SORBONNE UNIVERSITY

M1 - ML

Project: Neural Networks - DIY

Authors:
Sama SATARIYAN
Missoum YOUCEF

May 11, 2025

Abstract

The project implements modular neural networks from scratch for binary, multi-class, and autoencoding tasks. Our best binary classifier achieved a test accuracy of 100%, while the multi-class model reached 96.67%. Starting with a linear perceptron, it introduces non-linear activations and Linear and Sequential wrapper for clean architecture. Hyperparameter searches reveal that solely mean validation loss isn't always a reliable selection method. The autoencoder compresses MNIST images and enables denoising and generation from latent space. The model reconstructed digits with high fidelity, achieving a 91% test accuracy using a k-NN classifier in the latent space.

The structure of the report is divided into five parts:

- **Section 1:** Implementation of a linear neural network for binary classification, along with testing the model on a synthetically generated dataset, and conducting a hyperparameter search.
- **Section 2:** Introduction of non-linear activation functions and construction of a two-layer perceptron to improve classification performance.
- **Section 3:** Abstraction using a `Sequential` module to streamline network construction and training logic for binary classification.
- **Section 4:** Extension to multi-class classification, incorporating new activation and loss functions.
- **Section 5:** Implementation and evaluation of an autoencoder to learn compressed representations of MNIST digits, including image reconstruction and generation from latent space.

1 Section One: Linear NN Model - Binary Classification

1.1 Objective

In the first section, we aim to implement the foundational components required for linear regression using a modular neural network framework.The goal is to understand how to build a forward and backward pass manually for a perceptron, compute gradients, and update parameters through gradient descent.

1.2 Method

Synthetic Dataset Generation

We created a configurable synthetic dataset using the `data_creation` (`data_utils.py`). This allows controlled experimentation by generating input-output pairs under known distributions and parameters.

Data Generation Process The synthetic data is generated as follows:

- N input vectors $\mathbf{x}_i \in \mathbb{R}^d$ are sampled from a standard normal distribution $\mathcal{N}(0, I)$, where $d =$ `input_dim`.
- A ground-truth weight matrix $W \in \mathbb{R}^{d \times C}$ and bias $b \in \mathbb{R}^{1 \times C}$ are sampled from $\mathcal{N}(0, 1)$.
- The logits are computed via a noisy linear transformation:

$$\texttt{logits} = XW + b + \varepsilon \quad \text{where } \varepsilon \sim \mathcal{N}(0, 0.5)$$

Target Generation The targets y are computed depending on the number of classes:

- *Binary classification ($C = 1$):* The output is passed through a sigmoid function. Labels are assigned as 1 if the probability exceeds 0.5.
- *Multi-class classification ($C > 1$):* A noisy argmax is applied to the logits to assign discrete class labels, followed by one-hot encoding.

Dataset Splitting The dataset is randomly split into three disjoint subsets: *Training set* (60%), *Validation set* (20%), *Test set* (20%).

To ensure reproducibility, a fixed random seed (`SEED_NUMBER = 0`) is used via the function `init_random_seed`, which sets the global NumPy seed before data generation or training.

The NN Structure

We implemented a custom `Linear` class (please check out the file `layers.py` representing a fully connected layer in a feedforward neural network. This class is designed to support both forward and backward passes for training using gradient-based optimization.

The constructor `__init__` initializes the layer's parameters:

- The weights W are initialized using He initialization:

$$W \sim \mathcal{N}\left(0, \frac{1}{\text{input\_dim}}\right)$$

  which is well-suited for layers using Tanh activation functions.

- The biases b are initialized to zero.
- A gradient dictionary is created to store the gradients of the loss with respect to both W and b, initialized to zero.

Forward Pass: The `forward(X)` method computes the linear transformation:

$$y = XW + b$$

where:

- $X \in \mathbb{R}^{N \times D}$ is the input batch of size N,
- $W \in \mathbb{R}^{D \times M}$ is the weight matrix,
- $b \in \mathbb{R}^{1 \times M}$ is the bias vector,
- $y \in \mathbb{R}^{N \times M}$ is the resulting output.

The input X is stored for use in backpropagation.

Zeroing Gradients: The `zero_grad()` method resets the stored gradients to zero. This is necessary before beginning a new gradient computation to avoid accumulating gradients across multiple batches.

Backward Pass (Gradient Accumulation): `backward_update_gradient(input,delta)` method computes and accumulates the gradients of the loss L with respect to the parameters:

$$\frac{\partial L}{\partial W} = X^\top \delta, \quad \frac{\partial L}{\partial b} = \sum \delta$$

where δ is the gradient of the loss with respect to the output of the layer.

Backward Pass (Delta Propagation): The `backward_delta(input, delta)` method computes the gradient of the loss with respect to the input:

$$\delta_{\text{prev}} = \delta W^\top$$

This quantity is propagated to the previous layer during backpropagation.

Parameter Update: The `update_parameters(gradient_step)` method updates the weights and biases using standard gradient descent:

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}, \quad b \leftarrow b - \eta \frac{\partial L}{\partial b}$$

where η is the learning rate.

Evaluation: The `score(X, y, label="Test")` method performs a forward pass followed by an optional activation function (e.g., sigmoid or softmax), then computes the classification accuracy. For binary classification, a threshold of 0.5 is applied; for multiclass tasks, predictions are made via `argmax`. The method prints and returns the resulting accuracy.

1.3 Training Procedure:

In `training_loop_linear_binary` (please check out the file `training.py`) we trained the binary classifier using a simple stochastic gradient descent loop. The model consists of a single linear layer trained using the MSE loss. At each epoch, the training data is shuffled and divided into mini-batches. For each batch, the model performs a forward pass to compute predictions, then a backward pass to compute gradients, which are used to update the parameters.

Validation and test losses are computed at each epoch. The model achieving the lowest validation loss is saved to avoid overfitting. After training, the model is evaluated on the training and test sets using accuracy. Loss curves can also be visualized to assess training progress.

