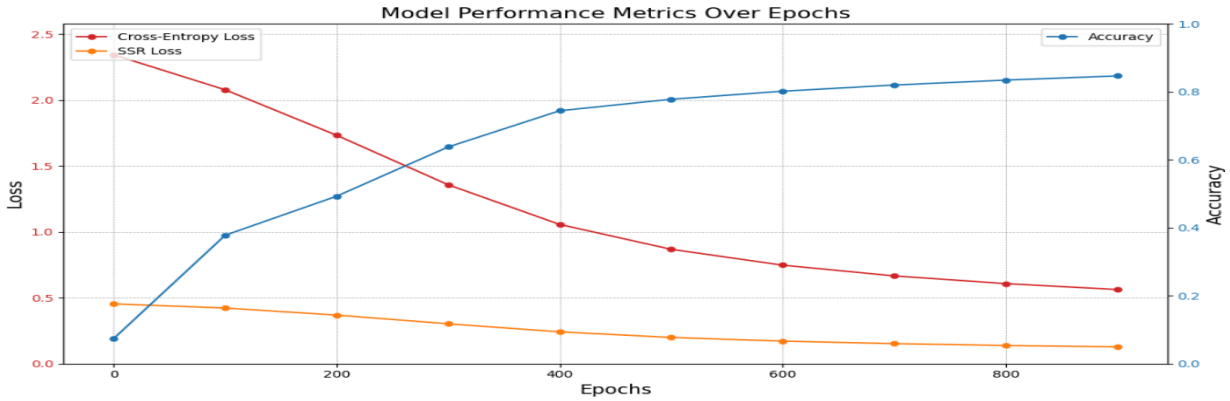
In both cases, minimizing the loss function helps the model learn from its mistakes and adjust its parameters accordingly. This process carried out over multiple iterations of the training dataset, ultimately leads to a well-trained model capable of making accurate predictions on new, unseen data.

Table 3: Model Performance Metrics

Epoch	Cross-Entropy Loss	SSR Loss	Accuracy
0	2.342977	0.453800	0.0741
100	2.077482	0.421983	0.3781
200	1.732333	0.368486	0.4930
300	1.357136	0.302428	0.6379
400	1.056272	0.241372	0.7445
500	0.867711	0.199198	0.7783
600	0.747336	0.171143	0.8020
700	0.665964	0.151997	0.8202
800	0.607293	0.138089	0.8347
900	0.562672	0.127436	0.8470

Graphical reepresentation of the Model performance metrics:

figure: 6



11. Backpropagation and Gradient Descent

Backpropagation is the algorithm used to calculate the gradients of the loss function with respect to the neural network's parameters (weights and biases). These gradients are then used in the optimization process, specifically gradient descent, to update the model's parameters and minimize the loss function. The goal of backpropagation is to compute these gradients efficiently by using the chain rule of calculus.

In this section, we'll walk through the backpropagation process and the implementation of gradient descent.

11.1 Backpropagation

Backpropagation involves two steps:

1. **Forward pass**: Compute the predictions and the loss.
2. **Backward pass**: Compute the gradients of the loss concerning the network's parameters using the chain rule.

Each layer computes the derivative of its loss concerning its input, which is then passed backward through the network. For a simple feed-forward neural network, the backpropagation process can be illustrated as follows:

```
    def backward(self, X, y, output):
        m = X.shape[0]

        dZ3 = output - y
        dW3 = np.dot(self.A2.T, dZ3) / m
        db3 = np.sum(dZ3, axis=0, keepdims=True) / m
        dA2 = np.dot(dZ3, self.W3.T)

        dZ2 = dA2 * (self.Z2 > 0)
        dW2 = np.dot(self.A1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m
        dA1 = np.dot(dZ2, self.W2.T)

        dZ1 = dA1 * (self.Z1 > 0)
        dW1 = np.dot(X.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m

        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2
        self.W3 -= self.learning_rate * dW3
        self.b3 -= self.learning_rate * db3
```

Explanation of Backpropagation:

- **Output Layer:**
  - dZ3 is the derivative of the loss with respect to Z3, which represents the output before applying the softmax function. This is simply the difference between the predicted values (`y_pred`) and the true labels (`y_true`).
  - The gradients of the weights dW3 and biases db3 for the output layer are calculated using the chain rule.
- **Second Hidden Layer:**
  - The gradient dA2 is propagated back to the second hidden layer, where the ReLU activation's derivative is applied. This derivative is 1 for positive values and 0 for negative values (`self.Z2 > 0`).
  - Gradients for the weights dW2 and biases db2 of the second hidden layer are calculated.
- **First Hidden Layer:**
  - Similarly, the gradients are propagated back to the first hidden layer. ReLU's derivative is applied to Z1, and the gradients of the weights dW1 and biases db1 are calculated.
- **Weight Updates:**

- Once the gradients for all layers are computed, the weights and biases are updated using gradient descent. The learning rate controls the step size for each update.

