Neural Network from scratch

The following script represents my 4-day journey of making a neural network from scratch without any previous knowledge. I try to represent my thought process and keep a record of what I am doing at each step, also my thoughts and some recommendations of what could be improved or changed.

Project Description

The main concept of the project is a neural network trained and tested on the MNIST dataset for recognizing handwritten numbers. The dataset can be found and downloaded at:

MNIST-DATASET

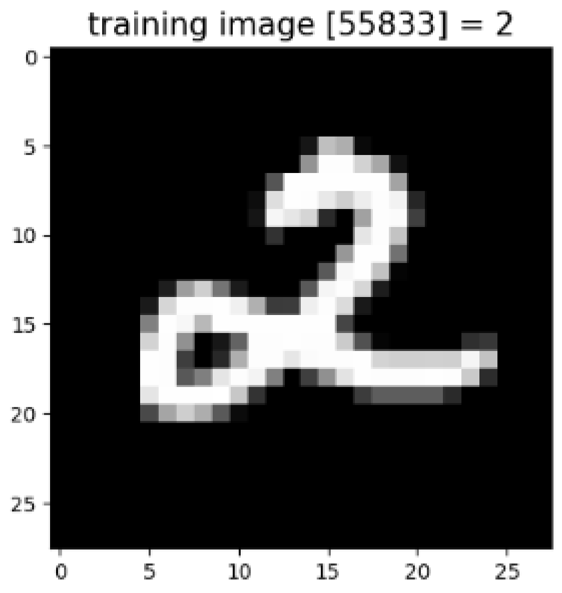
Author: Krzysztof Nowak

Part One – dataset preparation

1. Dataset setup

Starting with Kaggle’s MNIST scripts, I verified the dataset integrity. The original files split into 60,000 training and 10,000 test images—a standard ratio to prevent overfitting. Each 28x28 grayscale image is stored as a 3D array (samples × width × height), but most fully connected layers expect 2D input (samples × flattened_features).

A sample image representing number 2 looks like this:



Img. Train image representing number 2

2. Normalization

Pixel values are 8-bit integers [0, 255]. To stabilize gradient calculations during backpropagation, I normalized them to [0, 1] by dividing by 255. This prevents large input values from causing erratic weight updates, which could slow training or destabilize the loss landscape.

3. Flattening

I reshaped the 3D arrays (e.g., 60,000×28×28) into 2D (60,000×784). Each 28x28 image becomes a 784-pixel vector—essentially ‘unfolding’ the grid into a single row. While this loses spatial locality it’s necessary for initial dense layer compatibility in our neural network.

