Report on CS 419 M Project: Neural Network from Scratch

By Justin Queiros de Oliveira, 6 March 2025 ⋆, *IIT Bombay*

(CS 419 M: Neural Network from Scratch)

A B S T R A C T

This report describes how I developed a network to classify MNIST images from the ground up using NumPy without relying on pre existing libraries.The structure allows for settings and efficient computation, for both backward movements along with optimization techniques implemented without off the shelf tools. Experimental outcomes show the models success, in handling class categorization through important performance indicators and visual aids that showcase the training journey.

Keywords: MNIST Classification, Neuronal Network, Numpy

1. Introduction

Constructing a neural network from scratch for MNIST classification provides a valuable opportunity to deeply explore the fundamentals of machine learning. By relying solely on NumPy, I implemented critical components such as forward and backward computations, optimization techniques, and a modular, parallelized design, ensuring both computational efficiency and adaptability. To further enhance the solution, I introduced data augmentation, applying transformations to the training dataset to increase diversity and improve the model's generalization capabilities. Additionally, I incorporated dropout as a regularization technique during training, deactivating neurons to prevent overfitting and improve robustness. These methods align with and address Requirement 15, which focuses on exploring techniques to boost model performance.

In this report, I will detail the architectural decisions behind the network design, including the selection of layers, their specific roles, and the conventions used for naming parameters and variables to maintain clarity throughout the codebase. The mathematical foundations of the forward pass will be explained, illustrating how data flows through the network to generate predictions, alongside a discussion of the gradient calculation equations that enable backpropagation and weight updates. These theoretical aspects are supported by experimental results and insights into the training process, seamlessly integrating the theoretical and practical components of the project.

⋆ Corresponding author. e-mail: queirosdeoliveira.justin@gmail.com

2. Naming Conventions

In this project, consistent and meaningful naming conventions have been adopted to enhance code readability, maintainability, and clarity. The following conventions were applied:

Parameters and Hyperparameters are named using lowercase, snake_case formatting. This convention improves readability and distinguishes these variables from class names or functions. Examples include `input_size`, which represents the dimensionality of input data; `hidden_layers`, a list specifying the sizes of the hidden layers in the network; `learning_rate`, the step size used for gradient updates; `num_epochs`, the number of epochs for training; and `batch_size`, the number of samples per training batch. **General-purpose Variables** follow similar snake_case formatting for clarity. For instance, `X_train`, `X_val`, and `X_test` represent the training, validation, and test datasets, respectively. Similarly, `y_train`, `y_val`, and `y_test` denote the corresponding label sets. Gradients of weights and biases during backpropagation are denoted as `dW` and `db`. The activations at each layer of the network during the forward pass are stored in a list named `activations`, while `pre_activations` holds the weighted sums (pre-activation values) at each layer. **Layer-specific Variables and Functions** are named descriptively, often reflecting their purpose or computation. For instance, `weights` and `biases` represent the learnable parameters of the network, organized as lists for each layer. Functions such as `adam_update` and `rmsprop_update` implement specific optimization algorithms. The `forward` and `backward` functions handle the core forward and backward passes through the network, while `apply_dropout` applies dropout regularization during training. **Activation Functions** and their deriva-

tives are named using lowercase, snake_case formatting, with the prefix indicating the function type. For example, `relu` and `relu_derivative` represent the ReLU function and its derivative. Similarly, `sigmoid` and `sigmoid_derivative` denote the sigmoid function and its derivative, while `tanh` and `tanh_derivative` represent the hyperbolic tangent function and its derivative. The **Global Configuration** is represented by the `config` dictionary, which encapsulates all hyperparameters and boolean flags in a centralized, structured format. Each key is named descriptively to indicate its role. Examples include `use_data_augmentation`, `dropout_rate`, and `random_state`.

3. Architecture

The code base of my project is built to be flexible and adaptable, through its design and customizable features. The structure is specified using a configuration dictionary "*config*" to maintain reproducibility and facilitate modifications across various parameters, which will be argued in the following sections.

