MPCS 51087
Project Set 3, Milestone 1
Machine Learning for Image Classification

due: Sunday, Feb. 23 @ 6pm

1 Building and Training a Neural Network for Rasterized Digit Classification

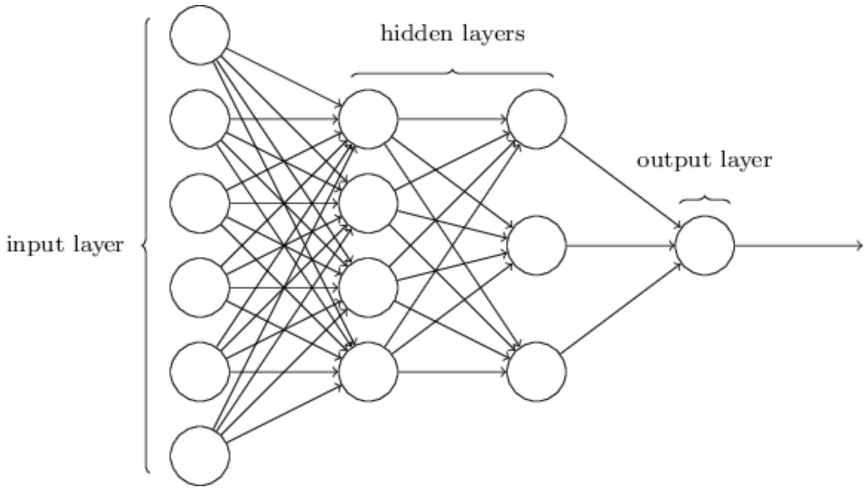

Figure 1: An example neural network with four densely connected layers. The first layer is known as the *input layer*, and represents the predictive variables; the final (fourth) layer is the *output layer* and represents the model's prediction. Neural networks can also include an arbitrary number of intermediate (*hidden)* layers to increase the complexity of the model.

In this exercise, you will implement a basic feedforward neural network (FNN) for both training and inference. While the code will eventually be optimized and ported to GPUs, this first milestone focuses on developing and testing a CPU implementation. Although the approach is general, the application is a classic machine learning problem: handwritten digit recognition.

The basic structure of an FNN is shown in Figure 1. The network consists of four layers: an input layer, an output layer, and two hidden layers. In this example, the layers contain 6, 4, 3, and 1 neurons, respectively, and are fully connected (each neuron is connected to every neuron in the next layer). These design choices are largely heuristic and can vary between implementations. Each connection in the network has an associated weight (not shown in the figure for clarity), which represents the contribution of the source neuron to the activation of the target neuron. These weights are initially randomized, and the learning process aims to optimize them by minimizing a cost function based on a given training dataset.

In addition to weights, each node (neuron) has a value known as its *activation*. In the first layer, activations correspond directly to the input values. In subsequent layers, activations are computed as linear combinations of the weighted activation values from all incoming connections, with an activation function, $\sigma(x)$, applied to map the output to a continuous value within a predefined range. Additionally, a bias term

is added to each weighted sum. Using index notation and numbering the layers $1, \ldots, L$, we denote the weights, biases, and activation values as follows:

- w^l_{jk}: The weight of the connection from the kth neuron in layer $l-1$ to the jth neuron in layer l.
- b^l_j: The bias of the j'th neuron in layer l.
- a^l_j: The activation of the j'th neuron in layer l.

1.1 Forward Propagation

The first step in training, known as forward propagation, takes a set of inputs in layer 1 and uses the weights, biases, and an activation function to compute the neuron values in the next layer. This process continues layer by layer until the output is determined for the given set of weights.

The neuron activations in the l'th layer given the activations from layer $l-1$ are computed as:

$$a^l_j = \sigma\left[\sum_{k=1}^{n_{l-1}} \left(w^l_{jk} a^{l-1}_k\right) + b^l_j\right]$$

or in matrix notation

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{1}$$

In this first milestone, the activation function $\sigma(x)$ for the hidden layers will be the *Rectified Linear Unit (ReLU)* function, defined as

$$f(x) = \max(0, x)$$

which maps all negative inputs to zero while allowing positive inputs to pass through unchanged.