Epoch Loop: In the training process at each epoch:

- The training data is shuffled to improve generalization.
- The data is split into mini-batches of size B.
- For each mini-batch:
  1. A forward pass computes the predicted output $\hat{y}$.
  2. The MSE loss is computed.
  3. A backward pass computes the gradient of the loss with respect to model parameters.
  4. Gradients are reset to zero before each update to prevent accumulation.
  5. The model parameters are updated using gradient descent:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$$

  where η is the learning rate.

Validation: At the end of each epoch, we compute the validation loss. The model achieving the lowest validation loss during training is stored as the best model to help mitigate overfitting.

Evaluation: After training, the model's performance is evaluated on both the training and test sets using accuracy.

1.4 Experiments

We train the model on the generated data to see the performance. The model is trained on the training dataset, then it is tuned on the validation dataset, and finally the performance is evaluated on the test dataset.

Hyperparameter Search:. In order to tune the models, we can adjust their parameters, in this case *Learning Rate*. In order to conduct a hyperparam search, we use the `param_search_p1` method (`param_search.py`). This function focuses on identifying an effective learning rate and estimating the optimal number of training epochs to minimize validation loss.

In this case, the function performs a systematic search over a range of learning rates using `np.linspace(0.0000001, 0.00003, 10)`. For each candidate learning

rate, a training loop is executed for 1000 epochs. During training, the validation loss is recorded at each epoch. The performance metric that is collected for each learning rate is the Average of the Last 10 Validation Losses: a smoothed metric used to mitigate fluctuations near convergence.

The function selects the best learning rate based on the lowest average validation loss over the final 10 epochs.

After determining the best learning rate, the model is retrained using this rate, and the epoch corresponding to the minimum validation loss is selected as the optimal number of training epochs.

This approach ensures that both the learning rate and training duration are tuned to improve the efficiency and performance of the model.

1.5 Results

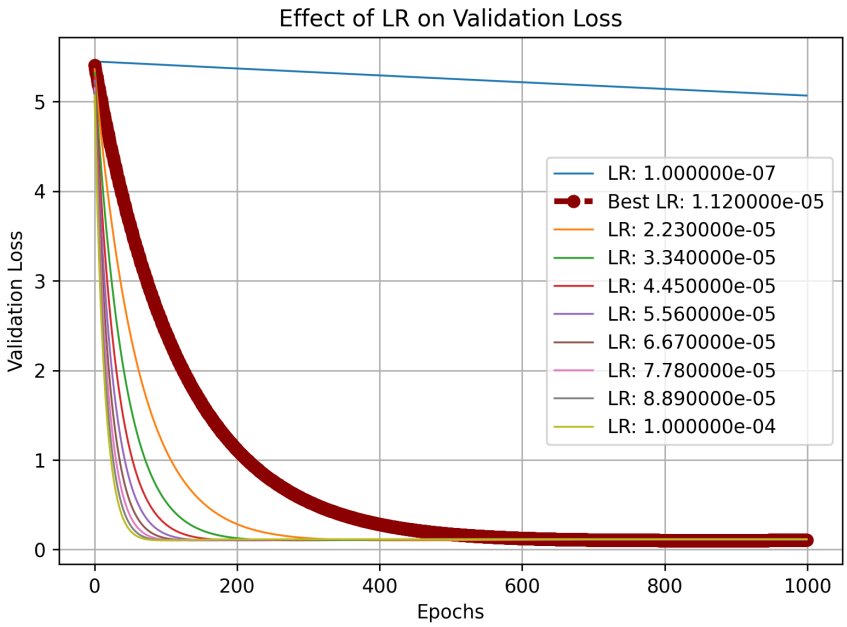

FIGURE 1: Section one - Heatmap for each learning-rate validation loss

After conducting the hyperparameter search, the lowest validation loss for each combination of parameters gives the best parameters, as shown in the Heatmap 1. Then , we train the model with the best parameters.

As shown in Figure 2, the linear perceptron is able to classify the binary classes with a training accuracy of 95.56% and a test accuracy of 95.00%, using an optimized learning rate as the hyperparameter.

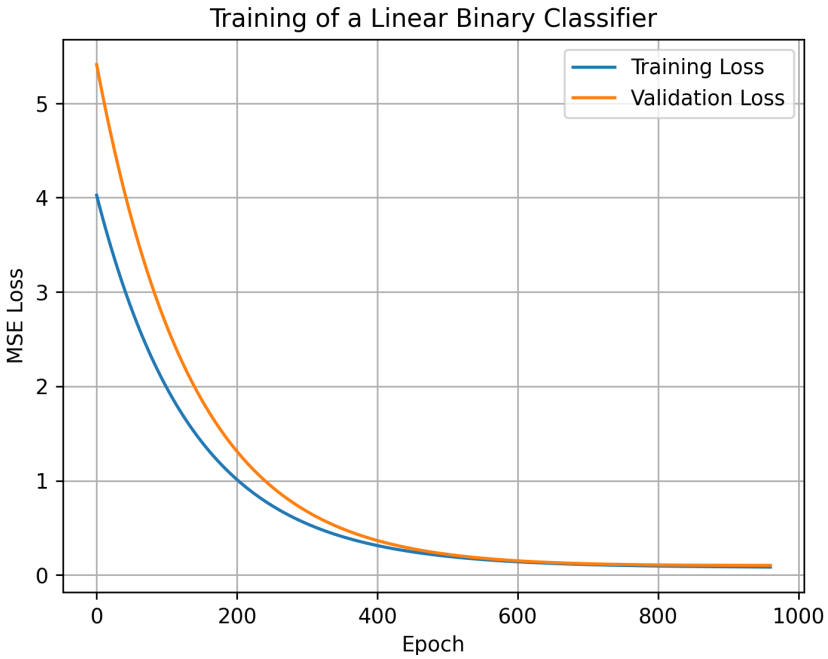

FIGURE 2: Section one - Training and validation loss over epochs

2 Section Two: Nonlinear NN Model - Binary Classification

2.1 Objective

The objective of this part is to implement non-linear activation functions—TanH and Sigmoid—as independent modules within the neural network framework. These modules were designed to inherit from the abstract `Module` class.

