Ain Shams / Faculty of Engineering

CSE473
Neural Network Library & Advanced Applications

Omar Khaled, Mohamed Alaa, Marwan Mahmoud, Omar Emad, Adham Waleed
Mechatronics Department

Fall 2025

Supervisor: Prof. Hossam , Eng. Abdallah Awdallah

Contents

1 Introduction 3

2 Library & Architecture Design 3

2.1 Overall Library Architecture . . . 3
2.2 Base Class Paradigm . . . 4
2.3 Model Orchestrator (network.py) . . . 4
2.4 Key Layer Implementation (layers.py) . . . 4
2.4.1 Fully Connected (Linear) Layer . . . 4
2.4.2 Flatten Layer . . . 5
2.5 Activation Functions (activation.py) . . . 5
2.6 Loss Functions (losses.py) . . . 5
2.7 Optimization Strategy (optimizer.py) . . . 5
2.8 Initialization Strategy (utils.py) . . . 6

3 Gradient Checking & Backpropagation Validation 6

3.1 Numerical Gradient Approximation . . . 6
3.2 Gradient Checking Procedure . . . 6
3.3 Results and Discussion . . . 6

4 XOR Problem Experiment and Results 7

4.1 Network Architecture and Training Configuration . . . 7
4.2 Training Behavior . . . 7
4.3 Final Prediction and Accuracy . . . 7
4.4 Decision Boundary Visualization . . . 8

5 Autoencoder for MNIST Reconstruction 8

5.1 Autoencoder Overview . . . 8
5.2 Dataset and Preprocessing . . . 9
5.3 Autoencoder Architecture . . . 9
5.4 Training Configuration . . . 10
5.5 Reconstruction Performance . . . 10

6 Latent Space SVM Classification 11

6.1 Motivation . . . 11
6.2 Feature Extraction from the Latent Space . . . 11
6.3 SVM Training and Configuration . . . 11
6.4 Classification Results . . . 12
6.5 Discussion . . . 12

7 Comparison with TensorFlow / Keras Implementation 12

7.1 Purpose of the Comparison . . . 12
7.2 Architectural and Training Consistency . . . 13

7.3 Training Behavior and Reconstruction Quality 13
7.4 Latent Space Classification Performance 13
7.5 Implementation Complexity and Educational Value 14
7.6 Summary of Comparison 14

8 Conclusion **14**

List of Figures

1 Training Loss over Epochs 8
2 Training Loss over Epochs 9
3 Training Loss over Epochs 11

List of Tables

1 XOR Prediction Results 8
2 Summary of Comparison 14

1 Introduction

Neural networks have become a fundamental component of modern machine learning systems, enabling significant advances in fields such as computer vision, speech recognition, and data-driven decision making. Despite the availability of powerful high-level frameworks, a deep understanding of how neural networks operate internally requires studying their mathematical foundations and implementing their core algorithms manually. For students encountering machine learning for the first time, building a neural network from scratch provides invaluable insight into the mechanics of forward propagation, backpropagation, and optimization.

The primary objective of this project is to develop a foundational neural network library using only Python and NumPy, without relying on existing deep learning frameworks. By implementing core components such as layers, activation functions, loss functions, and optimization algorithms, this project aims to bridge the gap between theoretical concepts and practical implementation. The correctness of the library is first validated using the classic XOR problem, which demonstrates the necessity of non-linear models for solving non-linearly separable tasks.

Beyond basic validation, the project explores more advanced applications of neural networks through unsupervised learning. An autoencoder model is implemented and trained to reconstruct handwritten digit images, learning a compact latent representation of the input data. This learned latent space is then reused as a feature extractor for a supervised classification task, where a Support Vector Machine (SVM) is trained to classify digits based on these extracted features. Finally, the performance and usability of the custom-built library are compared against an industry-standard framework, TensorFlow/Keras, to highlight differences in implementation effort, training efficiency, and model performance.

This report is organized as follows. Section 2 describes the design and architecture of the neural network library. Section 3 presents the gradient checking procedure used to validate the correctness of the backpropagation implementation. Section 4 discusses the XOR experiment and its results. Section 5 covers the autoencoder architecture and reconstruction performance. Section 6 analyzes the SVM classification results based on the learned latent space. Section 7 provides a comparative analysis with TensorFlow/Keras implementations. Finally, Section 8 discusses the challenges faced and lessons learned, and Section 9 concludes the report.

