

Neural Network from Scratch

Complete Documentation & Debugging Guide

NumPy · MNIST · ~95% Accuracy

Mahmut Esat Kolay

April 2026

1. Project Overview

This project implements a fully hand-crafted Multi-Layer Perceptron (MLP) using only NumPy — no PyTorch, no TensorFlow, no autograd. Every forward pass, every gradient computation, and every weight update is written from scratch.

The model is trained on the MNIST handwritten digit dataset and achieves ~95% test accuracy in under 20 seconds on modern hardware.

Architecture

Input (784) → MLP(128) → ReLU → MLP(64) → ReLU → MLP(10) → LogSoftmax

Dataset : MNIST (60,000 train / 10,000 test samples)
Loss : NLLLoss + LogSoftmax = Cross-Entropy
Optimizer: SGD (lr=0.01, batch=128, epochs=5000)

2. Code Walkthrough

2.1 — MLP Layer (Fully Connected Layer)

The fundamental building block of the network. Each MLP layer holds a weight matrix W and a bias vector b , applying a linear transformation to its input.

Mathematical Formulas

Forward : $Z = X @ W.T + b$
X shape: (batch_size, din) Z shape: (batch_size, dout)

Backward : $dW = \text{gradout}.T @ X$ (gradient w.r.t. W)
 $db = \text{gradout.sum}(0)$ (gradient w.r.t. b)
 $dX = \text{gradout} @ W$ (gradient passed to previous layer)

Weight Initialization — Xavier / Glorot

Weights are initialized using Xavier/Glorot scaling. This prevents vanishing or exploding gradients at the start of training by keeping the variance of activations consistent across layers.

```
scale = sqrt(6) / sqrt(din + dout)
W = (2 * random(dout, din) - 1) * scale
b = (2 * random(dout) - 1) * scale
```

Code

```
class MLP():
    def __init__(self, din, dout):
        scale = np.sqrt(6) / np.sqrt(din + dout)
        self.W = (2 * np.random.rand(dout, din) - 1) * scale # (dout, din)
        self.b = (2 * np.random.rand(dout) - 1) * scale # (dout,)

    def forward(self, x): # x: (batch_size, din)
        self.x = x # save for backward
        return x @ self.W.T + self.b

    def backward(self, gradout): # gradout: (batch_size, dout)
        self.deltaW = gradout.T @ self.x
        self.deltab = gradout.sum(0)
        return gradout @ self.W
```

2.2 — ReLU Activation Function

Without an activation function, stacking multiple linear layers would still result in a single linear transformation. ReLU breaks this linearity by zeroing out negative values, giving the network the capacity to learn non-linear patterns.

Mathematical Formulas

```
Forward : f(x) = max(0, x)
Backward : x > 0 --> pass gradient through unchanged
           x <= 0 --> zero out gradient
```

Code

```
class ReLU():
    def forward(self, x):
        self.x = x
        return np.maximum(0, x)

    def backward(self, gradout):
        grad = gradout.copy() # never mutate original gradient
        grad[self.x < 0] = 0.
        return grad
```

Note: `gradout.copy()` is critical. Mutating `gradout` in-place would corrupt any other computation sharing the same object.

2.3 — LogSoftmax Activation Function

Converts raw logit scores from the final MLP layer into log-probabilities. Using `logsumexp` from `scipy` is more numerically stable than computing `log(softmax(x))` directly — it prevents overflow errors caused by large `exp()` values.

Mathematical Formulas

```
Forward : log_softmax(x) = x - log( sum(exp(x)) )
          Computed via scipy.logsumexp for numerical stability

Backward : Jacobian = I - softmax(x)    shape: (batch_size, dout, dout)
          Output    = Jacobian @ gradout
```

Code

```
class LogSoftmax():
    def forward(self, x):
        self.x = x
        return x - logsumexp(x, axis=1)[..., None]

    def backward(self, gradout):
        jacobian = np.eye(self.x.shape[1])[None, ...]
        softmax = np.exp(self.x) / np.sum(np.exp(self.x), axis=1,
keepdims=True)
        jacobian = jacobian - softmax[..., None]
        return (np.matmul(jacobian, gradout[..., None]))[:, :, 0]
```

`[..., None]` adds a new axis in NumPy — required to align tensor shapes for batched matrix multiplication.