To validate their functionality, the modules were integrated into a simple neural network composed of two linear layers, with a TanH activation between them and a Sigmoid activation at the output. This architecture was trained on a binary classification task, using gradient descent. The goal was to observe how non-linearity improves the model's ability to classify data beyond what a purely linear model can achieve. Here is the scheme:

$$X \rightarrow Linear \rightarrow Tanh \rightarrow Linear \rightarrow Softmax \rightarrow y$$

2.2 Method

Dataset

The dataset used in this part is the same as the one described in Section 1.2.

The NN Structure

The `Module` Class

We implemented an abstract `Module` class (`layers.py`) which serves as the base for all neural network components, including layers and activation functions. This class defines a common interface for forward and backward passes, and enforces the modular structure of the network.

Forward Pass: The method `forward(X)` is defined as abstract and must be implemented by all subclasses. It takes input data X and returns the output of the module. This enforces a consistent interface across all types of layers and functions.

Gradient Handling: The `zero_grad()` method is used to reset the gradient values to zero before starting a new backward pass. The following methods are defined to be overridden:

- `backward_update_gradient(input, delta)`: computes the gradient of the loss with respect to the module's parameters and accumulates it.
- `backward_delta(input, delta)`: computes the gradient of the loss with respect to the module's input, used for backpropagation.
- `update_parameters(gradient_step)`: updates parameters using the accumulated gradient and a specified step size.

Building on this abstract base class, we implemented a `Sequential` class that simplifies the chaining of modules in a feedforward architecture.

Activation Function Modules: `TanH`, `Sigmoide`, and `Softmax`:

We implemented several non-linear activation functions as subclasses of the abstract `Module` class. Although these modules do not have trainable parameters, they are essential to introduce non-linearity into the network and must support both forward and backward computations.

`TanH`

The `TanH` class applies the hyperbolic tangent function to the input.

TanH - Forward Pass: The `forward(X)` method computes:

$$\text{output} = \tanh(X)$$

where X is the input batch. The result is stored for use in the backward pass.

TanH - Backward Delta: The `backward_delta(input, delta)` method computes the gradient of the loss with respect to the input using the derivative of $\tanh(x)$:

$$\delta_{\text{prev}} = \delta \cdot (1 - \tanh^2(X))$$

`Sigmoide`

The `Sigmoide` class applies the sigmoid activation function.

Sigmoide - Forward Pass: The `forward(X)` method computes:

$$\text{output} = \frac{1}{1 + e^{-X}}$$

Sigmoide - Backward Delta: The `backward_delta(input, delta)` method computes:

$$\delta_{\text{prev}} = \delta \cdot \sigma(X) \cdot (1 - \sigma(X))$$

where $\sigma(X)$ is the sigmoid function.

`Softmax`

The `Softmax` class applies a softmax normalization across the last axis of the input, converting raw scores into probabilities.

Softmax - Forward Pass: The `forward(X)` method computes:

$$\text{output}_i = \frac{e^{X_i}}{\sum_j e^{X_j}}$$

for each input row. A numerical stability trick is used by subtracting the row-wise maximum before exponentiation.

Softmax - Backward Delta: The `backward_delta(input, delta)` method returns the delta unchanged:

$$\delta_{\text{prev}} = \delta$$

This assumes that the cross-entropy loss has already computed the correct gradient with respect to the softmax output (as is commonly done for numerical stability).

Training Procedure

The function `training_testing_nonlinear_binary` (`training.py`) trains a feedforward neural network consisting of two linear layers with non-linear activations: `TanH` in the hidden layer and `Sigmoide` in the output. The loss function used is binary cross-entropy (`BCELoss`).

The main differences from the linear case are:

- The model architecture includes two layers with non-linear activations, enabling it to learn more complex decision boundaries.
- The loss function is suited for binary classification with probabilistic outputs in $[0, 1]$.
- Gradients are propagated manually through both activation and linear layers using `backward_delta` and `backward_update_gradient`.

Validation loss is computed at each epoch to monitor generalization, and classification accuracy is reported at the end of training.

2.3 Experiments

Hyperparameter Search

The model has two parameters that can change the performance of the model : *Learning Rate* and the *Hidden Dimension Size*.

To optimize the performance of the nonlinear binary classifier, we conducted a hyperparameter search using the `param_search_p2` function (`param_search.py`). This function explores combinations of training parameters to minimize validation loss and improve model generalization.

Joint Search over Learning Rate and Hidden Dimension: When called with the argument `"LR_and_middleDim"`, the function performs a grid search over two parameters:

- The number of hidden units in the middle layer (`middle_dim`) from the set $\{3, 5, 6, 7, 8, 12\}$,
- The learning rate from the set $\{0.050075, 0.06, 0.07, 0.09, 0.2, 0.3\}$.

For each configuration, the model is trained for 1000 epochs and evaluated using the average validation loss over the last 10 epochs. The results are stored in a matrix and visualized as a heatmap for better interpretability. The best-performing combination is selected, and the model is then retrained for 4000 epochs to determine the best stopping time.

2.4 Result

FIGURE 3: Section two - Heatmap for validation loss mean per learning-rate and hidden-dimension

The criterion used for selecting the best hyperparameter combination was the mean validation loss. The underlying assumption was that minimizing this average across training epochs would yield the most effective and generalizable model.

However, a comparison of the loss curves corresponding to two distinct points from the heatmap in Figure 3 reveals a counterintuitive outcome. The configuration with the lowest validation loss mean (0.0067, test accuracy: 98%), shown in Figure 5, displays significant fluctuations and less stable convergence. In contrast, a near-optimal configuration with a slightly higher validation loss mean (0.0148, test accuracy: 100%), illustrated in Figure 4, exhibits smoother and more stable convergence behavior.