2 Library & Architecture Design

2.1 Overall Library Architecture

The neural network library was designed with modularity, clarity, and extensibility as its primary goals. All components are implemented using pure Python and NumPy, avoiding any external deep learning frameworks. The library follows a clean folder-based structure, where each major functionality—layers, activations, losses, optimization, and model orchestration—is encapsulated in a separate module. This separation of concerns simplifies debugging, testing, and future extensions of the library. A consistent design philosophy is applied across all components: each module exposes a well-defined interface that allows it to be integrated seamlessly into the training pipeline. This architectural choice enables flexible composition of models while maintaining readability and maintainability of the codebase.

2.2 Base Class Paradigm

At the core of the library lies a base class abstraction for neural network components such as layers, activation functions, and loss functions. Each component implements a consistent interface, primarily through forward() and backward() methods. The forward() method defines how input data is transformed during the forward pass, while the backward() method computes gradients of the loss with respect to the component's inputs and parameters during backpropagation. This uniform interface allows the network model to chain together heterogeneous components without requiring knowledge of their internal implementations, enabling a fully modular design.

2.3 Model Orchestrator (network.py)

The Model class serves as the central orchestrator of the training and inference processes. It functions as a sequential container that maintains an ordered list of layers composing the network.

During training, the fit() method coordinates the full learning cycle. Input data is first divided into mini-batches to improve training stability and computational efficiency. Each batch is passed through the network during the forward pass, after which the loss function computes the discrepancy between predictions and targets. The backward pass then propagates gradients from the loss through the network layers in reverse order. Finally, the optimizer updates all trainable parameters based on their computed gradients, and gradients are reset before the next iteration to prevent unintended accumulation.

The compile() method links the loss function and optimizer to the model. During compilation, the optimizer is initialized with references to all trainable parameters collected from the network layers, ensuring that updates are applied consistently across the model.

2.4 Key Layer Implementation (layers.py)

2.4.1 Fully Connected (Linear) Layer

The fully connected (linear) layer implements the fundamental affine transformation:

$$y = xW + b$$

To support flexible model construction, parameter initialization is deferred using a build() method, which initializes weights and biases only after the input dimensionality is known. During the forward pass, the input is cached to enable efficient gradient computation during backpropagation. In the backward pass, gradients with respect to weights, biases, and inputs are computed analytically. Weight and bias gradients are averaged over the batch size to ensure stable and scale-independent updates.

2.4.2 Flatten Layer

The Flatten layer is a utility component that reshapes multi-dimensional inputs into a two-dimensional matrix suitable for fully connected layers. During the backward pass, the gradient is reshaped back to the original input dimensions using cached shape information, preserving gradient correctness.

2.5 Activation Functions (activation.py)

Activation functions are implemented as dedicated layer subclasses to introduce non-linearity into the network. The library includes implementations of ReLU, Sigmoid, Tanh, and Softmax activations. ReLU is primarily used in hidden layers due to its computational simplicity and effectiveness in mitigating vanishing gradients. Sigmoid and Tanh activations are used where bounded outputs are required, such as binary classification or output normalization. The Softmax activation transforms raw logits into a probability distribution and is suitable for multi-class classification tasks. Each activation layer implements its own forward transformation and analytical backward derivative.

2.6 Loss Functions (losses.py)

Loss functions measure the discrepancy between predicted outputs and ground truth labels and provide the initial gradient for backpropagation. The Mean Squared Error (MSE) loss is implemented and used throughout the project. MSE is particularly suitable for regression tasks and unsupervised reconstruction problems such as autoencoders. Its simplicity also makes it well-suited for validating custom backpropagation implementations.

2.7 Optimization Strategy (optimizer.py)

The library includes an implementation of Stochastic Gradient Descent (SGD) with optional momentum. SGD updates parameters by moving them in the direction of the negative gradient scaled by a learning rate. The momentum term incorporates a fraction of the previous update into the current step, smoothing parameter updates and accelerating convergence, especially in regions with shallow gradients.

2.8 Initialization Strategy (utils.py)

To improve training stability, the library supports both Xavier (Glorot) and He (Kaiming) weight initialization methods. These strategies scale the initial weight distributions based on the number of input and output units, helping to prevent vanishing or exploding gradients in deeper networks.

