Neural Networks

This project implements a neural network framework from first principles, beginning with linear regression and extending to more complex architectures. Our experiments on MNIST identify optimal hyperparameters while demonstrating effective dimensionality reduction and robust denoising capabilities in autoencoder models.

Preliminaries

Linear Regression and Classification. Neural networks begin with linear regression, a foundation for more complex models. For input $x \in \mathbb{R}^d$, a linear model computes $\hat{y} = Wx + b$ where $W \in \mathbb{R}^{\{p \times d\}}$ and $b \in \mathbb{R}^p$ are parameters.

Training involves minimizing a loss function through gradient descent. For mean squared error:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{\{i=1\}}^{n} (y_i - \hat{y}_i)^2$$

The gradient with respect to parameters is computed:

$$\nabla_W \mathcal{L} = -\frac{2}{n}(y - \hat{y})x^T$$

$$\nabla_b \mathcal{L} = -\frac{2}{n}\sum(y - \hat{y})$$

These gradients guide parameter updates iteratively:

$$W \leftarrow W - \eta \nabla_W \mathcal{L}$$

$$b \leftarrow b - \eta \nabla_b \mathcal{L}$$

For classification tasks, we transform continuous outputs to class probabilities. The softmax function performs this transformation:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

Combined with cross-entropy loss:

$$\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i)$$

Adding Non-linearity. Linear models cannot solve problems with non-linear decision boundaries. Activation functions introduce non-linearity:

- ReLU: $f(x) = \max(0, x)$
- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
- TanH: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Multi-layer Perceptrons. A multi-layer perceptron stacks linear transformations with non-linear activations:

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$$a^{(1)} = f\big(z^{(1)}\big)$$

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$

$$a^{(2)} = f\big(z^{(2)}\big)$$

This creates a function composition:

$$\hat{y} = f^{(L)}\big(W^{(L)} f^{(L-1)}\big(...f^{(1)}\big(W^{(1)}x + b^{(1)}\big)...\big) + b^{(L)}\big)$$

Backpropagation. Training neural networks requires computing gradients through the chain rule. Backpropagation computes these gradients efficiently:

Modular Implementation.

Our framework implements neural networks as modules with standardized interfaces:

– `Module`: Abstract base class defining forward/backward methods
– `Linear`: Implements linear transformations with parameter management
– `Activation`: Implements non-linear transformations (ReLU, Sigmoid, TanH)
– `Loss`: Computes loss values and gradients (MSE, CrossEntropy)
– `Sequential`: Chains modules for forward/backward propagation
– `Optimizer`: Updates parameters using gradient information

Each module requires:
1. Forward pass to compute outputs
2. Backward pass to compute gradients with respect to inputs
3. Parameter update mechanisms

This modular design enables flexible network construction and experimentation with different architectures and hyperparameters.

Backprop

Input: $\mathcal{N}$, x, y, η
Ouput: $W^{\{(l)\}}, b^{\{(l)\}} \quad \forall l = 1, ..., L$

Compute $a^{(l)}$ for all layers
$\delta^{(L)} = \nabla_{a^{(L)}} \mathcal{L} \cdot f'^{(L)}\left(z^{(L)}\right)$
$\delta^{(l)} = \left(\left(W^{(l+1)}\right)^T \delta^{(l+1)}\right) \cdot f'^{(l)}\left(z^{(l)}\right)$
$\nabla_{W^{(l)}} \mathcal{L} = \delta^{(l)} \left(a^{(l-1)}\right)^T$
$\nabla_{b^{(l)}} \mathcal{L} = \delta^{(l)}$
$W^{(l)} \leftarrow W^{(l)} - \eta \nabla_{W^{(l)}} \mathcal{L}$
$b^{(l)} \leftarrow b^{(l)} - \eta \nabla_{b^{(l)}} \mathcal{L}$

Fig. 1. – Backpropagation Algorithm

Basic Supervised Learning

Linear Regression Implementation. To implement linear regression, we created a `Linear` module that performs the forward computation $\hat{y} = Wx + b$ and computes gradients during backpropagation. The `MSELoss` module calculates the mean squared error and its gradient.

Training follows the gradient descent algorithm, with the optimizer handling parameter updates. Our implementation supports batch processing, where multiple samples are processed simultaneously.