2.4 — NLLLoss (Loss Function)

Negative Log Likelihood Loss. Takes the log-probabilities from `LogSoftmax` and picks only the log-probability of the correct class, then negates it. The higher the probability the model assigns to the correct class, the closer the loss gets to zero.

Mathematical Formulas

```
Forward : loss = -sum( log_prob[b, true[b]] ) for b in batch
Backward : jacobian[b, true[b]] = -1 (0 elsewhere)
```

Code

```
class NLLLoss():
    def forward(self, pred, true):
        self.pred, self.true = pred, true
        loss = 0
        for b in range(pred.shape[0]):
            loss -= pred[b, true[b]]
        return loss

    def backward(self):
        jacobian = np.zeros((self.pred.shape[0], self.pred.shape[1]))
        for b in range(self.pred.shape[0]):
            jacobian[b, self.true[b]] = -1
        return jacobian

    def __call__(self, pred, true): # enables: loss(pred, true)
        return self.forward(pred, true)
```

2.5 — SequentialNN (Network Container)

Chains all layers together in sequence. This is the hand-written equivalent of PyTorch's `nn.Sequential`. Forward pass runs left to right; backward pass runs right to left as required by the chain rule.

Code

```
class SequentialNN():
    def __init__(self, blocks: list):
        self.blocks = blocks

    def forward(self, x):
        for block in self.blocks:
            x = block.forward(x)
        return x

    def backward(self, gradout):
        for block in self.blocks[::-1]: # reversed - chain rule
            gradout = block.backward(gradout)
        return gradout
```

2.6 — Optimizer (SGD)

After the backward pass, every MLP layer holds `deltaW` and `deltab`. The optimizer uses these to update weights. ReLU and LogSoftmax have no learnable parameters and are skipped.

Update Rule

```
W = W - lr * dW
b = b - lr * db
```

Code

```
class Optimizer():
    def __init__(self, lr, compound_nn):
        self.lr = lr
        self.compound_nn = compound_nn

    def step(self):
        for block in self.compound_nn.blocks:
            if block.__class__ == MLP: # skip ReLU, LogSoftmax
                block.W -= self.lr * block.deltaW
                block.b -= self.lr * block.deltab
```

2.7 — Training Loop

The main loop that ties everything together. Runs mini-batch SGD: a random subset of training data is sampled each epoch, and the full forward → loss → backward → update cycle is executed.

Each Epoch

1. Sample a random mini-batch (batch_size = 128)
2. Forward pass — produce predictions
3. Compute loss — measure how wrong the predictions are
4. Backward pass — compute all gradients via chain rule
5. Optimizer step — update all weights

Code

```
def train(model, optimizer, trainX, trainy,
          loss_fct=NLLLoss(), nb_epochs=5000, batch_size=128):
    training_loss = []
    for epoch in tqdm(range(nb_epochs)):
        idx = [np.random.randint(0, trainX.shape[0]) for _ in
               range(batch_size)]
        x = trainX[idx]
        target = trainy[idx]
        prediction = model.forward(x)
        loss_value = loss_fct(prediction, target)
        training_loss.append(loss_value)
        gradout = loss_fct.backward()
        model.backward(gradout)
        optimizer.step()
    return training_loss
```

3. Debugging: The 9.80% Accuracy Problem

During development, the model produced only ~9.80% test accuracy — essentially random guessing across 10 classes. This section documents the root cause, the error messages, and how it was fixed.