This observation suggests that relying solely on the mean validation loss as a selection criterion may be insufficient or even misleading. It fails to account for training dynamics such as convergence stability and robustness, which may better correlate with downstream performance. Consequently, incorporating additional metrics—such as validation loss variance, smoothed minimum loss, or convergence speed—may lead to more reliable hyperparameter selection.

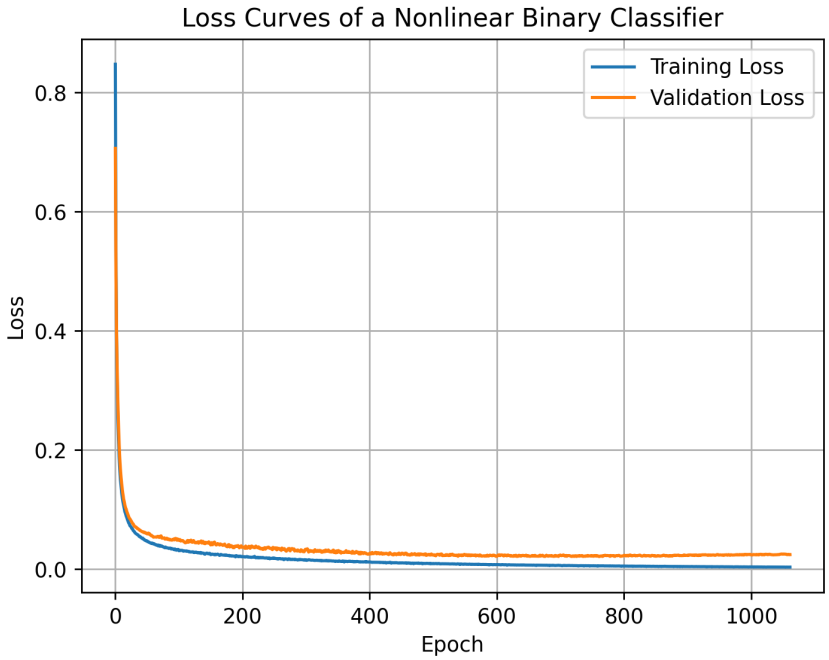

FIGURE 4: Section two - Training and validation loss over epochs - using suboptimal point

Even considering the model with 98% test accuracy, we observe that the non-linear two-layered perceptron was able to classify the binary classes better than the linear percepron in the section one, explaining higher complexity in the dataset.

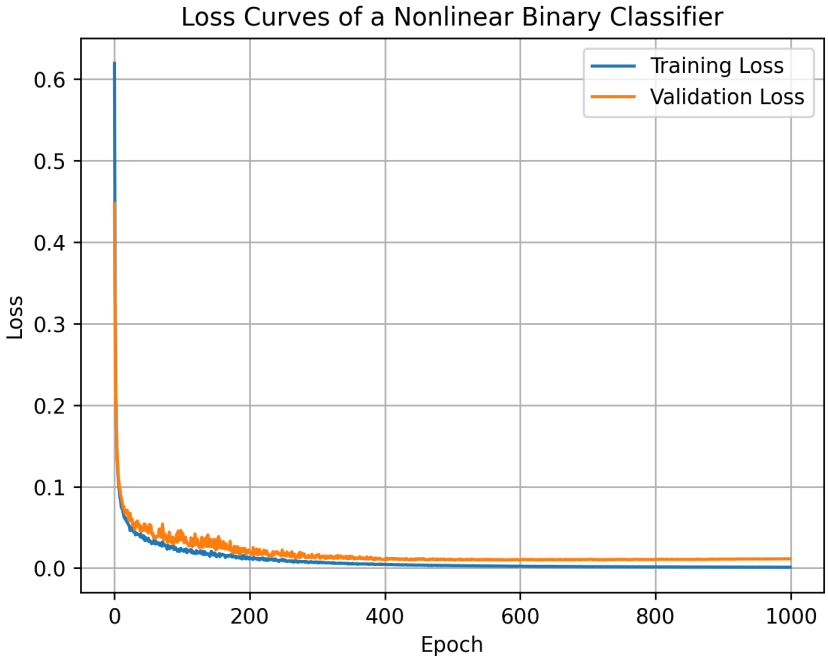

FIGURE 5: Section two - Training and validation loss over epochs - using optimal point

3 Section Three - Nonlinear MLP Wrapped in a Sequence - Binary Classification

3.1 Objective

The objective of this part is to simplify the construction and training of neural networks by implementing a `Sequential` class. This class encapsulates a list of modules (layers and activations) and automates the forward and backward passes across them.

By using `Sequential`, we can define complex networks in a modular and readable way, avoiding repetitive code when chaining multiple layers. This abstraction supports batch-based computation, gradient accumulation, and parameter updates for all submodules.

In addition, we implemented an `Optim` class that wraps the training step, including forward pass, loss computation, backward pass, and parameter updates, further streamlining the training loop. Together, these tools enable cleaner code and faster experimentation when working with various network architectures.

3.2 Method

Dataset

The dataset used in this part is the same as the one described in Section 1.2.

The NN Structure

The `Sequential` class (in `layers.py`) encapsulates a list of modules to be applied in sequence. This enables construction of full networks by simply stacking modules.

Forward Pass: The `forward(X)` method performs a forward pass through all submodules sequentially:

$$X \to M_1(X) \to M_2(M_1(X)) \to \cdots \to M_n(\cdots) \to y$$

The output of each module is passed as input to the next.

Gradient Reset: The `zero_grad()` method calls `zero_grad()` on each submodule, resetting all gradients in preparation for a new backpropagation step.

Backward Pass: The `backward_update_gradient(input, delta)` method performs a full backward pass:

1. It first stores all intermediate activations during the forward pass.
2. Then, it loops backward through the modules to compute parameter gradients and propagate deltas using:

$$\delta^l = \frac{\partial L}{\partial z^l}, \quad \frac{\partial L}{\partial \theta^l} = \delta^l \cdot \frac{\partial z^l}{\partial \theta^l}$$

Backward Delta Propagation:

The `backward_delta(input, delta)` method computes the overall delta to propagate to earlier layers by chaining calls to each module's `backward_delta` method.