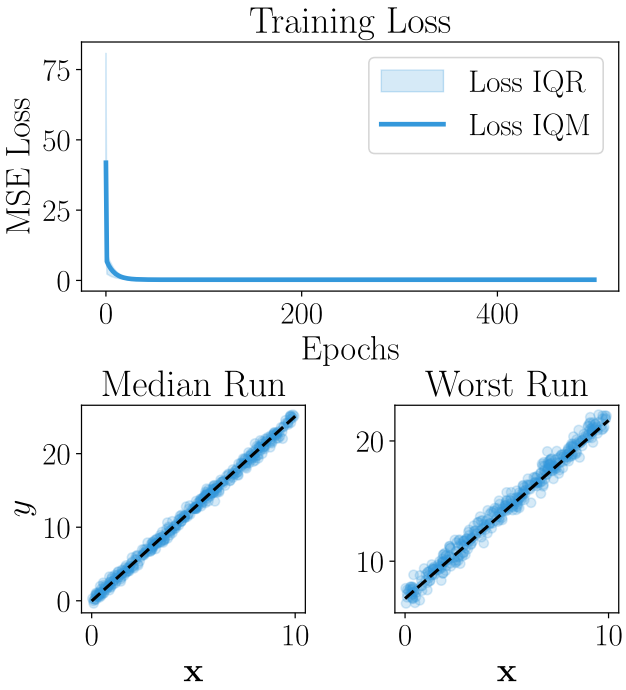

Fig. 2. – **Linear regression training results.** Top: Loss evolution over training epochs, showing interquartile mean (IQM) and interquartile range (IQR). Bottom: Comparison of median and worst models from 25 training runs. Both successfully learn linear relationships despite initialization differences. Models were trained for 200 epochs with learning rate 0.01 and batch size 32.

Fig. 2 shows our linear regression results. The top plot displays loss evolution during training, with rapid initial decrease followed by convergence. The bottom plots compare median and worst-performing models, both capturing the underlying linear pattern with slight differences in fit quality.

Binary Classification with Linear Boundaries. For classification tasks, we extended our framework with a `Softmax` module and `CrossEntropyLoss`. The softmax activation transforms logits into probabilities, while cross-entropy quantifies prediction error.

Implementing classification required modifying our training loop to handle one-hot encoded labels and evaluate using accuracy rather than MSE.

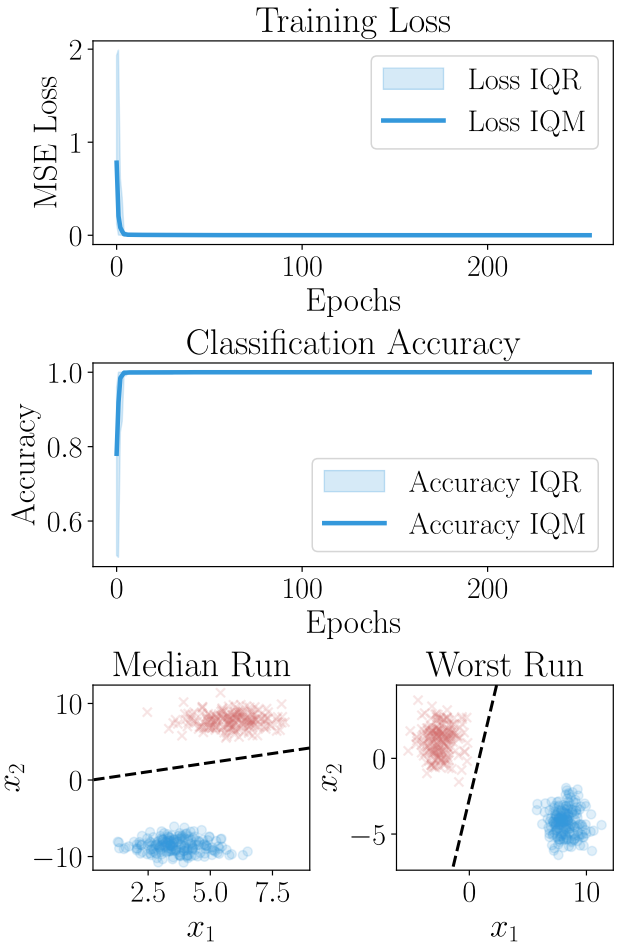

Fig. 3. – **Linear classification training results.** Top: Loss evolution over training epochs, showing interquartile mean (IQM) and interquartile range (IQR). Middle: Accuracy improvement during training. Bottom: Comparison of median and worst models, showing decision boundaries separating two classes (red and blue points). Models were trained for 256 epochs with learning rate 0.01 and batch size 32.

Fig. 3 presents linear classification results. Loss quickly decreases while accuracy reaches nearly 100%, indicating successful convergence. The bottom plots show decision boundaries learned by median and worst models. Both successfully separate the two classes, though the worst model's boundary has a suboptimal orientation.

XOR Problem and Non-linearity. The XOR problem requires non-linear decision boundaries. We implemented activation functions (`ReLU`, `Sigmoid`, `TanH`) as modules and created multi-layer networks using the `Sequential` container.

A key implementation challenge was managing the backward pass through multiple layers, correctly propagating gradients through non-linearities.