The **Network Structure** consists as usual for the MNIST dataset of an input layer with 784 features, corresponding to the flattened 28×28 MNIST images. I applied two fully connected hidden layers with 128 and 64 neurons respectively, followed by an output layer with 10 neurons, one for each digit class. The choice for these particular hidden layer sizes was made initially through research and was validated by my tests later on. The output layer uses the softmax activation function to produce class probabilities, while the hidden layers utilize activation functions specified in the configuration, such as **ReLU**, **Sigmoid**, or **Tanh**, which can again be elected freely in the config dictionary. Considering the **Weight Initialization**, weights are set up by using an adjusted normal distribution to ensure stability in the training process; biases are set to zero during initialization to avoid problems, like disappearing or overly increasing gradients.

The **Training Process** is implemented with a loop over a fixed number of epochs, during which the network iteratively learns by minimizing the loss function. At the beginning of each epoch, the training dataset is shuffled to ensure the network does not learn patterns tied to the data order. The data is then divided into mini-batches of a configurable size, enabling efficient training using batch gradient descent.

During each batch iteration, a **Forward Pass** propagates the input data through the layers of the network to compute predictions. For an input vector X, the propagation starts with the first layer. The weighted sum of inputs and biases is computed as:

$$Z^{(1)} = XW^{(1)} + b^{(1)}$$

This pre-activation $Z^{(1)}$ is passed through either one of our specified activation functions, to produce the output of the first hidden layer:

$$A^{(1)} = \sigma(Z^{(1)})$$

where σ represents the activation function. The output of the first layer $A^{(1)}$ then serves as the input to the second hidden layer, where the process is repeated:

$$Z^{(2)} = A^{(1)}W^{(2)} + b^{(2)}, \quad A^{(2)} = \sigma(Z^{(2)})$$

This propagation continues layer by layer until the output layer is reached. In the final layer, the network produces logits $Z^{(L)}$, which are converted into class probabilities using the softmax function:

$$A^{(L)} = \text{softmax}(Z^{(L)}) = \frac{\exp(Z^{(L)})}{\sum_j \exp(Z_j^{(L)})}$$

Here, $A^{(L)}$ represents the predicted probabilities for the classes. Optionally, dropout is applied to the activations of the hidden layers to randomly deactivate neurons:

$$A^{(l)}_{\text{dropout}} = \frac{A^{(l)} \cdot \text{Mask}}{1 - p}$$

where Mask is a binary mask and p is the dropout rate. This regularization technique helps reduce overfitting by encouraging redundancy in the network's learned features. Once the **Forward Pass** is complete, the **Loss** for the current batch is calculated using the cross-entropy loss function:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_{i,k} \log(A_{i,k}^{(L)})$$

where m is the number of samples in the batch, K is the number of classes, $y_{i,k}$ is the true label, and $A_{i,k}^{(L)}$ is the predicted probability for class k.

Furthermore, the **Backward Pass** propagates the gradients through the network using the chain rule of calculus. Starting from the output layer, the gradient of the loss with respect to the activations is computed as:

$$\delta^{(L)} = A^{(L)} - Y$$

The gradients of the weights and biases for layer l are:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{1}{m} A^{(l-1)^\top} \delta^{(l)}, \quad \frac{\partial \mathcal{L}}{\partial b^{(l)}} = \frac{1}{m} \sum_{i=1}^{m} \delta^{(l)}$$

For hidden layers, the gradients are propagated backward as:

$$\delta^{(l)} = \delta^{(l+1)} W^{(l+1)^\top} \odot \sigma'(Z^{(l)})$$

where $\odot$ denotes element-wise multiplication and $\sigma'(Z^{(l)})$ is the derivative of the activation function.