Parameter Update: The `update_parameters(gradient_step)` method updates the parameters of all submodules using the accumulated gradients and a learning rate

η.

Evaluation:
The `score(X, y, label="Test")` method gives the evaluation of the performance.

Training Procedure

The function `training_testing_sequential_binary` (`training.py`) trains a binary classifier using the `Sequential` class to encapsulate the entire network architecture. This abstraction simplifies the training pipeline by automating both forward and backward passes across a sequence of layers.

The model used here consists of two linear layers, a `TanH` activation in the hidden layer, and a `Sigmoide` activation in the output. Training is performed using binary cross-entropy loss (`BCELoss`) and mini-batch stochastic gradient descent.

Epoch Loop: Each epoch proceeds as follows:

- The training data is shuffled to promote better generalization.
- Data is split into mini-batches of size B.
- For each batch:
  1. A forward pass is computed using `model.forward(batch_x)`.
  2. The binary cross-entropy loss is computed between predictions and labels.
  3. A backward pass is executed using the model's `backward_update_gradient` method, automatically propagating through all submodules.
  4. Gradients are cleared using `model.zero_grad()` before each update.
  5. Parameters are updated using `model.update_parameters(learning_rate)`.

Validation and Evaluation: At the end of each epoch, forward passes are performed on the validation set and loss is recorded. Final accuracy on the test and training sets is computed using `model.score()`.

3.3 Experiments

Hyperparameter Search

The model has two parameters that can change the performance of the model : *Learning Rate* and the *Hidden Dimension Size*.

To optimize the performance of the model built with the `Sequential` class, we used the `param_search_p3` function (`param_search.py`). This procedure explores key training hyperparameters to minimize validation loss and determine the best training configuration.

Joint Search: Learning Rate and Hidden Layer Size When `"LR_and_middleDim"`, the function performs a grid search over:

- Hidden layer size (`middle_dim`) in $\{1, 2, 3, 4, 6\}$,
- Learning rates in $\{0.09, 0.1, 0.3, 0.5\}$.

For each configuration, the model is trained for 1000 epochs and evaluated based on the average validation loss over the last 10 epochs. Results are stored in a matrix and visualized using a heatmap to identify the most promising configuration.

The best-performing pair of hyperparameters is used for a final 4000-epoch training session to determine the optimal stopping epoch.

3.4 Result

FIGURE 6: Section three - Heatmap for validation loss mean per learning-rate and hidden-dimension

The situation is the same described in the section two 2.4. The validation loss plot 7 with suboptimal parameters (namely 0.0087 loss) is more stable than the validation loss plot 8 with optimal hyperparameter (namely 0.0048 loss); though the test accuracies are the same (100%) in both cases.

This phenomenon again 2.4 shows that the current process used for the selection can lead to suboptimal solution. Thus, this process is yet to be improved to maximize the chance of obtaining the optimal hyperparameter.

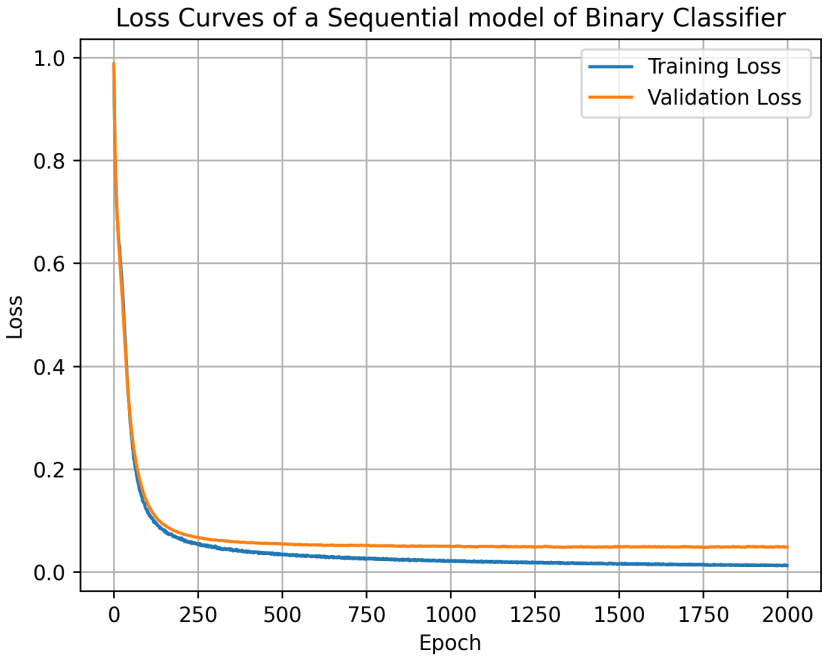

FIGURE 7: Section three - Training and validation loss over epochs - using optimal validation loss mean

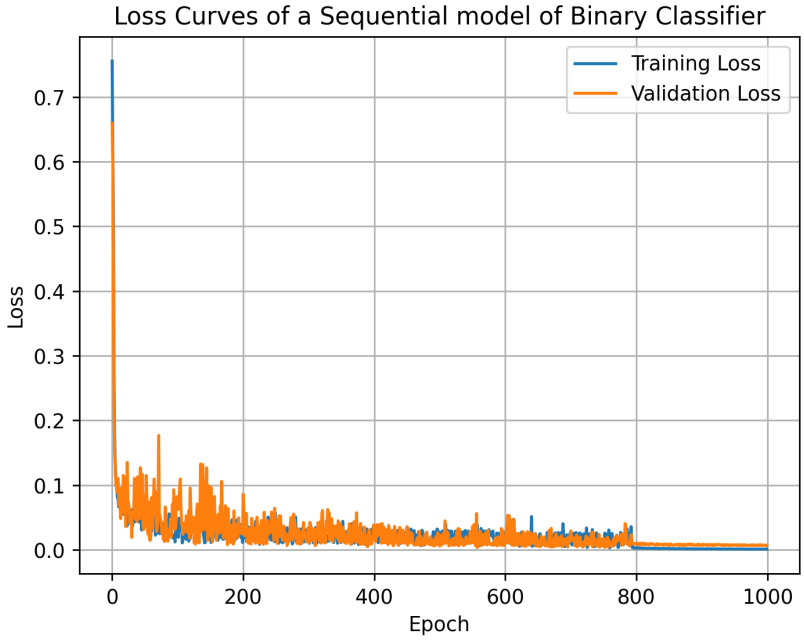

FIGURE 8: Section three - Training and validation loss over epochs - using optimal validation loss mean

4 Section Four - Nonlinear MLP Wrapped in a Sequence - Multi-class Classification

4.1 Objective

The objective of this part is to extend the neural network framework to handle multi-class classification problems.