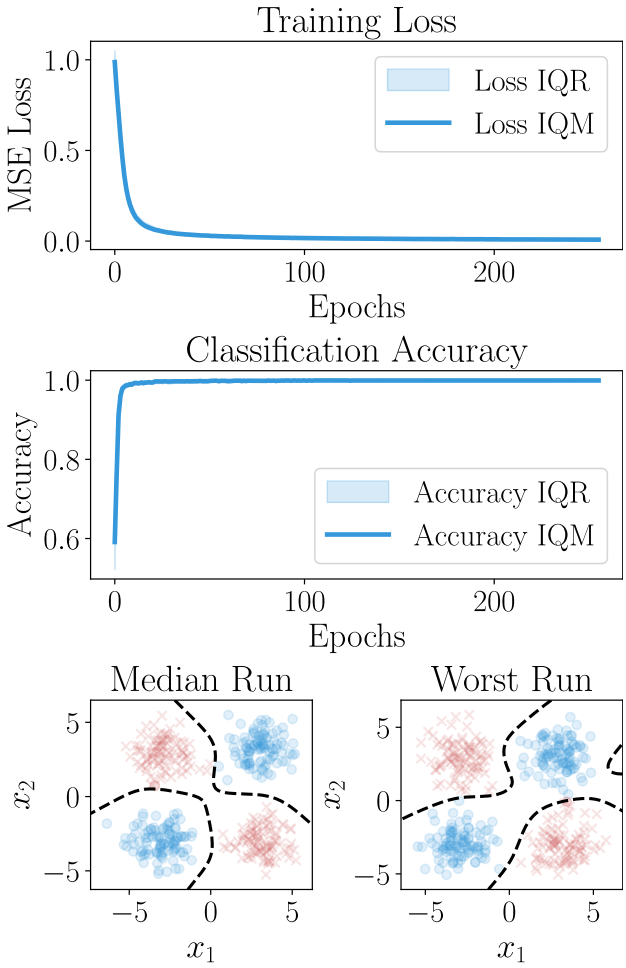

Fig. 4. – **Non-linear classification training results.** Top: Loss evolution over training epochs, showing interquartile mean (IQM) and interquartile range (IQR). Middle: Accuracy improvement during training. Bottom: Comparison of median and worst models, showing complex decision boundaries separating XOR pattern classes. Models contained one hidden layer with 8 neurons and sigmoid activation, trained for 256 epochs with learning rate 0.01 and batch size 32.

Fig. 4 shows our non-linear classification results on the XOR problem. The model successfully learns a non-linear decision boundary, separating the four quadrants into two classes. Both median and worst models capture the XOR pattern, though the worst model shows more classification

errors. The curved decision boundaries demonstrate the network's ability to model non-linear relationships.

MNIST Classification Implementation. Building on our multi-layer framework, we implemented MNIST digit classification using deeper networks. We created a `Sequential` model with two hidden layers followed by a softmax output layer.

Training on MNIST required handling larger data volumes and implementing data batching strategies. We also added validation during training to monitor for overfitting.

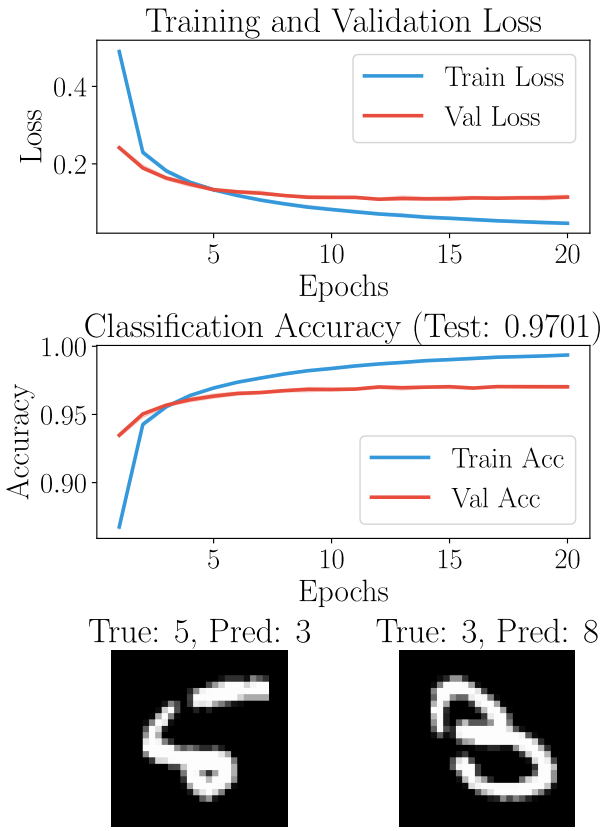

Fig. 5. – **MNIST classification training results.** Top: Training and validation loss over epochs. Middle: Training and validation accuracy over epochs, reaching 97% test accuracy. Bottom: Examples of misclassified digits, showing challenging cases where the model made errors. Model used two hidden layers (64 and 32 neurons) with ReLU activation, trained with Adam optimizer, learning rate 0.001, and batch size 128.