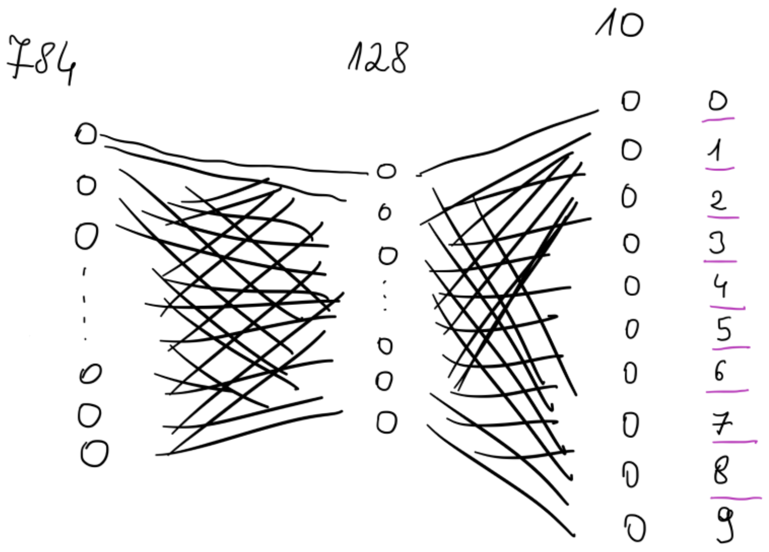
4. Label encoding

The labels (y_train/y_test) are integers 0-9 but neural networks output probabilities via softmax. To align labels with this probabilistic framework, I applied one-hot encoding. For example, a label ‘5’ becomes [0,0,0,0,0,1,0,0,0,0] - a 10-dimensional vector where the 6th index (zero-based) is 1. This directly matches the 10 output neurons I’ll configure in the final layer. After this step our y_test becomes (60000, 10) vector and y_train becomes (10000, 10) vector. In summary we normalize the data we will be comparing later on by making sure that when we get the output from our neural network which is 10 values stored in a vector, we can easily compare it with the hot encoded y label vector.

Part Two – Neural network

1. Neural network design

I will be aiming for a simple yet effective design composed of input layer then one hidden layer and output layer. The hidden layer will have 128 neurons. My design looks like this:



Img. Neural network design

2. Weights and bias initialization

Having the design we can delve into implementation. Neural networks learn through adjustable parameters called weights and biases. These parameters are fundamental to what we call "AI learning" - the process of calculating and fine-tuning these values to achieve desired outputs.

Weights represent the strength of connections between neurons. Each connection has an assigned weight that determines how much influence one neuron's output has on the next neuron's input. What helped me was to think of weights as determining the "importance" of each connection in the network.

A **bias** is an additional parameter added to each neuron that allows the network to learn more effectively. It shifts the neuron's activation function horizontally, enabling the network to represent patterns that don't necessarily pass through the origin. This is crucial for proper model fitting.

There are several established methods for initializing weights, with two particularly common approaches:

- *He initialization (Best for ReLU)*
- *Xavier initialization*

In my approach we will be using He initialization, but I implemented other as well to test the performance on them.

In this implementation, we initialize all biases to zero. While some approaches use small random values, zero initialization for biases is a common and effective practice. The He initialization formula is given as :

$$W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$

We approximate weights matrix with gaussian distribution of mean value 0 and variance equal to **2/n^l** where **n^l** is a number of neurons in layer l.

For a network with:

- Input layer: 784 neurons (28×28 input image)
- Hidden layer: 128 neurons
- Output layer: 10 neurons (for digit classification)

The initialization requires two weight matrices (W1, W2) and two bias vectors (b1, b2):

```
# Weight initialization using He initialization

W1 = np.random.randn(input_size, hidden_size) *
np.sqrt(2/input_size)

W2 = np.random.randn(hidden_size, output_size) *
np.sqrt(2/hidden_size)


# Bias initialization
```

```
b1 = np.zeros((1, hidden_size))

b2 = np.zeros((1, output_size))
```

3. Activation function

Activation functions are mathematical operations applied to a neuron's output, serving two critical purposes:

1. Introducing non-linearity into the model
2. Enabling the network to learn and represent complex patterns in data

While there are numerous activation functions available (including ReLU, Leaky ReLU, Tanh, and Softmax), this implementation focuses on two specific functions:

- ReLU is commonly used in modern networks and is best for hidden layers. It works by ‘zeroing’ negative values .

$$\mathrm{ReLU}(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0, \\ 0 & x \leq 0 \end{cases}$$

Img. ReLU function
Source: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

- Softmax converts a vector of n real numbers into a probability distribution of n possible outcomes. It is commonly used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

$$\mathrm{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \qquad \mathrm{softmax}(x)_i = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

Img. Softmax (left), Numerically stable softmax (right)

Apart from normal softmax we use a numerically stable softmax to ensure that all exponentiated values will be between 0 and 1 , since the value in the exponent is always negative. This prevents overflow errors.

This combination of ReLU for hidden layers and stabilized softmax for the output layer is particularly effective for classification tasks, where we need both non-linear feature extraction (ReLU) and probabilistic output interpretation (softmax).