This part demonstrates how modular components such as `Softmax` and cross-entropy can be integrated into the existing `Sequential` framework, enabling the network to solve classification tasks with more than two classes.Here is the scheme:

$$X \rightarrow Linear \rightarrow Tanh \rightarrow Linear \rightarrow Softmax \rightarrow y$$

4.2 Method

Dataset

The dataset used in this part is the same as the one described in Section 1.2, considering the fact that now the y is not binary but has 3 classes.

The NN Structure

The network is composed of two linear layers with a non-linear activation (`TanH`) in the hidden layer and a `Softmax` activation at the output layer to represent class probabilities. The output dimension equals the number of classes, and the target labels are encoded as one-hot vectors.

Training Procedure

The function `training_testing_sequential_multiclass` is used to train a neural network for multi-class classification using the `Sequential` architecture. The network outputs a vector of class scores which are converted into probabilities using the `Softmax` activation, and the model is trained using a cross-entropy loss.

Training Loop The training procedure closely follows the design used in previous parts, with the following specific features:

- The final activation is `Softmax`, followed by cross-entropy loss for stable training.
- The label vectors are one-hot encoded to match the network's output.
- Accuracy is computed using `argmax` over the predicted class probabilities.

Validation loss is computed at each epoch, and final classification accuracy is reported. Loss curves can be plotted to assess convergence behavior and generalization.

4.3 Experiments

Hyperparameter Search

The model has two parameters to be optimized : *Learning Rate* and the *Hidden Dimension Size*.

To optimize the multi-class classification model, we used the `param_search_p4` function (`param_search.py`). This function performs systematic searches over key hyperparameters, such as learning rate and the number of hidden units in the network, to identify the configuration that minimizes validation loss.

Grid Search over Learning Rate and Hidden Size When param `"LR_and_middleDim"` is used, a grid search is conducted over combinations of:

- Hidden dimensions: $\{2, 3, 4, 7, 11, 12\}$
- Learning rates: $\{0.005, 0.05, 0.08, 0.4, 0.6, 0.8\}$

For each pair, the model is trained for 1000 epochs and evaluated based on the average of the last 10 validation losses. The best configuration is selected and visualized using a heatmap. A final training is then performed using this configuration for 4000 epochs to determine the best epoch for early stopping.

4.4 Result

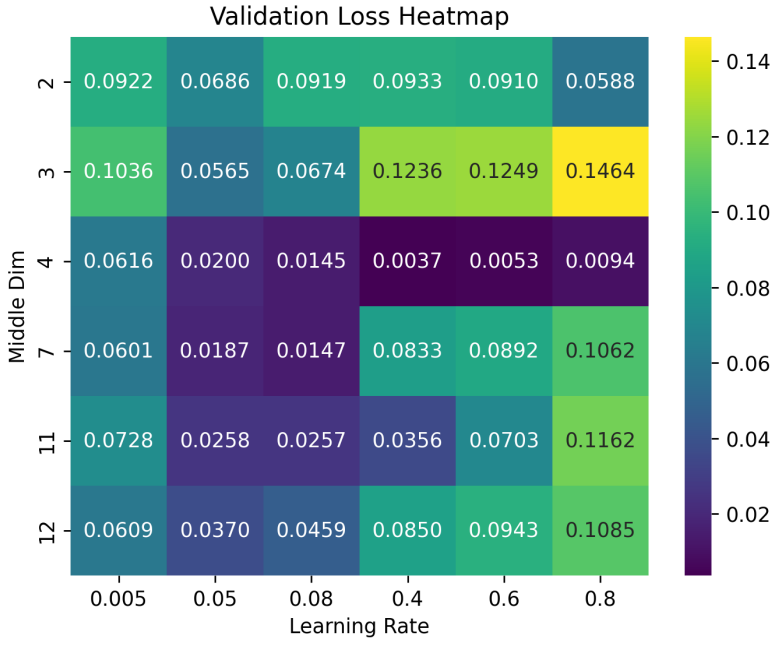

FIGURE 9: Section four - Heatmap for validation loss per learning-rate and hidden-dimension combination

For the case of multiclass classification, the mean validation loss criterion is not making issues as in the section two and three 2.4. The optimal hyperparameter gives a stable model with the best test accuracy.