Fig. 5 presents MNIST classification results. Training and validation losses decrease smoothly, while accuracy increases to approximately 97%. The model converges rapidly, with most improvement occurring in the first 5-10 epochs. The bottom images show examples of misclassified digits, illustrating challenging cases where visual ambiguity led to errors.

MNIST Classification & Hyperparameter Analysis

After establishing our base MNIST model, we conducted systematic experiments to analyze how different hyperparameters affect performance. These experiments help understand which factors most significantly impact neural network training and generalization.

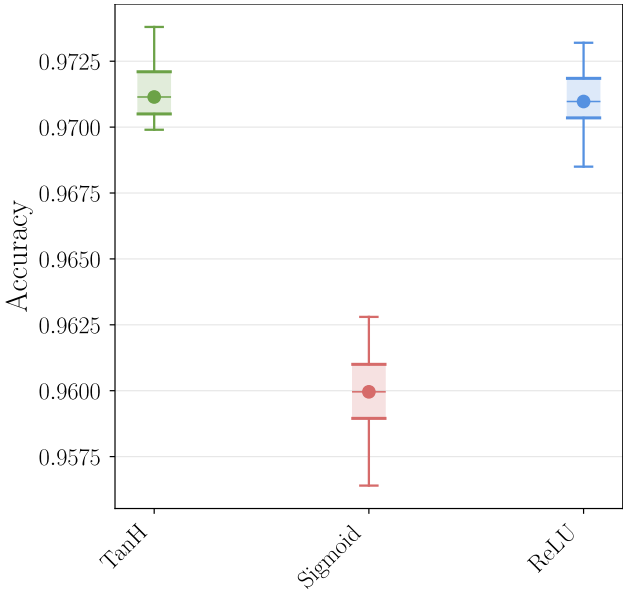

Fig. 6. – **Activation function performance comparison.** Box plots showing accuracy distributions across 15 training runs for different activation functions (ReLU, Sigmoid, TanH). Each model used identical architecture (two hidden layers with 64 and 32 neurons) and training parameters (Adam optimizer, learning rate 0.001, batch size 128, 50 epochs). TanH and ReLU demonstrate similar performance, both significantly outperforming Sigmoid.

Activation Function Comparison. Activation functions determine the non-linear properties of

neural networks. We compared three common functions: ReLU, Sigmoid, and TanH, while keeping all other parameters constant.

Fig. 6 reveals that TanH and ReLU significantly outperform Sigmoid on MNIST classification. TanH shows slightly higher median accuracy, but ReLU offers computational advantages due to its simpler derivative. The poor performance of Sigmoid likely stems from its tendency to saturate, causing gradient vanishing during backpropagation. These results confirm ReLU as a sensible default for most neural network applications.

Batch Size Optimization. Batch size affects both training stability and computational efficiency. We tested batch sizes ranging from 8 to 256 samples to identify optimal values.

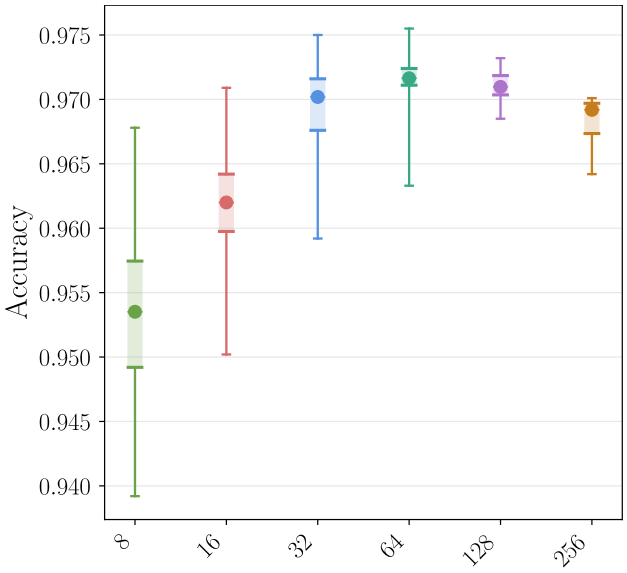

Fig. 7. – **Batch size impact on model performance.** Box plots showing accuracy distributions across 15 training runs for different batch sizes (8, 16, 32, 64, 128, 256). All models used identical architecture (two hidden layers with 64 and 32 neurons, ReLU activation) and training parameters (Adam optimizer, learning rate 0.001, 50 epochs). Medium batch sizes (64-128) achieve the best balance between performance and stability.