The computed gradients are then used to update the weights and biases. Additionally this process can be further enhanced by enabling the optimization algorithms **Adam** or **RMSprop** in the *config*, which have been implemented to ensure efficient

gradient updates and robust convergence:

Adam Optimizer: Adam combines the advantages of both momentum and RMSprop, using first- and second-moment estimates of the gradients. The updates for the weights W are computed as follows:

$$m_w = \beta_1 m_w + (1-\beta_1)\frac{\partial \mathcal{L}}{\partial W}, \quad v_w = \beta_2 v_w + (1-\beta_2)\left(\frac{\partial \mathcal{L}}{\partial W}\right)^2$$

$$W = W - \frac{\alpha m_w}{\sqrt{v_w} + \epsilon}$$

where β_1 and β_2 control the decay rates for the moment estimates, α is the learning rate, and ϵ prevents division by zero. This optimizer provides adaptive learning rates and is well-suited for problems with sparse gradients.

RMSprop Optimizer: RMSprop focuses on scaling the learning rate based on the magnitude of recent gradients, which helps with stabilization in non-stationary objectives. The updates for the weights are:

$$s_w = \beta s_w + (1-\beta)\left(\frac{\partial \mathcal{L}}{\partial W}\right)^2$$

$$W = W - \frac{\alpha \frac{\partial \mathcal{L}}{\partial W}}{\sqrt{s_w} + \epsilon}$$

where β is the decay rate, s_w is the running average of the squared gradients, α is the learning rate, and ϵ prevents division by zero. RMSprop works particularly well in scenarios where the scale of gradients varies significantly.

After processing all batches in an epoch, the **Model's Performance** is evaluated on the validation dataset by computing the validation loss and accuracy. Early stopping is implemented to terminate training if the validation loss does not improve for a specified number of consecutive epochs, preventing overfitting and saving computational resources.

Last but not least, for the **Evaluation and Visualization** of the model's performance, there are metrics such as accuracy and loss, visualized across epochs for both training and validation datasets, helping to identify potential issues such as underfitting or overfitting. Additionally, predictions on test data are evaluated and randomly visualized to provide insights into the model's performance and areas for potential improvement.

4. Evaluation

The overall outcome of the project is surprisingly good. The accuracy remained stable around 96–97% for all possible combinations of activation functions for two hidden layers, as well as for batch sizes of 16, 32, and 64. The following subsections present the results in detail.

[ht]

Function 1	Function 2	Accuracy
relu	tanh	0.9772
sigmoid	sigmoid	0.9753
relu	relu	0.9753
sigmoid	tanh	0.9739
relu	sigmoid	0.9734
tanh	sigmoid	0.9734
tanh	relu	0.9729
sigmoid	relu	0.9692
tanh	tanh	0.9716
Average		**0.9736**

Table 1. Accuracy for Different Activation Function Combinations

System	Accuracy
standard	0.9772
data augmentation	0.9769
dropout	0.9669
data augmentation & dropout	0.9615

Table 2. Comparison of Standard Training and Enhanced Techniques

The first table shows all **Combinations of Activation Functions** and their accuracies, sorted in descending order.
The second table presents the comparison between the **Standard Training Procedure** and tests with either **Data Augmentation**, **Dropout**, or both enabled. Sadly, further improvements were not achieved, likely due to the chosen parameters. Testing different parameters took significant time due to limited computational resources.

The following table lists the **Configuration Parameters** used for the experiments, including their values and descriptions.

This wraps up the theoretical part of my assignment. Thank you for the opportunity to be a part of this course!