4. Forward propagation

After we initialized weights and biases and chose activation functions for each layer, the next step would be to define the propagation function. Lets' define it.

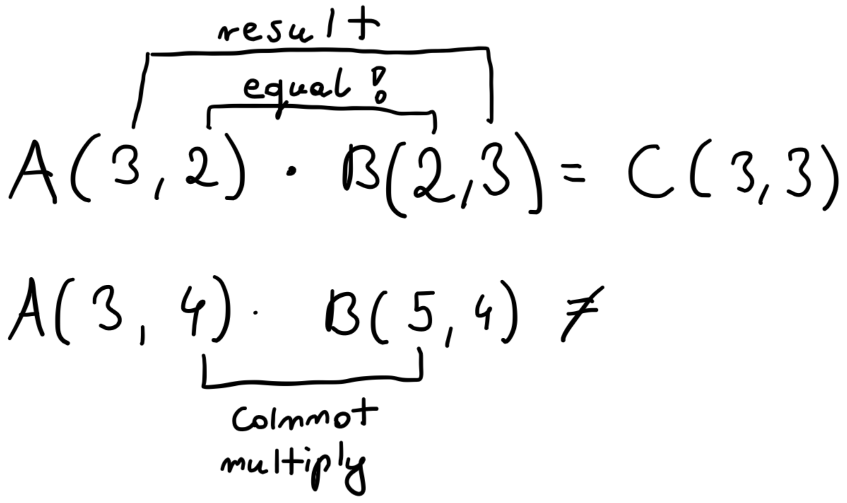
Forward propagation is the process of moving input data through a neural network to generate predictions. It consists of sequential matrix operations and activation functions applied layer by layer, from input to output.

Forward propagation is described by linear combination:

$$Z = W \times X + b$$

- Z: Layer output before activation
- W: Weight matrix
- X: Input matrix
- b: Bias vector

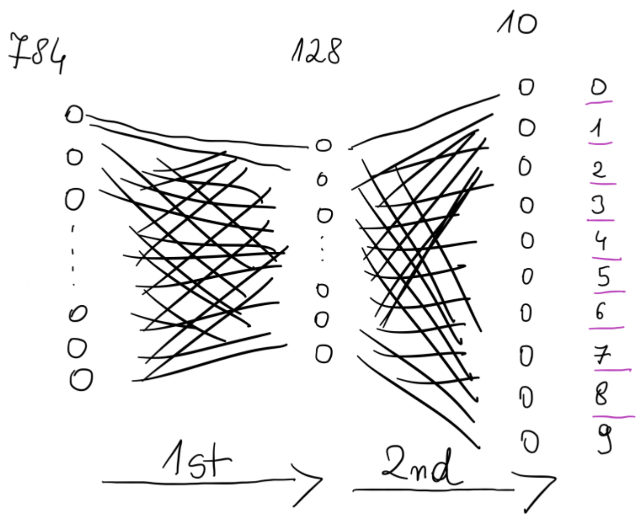
Matrix Dimensions and Transposition – Recap



Img. Matrices multiplication recap

For matrix multiplication to be valid, the dimensions must be compatible:

- If matrix A is (m × n)
- And matrix B is (n × p)
- Then A × B results in a matrix of size (m × p)
- The inner dimensions (n) must match



Img. 2 forward passes structure

First Forward Pass

$Z1 = W1^T \times X + b1$

$A1 = \text{ReLU}(Z1)$

Where:

- $W1^T$: Transposed weight matrix for first layer
- X: Input data
- b1: First layer bias
- A1: First layer activation output

Second Forward Pass

$Z2 = W2^T \times A1 + b2$

$A2 = \text{Softmax}(Z2)$

Where:

- $W2^T$: Transposed weight matrix for second layer
- A1: Output from first layer
- b2: Second layer bias
- A2: Final network output (probabilities)

5. Loss function

A loss function serves as a "performance metric" or "error measure" that tells us how well our neural network is performing. We can think of it as a report card that grades how close the network's predictions are to the actual correct answers.