Fig. 7 demonstrates that medium batch sizes (64-128) achieve the best performance. Smaller batches (8-16) show higher variance in results, indicating less stable training. Very large batches (256) slightly reduce accuracy, possibly due to less frequent parameter updates. The batch size of 64 yields the highest median accuracy, though with higher variance than 128, suggesting a trade-off between performance and training stability.

Hidden Layer Architecture. The size and structure of hidden layers determine the network's capacity to model complex patterns. We experimented with different hidden layer configurations to assess their impact on performance.

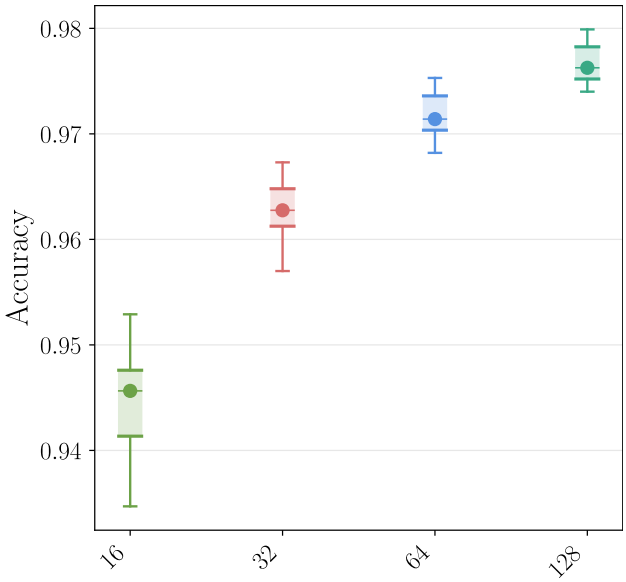

Fig. 8. – **Hidden layer size comparison.** Box plots showing accuracy distributions across 15 training runs for different hidden layer sizes (16, 32, 64, 128 neurons per layer in a two-layer network). All models used identical training parameters (ReLU activation, Adam optimizer, learning rate 0.001, batch size 128, 50 epochs). Larger hidden layers generally yield better performance up to a point of diminishing returns.

Fig. 8 shows that increasing hidden layer size generally improves performance up to a point. The model with 64 neurons per layer achieves high accuracy, while further increase to 128 offers marginal improvement at the cost of additional parameters. Smaller networks (16-32 neurons) underperform, likely lacking capacity to capture the full complexity of handwritten digits. These

results indicate that moderate-sized networks offer the best balance between performance and computational efficiency for MNIST.

Optimizer Comparison. Optimizers control how parameters are updated during training. We compared three optimization algorithms: standard SGD, SGD with momentum, and Adam.

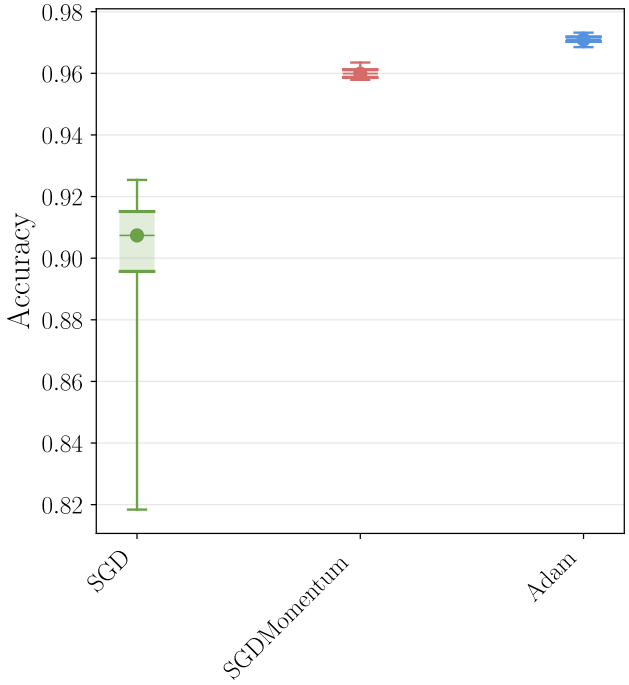

Fig. 9. – **Optimizer performance comparison.** Box plots showing accuracy distributions across 15 training runs for different optimizers (SGD, SGD with Momentum, Adam). All models used identical architecture (two hidden layers with 64 and 32 neurons, ReLU activation) and consistent hyperparameters (learning rates adjusted per optimizer: 0.01 for SGD variants, 0.001 for Adam; batch size 128, 50 epochs). Adam consistently outperforms both SGD variants.

Fig. 9 demonstrates Adam's superior performance compared to both SGD variants. Adam combines adaptive learning rates with momentum, allowing faster convergence and better final accuracy. SGD with momentum shows improvement over vanilla SGD but still lags behind Adam. The reduced variance in Adam's results also indicates more consistent training outcomes, making it a reliable choice for neural network optimization.