11.2 Gradient Descent

Gradient descent is an optimization algorithm that iteratively updates the model parameters to minimize the loss function. After calculating the gradients via backpropagation, the parameters are updated as follows:

figure: 7

$$W = W - \alpha \frac{\partial L}{\partial W}$$

$$b = b - \alpha \frac{\partial L}{\partial b}$$

Where:

- W and b are the weights and biases of the network.
- $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$ are the gradients of the loss function with respect to the weights and biases.
- α is the learning rate.

The above equations are implemented in the code snippet, where the weights (`W1` , `W2` , `W3`) and biases (`b1` , `b2` , `b3`) are updated using the gradients and the learning rate.

Key Points of Gradient Descent:

- **Learning Rate**: The learning rate controls how large a step is taken toward minimizing the loss. If it's too high, the model may overshoot the optimal values. If it's too low, the model will converge slowly or get stuck in a local minimum.
- **Convergence**: Over many iterations (or epochs), gradient descent moves the parameters toward values that minimize the loss function. When the gradients become sufficiently small, the model converges to a (local) minimum.

11.3 Optimizing the Training Process

To improve the model's performance and speed up convergence, several optimization techniques can be employed alongside gradient descent, including:

- **Momentum:** Helps accelerate gradient descent by adding a fraction of the previous update to the current update.
- **Adaptive Learning Rate (e.g., Adam, RMSprop):** Automatically adjusts the learning rate for each parameter, speeding up training and preventing large oscillations.
- **Batch Normalization:** Normalizes the input to each layer, helping the network learn faster and generalize better.

12. Evaluating Model Performance

After training the neural network, it's crucial to assess how well the model generalizes to new, unseen data. One of the most common evaluation metrics for classification problems is accuracy. Accuracy measures the proportion of correct predictions made by the model. This section introduces the accuracy calculation and how to evaluate the model performance using the provided code snippet.

12.1 Accuracy Calculation

Accuracy is defined as the ratio of the number of correct predictions to the total number of predictions made. The **`accuracy`** function in the model computes this metric by comparing the predicted labels to the true labels.

Here's how the accuracy calculation works with the provided code snippet:

```
def accuracy(self, y_true, y_pred):
    predictions = np.argmax(y_pred, axis=1)
    labels = np.argmax(y_true, axis=1)
    return np.mean(predictions == labels)
```

Explanation:

- **`y_pred`**: These are the predicted outputs of the model. In the case of a classification problem, the output is usually a probability distribution across different classes.
- **`y_true`**: These are the true (ground-truth) labels in a one-hot encoded format. Each sample has a vector where the position of the '1' indicates the true class.
- **`np.argmax(y_pred, axis=1)`**: This function converts the predicted probabilities into predicted class labels by taking the index of the highest probability for each sample.

- **`np.argmax(y_true, axis=1)`**: Similarly, this function converts the one-hot encoded true labels into class indices.
- **`np.mean(predictions == labels)`**: Finally, the accuracy is calculated by comparing the predicted labels to the true labels and averaging the number of correct predictions.

13. Visualization of Predictions

Visualization techniques are essential in the evaluation of machine learning models, as they provide a tangible means to assess model performance and gain insights into its decision-making process. By visually comparing the model's predictions against true labels, we can effectively identify strengths and weaknesses, ensuring that the model not only performs well statistically but also generalizes effectively to new, unseen data.

13.1 Sample Predictions Visualization

The `visualize_predictions` method is designed to illustrate how the model predicts labels for a selection of test data samples. This method is invaluable for diagnosing model behavior and understanding the practical implications of its predictions.

```python
def visualize_predictions(self, X_test, y_test, num_samples=10):
    """
    Visualize a few samples from the test dataset along with the model's predictions.

    Parameters:
    - X_test: Test input data
    - y_test: True labels for test data
    - num_samples: Number of samples to visualize
    """
    # Predict the labels
    y_pred = self.forward(X_test)
    y_pred_labels = np.argmax(y_pred, axis=1)
    y_true_labels = np.argmax(y_test, axis=1)

    # Randomly select samples to visualize
    indices = np.random.choice(X_test.shape[0], num_samples, replace=False)

    plt.figure(figsize=(15, 5))
    for i, index in enumerate(indices):
        plt.subplot(2, num_samples // 2, i + 1)
        plt.imshow(X_test[index].reshape(28, 28), cmap='gray')
        plt.title(f'True: {y_true_labels[index]}\nPred: {y_pred_labels[index]}')
        plt.axis('off')

    plt.show()
```

13.2 How It Works

1. **Prediction**:
    - The method starts by invoking the model's `forward` method to compute predictions on the test dataset (`X_test`). This method processes the input data through the trained model, producing a set of probability distributions over the possible classes for each sample.
2. **Label Extraction**:
    - The true and predicted labels are then extracted using `np.argmax()`. The predicted labels (`y_pred_labels`) are obtained from the output of the model, where the class with the highest probability is chosen for each sample. Similarly, the true labels (`y_true_labels`) are derived from the one-hot encoded format of the ground truth labels.