The activation function for the output layer will be the *Softmax* function, defined as

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } i = 1, \ldots, K$$

where K is the number of output classes. The Softmax function converts the output layer's activations into probabilities that sum to 1.

Equation 1 defines the basic algorithm for forward propagation. The training process begins with a given set of inputs and randomly initialized weights and biases, using forward propagation to compute the outputs. Once the outputs are obtained, a cost function is evaluated for each training input set and averaged across all training examples.

1.2 Cost Function

In this example, we use the *cross-entropy* cost function, which is well-suited for classification problems with a Softmax output layer. It is defined as:

$$C = -\frac{1}{n} \sum_x \sum_{i=1}^{K} y_i(x) \log a^L_i(x) \tag{2}$$

where:

- K is the number of output classes (10 for MNIST).
- n is the total number of training examples.
- $y_i(x)$ is the true one-hot encoded label for class i.
- $a^L_i(x)$ is the predicted probability for class i from the Softmax function.

For each training example x, the cross-entropy loss measures how well the predicted probability distribution $a^L(x)$ aligns with the true distribution $y(x)$. Because $y(x)$ is one-hot encoded (only one entry is 1, the rest are 0), this simplifies to:

$$C = -\frac{1}{n} \sum_x \log a_j^L(x) \tag{3}$$

where j is the index of the correct class for x. This formulation ensures that:

- If the model predicts a high probability for the correct class ($a_j^L \approx 1$), the loss is small.
- If the probability for the correct class is low ($a_j^L \approx 0$), the loss is large, forcing the network to improve.
- The gradient updates remain large even when a_j^L is close to 0, preventing vanishing gradients.

Since $C = C(w, b)$, we seek the values of w and b that minimize C across the training dataset. This is achieved iteratively using an efficient method called *backpropagation*, which computes the derivatives of C with respect to w and b.

1.3 Backpropagation

The optimal values of w and b are found using an approach called *gradient descent*. Without an efficient algorithm to minimize the cost function, training deep neural networks would be impractical, as a naive search for the minimum suffers from the "curse of dimensionality." Fortunately, gradient descent is highly efficient, even if not always fully robust.

The process begins with randomly initialized values of (w, b), and then iteratively updates these parameters by computing the gradient of the cost function and adjusting them in the direction of steepest descent.

To minimize the cost function C, we compute its derivatives with respect to the weights and biases:

$$\frac{\partial C}{\partial w_{jk}^l}, \quad \frac{\partial C}{\partial b_j^l}$$

which are then used to update w_{jk}^l and b_j^l in each iteration.

1.3.1 Output Layer Backpropagation (Softmax + Cross-Entropy)

For a *Softmax output layer* combined with the *cross-entropy loss function*, the gradient simplifies significantly. If a^L represents the output of the Softmax function and y is the one-hot encoded true label, then:

$$\frac{\partial C}{\partial a^L} = a^L - y$$

which eliminates the need for an explicit derivative of the Softmax function since it naturally cancels out in the gradient computation. Thus, the error at the output layer L is:

$$\delta^L = a^L - y$$

This leads to the gradient updates:

$$\frac{\partial C}{\partial w^L} = \delta^L (a^{L-1})^T$$

$$\frac{\partial C}{\partial b^L} = \delta^L$$

1.3.2 Hidden Layer Backpropagation (ReLU)

For hidden layers with the *ReLU activation function*:

$$\sigma(z) = \max(0, z)$$

its derivative is:

$$\sigma'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$$

The error term for each hidden layer l is computed using:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

where $\odot$ denotes the element-wise (Hadamard) product. The gradients for updating weights and biases in hidden layers are:

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T$$

$$\frac{\partial C}{\partial b^l} = \delta^l$$

1.3.3 Summary of Backpropagation Equations

To summarize, the updated backpropagation equations for a network using *Softmax + cross-entropy in the output layer and ReLU in hidden layers* are:

$$\delta^L = a^L - y \tag{4}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad \text{(for hidden layers using ReLU)} \tag{5}$$

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{6}$$

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T \tag{7}$$

These computed gradients are then used in gradient descent to iteratively update the weights and biases, optimizing the neural network's performance on the classification task.