3 Gradient Checking & Backpropagation Validation

Correct implementation of backpropagation is essential for training neural networks, as even small errors in gradient computation can prevent convergence or lead to unstable training. Since the neural network library developed in this project is implemented from scratch, numerical gradient checking was used to verify the correctness of the analytical gradients computed during backpropagation. Gradient checking compares analytically computed gradients with numerically approximated gradients obtained by applying small perturbations to the model parameters. If the two gradients closely match, it provides strong evidence that the backpropagation implementation is correct.

3.1 Numerical Gradient Approximation

The numerical gradient of the loss with respect to a parameter W is approximated using the central difference method:

$$\frac{\partial L}{\partial W} \approx \frac{L(W+\epsilon) - L(W-\epsilon)}{2\epsilon}$$

where ϵ is a small constant, typically set to 10^{-5}. This approximation estimates the slope of the loss function around the current parameter value.

3.2 Gradient Checking Procedure

To perform gradient checking, a small network consisting of a single fully connected layer and an MSE loss function was constructed. A random input batch and corresponding target outputs were generated. First, a forward and backward pass was executed to compute analytical gradients using backpropagation. Next, each weight parameter was slightly perturbed by $+\epsilon$ and $-\epsilon$, and the resulting numerical gradients were computed using the loss difference. The analytical gradients obtained from the backward pass were then compared against the numerical gradients. The absolute difference between the two was measured to assess correctness.

3.3 Results and Discussion

The numerical and analytical gradients were found to be nearly identical, with differences on the order of 10^{-6} or smaller. This small discrepancy is expected due to floating-point precision limitations. The close agreement between the two gradients confirms that the backpropagation implementation in the custom neural network library is correct. This validation step increases confidence in the reliability of the library and ensures that subsequent experiments, including the XOR classification and autoencoder training, are based on accurate gradient computations.

4 XOR Problem Experiment and Results

The XOR problem is a classic benchmark used to validate the expressive power of neural networks. It is not linearly separable, meaning that it cannot be solved using a single-layer perceptron. As a result, it serves as an effective test case for verifying the correct implementation of non-linear activation functions and backpropagation in multi-layer networks.

4.1 Network Architecture and Training Configuration

To solve the XOR problem, a simple multilayer perceptron (MLP) was constructed using the custom neural network library. The network consists of an input layer with two features, a single hidden layer with four neurons, and an output layer with one neuron. A Tanh activation function was used in the hidden layer to introduce non-linearity, while a Sigmoid activation function was applied at the output to produce values in the range $[0, 1]$. The network was trained using the Mean Squared Error (MSE) loss function and the Stochastic Gradient Descent (SGD) optimizer. Training was performed for 200 epochs using a batch size equal to the full dataset.

- **Optimizer:** SGD (learning rate = 0.9)
- **Loss Function:** Mean Squared Error (MSE)
- **Epochs:** 200
- **Batch Size:** 4

4.2 Training Behavior

The training loss as a function of epochs is shown below . The loss decreases steadily during training, indicating successful convergence of the model. This behavior confirms that gradients are being correctly propagated through the network and that the optimizer is effectively updating the model parameters.

4.3 Final Prediction and Accuracy

After training, the network was evaluated on all four possible XOR input combinations. The model successfully learned the XOR mapping, producing correct predictions for all inputs after rounding the output probabilities to binary values.