- Without a loss function, we wouldn't have a numerical way to measure how "wrong" our predictions are
- It converts the concept of "correctness" into a measurable number
- The loss function provides a mathematical "landscape" that the network can use to improve
- By calculating the gradient of the loss, the network knows which direction to adjust its weights to get better results
- Lower loss = better predictions

In my project I will be using Cross-Entropy loss function.

$$\text{cross-entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k} t_{i,j}\log(p_{i,j})$$

Img. Cross-entropy function formula

Where:

- N – number of training examples
- K – number of classes
- ti,j – true value (0 or 1) for example I and class j
- pij – predicted probability for example I being in class j

6. Backward propagation

Backpropagation is a fundamental algorithm in training neural networks that calculates gradients of the loss function with respect to the network's weights and biases. It's essentially the "learning" part of the network's learning process.

The Learning Process

1. **Forward Propagation**: Network makes predictions
2. **Loss Calculation**: Measure prediction error
3. **Backpropagation**: Calculate gradients
4. **Weight Update**: Adjust parameters using gradients

Backpropagation efficiently computes how much each weight and bias contributed to the error, allowing us to:

- Determine which parameters need adjustment
- Calculate the direction and magnitude of necessary adjustments
- Distribute error responsibility through the network layers

The core of backpropagation relies on the **chain rule** from calculus. For a neural network with loss L, we need to find **∂L/∂W** and **∂L/∂b** for each layer. The derived equations are listed below:

```
m = x_train.shape[0] # batch size
dz2 = a2 - y_true
dW2 = np.dot(a1.T, dz2) / m
db2 = np.sum(dz2, axis=0, keepdims=True)

dz1 = np.dot(dz2, W2 * (z1 > 0)
dW1 = np.dot(x_train.T, dz1) / m
db1 = np.sum(dz1, axis=0, keepdims=True)
```

7. **Parameters update**

The parameter update step is the final stage in the training process where we actually "learn" by adjusting our network's weights and biases based on the computed gradients.

The general formula for gradient descent is:

parameter = parameter - learning_rate × gradient

- **Parameter**: The value we're trying to optimize (weights or biases)
- **Learning Rate**: How big of a step we take in the direction of the gradient
- **Gradient**: The direction and magnitude of the steepest increase in error

Gradients point in the direction of steepest increase, subtracting gradients moves parameters in direction of steepest decrease, this helps minimize the loss function. The learning rate controls the step size of parameter updates, too large will overshoot minimum and can lead to oscillations and too small will make the learning very slow.

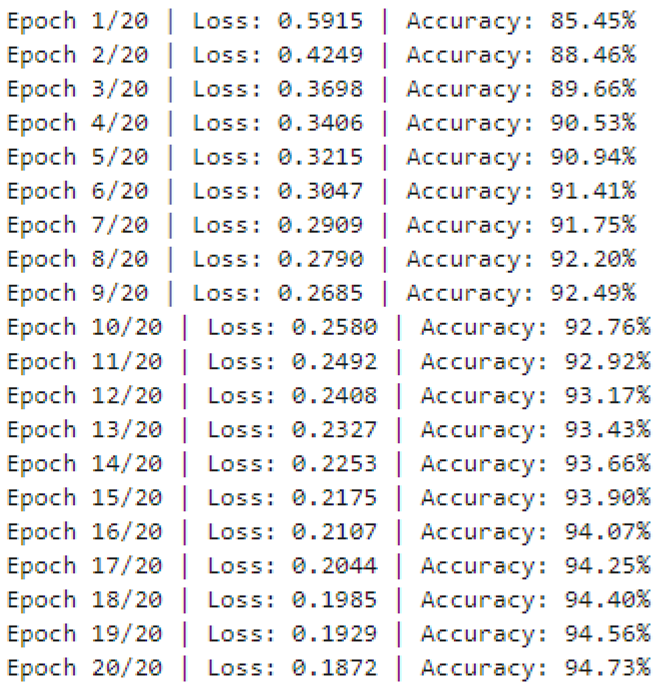
8. Training

The neural network training process begins once all initial setup is complete. At its core, the training is structured around two fundamental concepts: epochs and batches. An epoch represents a complete pass through the entire dataset, while batches allow for parallel processing of smaller subsets of data, optimizing computational efficiency.