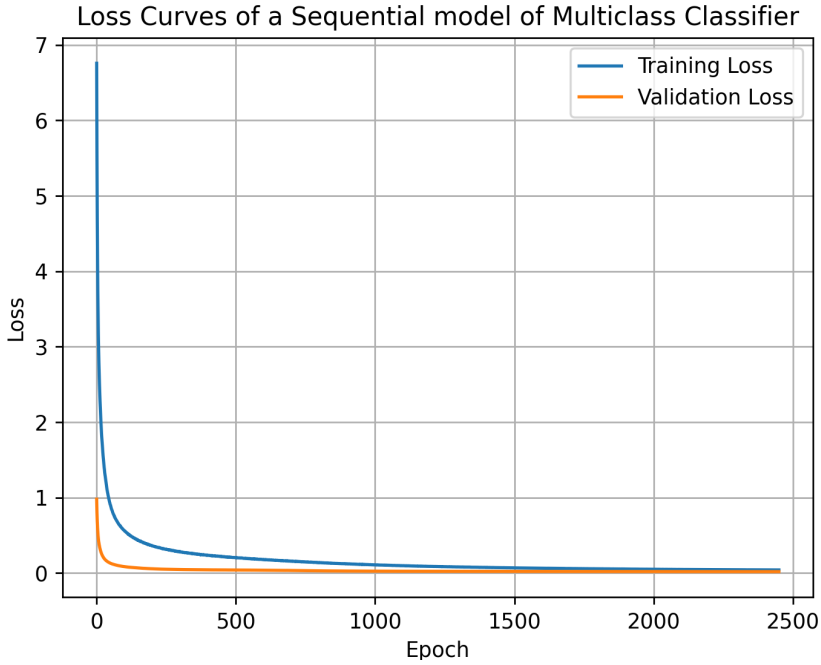

FIGURE 10: Section four - Training and validation loss over epochs

Therefore, as shown in Figure 10, the neural network is trained on a 3-class classification task. The model successfully captures the complexity of the data, achieving a training accuracy of 100.00% and a test accuracy of 96.67%.

5 Section Five - Auto-encoder

5.1 Objective

The objective of this part is to implement and experiment with an *autoencoder* containing two parts: *The encoder*, which compresses the input into a lower-dimensional latent representation, typically using linear layers and non-linear activations. *The decoder*, which reconstructs the original input from its latent representation, usually through a symmetric architecture.

5.2 Method

The NN Structure

The autoencoder is composed of two `Sequential` modules: an *encoder* and a *decoder*. The encoder compresses the input into a lower-dimensional latent representation, and the decoder attempts to reconstruct the original input from this compressed form.

Dataset

The experiments in this project were conducted using the **MNIST** dataset, a widely used benchmark for handwritten digit classification. It consists of grayscale images of digits (0 to 9), originally sized 28×28 pixels.

In our implementation, the dataset is loaded with function `loading_MNISTimages` (`data_utils.py`), with the following custom settings:

- All images are *resized to* 16×16 to reduce input dimensionality and training time.
- Each image is converted to a PyTorch tensor and then to a flattened NumPy array of shape $(256,)$ if `flatten=True`.
- A subset of the dataset is used, with a total of `100` images sampled from the training set for experimentation.

The 100 samples are split into: *70% training set*, *10% validation set*, and *20% test set*. Splitting is performed using `train_test_split` from `scikit-learn`, with a fixed random seed to ensure reproducibility.

Training Procedure

The function `training_testing_autoencoder` trains the autoencoder. The training is performed using mini-batch stochastic gradient descent and a reconstruction loss—specifically, binary cross-entropy (`BCELoss`) between the input and its reconstruction.

Training Loop Each epoch proceeds as follows:

- The training data is shuffled and divided into mini-batches.
- For each batch:
    1. A forward pass is performed through the encoder and decoder to produce a reconstruction.

2. The binary cross-entropy loss is computed between the original and reconstructed inputs.
3. A backward pass is performed:
   - The decoder is updated first using the reconstruction loss.
   - Gradients are then propagated to and through the encoder using the chain rule.
4. Parameters of both encoder and decoder are updated using the computed gradients.

- Validation loss is computed at the end of each epoch using forward passes only.

Evaluation: The evaluation of the autoencoder is carried out through both quantitative and qualitative means to assess its ability to learn meaningful representations.

1. Reconstruction Loss During training, the autoencoder is evaluated on its ability to reconstruct the input data, the images. This is to see how well the model could reconstruct the images. The loss curves are plotted over epochs to monitor convergence behavior and generalization.

2. Encoder Evaluation via k_NN The encoder is expected to capture semantic information in the latent space. To quantitatively evaluate the usefulness of this representation, we adopt the following strategy:

1. After training, the encoder is applied to the training set to obtain compressed latent representations.
2. A k_NN classifier (with $k = 5$) is trained on these encoded vectors using their true class labels.
3. The trained kNN classifier is then used to predict class labels for the validation set
4. The classification **accuracy** is used as a measure of class separability in the learned latent space.

This indirect evaluation reflects how well the encoder preserves discriminative information, even though it was not trained with supervision.

5.3 Experiments

Hyperparameter Search

The parameters of the model that can change the performance of the model are: *Learning Rate*, *Hidden Layer Size*, *Latent Space Size*.

To optimize the autoencoder's performance, we implemented a grid search strategy in the function `param_search_p5` (`param_search.py`). This procedure explores a 3D grid of combinations over three critical hyperparameters:

- *Learning rate* $\eta \in \{0.001, 0.01, 0.02, 0.03\}$
- *Hidden layer size (middle_dim)* $\in \{50, 100, 220, 240\}$
- *Latent space dimension* $\in \{50, 70, 90, 120\}$

Grid Search over Learning Rate, Hidden Size and Latent Dimension: For each combination of parameters, a new encoder and decoder are instantiated and trained for 300 epochs on the resized MNIST dataset.

Evaluation is based on two key metrics:

- The *validation accuracy* of a k-nearest neighbors classifier trained on the latent representations. (Selecting the parameters that yields the highest validation accuracy)
- In the case where the accuracies are the same, *mean validation loss*

So both metrics are used in tandem to determine the best configuration. The model achieving the highest validation accuracy is selected, with reconstruction loss serving as a tie-breaker when multiple models share the same accuracy.

After evaluating all combinations, the function reports the best learning rate, middle dimension, and latent dimension.

This method ensures that the selected architecture yields not only accurate but also compact and efficient latent representations, balancing generalization, expressivity, and training stability.