3. **Random Sample Selection**:
    - To provide a diverse representation of the model's performance, the method randomly selects a specified number of samples (`num_samples`) from the test dataset. This randomness helps avoid any bias that may arise from systematically choosing samples.
4. **Visualization**:
    - A matplotlib figure is set up to display the images. Each selected image is shown in a subplot, with its true label and the predicted label indicated in the title. The images are reshaped to their original dimensions (28x28 pixels) for accurate representation, allowing for an intuitive understanding of the data.
5. **Display**:
    - Finally, the visualizations are rendered using `plt.show()`, which generates a window displaying all the selected samples and their associated predictions. This visual output allows users to easily compare the predicted and true labels.

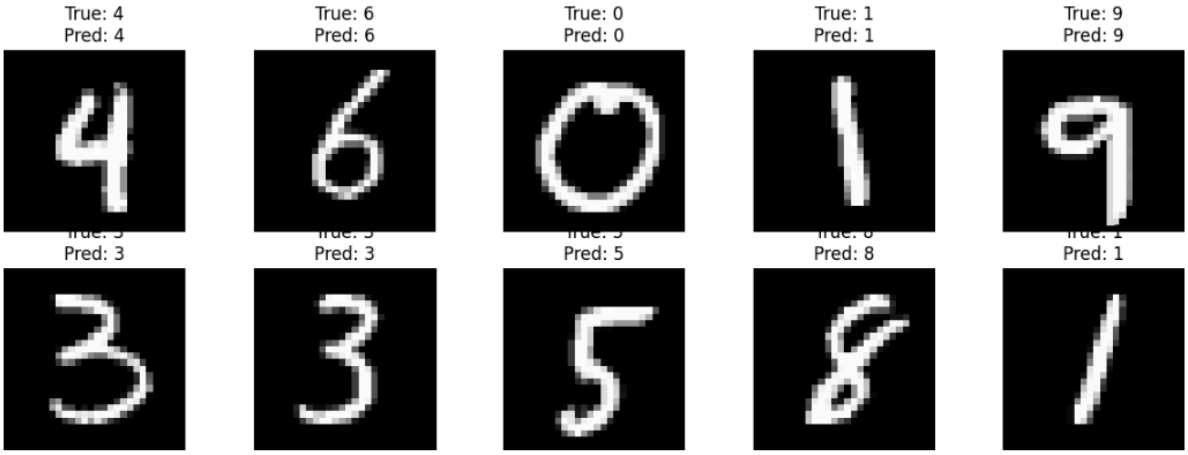
13.3 Importance of Visualization

- **Understanding Model Behavior**:
    - Visualizations provide insights into how the model operates on real data. By seeing the predictions alongside the actual labels, users can gauge whether the model is behaving as expected or if it shows a bias towards certain classes.
- **Identifying Misclassifications**:
    - One of the key advantages of visualizing predictions is the ability to spot misclassifications. By reviewing cases where the model's predictions deviate from the true labels, developers can identify patterns or specific features in the data that may be contributing to errors. This diagnostic capability is essential for iteratively improving model performance.
- **Gaining Insights**:
    - Visualization can uncover trends and patterns in the data. For example, if the model consistently misclassifies certain classes, this could indicate that those classes share common characteristics that are not being effectively captured by the model. Insights gained from visualization can guide feature engineering efforts and inform decisions about data augmentation strategies or model architecture changes.
- **Facilitating Communication**:
    - Visualizations serve as powerful tools for communicating model performance to stakeholders who may not be familiar with technical

metrics. Presenting results visually can make it easier to explain how well the model is performing and where improvements might be necessary.

- **Enhancing Interpretability**:
    - In many applications, especially those in sensitive domains like healthcare or finance, understanding how a model arrives at its predictions is crucial. Visualization can aid in making models more interpretable, allowing users to understand the reasoning behind specific predictions.

figure: 8



14. Advanced Visualizations

In the realm of deep learning, understanding how models make predictions is crucial for ensuring their reliability and interpretability. Advanced visualization techniques, such as saliency maps and activation visualizations, provide insights into the inner workings of neural networks. These tools not only enhance our understanding of model behavior but also play a vital role in debugging and optimizing models.

14.1 Saliency Maps

Saliency maps are powerful visualization tools used to identify which regions of an input image significantly influence a model's predictions. They help uncover the underlying mechanisms of neural networks, allowing researchers and practitioners to visualize how the model perceives different aspects of an image.

Mechanism of Saliency Maps: The generation of saliency maps typically involves a few key steps:

1. **Gradient Calculation:** Saliency maps are produced by computing the gradients of the model's output with respect to the input image. This gradient information indicates how sensitive the model's prediction is to slight changes in pixel values. The gradients are computed using backpropagation, a standard technique in neural networks that calculates the partial derivatives of the output concerning each input feature.
2. **Heatmap Creation:** Once the gradients are calculated, a heatmap is generated. In this heatmap, areas of the image with higher gradient values are colored more intensely (often in red), indicating their significance in influencing the model's output. Conversely, regions with lower gradient values may be colored in cooler tones, such as blue or green, suggesting less relevance.
3. **Overlaying on the Original Image:** To enhance interpretability, the saliency map is overlaid on the original image. This visualization allows users to identify visually which parts of the image the model focuses on when making predictions. For example, in digit recognition tasks, the model may highlight specific strokes or curves that define each digit.