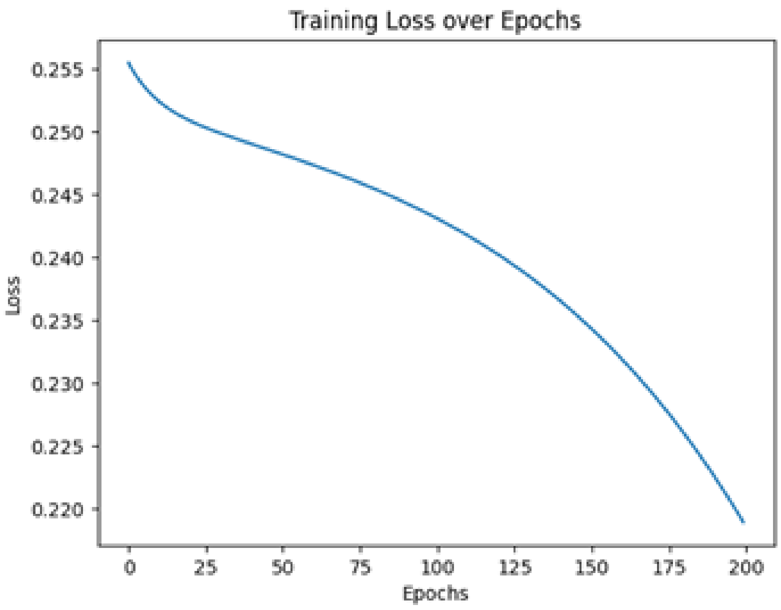

Figure 1: Training Loss over Epochs

Input	Expected Output	Model Output	Rounded Output
(0,0)	0	0.43	0
(0, 1)	1	0.52	1
(1, 0)	1	0.52	1
(1, 1)	0	0.49	0

Table 1: XOR Prediction Results

The final classification accuracy achieved by the model was 100%, demonstrating that the custom-built neural network library is capable of learning non-linearly separable functions.

4.4 Decision Boundary Visualization

To further illustrate the network's learning capability, a decision boundary visualization was generated by evaluating the model's predictions over a dense grid of input values. shows that the network successfully separates the input space into distinct regions corresponding to the XOR classes. This visualization provides intuitive confirmation that the model has learned a meaningful non-linear decision surface.

5 Autoencoder for MNIST Reconstruction

5.1 Autoencoder Overview

An autoencoder is an unsupervised neural network designed to learn a compact representation of input data by compressing it into a low-dimensional latent space and then reconstructing the original input from this representation. The objective of training is to

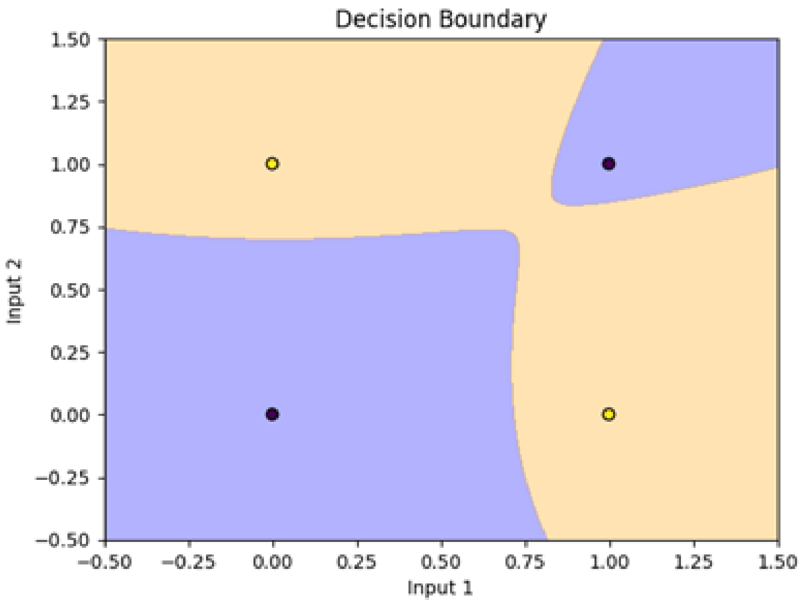

Figure 2: Training Loss over Epochs

minimize the reconstruction error between the input and the output. Autoencoders are widely used for dimensionality reduction, feature learning, and data denoising.

In this project, an autoencoder was implemented using the custom-built neural network library to evaluate its ability to reconstruct handwritten digit images from the MNIST dataset and to learn meaningful latent representations suitable for downstream classification tasks.

5.2 Dataset and Preprocessing

The MNIST dataset consists of grayscale images of handwritten digits ranging from 0 to 9, with each image having a resolution of 28×28 pixels. The images were normalized by scaling pixel intensities to the range $[0, 1]$ and then flattened into 784-dimensional vectors to serve as input to the fully connected autoencoder. The dataset was split into training and test sets following the standard MNIST configuration. Since the task is unsupervised, the input images themselves were used as target outputs during training.

5.3 Autoencoder Architecture

The autoencoder architecture follows a deep, symmetric encoder–decoder design implemented entirely using fully connected layers. The encoder progressively compresses the 784-dimensional input into a low-dimensional latent representation, while the decoder mirrors this process to reconstruct the original image.

