From Theory to Code: Implementing a Neural Network in 200 Lines of C

Neural networks are computational models inspired by the human brain, capable of learning complex patterns from data. Today, high-level languages and libraries like TensorFlow, PyTorch, or Keras make building neural networks trivial. However, implementing one in C provides a deeper understanding of the underlying mechanics and helps gain appreciation of the higher-level abstractions.

In this article, we will make our own 200 line neural network in C using only the standard library. The neural network will be trained and fine-tuned to recognize handwritten digits from the MNIST dataset.

Note: This article is based on miniMNIST-C : Minimal neural network implementation in C. You can explore the project and its source code on GitHub:

`https://github.com/konrad-gajdus/miniMNIST-c`

Neural Network Structure

Our neural network will consist of three layers: input, hidden, and output. The number of neurons in each layer is determined by the characteristics of the dataset we are using. In this case, we'll be working with the MNIST dataset, which provides handwritten digits in a $\mathbf{28 \times 28}$ pixel format. By flattening each image into a single vector, we obtain an array of **784** elements that will serve as our input layer. The output layer will comprise **10** neurons, each corresponding to one of the possible digits (0-9).

This can be summarized as:

- **Input Layer**: Accepts the $\mathbf{28 \times 28}$ pixel images (flattened into a **784**-dimensional vector).
- **Hidden Layer**: Contains **256** neurons
- **Output Layer**: Contains **10** neurons (one for each digit class)

Processing Input Data

The MNIST dataset comprises 60,000 training images and 10,000 testing images of handwritten digits, ranging from 0 to 9. Each image is represented as a $\mathbf{28 \times 28}$ pixel grayscale matrix. Before we can use this data in our neural network, we need to preprocess and format it in a way that aligns with the network's architecture.

We use two functions to read the images and labels from the IDX file format:

These functions:

- Open the IDX files in binary mode.
- Read and convert the header information (number of images, rows, columns).
- Allocate memory and read the image pixel data and labels.

Implementing the Neural Network Structure

We define a Layer struct to represent each layer in the network:

- **weights**: A flattened array representing the weight matrix.
- **biases**: An array for the biases of each neuron.
- **input_size**: Number of inputs to the layer.
- **output_size**: Number of neurons in the layer.

The weights and biases of each layer need to be initialized to a default value. They will later be updated by the neural network as it learns through the process of backpropagation.

We initialize the weights and biases in the 'init_layer' function:

We use He Initialization to set the weights:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right)$$

Biases are initialized to zero.

Forward Propagation

Forward propagation, also known as the forward pass, is the process of computing the output of a neural network given an input. It involves moving the input data through each layer of the network, applying linear transformations and activation functions (ReLU, Softmax etc.), to produce the final output:

Activation Functions

An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. We will use two kinds of activation functions in our implementation:

ReLU Activation

We apply the ReLU (Rectified Linear Unit) activation function in the hidden layer:
Mathematically:

$$\text{ReLU}(x) = \max(0, x)$$

Softmax Activation

We apply the softmax function to the output layer to compute probabilities:
Mathematically:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Forward propagation is essential because it:

1. **Generates Predictions**: By moving the input through the network, we obtain the predicted output, which is necessary for both training and inference.
2. **Computes Loss**: In training, the predicted output is compared against the true labels to compute the loss, which quan-

tifies the error.

3. **Facilitates Backpropagation**: The activations and intermediate outputs computed during forward propagation are used in backpropagation to compute gradients.

Backpropagation

Backpropagation, also known as the backward pass, is the reversed flow of the forward pass - we start by propagating the error from the output layer until reaching the input layer, passing through the hidden layer(s). The backward function updates the weights and biases based on the gradients:

1. **Weight update**:

$$w_{ij} = w_{ij} - \eta \cdot \frac{\partial L}{\partial w_{ij}}$$

   New weight equals old weight minus learning rate times gradient of loss with respect to weight.

2. **Bias update**:

$$b_i = b_i - \eta \cdot \frac{\partial L}{\partial b_i}$$

   New bias equals old bias minus learning rate times gradient of loss with respect to bias.

3. **Input gradient calculation**:

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial L}{\partial y_i} \cdot w_{ij}$$

   Gradient of loss with respect to input j is the sum of (gradient of loss with respect to each output i times the weight connecting input j to output i) over all outputs.

4. **Gradient calculation for weight update**:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial y_i} \cdot x_j$$

Gradient of loss with respect to weight equals gradient of loss with respect to output times input value.

Backpropagation is essential because it:

1. **Computes Gradients**: Calculates how much each weight and bias affects the network's error.
2. **Updates Parameters**: Adjusts weights and biases to reduce the error in future predictions.
3. **Propagates Error**: Sends error information backward through the network, allowing all layers to learn.
4. **Enables Learning**: Through iterative updates, allows the network to improve its performance on the given task.

Backpropagation is key to training neural networks, as it's how they learn to make better predictions over time.

Training

With forward and backward propagation now implemented, we are ready to train our neural network. Training involves calling these propagation functions in a loop to adjust the network's weights and biases, minimizing the loss and enhancing the model's ability to make accurate predictions with each pass.

The train function encapsulates the training logic for a single input-label pair:

This process represents one training iteration.

Training loop

Training is an iterative process spanning multiple epochs. Each epoch involves running the entire dataset through the network, performing forward and backward passes to adjust weights and biases. Thus improving the accuracy of the neural network.

This can be achieved with training loop, iterating over a set number of epochs (we will use 20 for our use case):

Evaluation

With the neural network trained, we can move on to evaluating its performance by running inference on an 'unseen' set of data. We split the input data (80% training, 20% test) to provide a benchmark for evaluation.

We assess accuracy using the predict function:

The predict function implements the forward pass of our neural network for running inference. It uses the same forward propagation and activation functions (ReLU, softmax) covered earlier, but instead of computing gradients, it returns the most likely digit prediction.

We combine that with out main training loop to evaluate the performance of the model:

Running the training loop gives us the following data:

The results demonstrate consistent improvement in accuracy and reduction in loss over the course of 20 epochs, with the model ultimately achieving an impressive 98.22% accuracy on the final test set. It's worth noting that state-of-the-art models typically achieve around 99% accuracy on the same dataset.

Taking this into account, our small, custom-built model implemented in C performs remarkably well, especially considering its simplicity.

Conclusion

You should now be able to combine the key steps above to create your own neural network in C.

You can explore the project and its source code on GitHub:

`https://github.com/konrad-gajdus/miniMNIST-c`

Leave a star to follow for further updates.