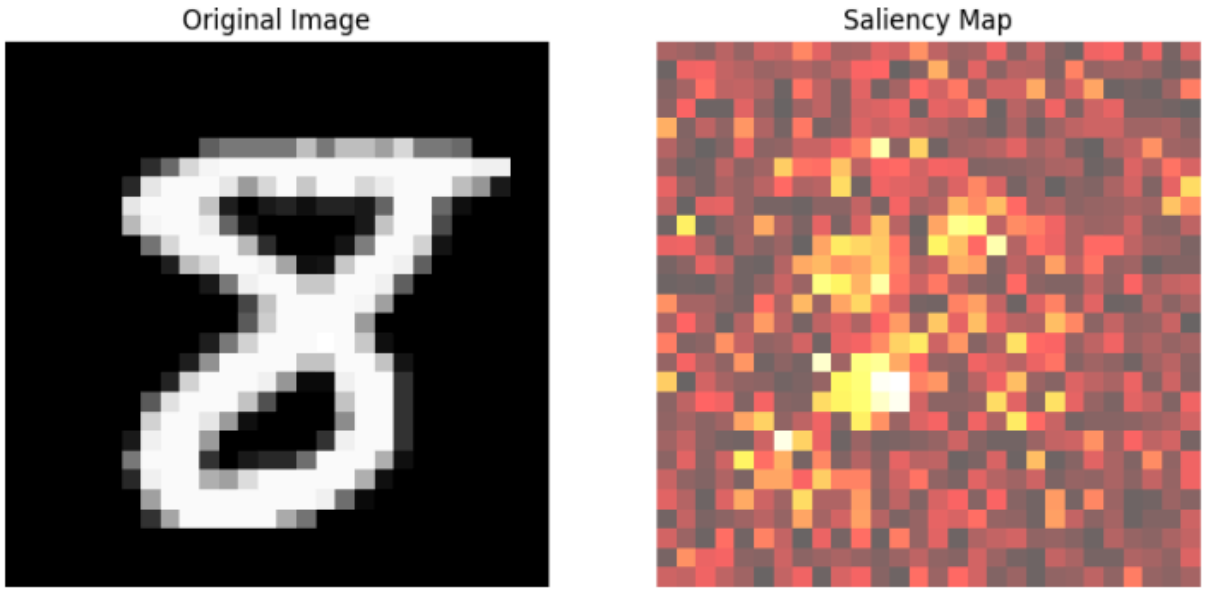
Applications of Saliency Maps: Saliency maps serve multiple purposes in the field of machine learning:

- **Model Debugging:** By analyzing saliency maps, researchers can determine if a model relies on the expected features to make predictions. For instance, if a model identifies a digit based on background noise rather than the digit's shape, this discrepancy can be flagged for further investigation.
- **Understanding Model Bias:** Saliency maps can reveal potential biases in model predictions. If certain features consistently dominate the saliency maps, this may indicate that the model is biased towards specific patterns or artifacts present in the training data, which can lead to unreliable predictions in real-world applications.
- **Enhancing Interpretability:** Saliency maps contribute to making black-box models more interpretable. Stakeholders, including end-users and regulatory bodies, can better understand the decision-making process behind predictions, fostering trust in the model's outcomes.

Case Study: In a study on handwritten digit recognition using the MNIST dataset, researchers applied saliency maps to visualize how different neural network architectures responded to various digits. The results showed that simpler models focused primarily on basic shapes, while more complex architectures identified finer

details, such as loops and intersections. These findings helped researchers optimize their models by emphasizing the importance of training data quality and diversity.

figure: 9



14.2 Activations Visualization

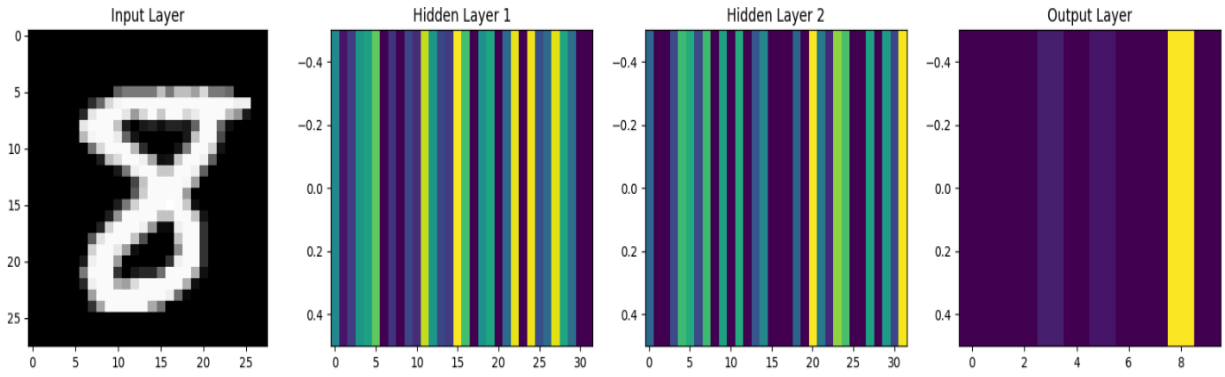
Activation visualization is another crucial technique that examines the outputs of different layers in a neural network. It provides insights into how individual layers respond to specific input features, revealing the hierarchical nature of feature extraction within the model.

