MPCS 51087
Problem Set 3
Machine Learning for Image Classification
Milestone 1

Winter 2024

due: Sunday Feb. 25 @6pm

1 Building and Training a Neural Network for Rasterized Digit Classtification

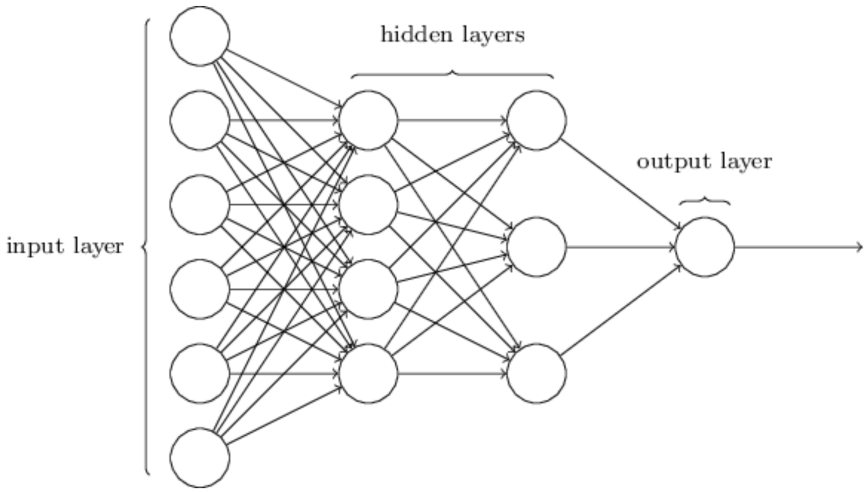

Figure 1: An example neural network with four densely connected layers. The first layer is known as the *input layer*, and represents the predictive variables; the final (fourth) layer is the *output layer* and represents the model's prediction. Neural networks can also include an arbitrary number of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

In this exercise you will write a basic feed foward neural network (FNN) code for both training and inference. The code will ultimately be optimized and ported to GPUs, but in this first milestone we focus on building and testing a CPU implementation on a classic problem – digit recognition.

The basic structure of an FNN is shown in Figure 1. The network shown has four layers – input, output, and two hidden layers. There are 6,4,3, and 1 neurons per layer in this example, and the layers are densely connected (each neuron is connected to every other neuron in the subsequent layer). These are all specific choices with a strong heuristic component that vary from implementation to implementation. Each incoming edge in the graph above is characterized by a decimal *weight*, not shown in the figure to avoid clutter, representing the significance of the source node's contribution to *activation* of the target node. These

weights are initially set randomly, and the learning process seeks to optimally determine them by minimizing a *cost function* relative to a set of training data.

In addition to weights, each node (neuron) has a value, known as its *activation*. In layer 1, the activations are just the input values. In subsequent layers they are computed as linear combinations of the weighted activation values of all incoming edges, with some activation function, $\sigma(x)$, applied to convert the output to a continuous value in some predefined range. Additionally, a bias is added to each weighted sum. Using index notation and numbering the layers $1, \ldots, L$, we denote the weights, bias, and activation values as:

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

1.1 Forward Propogation

The first step in training, known as *forward propogation*, takes a set of inputs in layer 1 and uses the weights, *biases*, and an *activation function* to compute node values in the next layer, and so forth, until the output is determined for the given set of weights.

The node values (activations) in the l'th layer given activation values in layer l-1 are given by:

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

or in matrix notation

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

where the activation function $\sigma(x)$ in the first milestone will be the *Sigmoid* function, $f(x) = \frac{1}{1+e^{-x}}$, which maps values to between 0 and 1. Other common choices are ReLU, Hyperbolic tangent, and Softmax. Softmax, as explained below, will be used in the next milestone just for the output layer. Equation 1 is the basic algorithm for forward propagation. The traning process starts with a given set of inputs and randomly chosen weights and biases, and uses forward propogation to calculate outputs. Once the outputs are calculated, a cost function is computed for each training input set and averaged across all sets.

1.2 Cost Function

In this example we use a quadratic cost funcion of the form

$$C = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 \tag{2}$$

where $||.||$ denotes the L2-norm, n is the total number of training examples; the sum is over individual training examples, x; $y = y(x)$ is the corresponding desired output (correct answer); L denotes the number of layers in the network; and $a^L = a^L(x)$ is the vector of activations output from the network when x is input. In other words, for each set of training data x, we compute the squared norm of the "errors": the vector difference between the correct output, $y(x)$, and the predicted output, $a^L(x)$. The average across n training sets then gives the value of C (the factor of $1/2$ is for convenience in computing the derivative). Since $C = C(w, b)$, we seek the values of w, b that minimize C across the training data set. This is done as an iterative process using an efficient process called *backpropagation* to compute derivitives of C with respect to b and w.