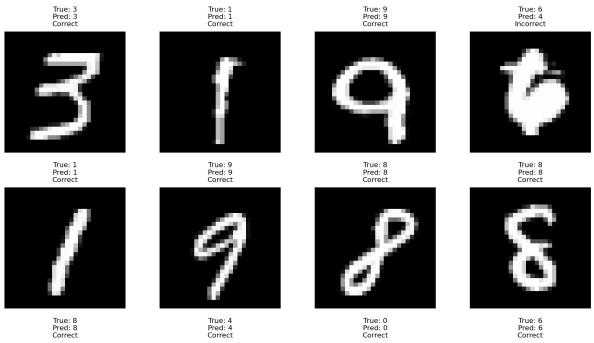

Fig. 1. 8 randomly test cases plotted with their label and their assigned output

Parameter	Value	Description
input_size	784	Dimensionality of input data
hidden_layers	[128, 64]	Sizes of the two hidden layers
output_size	10	Number of output classes
activation_functions	['relu', 'tanh']	Activation functions used
learning_rate	0.001	Step size for gradient updates
num_epochs	15	Number of training epochs
batch_size	16	Size of each training batch
optimizer	'adam'	Optimizer used for training
beta1	0.9	Beta1 for Adam optimizer
beta2	0.999	Beta2 for Adam optimizer
epsilon	1e-8	Small value to prevent division by zero
decay_rate	0.9	Decay rate for RMSProp optimizer
test_size	0.2	Fraction of dataset used for testing
val_size	0.25	Fraction of training data used for validation
random_state	42	Seed for reproducibility
patience	5	Patience for early stopping
use_data_augmentation	True	Whether to use data augmentation
use_dropout	True	Whether to use dropout regularization
dropout_rate	0.1	Dropout probability

Table 3. Configuration Parameters

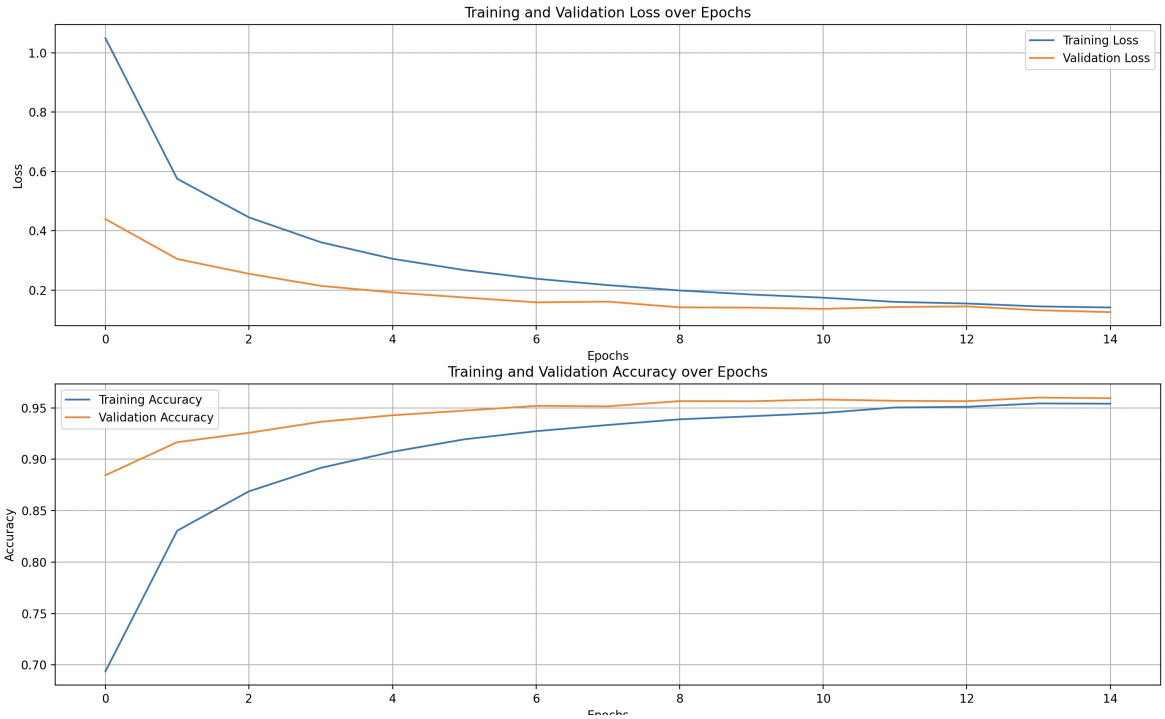

Fig. 2. Training and Validation Loss and Accuracy over Epochs for 'relu' and 'tanh' in order as activation functions.