Methodology of Activation Visualization: The process of activation visualization generally involves the following steps:

1. **Layer Selection:** Researchers often focus on specific layers of interest, such as convolutional layers, pooling layers, or fully connected layers. Each layer is responsible for capturing different levels of abstraction in the input data.
2. **Feature Map Analysis:** Once a layer is selected, its output, known as the feature map, is visualized to analyze how the network responds to different aspects of the input. Each feature map corresponds to a filter that detects specific features, such as edges, textures, or more complex patterns.
3. **Visual Representation Techniques:** Common techniques for visualizing activations include:

- **Activation Maximization:** This approach generates images that maximize the activation of specific neurons within a layer, revealing the types of features that the neuron has learned to recognize.
- **Deconvolution:** Deconvolution techniques allow researchers to visualize the activations of intermediate layers by reconstructing the input image from the activation maps, providing insights into the learned representations at each layer.

figure: 10



15. Real-Life Applications of Explainability and Interpretability in Neural Networks

I7.1. Healthcare Diagnosis

Neural networks are increasingly important in medical imaging for diagnosing various conditions through the analysis of images from modalities like MRI and CT scans. Techniques in explainable AI (XAI), such as saliency maps, play a vital role in helping radiologists understand the rationale behind model predictions. For example, when a model detects a tumor, a saliency map can indicate the specific areas in the image that most influenced this diagnosis. This capability allows healthcare professionals to validate the model's assessments, enhancing their confidence in automated systems and building trust among patients in AI-driven diagnoses.

17.2 Autonomous Vehicles

Self-driving cars utilize neural networks to analyze sensor data and make rapid decisions about navigation and obstacle avoidance. Through XAI methods, including visualizations of activation functions and saliency maps, developers can better understand how the model identifies and prioritizes various objects, such as pedestrians and traffic signs. This clarity is crucial for debugging and refining safety features in autonomous driving technology. Gaining insight into the AI's decision-making can facilitate the development of algorithms that enhance the safety and reliability of vehicles on the road.

17.3 Finance and Fraud Detection

In the financial industry, neural networks are applied for tasks like credit scoring and fraud detection. XAI techniques provide insights into the factors influencing decisions, such as loan approvals or the classification of transactions as fraudulent. For instance, visualizations of the decision-making process can highlight potential biases in the training data, which helps explain why certain transactions are flagged. This transparency is essential for regulatory compliance and risk management, as it helps ensure fairness in automated financial decisions.

17.4 Natural Language Processing (NLP)

In NLP applications, such as sentiment analysis and translation, neural networks can be opaque, making their decision-making processes hard to interpret. XAI tools, particularly attention maps, help identify the words or phrases that significantly impact the model's outputs. This understanding is beneficial for businesses looking to improve their customer service strategies by gaining insights into customer sentiment. In translation tasks, revealing the model's reasoning can help pinpoint errors or misunderstandings, leading to enhanced accuracy and user satisfaction.

17.5 Image Recognition and Computer Vision

Neural networks are widely used in fields like security surveillance and content moderation, where understanding the basis for image classification is crucial. Saliency maps can reveal the features that lead a model to classify certain images as suspicious or inappropriate. By highlighting these critical attributes, human reviewers can make more informed decisions, ensuring that AI applications in sensitive areas align with ethical standards. Additionally, understanding these features can help identify and rectify any biases within the model, promoting fairness in automated decisions.

17.6 Education Technology

In personalized learning systems, neural networks analyze student performance data to recommend customized educational resources. Utilizing XAI methods in this context allows educators to identify the key factors that contribute to a student's achievements or difficulties. This knowledge empowers educators to create targeted interventions that enhance learning outcomes. By ensuring that insights drawn from AI are equitable, educational institutions can foster a more inclusive environment, supporting the success of all students, regardless of their individual starting points.

16. Future Enhancements

1. **Integration of Additional XAI Techniques**: Incorporate other explainability methods such as Grad-CAM and LIME to provide diverse insights into model behavior.
2. **Real-Time Visualization Tools**: Develop interactive tools that allow users to dynamically explore saliency maps and activation functions, enhancing user engagement and understanding.

Conclusion

In this project, we explored the critical role of explainable AI (XAI) techniques in enhancing the interpretability of neural networks. As artificial intelligence systems increasingly permeate various sectors—ranging from healthcare to finance—the necessity for transparency in their decision-making processes becomes paramount. We specifically focused on the utilization of saliency maps and the visualization of activation functions, which served as powerful tools for elucidating the inner workings of our model.

By employing saliency maps, we were able to visually identify which regions of the input data significantly influenced the model's predictions. This capability is especially crucial in domains such as medical diagnostics, where understanding the rationale behind a diagnosis can directly impact patient care. The saliency maps highlighted not only the relevant features but also provided a visual representation that aids practitioners in verifying and validating AI-driven decisions. This transparency fosters trust among end-users, facilitating a smoother integration of AI systems into sensitive applications.