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1776450763.827890 12596 port.cc:153] oneDNN custom operations are on. You may see slight
ly different numerical results due to floating-point round-off errors from different computation order
s. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1776450765.680455 12596 port.cc:153] oneDNN custom operations are on. You may see slight
ly different numerical results due to floating-point round-off errors from different computation order
s. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
Loading MNIST dataset...
Training data shape: (60000, 784)
Test data shape: (10000, 784)
Training started...
0%|          | 0/5000 [00:00<?, ?it/s]C
:\Users\esatk\Desktop\nn_from_scratch\nn.py:87: RuntimeWarning: overflow encountered in exp
softmax = np.exp(self.x) / np.sum(np.exp(self.x), axis = 1, keepdims= True)
C:\Users\esatk\Desktop\nn_from_scratch\nn.py:87: RuntimeWarning: invalid value encountered in divide
softmax = np.exp(self.x) / np.sum(np.exp(self.x), axis = 1, keepdims= True)
100%|          | 5000/5000 [00:14<00:00, 343.32it/s]

Evaluating on test data...
Test Accuracy: 9.80%

Total learning time: 14.57 second (0.24) minute
```

Figure 1 — Broken run: 9.80% accuracy with RuntimeWarning on LogSoftmax (AMD Ryzen 3 3100)

3.1 — What Went Wrong

The terminal output showed two RuntimeWarnings, both originating from the LogSoftmax backward pass at nn.py line 87:

```
RuntimeWarning: overflow encountered in exp
softmax = np.exp(self.x) / np.sum(np.exp(self.x), axis=1, keepdims=True)

RuntimeWarning: invalid value encountered in divide
softmax = np.exp(self.x) / np.sum(np.exp(self.x), axis=1, keepdims=True)
```

Both warnings point to the same line: the raw softmax computation inside `LogSoftmax.backward()`. When the input `x` contains large values, `np.exp(x)` produces `inf`. Dividing any number by `inf` — or dividing `inf` by `inf` — produces `NaN`. Once `NaN` enters the gradient flow, every subsequent weight update becomes `NaN` as well, and the model degenerates to random chance (~10% on MNIST).

3.2 — Root Cause

The forward pass correctly uses `scipy.logsumexp` for numerical stability. However, the backward pass independently re-computes softmax using the raw `np.exp` formula — without any stability trick. As training progresses, logit values grow larger, `exp()` overflows, and the backward pass produces `NaN` gradients.

The forward pass was stable. The backward pass was not.

```
Forward : x - logsumexp(x, axis=1) <-- correct, stable
Backward : np.exp(self.x) / np.sum(np.exp(self.x), ...) <-- overflow risk
```

3.3 — The Fix

Two independent changes resolved the issue. Either one alone was sufficient, but both together make the implementation robust.

Fix 1 — Numerically Stable Softmax in Backward Pass

Subtract the row-wise maximum before computing `exp()`. This is the standard log-sum-exp trick applied to softmax:

```
# BEFORE (unstable):
softmax = np.exp(self.x) / np.sum(np.exp(self.x), axis=1, keepdims=True)

# AFTER (stable - subtract max before exp):
x_stable = self.x - np.max(self.x, axis=1, keepdims=True)
softmax = np.exp(x_stable) / np.sum(np.exp(x_stable), axis=1, keepdims=True)
```

Fix 2 — Lower Learning Rate

The original learning rate of 0.01 allowed logit values to grow too large too quickly, triggering overflow within the first few epochs. Reducing it to 0.001 (as used in the reference implementation) kept activations in a safe numerical range throughout training.