5.4 Result

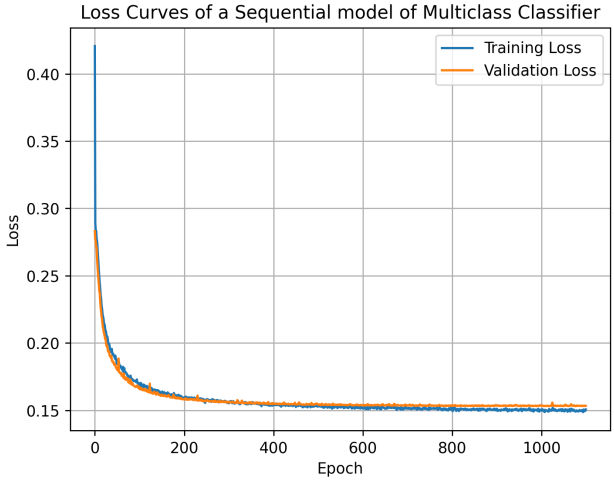

FIGURE 11: Section five - Training and validation loss over epochs

After performing hyperparameter search, the best parameters (learning rate, middle dimension and latent dimension size) are selected based on the highest validation accuracy of a knn classifier and lowest validation loss in the case of the same validation accuracy. Then, the model is trained with the selected hyperparameters.

To gain an intuitive understanding of how the model captures information about the digits, we visualize in Figure 12 the performance of a k-nearest neighbors classifier on 1,000 data points. These points are projected into a two-dimensional space using t-SNE after being compressed by the trained encoder.

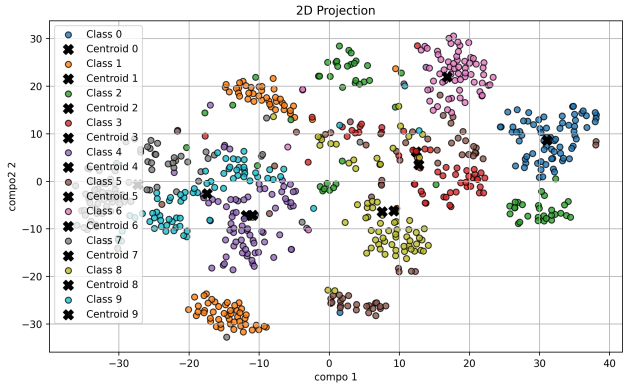

FIGURE 12: Section five - latent space projected on a 2D space

Figure 11 illustrates the learning process through training and validation loss curves over the epochs. As shown in the plot, the model trained on the MNIST dataset successfully converges. It effectively captures the complexity of the data, achieving a k-NN test accuracy of 91% in the latent space.

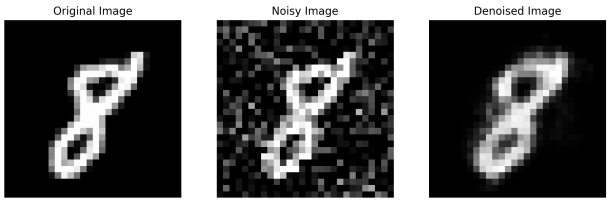

FIGURE 13: Section five - Adding noise and denoising it

To evaluate even more how well the model captures the essential information in the data, we feed the autoencoder a noisy image and assess its ability to reconstruct a clean version. To do this, we add noise to a randomly selected test image and then use the trained model to denoise it. As illustrated in Figure 13, the model successfully reconstructs the denoised image.

Image Generation

The trained autoencoder has learned to capture the essential features of the MNIST dataset and represent them in a compressed latent space. As shown in Figure 12, the centroids corresponding to each digit class are marked. For some digits, the latent representations are tightly clustered around their respective centroids, while for others, the samples are more dispersed. As a result, certain centroids do not fall within the densest regions of their corresponding digit clusters.

FIGURE 14: Section five - 8 generated samples from 8 randomly selected points in the latent space

The latent space enables the model to generate new images (digits), as it retains the essential information about each class. To generate a new image, one can sample a point from the latent space and pass it through the decoder to reconstruct a corresponding image.

For instance, in Figure 14, we sample random points from the latent space and decode them into images. These generated images do not resemble meaningful digits, as the sampled points do not originate from the dense regions—i.e., the latent clusters—corresponding to actual digits.

Alternatively, in Figure 16, we sample points uniformly along the latent space trajectory between the centroids of digits 8 and 9.

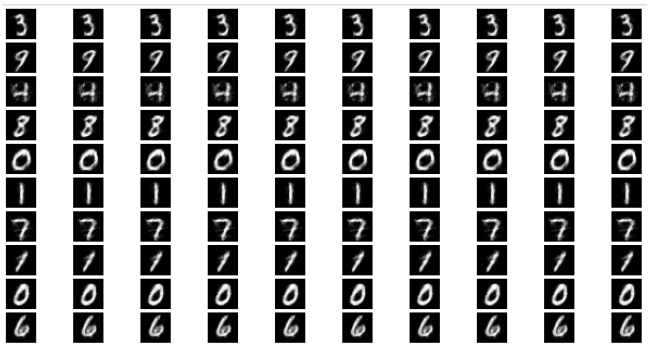

FIGURE 15: Section five - 10 generated samples from each of the 10 centroids in the latent space

On the other hand, Figure 15 shows 10 generated images for each of the 10 centroids in the latent space. Despite this more structured sampling, certain digits—such as 2—remain indistinct. This phenomenon can be explained by observing Figure 12, where the latent representations of digit 2 appear dispersed rather than concentrated, making it harder for the model to generate consistent representations.

FIGURE 16: Section five - generated samples uniformly drawn from latent between the digit 3 and 9

Summary

In summary, this project demonstrates how modular neural network components can be implemented and extended to solve increasingly complex tasks. Through manual training, evaluation, and hyperparameter tuning, we developed models aimed at capturing the complexity of the dataset. Starting with a simple perceptron, we observed that a linear model was insufficient to fully classify the binary dataset. By introducing non-linearity, the network was able to capture the underlying structure and achieve perfect classification. For a multi-class dataset, the non-linear model reached up to 96% accuracy. Incorporating a `Sequential` structure into the model significantly simplified the implementation and training pipeline. Although a hyperparameter search was conducted to optimize model performance, we found that the selection criterion—based on average validation loss—was not always reliable, as some suboptimal configurations yielded better test accuracy. Finally, the autoencoder model demonstrated the capability of unsupervised learning to extract and exploit meaningful latent representations for both reconstruction and data generation.