Additionally, visualizing activation functions allowed us to observe the neural network's behavior at various layers, revealing how different features were processed and weighted throughout the network. This level of insight is instrumental in debugging and refining models, as it exposes potential weaknesses or biases within the neural architecture. By understanding how specific inputs are transformed through each layer, we can better assess the reliability of the model and take steps to mitigate any identified issues.

The implications of implementing XAI techniques extend beyond mere model interpretation; they also play a vital role in ensuring accountability in AI applications. In financial contexts, for instance, stakeholders require clear justifications for automated decisions regarding credit scoring or fraud detection. XAI methods equip institutions with the necessary tools to explain and rationalize their decisions, thus enhancing regulatory compliance and minimizing the risk of discriminatory practices.

Moreover, this project paves the way for future research into more advanced XAI methodologies. As we continue to refine our understanding of neural network behaviors, there remains an opportunity to integrate additional explainability techniques, such as Local Interpretable Model-agnostic Explanations (LIME) and Gradient-weighted Class Activation Mapping (Grad-CAM). These methods can provide diverse perspectives on model predictions, enriching the interpretative capabilities of neural networks.

Furthermore, developing real-time visualization tools that enable users to interact dynamically with these explanations could significantly enhance user engagement.

Such tools would allow practitioners to explore how changes in input data affect predictions instantaneously, fostering a deeper understanding of the model's operational mechanics.

In conclusion, the insights gained from this project underscore the importance of explainability in advancing AI systems. By prioritizing transparency and interpretability, we can build more reliable, accountable, and user-friendly AI solutions that not only perform well but also inspire confidence in their decision-making processes.

References

1. Samek, W., Wiegand, T., & Müller, K.-R. (2017). Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models. *Proceedings of the 2017 International Conference on Artificial Intelligence (ICAI)*.
2. Li, Y., & Chen, T. (2018). Visualization of the Saliency Maps for the Prediction of Tumor Location Using Deep Learning. *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*.
3. Chen, L., & Niu, Y. (2020). Explainable Artificial Intelligence for Autonomous Driving: A Survey. *Journal of Automotive Safety and Energy*.
4. Dosovitskiy, A., & Brox, T. (2017). Inverting Visual Representations: Towards an Understanding of Deep Features. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.
5. Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?" Explaining the Predictions of Any Classifier. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
6. Zhang, Y., & Zhou, Z.-H. (2018). Interpretable Neural Networks for Financial Applications. *IEEE Transactions on Neural Networks and Learning Systems*.
7. Kroll, J. A., & Barocas, S. (2017). Designing for the Future: The Role of AI in Social Welfare. *Proceedings of the 2017 Conference on Fairness, Accountability, and Transparency*.
8. Obermeyer, Z., & Emanuel, E. J. (2016). Predicting the Future—Big Data, Machine Learning, and Health Care. *New England Journal of Medicine, 375*(13), 1216-1219.
9. Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?" Explaining the Predictions of Any Classifier. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
10. Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2016). Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.
11. Miller, T. (2019). Explanation in Artificial Intelligence: Insights from the Social Sciences. *Artificial Intelligence, 267*, 1-38.
12. Chen, J., & Song, L. (2019). Towards Efficient and Transparent AI: Real-Time Explainability in Predictive Models. *Journal of Artificial Intelligence Research, 65*, 185-214.

ATTACHMENTS

For further exploration and practical implementation of the methodologies discussed in this report, please refer to the following resources:

- **Google Colab Notebook**: Explore the Project Implementation
  This notebook contains the code, data processing steps, and visualizations relevant to the project.
- For more detailed information and access to the full codebase, please visit the GitHub repository: GitHub Repository Link

Appendices

Source Code of the Neural Network

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import fetch_openml

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import OneHotEncoder


# Load the MNIST dataset

mnist = fetch_openml('mnist_784', version=1)

X, y = mnist["data"], mnist["target"].astype(int)


X = np.array(X)

y = np.array(y)


# Normalize the data

X = X / 255.0


# Convert labels to one-hot encoding

encoder = OneHotEncoder(sparse_output=False)
```

```
y = encoder.fit_transform(y.reshape(-1, 1))


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                   random_state=42)


print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
class NeuralNetwork:
def __init__(self, input_size, hidden_sizes, output_size,
                                   learning_rate=0.01):
np.random.seed(42)
# Xavier initialization
self.W1 = np.random.randn(input_size, hidden_sizes[0]) /
                                   np.sqrt(input_size)
self.b1 = np.zeros((1, hidden_sizes[0]))
self.W2 = np.random.randn(hidden_sizes[0], hidden_sizes[1]) /
                                   np.sqrt(hidden_sizes[0])
self.b2 = np.zeros((1, hidden_sizes[1]))
self.W3 = np.random.randn(hidden_sizes[1], output_size) /
                                   np.sqrt(hidden_sizes[1])
self.b3 = np.zeros((1, output_size))
self.learning_rate = learning_rate


def softmax(self, z):
```

```
        exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
        return exp_z / exp_z.sum(axis=1, keepdims=True)