These experiments highlight the importance of hyperparameter selection in neural network performance. They demonstrate that even simple architectures can achieve strong results when configured appropriately. For MNIST, the optimal configuration uses ReLU or TanH activation, moderate batch sizes (64-128), hidden layers with 64+ neurons, and the Adam optimizer.

Autoencoder Implementation & Analysis

Autoencoder Architecture. Autoencoders are neural networks trained to reconstruct their input after passing through a constrained latent space. Our implementation consists of an encoder that compresses input data and a decoder that reconstructs it, sharing a symmetric architecture:

- **Encoder**: Sequence of linear layers with decreasing dimensions
- **Latent space**: Bottleneck layer representing compressed information
- **Decoder**: Sequence of linear layers with increasing dimensions

We implemented autoencoders using our modular framework, reusing components like `Linear` and various activation functions. This modularity allowed us to easily experiment with different architectures and hyperparameters.

Latent Space Dimensionality Study. We conducted a systematic study of how latent dimension and network depth affect reconstruction quality, comparing MSE and cross-entropy loss functions.

Fig. 10 reveals several key insights about autoencoder design. First, increasing latent dimension consistently improves reconstruction quality across both loss functions. This is expected as larger latent spaces can retain more information about the input. Second, deeper networks generally outperform shallow ones, with depth 3-5 showing the best results. The improvement plateaus at greater depths, suggesting diminishing

returns beyond a certain architectural complexity.

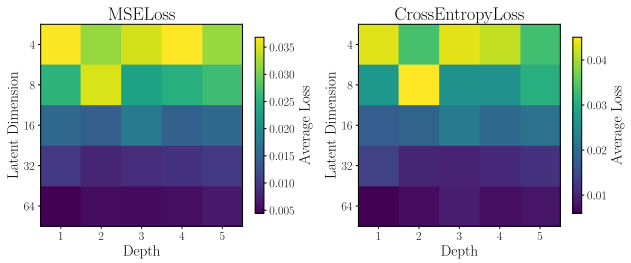

Fig. 10. – **Latent dimension and depth impact on reconstruction quality.** Heatmaps showing reconstruction loss for different combinations of latent dimensions (y-axis: 4, 8, 16, 32, 64) and network depths (x-axis: 1-5). Left: MSE loss results. Right: Cross-entropy loss results. Darker colors indicate better reconstruction quality. Models were trained on MNIST with identical parameters except for the studied dimensions.

Notably, cross-entropy loss appears more sensitive to latent dimension size than MSE, showing more dramatic improvement as dimension increases. This reflects cross-entropy's properties as a loss function that strongly penalizes incorrect probability predictions at pixel level.

Latent Space Visualization. To understand what information the autoencoder captures, we visualized the latent representations of MNIST digits.

Figure demonstrates that even with significant dimensionality reduction (from 784 to 32), the autoencoder successfully preserves the key visual characteristics of each digit. The reconstructions appear slightly smoother than originals, suggesting the model learns to filter noise while retaining essential features.

t-SNE Visualization and KNN Classification. To further examine the latent space structure, we applied t-SNE dimensionality reduction to visualize the 4-dimensional latent space in 2D.

Figure reveals that the autoencoder organizes digits into distinct clusters in latent space, despite being trained without class labels. This emergent structure suggests the model learns meaningful features that naturally separate digit classes. Digits with similar visual characteristics (like 4 and 9, or 3, 5, and 8) appear closer in the latent space, reflecting their visual similarity.

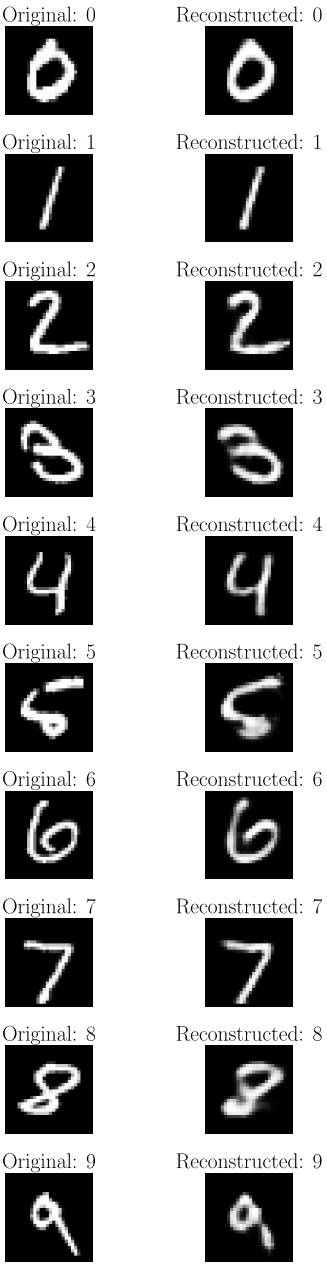

Fig. 11. – **Original and reconstructed MNIST digits.** Examples showing original digits (left) and their reconstructions (right) from an autoencoder with latent dimension 32 and depth 5. The model successfully captures essential visual characteristics of each digit while smoothing out noise and variations, indicating effective compression of relevant features.