1.3 Backpropogation

Finding optimal w and b is done by using approach called *gradient descent*. Without a highly efficient algorithm to find the values of w and b that minimize the cost function, FNNs would be completely impractical (a naive search for the min suffers the "curse of dimensionality"). Fortunately, gradient descent is extremely

efficient, even if it isn't always fully robust. We select random values of (w, b) as a starting point, and then go "downhill" until we (hopefully) reach a global minimum. Finding the downhill direction means computing all of the derivatives of the cost function with respect to the weights and biases.

Even when using gradient descent, finding the minimum quickly becomes intractable with increasing network size without an efficient method for computing the derivaties. We outline such a method here. We need to evaluate the derivitive of the cost function with respect to the weights and biases: $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$ and use this to update w^l_{jk} and b^l_j for the next iteration.

We start be defining the *error* at node l as:

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}$$

where z^l_j is defined as the *weighted input* to neuron j in layer l – that is, the input before the activation function is applied:

$$z^l_j = w^l_j a^{l-1} + b^l_j$$

Using the chain rule, the error in each node can then be written as:

$$\delta^l_j = \frac{\partial C}{\partial a^l_j}\sigma'(z^l_j)$$

In matrix form this can be written more concisely as:

$$\delta^l = \nabla_a C \odot \sigma'(z^l)$$

where ∇_a represents the derivative with respect to a^l_j. For our choice of quadratic cost function, $\nabla_a C = (a^L - y)$, and so we can immediately write an expression for the error at layer L:

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

Then we can compute the error at any layer from the previous layer by working backwards from layer L as

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

This then allows us to compute the desired derivatives:

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

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

To summarize, the equations for backpropogation are:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{3}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{4}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{5}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{6}$$

1.4 Training Epochs and Pseudocode

Everything is now in place to train a neural network given a set of training data. One final approximation will make the overall process more computationally tractable. We use a variant of gradient descent called *stochastic gradient descent* (SGD). Rather than use all the training data to compute each step in the gradient descent algorithm, we randomly select a small *batch* of training inputs of size m. A unique batch is chosen for each step in the algorithm until all training data is exhausted. This is referred to as a training *epoch*, at which point the process begins again with new randomly chosen batches. The pseudocode below represents on *epoch* of training and assumes an outer loop where the epic is chosen randomly from all of the input data.

1. Input a set of training examples
2. For each training example x: Set the corresponding input activation $a^{x,1}$ and perform the following steps:
   - **Feedforward**: For each $l = 2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.
   - **Output Error**: Compute $\delta^{x,L} = \nabla_a C \odot \sigma'(z^{x,L})$
   - **Backpropogate**: for each $l = L-1, L-2, ..., 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$
3. **Gradient Descent**: for each $l = L, L-1, L-2, ..., 2$ update the weights $w^l \implies w^l - \frac{\alpha}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ and the biases $b^l \implies b^l - \frac{\alpha}{m} \sum_x \delta^{x,l}$

1.5 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 perhaps the most widely used publicly-available image classification dataset in modern machine learning. It consists of 60,000 images in the training set, 10,000 images in the testing set, all grayscale and 28x28 pixels.

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

- The first layer (untrainable input layer) should consist of the 784 ($= 28^2$) pixels from a flattened 2D image.
- Then, there are an indeterminate number of intermediate hidden layers consisting of n_i units each. Generally, these are sized differently (search Google for "bottleneck layer", e.g.). For this benchmark you are encouraged to try different configurations, but to to have a common performance baseline across the class, you will report on results with two hidden layers of 10 neurons each.
- The last layer should consist of 10 nodes. For simplicity on Milestone 1 we will also use the Sigmoid activation function for the output layer. In the next milestone we will switch to *Softmax* activations for the last layer, roughly corresponding to the predicted probabilities of each digit class.

  $$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

  where $K = 10$ here.
- The cost will be based on the MSE loss, as discusssed earlier. For each training sample, there is a corresponding ground truth ("one-hot encoded") vector $\mathbf{y}$, which consist of a binary indicator for the correct class that is used in the MSE function. E.g for an input image of a handwritten 6, $\mathbf{y} = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)$.
- At a minimum, your code should take as command line arguments: 1) `nl` the number of dense (fully connected) linear layers in your NN (excluding the first and last layers), 2) `nh` the numnber of units in each of the hidden layers, 3) `ne` the number of training epochs, 4) `nb` the number of training samples per batch, and 5) α, the learning rate. It should output to stdout the average cost function and grind rate (samples/second) of each batch.

2 Submission and Documentation

The first milestone will consist of a working CPU version of the code using, C, C++, or Fortran together with a very short report showing

- success rate using the model described above (two hidden layers of 10 neurons each) across all 10,000 images in the test data sets

- grind rate (average number of input samples per second that can be processed)
- total time to train the model using all 60K training images
- total time to do inference on the 10K test images
- the learning rate used to calculate with the results above
- the number of images per batch used to calculate the results above

This milestone will be considered successful for a reasonable working implementation. No signficant optimization is required. However, you are encouraged to explore implementations that speed up training (including OpenMP), and these will be an important step in the next milestone.