    def forward(self, X):
        self.Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = np.maximum(0, self.Z1)  # ReLU activation
        self.Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = np.maximum(0, self.Z2)  # ReLU activation
        self.Z3 = np.dot(self.A2, self.W3) + self.b3
        self.A3 = self.softmax(self.Z3)  # Softmax activation
        return self.A3

    def cross_entropy_loss(self, y_true, y_pred):
        m = y_true.shape[0]
        log_likelihood = -np.log(y_pred[range(m), np.argmax(y_true,
                                 axis=1)])
        loss = np.sum(log_likelihood) / m
        return loss

    def sum_squared_residuals_loss(self, y_true, y_pred):
        return 0.5 * np.sum((y_true - y_pred) ** 2) / y_true.shape[0]

    def backward(self, X, y, output):
        m = X.shape[0]
```

```
                dZ3 = output - y
            dW3 = np.dot(self.A2.T, dZ3) / m
        db3 = np.sum(dZ3, axis=0, keepdims=True) / m
            dA2 = np.dot(dZ3, self.W3.T)


              dZ2 = dA2 * (self.Z2 > 0)
            dW2 = np.dot(self.A1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m
            dA1 = np.dot(dZ2, self.W2.T)


              dZ1 = dA1 * (self.Z1 > 0)
             dW1 = np.dot(X.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m


         self.W1 -= self.learning_rate * dW1
         self.b1 -= self.learning_rate * db1
         self.W2 -= self.learning_rate * dW2
         self.b2 -= self.learning_rate * db2
         self.W3 -= self.learning_rate * dW3
         self.b3 -= self.learning_rate * db3


       def accuracy(self, y_true, y_pred):
```

```
                predictions = np.argmax(y_pred, axis=1)
                labels = np.argmax(y_true, axis=1)
                return np.mean(predictions == labels)


            def train(self, X_train, y_train, epochs=1000):
                for epoch in range(epochs):
                output = self.forward(X_train)
                self.backward(X_train, y_train, output)


        cross_entropy_loss = self.cross_entropy_loss(y_train, output)
        ssr_loss = self.sum_squared_residuals_loss(y_train, output)
                acc = self.accuracy(y_train, output)


                        if epoch % 100 == 0:
                print(f"Epoch {epoch}, Cross-Entropy Loss:
{cross_entropy_loss}, SSR Loss: {ssr_loss}, Accuracy: {acc}")
    def visualize_predictions(self, X_test, y_test, num_samples=10):
                                """
        Visualize a few samples from the test dataset along with the
                        model's predictions.


                                Parameters:
                        - X_test: Test input data
                    - y_test: True labels for test data
```

```
            - num_samples: Number of samples to visualize

            """

            # Predict the labels

            y_pred = self.forward(X_test)

            y_pred_labels = np.argmax(y_pred, axis=1)

            y_true_labels = np.argmax(y_test, axis=1)


            # Randomly select samples to visualize

        indices = np.random.choice(X_test.shape[0], num_samples,
                                   replace=False)


            plt.figure(figsize=(15, 5))

            for i, index in enumerate(indices):

            plt.subplot(2, num_samples // 2, i + 1)

        plt.imshow(X_test[index].reshape(28, 28), cmap='gray')

            plt.title(f'True: {y_true_labels[index]}\nPred:
                 {y_pred_labels[index]}')

                        plt.axis('off')


                        plt.show()



# Initialize the neural network with a smaller learning rate
```

```
nn = NeuralNetwork(input_size=784, hidden_sizes=[32, 32], output_size=10,
                                learning_rate=0.01)


                            # Train the neural network

                        nn.train(X_train, y_train, epochs=1000)


                            # Evaluate on the test set

                        test_output = nn.forward(X_test)

                    test_accuracy = nn.accuracy(y_test, test_output)

                        print(f"Test Accuracy: {test_accuracy}")

                    def saliency_map(nn, X, target_class_index):

    # Generate a saliency map for a specific class in the neural network.

                            output = nn.forward(X)

                        target = np.zeros_like(output)

          target[0, target_class_index] = 1  # One-hot for the target class

        dZ3 = output - target  # Gradient of the loss with respect to output

                    dA2 = np.dot(dZ3, nn.W3.T)  # Gradient to A2

                     dZ2 = dA2 * (nn.Z2 > 0)  # ReLU derivative

                    dA1 = np.dot(dZ2, nn.W2.T)  # Gradient to A1

                     dZ1 = dA1 * (nn.Z1 > 0)  # ReLU derivative

                  saliency = np.dot(dZ1, nn.W1.T)  # Compute saliency

                            return np.abs(saliency)


                def visualize_saliency(saliency, original_image):
```

```
                plt.figure(figsize=(10, 5))


                    # Plot original image

                    plt.subplot(1, 2, 1)

            plt.imshow(original_image.reshape(28, 28), cmap='gray')

                    plt.title('Original Image')

                        plt.axis('off')


                     # Plot saliency map

                    plt.subplot(1, 2, 2)

    plt.imshow(saliency.reshape(28, 28), cmap='hot', alpha=0.6)  # Heatmap

                    plt.title('Saliency Map')

                        plt.axis('off')


                        plt.show()