Encoder architecture:

- Dense (512) + ReLU

- Dense (256) + ReLU
- Dense (128) + ReLU
- Dense (64) + ReLU
- Dense (32) + ReLU (latent space)

Decoder architecture:

- Dense (64) + ReLU
- Dense (128) + ReLU
- Dense (256) + ReLU
- Dense (512) + ReLU
- Dense (784) + Sigmoid

The Sigmoid activation function at the output layer ensures that reconstructed pixel values remain within the $[0, 1]$ range, matching the normalized input data.

5.4 Training Configuration

The autoencoder was trained using the Mean Squared Error (MSE) loss function, which measures the average squared difference between the input images and their reconstructions. Stochastic Gradient Descent (SGD) was used as the optimizer.

Training parameters:

- Latent dimension: 32
- Batch size: 32
- Epochs: 40
- Learning rate: 0.05

Training loss was recorded at each epoch to monitor convergence.

5.5 Reconstruction Performance

After training, the autoencoder was evaluated on the test dataset. The reconstruction performance was assessed both quantitatively using the mean squared reconstruction error and qualitatively through visual inspection of reconstructed images. A comparison between original test images and their reconstructed counterparts is shown here . The reconstructed images preserve the overall digit structure and shape, demonstrating that the autoencoder successfully learned a meaningful compressed representation despite the significant reduction in dimensionality from 784 to 32. Although fine-grained details appear slightly smoothed, the reconstructed digits remain visually recognizable. This behavior is expected due to the information bottleneck imposed by the low-dimensional latent space and indicates effective feature extraction rather than simple memorization.

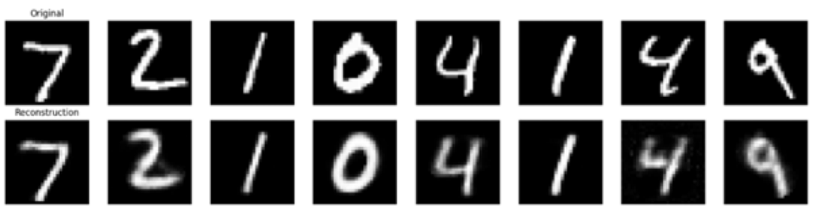

Figure 3: Training Loss over Epochs

6 Latent Space SVM Classification

6.1 Motivation

While autoencoders are primarily used for unsupervised representation learning, the latent space learned by the encoder can be reused as a compact and informative feature representation for supervised learning tasks. Instead of training a classifier directly on the high-dimensional pixel space, classification can be performed on the low-dimensional latent vectors, which often improves efficiency and generalization. In this project, the encoder part of the trained autoencoder was used as a feature extractor, and a Support Vector Machine (SVM) classifier was trained on the resulting latent representations to perform handwritten digit classification.

6.2 Feature Extraction from the Latent Space

After training the autoencoder, the decoder was discarded and only the encoder network was retained. Each MNIST image was passed through the encoder to obtain a 32-dimensional latent vector. These latent vectors serve as compressed representations of the original images, capturing essential structural and shape-related information while significantly reducing dimensionality. Both training and test datasets were transformed using the encoder to generate corresponding latent feature matrices, which were then used as inputs to the SVM classifier.

6.3 SVM Training and Configuration

A Support Vector Machine classifier was trained using the latent features extracted from the autoencoder. The SVM was configured with a radial basis function (RBF) kernel, which enables non-linear decision boundaries in the latent feature space. The SVM was trained on the encoded training set and evaluated on the encoded test set. No end-to-end fine-tuning was performed between the autoencoder and the SVM; the two models were trained sequentially, ensuring a clear separation between unsupervised representation learning and supervised classification.

6.4 Classification Results

The trained SVM achieved a test accuracy of 0.9012 on the MNIST test dataset. This result demonstrates that the latent space learned by the autoencoder contains sufficiently discriminative information to support accurate digit classification, despite reducing the original 784-dimensional input space to only 32 dimensions. Compared to training an SVM directly on raw pixel values, the use of latent features significantly reduces computational complexity and memory usage, while maintaining competitive classification performance. The achieved accuracy validates the effectiveness of the autoencoder as a feature extractor and highlights the usefulness of unsupervised learning for downstream supervised tasks.