```
# BEFORE:
optimizer = Optimizer(lr=0.01, compound_nn=model)
```

```
# AFTER:
optimizer = Optimizer(lr=0.001, compound nn=model)
```

3.4 — Why 9.80% Specifically?

MNIST has 10 classes (digits 0–9). A model that outputs NaN predictions will have all `argmax` values collapse to the same class index (typically 0 or the first index, due to how `np.argmax` handles NaN). This means it always predicts the same digit. The natural frequency of that single digit in the test set is approximately 10%, which explains the suspiciously clean-looking wrong answer.

3.5 — Results After Fix

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1776446484.776766    1172 port.cc:153] oneDNN custom operations are on. You may see slight
ly different numerical results due to floating-point round-off errors from different computation order
s. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1776446486.736928    1172 port.cc:153] oneDNN custom operations are on. You may see slight
ly different numerical results due to floating-point round-off errors from different computation order
s. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
Loading MNIST dataset...
Training data shape: (60000, 784)
Test data shape: (10000, 784)
Training started...
100%|██████████████████████████████████████████████████████████████████████████████| 5000/5000 [00:16<00:00, 311.61it/s]

Evaluating on test data...
Test Accuracy: 95.73%
```

Figure 2 — Fixed run: 95.73% accuracy, no warnings (AMD Ryzen 3 3100, 5000 epochs, 311 it/s)

```
Loading MNIST dataset...
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ————— 5s 0us/step
Training data shape: (60000, 784)
Test data shape: (10000, 784)
Training started...
100%|██████████████████████████████████████████████████████████████████████████| 5000/5000 [00:04<00:00, 1104.59it/s]

Evaluating on test data...
Test Accuracy: 93.32%

Total learning time: 4.55 second (0.08) minute
```

Figure 3 — Fixed run: 93.32% accuracy on Apple M1 (5000 epochs, 1104 it/s, 4.55 seconds)

After applying the fixes, training completed cleanly with no warnings. Accuracy jumped from ~9.80% to ~95.73% on the same hardware with the same number of epochs. The variation between the two fixed runs (93.32% vs 95.73%) is due to random weight initialization and random mini-batch sampling — not hardware differences.

4. Hardware & Performance Benchmark

All computation runs on CPU using pure NumPy. The same code was benchmarked on two different CPU architectures to measure the impact of BLAS backend, SIMD instructions, and memory bandwidth.

Hardware	Architecture / Backend	Epochs	Speed	Time	Accuracy
AMD Ryzen 3 3100 4C/8T @ 3.9GHz	x86-64 (Zen 2) Intel oneDNN / OpenBLAS (AVX2)	5000	~311 it/s	~15.0 sec	~95.73%
Apple M1 SoC 8-Core: 4P+4E	ARM64 Apple Accelerate (AMX + NEON)	5000	~1104 it/s	4.55 sec	~93.32%

Key Takeaways

- M1 is ~3.5x faster than the Ryzen 3 3100 on this workload
- M1's Apple Accelerate framework routes matrix math through the dedicated AMX coprocessor and NEON SIMD instructions
- Neural network training is memory-bandwidth bound — M1's Unified Memory Architecture delivers significantly higher bandwidth than a standard desktop DDR4 setup
- Accuracy differences between runs are caused by random weight initialization and mini-batch sampling, not hardware

5. Results Summary

Run	Epochs	Batch Size	Learning Rate	Test Accuracy
Broken (overflow)	5000	128	0.01	9.80% — NaN gradients
Fixed — Ryzen 3	5000	128	0.001	~95.73%
Fixed — Apple M1	5000	128	0.001	~93.32%

Note: Accuracy fluctuates slightly between runs due to random initialization. Results above are representative single runs.

6. Setup & Usage

```
# 1. Create virtual environment
python -m venv venv
venv\Scripts\activate      # Windows
source venv/bin/activate   # Linux / macOS
```

```
# 2. Install dependencies
pip install numpy tqdm scipy tensorflow

# 3. Run
python nn.py
```

7. Dependencies

Package	Purpose
numpy	All matrix operations — forward and backward passes
scipy	logsumexp — numerically stable LogSoftmax forward pass
tensorflow / keras	MNIST dataset loader only — not used for training or inference
tqdm	Progress bar during training