# Assuming you have trained your network and have X_test and y_test ready

                    # Example usage:

      target_class_index = 3  # Replace with your desired class index

       saliency = saliency_map(nn, X_test[0:1], target_class_index)

                visualize_saliency(saliency, X_test[0])

                  # Visualize predictions on test data

          nn.visualize_predictions(X_test, y_test, num_samples=10)
```

```
def visualize_activations(nn, X):
    activations = [X]
    Z1 = np.dot(X, nn.W1) + nn.b1
    A1 = np.maximum(0, Z1)
    activations.append(A1)
    Z2 = np.dot(A1, nn.W2) + nn.b2
    A2 = np.maximum(0, Z2)
    activations.append(A2)
    Z3 = np.dot(A2, nn.W3) + nn.b3
    A3 = nn.softmax(Z3)
    activations.append(A3)
    return activations

# Visualize activations for a sample image
sample_image = X_test[0].reshape(1, -1)
activations = visualize_activations(nn, sample_image)

# Plot the activations
fig, axes = plt.subplots(1, 4, figsize=(20, 4))
layer_names = ['Input Layer', 'Hidden Layer 1', 'Hidden Layer 2', 'Output Layer']

for i, (activation, name) in enumerate(zip(activations, layer_names)):
    if i == 0:
```

```
            axes[i].imshow(activation.reshape(28, 28), cmap='gray')
                else:
                    axes[i].imshow(activation, aspect='auto')
                axes[i].set_title(name)
            plt.show()



def plot_model_performance(metrics_dict):
    # Extract data from the input dictionary
    epochs = list(metrics_dict.keys())
    cross_entropy_loss = [data['Cross-Entropy Loss'] for data in
metrics_dict.values()]
    ssr_loss = [data['SSR Loss'] for data in metrics_dict.values()]
    accuracy = [data['Accuracy'] for data in metrics_dict.values()]

    # Create a figure and axis
    fig, ax1 = plt.subplots(figsize=(12, 6))

    # Plotting Cross-Entropy Loss and SSR Loss
    ax1.set_xlabel('Epochs', fontsize=14)
    ax1.set_ylabel('Loss', fontsize=14)
    ax1.plot(epochs, cross_entropy_loss, label='Cross-Entropy Loss',
color='tab:red', marker='o', markersize=5, linewidth=1.5)
```

```
ax1.plot(epochs, ssr_loss, label='SSR Loss', color='tab:orange',
         marker='o', markersize=5, linewidth=1.5)

ax1.tick_params(axis='y', labelcolor='tab:red')

ax1.set_ylim(0, max(cross_entropy_loss[0], ssr_loss[0]) * 1.1)

ax1.legend(loc='upper left')


# Create a second y-axis to plot Accuracy

ax2 = ax1.twinx()

ax2.set_ylabel('Accuracy', fontsize=14)

ax2.plot(epochs, accuracy, label='Accuracy', color='tab:blue',
         marker='o', markersize=5, linewidth=1.5)

ax2.tick_params(axis='y', labelcolor='tab:blue')

ax2.set_ylim(0, 1)  # Accuracy ranges from 0 to 1

ax2.legend(loc='upper right')


# Adding title and gridlines

plt.title('Model Performance Metrics Over Epochs', fontsize=16)

fig.tight_layout()  # Adjust layout to prevent clipping of ylabel

ax1.grid(color='gray', linestyle='--', linewidth=0.5, alpha=0.7)


# Show the plot

plt.show()


# Example dictionary with your model performance metrics
```

```
metrics = {

 0: {'Cross-Entropy Loss': 2.3429773641229072, 'SSR Loss':
 0.4537995987598951, 'Accuracy': 0.07405357142857143},

 100: {'Cross-Entropy Loss': 2.07748152646247, 'SSR Loss':
 0.4219825075361126, 'Accuracy': 0.37810714285714286},

200: {'Cross-Entropy Loss': 1.7323332098456725, 'SSR Loss':
       0.36848619436757296, 'Accuracy': 0.493},

300: {'Cross-Entropy Loss': 1.3571361136325242, 'SSR Loss':
 0.30242777545716004, 'Accuracy': 0.6379285714285714},

400: {'Cross-Entropy Loss': 1.0562721665518904, 'SSR Loss':
 0.2413717533303831, 'Accuracy': 0.7444642857142857},

500: {'Cross-Entropy Loss': 0.8677114834211727, 'SSR Loss':
 0.19919756444166853, 'Accuracy': 0.7783214285714286},

600: {'Cross-Entropy Loss': 0.7473361006832275, 'SSR Loss':
 0.17114346910783404, 'Accuracy': 0.8019642857142857},

 700: {'Cross-Entropy Loss': 0.665963624755651, 'SSR Loss':
 0.1519965992047637, 'Accuracy': 0.8201607142857142},

800: {'Cross-Entropy Loss': 0.6072933462786502, 'SSR Loss':
 0.1380890439767003, 'Accuracy': 0.8347321428571428},

900: {'Cross-Entropy Loss': 0.5626715927860536, 'SSR Loss':
 0.12743643856404233, 'Accuracy': 0.8469642857142857},

}


    # Call the function with the metrics dictionary

plot_model_performance(metrics)
```