Basic Neuarl Net Architecture

kartik virmani

September 2025

Building and Training a Simple Neural Network

Part (a): Randomizing the Data

We first implemented a helper function `select_half_per_class` that samples 50% of the images from each class of the MNIST dataset. This guarantees balanced subsets for both training and validation. We also added visualization to confirm that sampled images cover all digit classes.

Part (b): Embedding Layer

We designed a custom embedding layer that partitions each 28×28 image into 7×7 patches of size 4×4. Each patch is then embedded into an 8-dimensional feature vector by elementwise multiplication with 8 learnable 4×4 filters followed by summation. Thus, each image is transformed into a $(7 \times 7 \times 8) = 392$-dimensional vector.

$$patch\_out_{x,y,c} = \sum_{i=1}^{4} \sum_{j=1}^{4} W_{ijc} \cdot patch_{x,y}(i,j) + b_c$$

This embedding step mimics a convolution with stride 4 and kernel size 4.

Part (c): Forward and Backward Pass

The forward pass produces a $(b, 392)$ feature vector for a batch of size b. The backward pass propagates gradients through the embedding operation, updating both weights W and biases b, while redistributing gradient contributions back to the 4×4 patches.

Part (d): Linear Layer

We implemented a fully connected layer with parameters $W \in R^{392 \times 10}$ and $b \in R^{10}$, mapping the embedded feature vector into class logits:

$$h = xW + b$$

Backward propagation accumulates $\nabla W, \nabla b$ and passes ∇x upstream.

Part (e): Activation Layer

We used a ReLU activation defined as

$$ReLU(z) = \max(0, z).$$

Backward propagation sets gradients to zero where $z \leq 0$ and propagates unchanged where $z > 0$.

Part (f): Loss Layer

We applied a softmax function to convert logits into probabilities:

$$p_i = \frac{\exp(h_i)}{\sum_j \exp(h_j)}.$$

The cross-entropy loss for target y is

$$\ell = -\log p_y.$$

Gradients w.r.t logits are simply:

$$\nabla h = p - one\_hot(y).$$

Gradient Checks

For embedding, linear, and activation layers, we verified gradients using finite difference approximation:

$$\nabla f(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}.$$

All layers passed checks with relative error $\sim 10^{-6}$, confirming correctness of backward implementations.

Part (g): Training the Custom Network

We assembled the pipeline:

$$Embedding \rightarrow Linear \rightarrow ReLU \rightarrow Softmax + CE.$$

Training setup:

- Batch size = 32
- Learning rate = 0.1
- Updates = 1000–10000
- Optimizer = plain SGD

Results: training error decreased to $\sim$ 40%, validating that the network was learning. Loss fluctuated due to simplicity of the architecture.

Part (h): Validation

We implemented a validation function:

```
def validate(l1,l2,l3,l4,Xva,Yva):
    for i in range(0,N,32):
        xb,yb = Xva[i:i+32], Yva[i:i+32]
        loss, err = forward_pass(...)
        accumulate
    return avg_loss, avg_err
```

Validation was computed every 1000 updates, with results plotted for both loss and error vs. number of updates.

Part (i): PyTorch Implementation

We repeated the entire pipeline using PyTorch. The embedding layer was implemented via:

`nn.Conv2d(in_channels=1, out_channels=8, kernel_size=4, stride=4)`,

producing an output of shape $(b, 8, 7, 7)$ which was flattened to 392 features. The rest of the network was:

$$\texttt{Linear(392,10)} \rightarrow \texttt{ReLU} \rightarrow \texttt{CrossEntropyLoss}.$$

We trained for 10,000 updates with the same minibatch size and learning rate. Results were plotted:

- Training loss vs. updates
- Training error vs. updates
- Validation loss vs. updates
- Validation error vs. updates

Observations

- Our custom NumPy network achieved reasonable learning, but plateaued around 40–60% accuracy.
- PyTorch implementation was cleaner and more efficient, producing smoother curves and higher stability.
- The embedding operation with stride acted like a fixed receptive field convolution, enriching features beyond raw pixels.