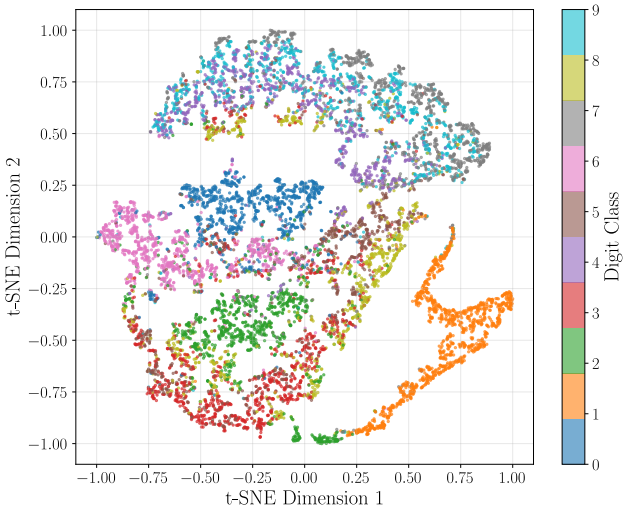

Fig. 12. – **t-SNE visualization of latent space representations.** 2D projection of the 4-dimensional latent space representations of MNIST test set, colored by digit class. The clear clustering by digit class indicates the autoencoder has learned digit-specific features despite being trained without class labels. Note how similar digits (e.g., 4-9, 3-5-8) appear closer in the latent space.

To quantify the discrimination power of these learned representations, we trained a KNN classifier (k=5) on the latent codes:

Class	TP	FP	FN	TN	Accuracy
0	914	123	66	8897	0.9811
1	1113	42	22	8823	0.9936
2	889	172	143	8796	0.9685
3	762	330	248	8660	0.9422
4	790	356	192	8662	0.9452
5	534	209	358	8899	0.9433
6	869	146	89	8896	0.9765
7	853	118	175	8854	0.9707
8	681	167	293	8859	0.9540
9	692	240	317	8751	0.9443
Total	8097	1903	1903	88097	0.9619

The high classification accuracy (96.19%) confirms that the latent space effectively captures digit-specific features. Notably, digits like « 1 » achieve exceptional accuracy (99.36%), likely due to their distinctive shape, while more ambiguous digits like « 5 » and « 3 » show lower performance.

Latent Space Centroids and Interpolation. To explore the semantic structure of the latent space, we computed centroid vectors for each digit class and visualized interpolations between them.

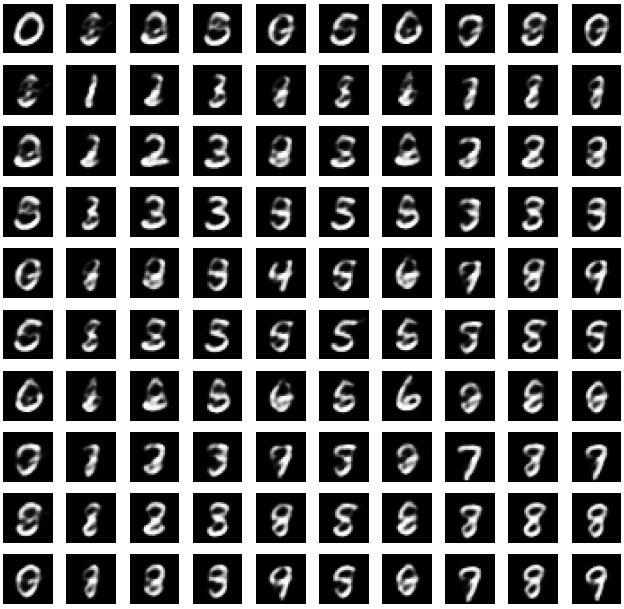

Fig. 13. – **Interpolation between digit centroids in latent space.** Grid showing reconstructions from latent vectors interpolated between digit class centroids. Diagonal elements represent pure class centroids, while off-diagonal elements show 50% interpolation between row and column digit centroids. The smooth transitions between digits reveal the continuous nature of the learned latent space.

Figure provides insights into the semantic organization of the latent space. The diagonal elements represent « prototypical » digits reconstructed from class centroids, showing idealized versions of each digit. The off-diagonal elements reveal how the model interpolates between digit classes. Some transitions appear natural (e.g., $4 \to 9$ or $3 \to 8$) while others create unusual hybrid digits. This confirms that the latent space is continuous and semantically organized, with nearby points representing similar visual concepts.