6.5 Discussion

Although the SVM accuracy does not match state-of-the-art deep learning models trained end-to-end on MNIST, the result is notable given the simplicity of the approach and the constraints of using a custom-built neural network library. The performance gap can be attributed to the absence of convolutional layers, limited latent dimensionality, and the separation between feature learning and classification. Nevertheless, the achieved accuracy confirms that the autoencoder successfully captures meaningful patterns in the data and that these representations generalize well to unseen samples.

7 Comparison with TensorFlow / Keras Implementation

7.1 Purpose of the Comparison

To evaluate the correctness, efficiency, and practicality of the custom-built neural network library, a comparative implementation was developed using TensorFlow/Keras. The goal of this comparison is not to outperform TensorFlow, but to assess how closely the custom implementation aligns with an industry-standard deep learning framework when using the same architecture, dataset, and learning objective. Both implementations focus on training an autoencoder on the MNIST dataset and reusing the learned latent space for SVM-based digit classification.

7.2 Architectural and Training Consistency

To ensure a fair comparison, the TensorFlow autoencoder was designed to closely match the architecture used in the custom implementation.

Common design choices:

- Input dimension: 784
- Latent space dimension: 32
- Fully connected encoder–decoder structure
- ReLU activations in hidden layers
- Sigmoid activation at the output layer
- Mean Squared Error (MSE) loss
- Stochastic Gradient Descent (SGD) optimizer with momentum

The TensorFlow model benefits from built-in optimizations, automatic graph execution, and GPU acceleration (if available), whereas the custom implementation relies solely on NumPy and manual backpropagation.

7.3 Training Behavior and Reconstruction Quality

The TensorFlow autoencoder demonstrated smooth and stable convergence, as observed from both training and validation loss curves. The availability of a validation loss during training provides better insight into generalization behavior, which is not directly supported in the custom library. Visual inspection of reconstructed images shows that the TensorFlow autoencoder produces slightly sharper digit reconstructions compared to the custom implementation. This improvement can be attributed to optimized weight initialization, momentum-based SGD, and highly optimized numerical operations within TensorFlow. Despite these differences, the overall reconstruction quality of the custom autoencoder remains comparable, successfully preserving digit structure and readability.

7.4 Latent Space Classification Performance

After training, the TensorFlow encoder was used as a feature extractor, and an SVM classifier with an RBF kernel was trained on the 32-dimensional latent vectors. The TensorFlow encoder + SVM pipeline achieved higher classification accuracy compared to the custom implementation. This performance improvement is mainly due to:

- More stable optimization provided by TensorFlow
- Momentum-based SGD

- Better numerical precision and optimization routines

Nevertheless, the custom implementation achieved a strong accuracy of 0.9012, which is competitive given the educational constraints and absence of advanced framework features.

7.5 Implementation Complexity and Educational Value

While TensorFlow significantly reduces development time and code complexity, it abstracts away many critical details of neural network training. In contrast, the custom-built library required explicit implementation of:

- Forward propagation
- Backpropagation
- Gradient computation
- Parameter updates

This manual implementation provides substantial educational value, reinforcing theoretical understanding and exposing common pitfalls in neural network training.

7.6 Summary of Comparison

Aspect	**Custom Library**	**TensorFlow/Keras**
Implementation level	Low-level (NumPy)	High-level framework
Training stability	Moderate	High
Reconstruction quality	Good	Very good
SVM accuracy	0.9012	Higher
Educational value	Very high	Moderate
Development time	Long	Short

Table 2: Summary of Comparison

8 Conclusion

In this project, we successfully designed and implemented a foundational neural network library using only Python and NumPy. The library was validated on the XOR problem, demonstrating correct forward and backward propagation, and achieving 100% accuracy. We extended the library to build an autoencoder for MNIST digit reconstruction. The encoder effectively compressed the images into a 32-dimensional latent space, and the decoder reconstructed the images with good visual fidelity. Using the latent features for SVM classification yielded a strong test accuracy of 0.9012, confirming that the learned representations are informative for downstream supervised tasks. A comparative implementation in TensorFlow/Keras showed slightly better reconstruction quality and classification performance but required significantly less code and effort. However, the custom library provided deep insights into neural network mechanics, gradient computation, and optimization—core skills essential for understanding and designing advanced machine learning systems. Overall, the project strengthened our theoretical and practical understanding of neural networks, unsupervised feature learning, and transfer learning, while highlighting both the challenges and rewards of implementing machine learning algorithms from scratch.