1.4 Training Procedures and Pseudocode

Everything is now in place to train a neural network given a set of training data. One final approximation improves the computational efficiency of the process: we use a specific form of gradient descent called *mini-batch stochastic gradient descent* (SGD).

Rather than computing each step of gradient descent using the full dataset, we approximate it by randomly selecting a batch of m training samples. A new batch is selected for each step of the algorithm until all training data has been processed once. This complete pass through the dataset is called a *training epoch*, after which the process repeats with newly selected batches. The pseudocode below represents one epoch of training.

1.5 Mini-Batch Stochastic Gradient Descent

1. **Shuffle the training dataset**.
2. **Divide the dataset into mini-batches** of size m.
3. **For each batch of training examples**:
    - **For each training example** x in the batch:

- **Forward propagation:**
  - **For each hidden layer** $l = 1, \ldots, L-1$:

$$z^{x,l} = w^l a^{x,l-1} + b^l, \quad a^{x,l} = \max(0, z^{x,l}) \quad \text{(ReLU activation)}$$

  - **For the output layer** L:

$$z^{x,L} = w^L a^{x,L-1} + b^L, \quad a^{x,L} = \text{Softmax}(z^{x,L})$$

- **Compute output error using cross-entropy loss:**

$$\delta^{x,L} = a^{x,L} - y$$

  where y is the one-hot encoded ground truth label.
- **Backpropagate the error for hidden layers:**

$$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \mathbf{1}(z^{x,l} > 0)$$

  where $\mathbf{1}(z^{x,l} > 0)$ is the derivative of the ReLU activation function, which is 1 for positive inputs and 0 otherwise.

4. **After processing the entire batch, update weights and biases:**

$$w^l \leftarrow w^l - \frac{\alpha}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T, \quad b^l \leftarrow b^l - \frac{\alpha}{m} \sum_x \delta^{x,l}.$$

1.6 Using Machine Learning to Identify Rasterized Digits

The MNIST database of handwritten digits (`https://web.archive.org/web/20220331130319/https://yann.lecun.com/exdb/mnist/`) is one of the most widely used publicly available datasets for image classification. It consists of 60,000 grayscale images for training and 10,000 for testing, each with a resolution of 28×28 pixels.

Using C/C++/Fortran, implement a CPU version of a multilayer neural network from scratch and train it using stochastic gradient descent on the MNIST training dataset.

- The input layer consists of 784 ($= 28^2$) neurons, representing the flattened 2D image.
- The network has two fully connected hidden layers with 128 and 256 neurons, respectively.
- The output layer consists of 10 neurons, corresponding to the 10 digit classes.
- The model is trained using the cross-entropy loss function with stochastic gradient descent.
- Each training sample has a corresponding *one-hot encoded* ground truth vector $\mathbf{y}$, where each element represents a binary indicator for the correct class. For example, for an input image of a handwritten "6," the target vector is:

$$\mathbf{y} = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0).$$

2 Submission Requirements and Documentation

The first milestone consists of a working (though not necessarily efficient) CPU implementation of the code using C, C++, or Fortran, along with a brief report that includes the following:

- **Success rate**: Accuracy of the model on the test dataset.
- **Grind rate**: Average number of input samples processed per second.
- **Total training time**.

- **Total inference time**.
- **Learning rate** used for training.
- **Batch size** used for training.

This milestone will be considered successful if a reasonable, functioning implementation is provided. No significant optimization is required at this stage. However, you are encouraged to explore optimizations that improve training speed (including OpenMP), as these will be important in the next milestone.