This interpolation capability demonstrates that the autoencoder has learned a meaningful compressed representation rather than simply memorizing training examples. The latent space captures the underlying manifold of handwritten digits, allowing navigation between different digit concepts.

Denoising Capabilities. We extended our autoencoder to perform denoising by training it to reconstruct clean images from artificially corrupted inputs. The model was trained with Gaussian noise at a moderate level ($\sigma = 0.2$) but proved capable of handling much stronger corruption.

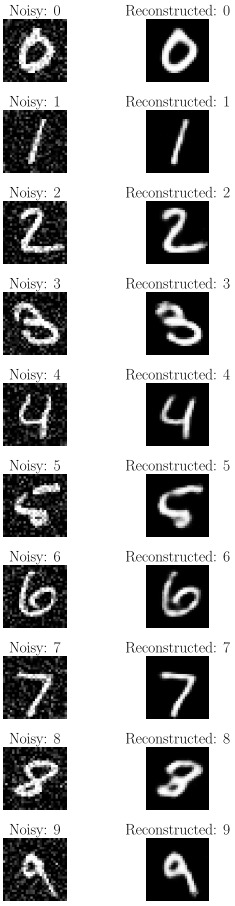

Fig. 14. – **Denoising results at training noise level.** Pairs of noisy inputs (left) and their reconstructions (right) for each digit class (0-9), using the same noise level ($\sigma = 0.2$) that the model was trained on. The reconstructions clearly preserve digit identity while removing noise artifacts, demonstrating the model's effectiveness at its trained task.

Figure shows the model's performance at its trained noise level ($\sigma = 0.2$). The reconstructions are clean and accurate, with noise effectively removed while maintaining the essential characteristics of each digit. This confirms the model successfully learned to distinguish between meaningful digit features and random noise during training.

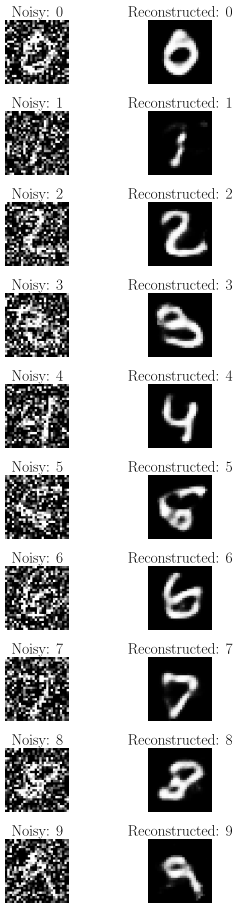

Fig. 15. – **Denoising generalization to severe noise.** Pairs of heavily corrupted inputs (left) and their reconstructions (right) for each digit class (0-9), using noise level ($\sigma = 0.6$) three times stronger than training conditions. Despite never seeing such extreme corruption during training, the autoencoder successfully recovers recognizable digits, demonstrating remarkable generalization capability.

Figure demonstrates the model's remarkable generalization capability when faced with much more severe noise ($\sigma = 0.6$) than it was trained on. Even with corruption levels that make digits nearly unrecognizable to human observers,

the autoencoder successfully reconstructs clear, identifiable digits. This suggests the model has learned deep, robust representations that capture the fundamental structure of each digit class.

The strong generalization to higher noise levels indicates the autoencoder didn't simply memorize noise-removal patterns but instead learned to identify invariant features that define digit identity regardless of noise intensity. This ability to extract essential signal from previously unseen levels of noise suggests potential applications in various domains where signal recovery from unexpected corruption levels is required.

The preserved classification accuracy even under extreme noise demonstrates the robustness of the learned representations and their potential utility as preprocessing for downstream tasks in challenging, noisy environments.

Conclusion

This project implemented a modular neural network framework from first principles using NumPy. Beginning with linear regression, we progressively developed components for classification, multi-layer architectures, and autoencoders. The modular design proved valuable, allowing systematic experimentation with different architectures and hyperparameters.

Our experiments yielded several practical insights. For MNIST classification, we found that ReLU and TanH activations significantly outperform Sigmoid, medium batch sizes (64-128) offer the best balance between performance and stability, and the Adam optimizer consistently delivers superior results compared to SGD variants. These findings align with established best practices in the field.

The autoencoder experiments demonstrated how neural networks can learn meaningful representations without explicit supervision. The latent space analysis revealed emergent organization of digit classes and robust feature extraction, as evidenced by the high KNN classification accuracy (96.19%) and impressive denoising capabilities even under severe corruption conditions.

While our implementation is not optimized for production use, it successfully demonstrates core neural network principles and provides a foundation for understanding more complex architectures. The framework's limitations include lack of GPU acceleration and absence of more advanced techniques like batch normalization or dropout, which would be natural extensions to this work.