To enhance the model's learning capability, I implemented dataset shuffling through random permutation of both the dataset and its corresponding labels. This crucial technique prevents the model from learning fixed positions within the data sequence, which could otherwise lead to position-based predictions rather than true pattern recognition.

The implementation uses Mini-batch Stochastic Gradient Descent as the primary training algorithm. During each batch iteration, the process flows through three main steps: forward propagation to compute activation matrices, back propagation to calculate gradients, and parameter updates based on these calculations. While there exists an alternative approach called Stochastic Gradient Descent that processes the entire dataset at once per epoch, I opted for the mini-batch method due to its better balance of computational efficiency and accuracy. The standard SGD approach, while potentially more accurate, requires significantly more epochs and computational time, and thus remains an unexplored option in my current implementation.

The evaluation process occurs after each epoch completion. It begins with a forward pass through the network to generate outputs, followed by loss computation. To determine specific predictions, I employed an argmax function with axis=1, which identifies the highest probability class by selecting the maximum value in each row vector. The final accuracy measurement compares these predictions against the true labels, essentially calculating the proportion of correct predictions or true positives in the dataset. This comprehensive process provides a clear metric for assessing the model's performance and learning progress.

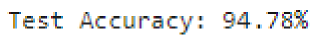


```
Epoch 1/20 | Loss: 0.5915 | Accuracy: 85.45%
Epoch 2/20 | Loss: 0.4249 | Accuracy: 88.46%
Epoch 3/20 | Loss: 0.3698 | Accuracy: 89.66%
Epoch 4/20 | Loss: 0.3406 | Accuracy: 90.53%
Epoch 5/20 | Loss: 0.3215 | Accuracy: 90.94%
Epoch 6/20 | Loss: 0.3047 | Accuracy: 91.41%
Epoch 7/20 | Loss: 0.2909 | Accuracy: 91.75%
Epoch 8/20 | Loss: 0.2790 | Accuracy: 92.20%
Epoch 9/20 | Loss: 0.2685 | Accuracy: 92.49%
Epoch 10/20 | Loss: 0.2580 | Accuracy: 92.76%
Epoch 11/20 | Loss: 0.2492 | Accuracy: 92.92%
Epoch 12/20 | Loss: 0.2408 | Accuracy: 93.17%
Epoch 13/20 | Loss: 0.2327 | Accuracy: 93.43%
Epoch 14/20 | Loss: 0.2253 | Accuracy: 93.66%
Epoch 15/20 | Loss: 0.2175 | Accuracy: 93.90%
Epoch 16/20 | Loss: 0.2107 | Accuracy: 94.07%
Epoch 17/20 | Loss: 0.2044 | Accuracy: 94.25%
Epoch 18/20 | Loss: 0.1985 | Accuracy: 94.40%
Epoch 19/20 | Loss: 0.1929 | Accuracy: 94.56%
Epoch 20/20 | Loss: 0.1872 | Accuracy: 94.73%
```

Img. Training step

9. Evaluation

After completing the training process, the model's performance must be evaluated on unseen data to demonstrate its true predictive capabilities. The final evaluation function implements a forward pass using the optimized parameters learned during training. This pass processes the test dataset through the trained network architecture, generating predictions that are then compared against the true labels. The accuracy calculation follows the same methodology used during training: comparing predicted classes (determined by the argmax function) against actual labels to compute the proportion of correct predictions. This final score serves as a crucial metric of the model's generalization ability, indicating how well it performs on data it hasn't encountered during the training phase.

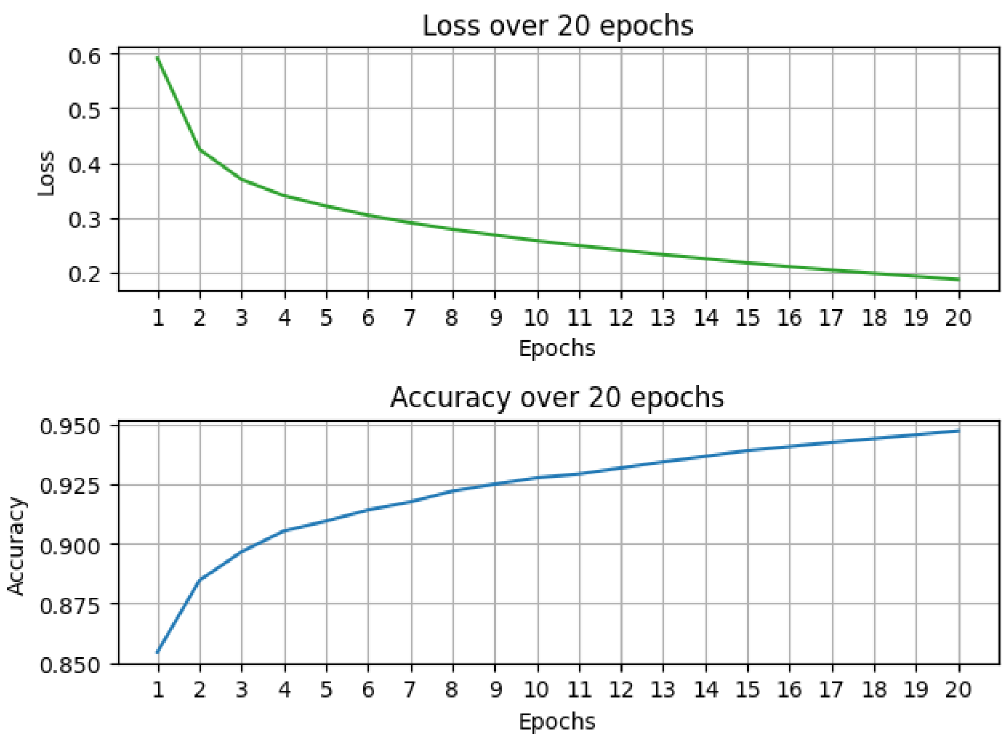


```
Test Accuracy: 94.78%
```

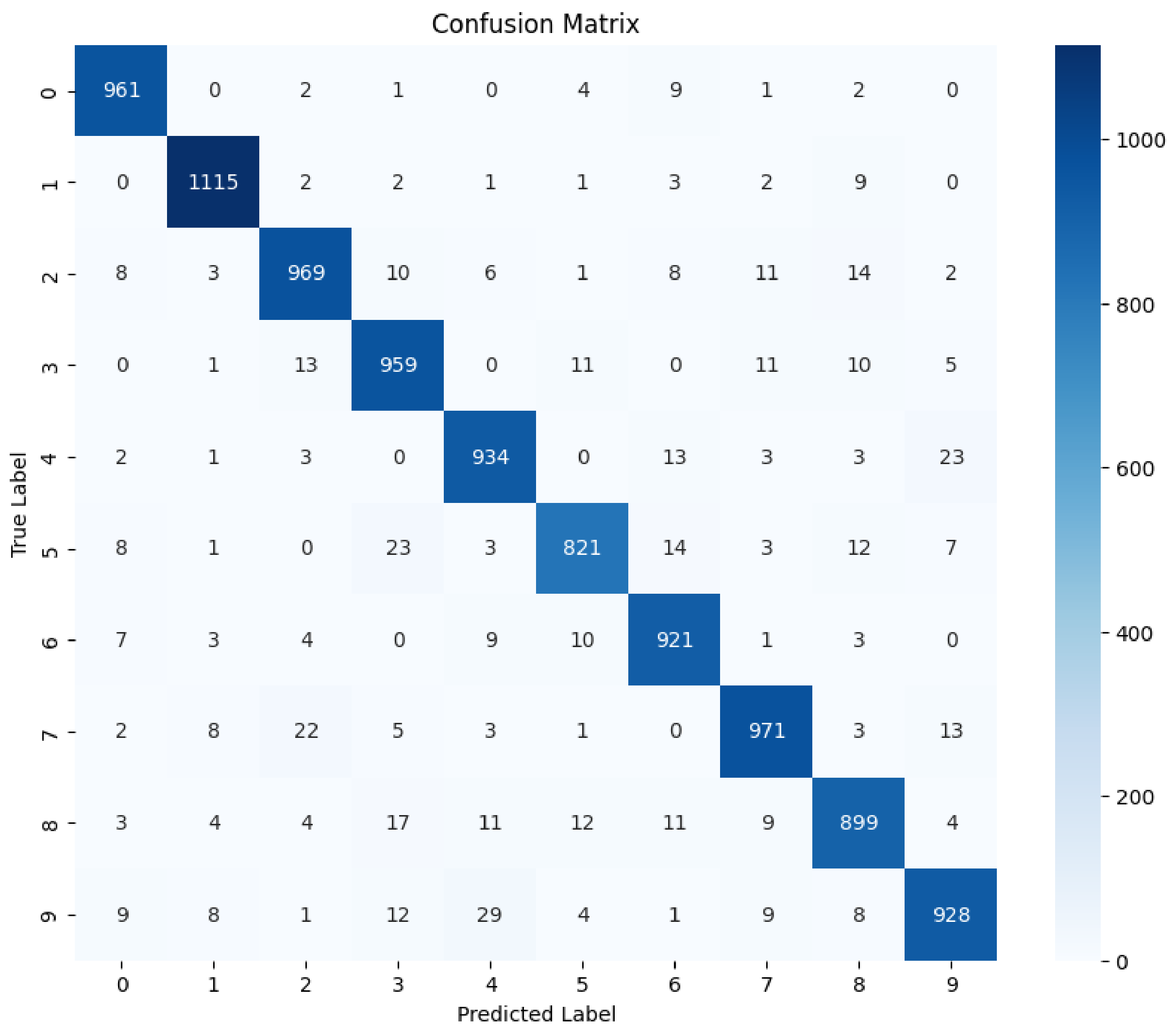
Img. Accuracy of the model

This evaluation step is essential as it validates that the model hasn't simply memorized the training data but has learned meaningful patterns that can be applied to new, unseen examples. The final accuracy score provides a concrete measure of the model's practical utility and effectiveness.

I though it would be nice to visualize some of the parameters that we used so I created a loss and accuracy plots showing clearly the learning capabilities of the model, loss goes down while accuracy goes up each epoch. The second graph is a 10 class confusion matrix showing how many times the model predicted correctly and how many times it predicted falsely numbers.



Img. Loss and accuracy over 20 epochs



Confusion Matrix

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	961	0	2	1	0	4	9	1	2	0
1	0	1115	2	2	1	1	3	2	9	0
2	8	3	969	10	6	1	8	11	14	2
3	0	1	13	959	0	11	0	11	10	5
4	2	1	3	0	934	0	13	3	3	23
5	8	1	0	23	3	821	14	3	12	7
6	7	3	4	0	9	10	921	1	3	0
7	2	8	22	5	3	1	0	971	3	13
8	3	4	4	17	11	12	11	9	899	4
9	9	8	1	12	29	4	1	9	8	928

True Label
Predicted Label
1000
800
600
400
200
0

Img. 10 class confusion matrix