MPCS 51087
Problem Set 4'
Machine Learning for Image Classification

Spring 2022

1 Intro: Basic Curve Fitting with Gradient Descent

Milestone 1 due Thurs, May 26 @6pm: Prototype using High Level Langage
Final Submission due Thurs, June 2 @6PM

1.1 Linear Fit

As warm-up, consider minimization by gradient descent in a simpler context, with a known linear function of just several independent variables. This will guide your intuition on applying the same algorithm in a more complicated context in the next section. Consider the problem of fitting a line to a set of points in 2D space, where the points generally follow a linear trend but there is noise such that no straight line can pass through all points. We want to come up with a line of the form:

$$y = mx + b \tag{1}$$

where m is the line's slope and b is the line's y-intercept. There are many different approaches to solving this proble. One method is to cast this as an optimization problem where we need to find "optimal" values for m and b, which has many analogues in machine learning.

To begin, we will need to come up with a way of defining what an "optimal" line looks like. For the first part of this assignment, we will use the mean squared error (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2}$$

where there are n 2D points in our data set, y_i is the y-coordinate of a point i, and $\hat{y}_i$ is the y-coordinate of our best fit line given an x-coordinate at location i. That is, we are defining error as the average of the squared distance between the best fit line and each of our data points. Given our curve fit model of $y = mx + b$, we can write a final *error function* E in terms of the line coefficients as:

$$E(m, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (mx_i + b))^2 \tag{3}$$

We therefore want to find the the coefficients m and b such that the error function $E(m, b)$ is minimized. Recall from calculus that the minimum of a function is found at a point where the function's gradient is zero ($\nabla E = 0$). So, to use gradient descent, we will want to be able to find the gradient of our error function as well. In this case, we can find the gradient analytically as follows:

$$E = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - (mx_i + b)\right)^2 \tag{4}$$

$$E = \frac{1}{n}\sum_{i=1}^{n}\left(y_i^2 + b^2 + 2bmx_i - 2by_i + m^2x_i^2 - 2mx_iy_i\right) \tag{5}$$

$$\nabla E = \left(\frac{\partial E}{\partial m}, \frac{\partial E}{\partial b}\right) \tag{6}$$

$$\frac{\partial E}{\partial m} = \frac{1}{n}\sum_{i=1}^{n}\left(2bx_i + 2mx_i^2 - 2x_iy_i\right) \tag{7}$$

$$\frac{\partial E}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}\left(2b + 2mx_i - 2y_i\right) \tag{8}$$

Taking inventory, we now have:

- a set of points we want to generate a linear fit for
- a way of defining how "good" the fit is (an error function) given line coefficients m and b
- a way of determining the gradient of our error function given coefficients m and b

Combined, these are all the elements we need to use the method of steepest descent (MSD), often referred to as gradient descent in the context of machine learning algorithms, shown in Algorithm 1. We start with some random guess for m and b, and compute the gradient $(\frac{\partial E}{\partial m}, \frac{\partial E}{\partial b})$ using our analytical expressions. We then follow the gradient for some distance (using a "learning rate" α that we can choose via experimentation) and then repeat the process until we have arrived at values for m and b where the norm of our error function gradient is (nearly) zero – which indicates we have arrived at a minima of the error function.

Algorithm 1 Gradient Descent Curve Fitting Pseudocode For Linear Fit

```
Input: n points, α learning rate, ε convergence criteria
Guess m and b
loop
    ∇m = 0                                          ▷ Gradient of m
    ∇b = 0                                          ▷ Gradient of b
    for each point i do
        ∇m = ∇m + (2bx_i + 2mx_i^2 − 2x_i y_i)
        ∇b = ∇b + (2b + 2mx_i − 2y_i)
    end for
    ∇m = ∇m / n
    ∇b = ∇b / n
    m = m − α∇m
    b = b − α∇b
    if ‖(∇m, ∇b)‖ < ε then
        Convergence detected, m and b are optimal
    end if
end loop
```

1.2 Second Order Fit

Your job for this portion of the assignment is to implement a gradient descent solver capable of fitting a second order curve given the set of 2D data points found in file `points.txt`. Do this by adapting Algorithm 1 to fit a curve of the form:

$$y = hx^2 + mx + b \tag{9}$$

Your error function and gradient will now have three dimensions. You can use any language to code your solver.

Note that this method is not unconditionally stable, and may require a fairly small learning rate to converge. $\alpha = 0.00025$ is a good place to start. You may find that your solver requires a lot of iterations to converge as well, which is expected. Just note that there are a variety of ways of accelerating gradient descent for curve fitting that are outside the scope of this problem set.

1.3 Analysis

Discuss (and illustrate) the effect of the choice of learning rate (α) on the resulting solution and its speed of convergence, as well as how the norm of the gradient evolves through the iteration. Around what value for *alpha* does the iterative process diverge? Plot the norm of the gradient vs. the iteration number for a variety of (convergent) learning rates.

Also, briefly discuss how you would approach this problem if you were unable to determine the gradient of the error function analytically, but still wanted to use gradient descent.

Finally, generate a plot showing both the inputted data points and your optimal second order curve fit. Report the coefficients h, m, and b for your optimal fit.

2 Building and Training a Neural Network for Rasterized Digit Classtification

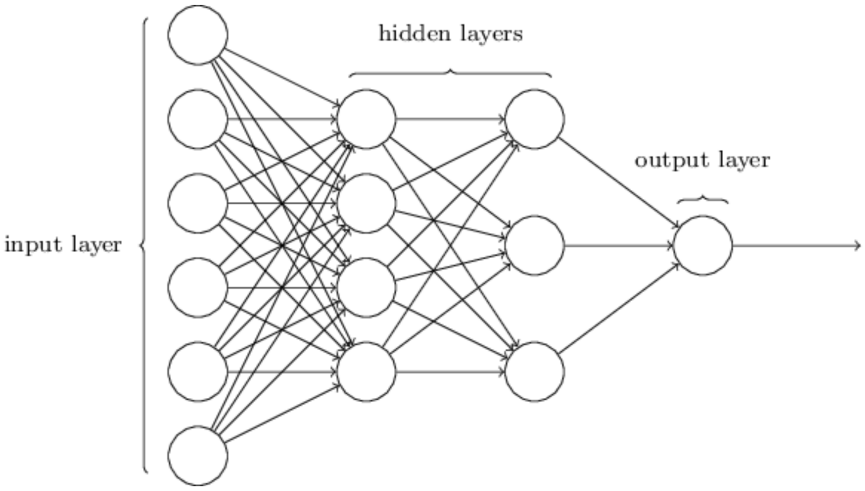

Figure 1: An example neural network with four densely connected layers. The first layer is known as the *input layer*, and represents the predictive variables; the final (fourth) layer is the *output layer* and represents the model's prediction. Neural networks can also include an arbitrary number of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

Each incoming edge in the graph above is characterized by a decimal *weight*, representing the singnificance 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.

2.1 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:

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

or in matrix notation

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{10}$$

where the activation function $\sigma(x)$ in our implemetnation will be the sigmoid function, $f(x) = \frac{1}{1+e^{-x}}$, which maps values to between 0 and 1. Equation 10 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.

2.2 Cost Function

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

$$C = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 \tag{11}$$

where 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 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 iterartive process using an efficient process called *backpropagation* to compute derivitives of C with respect to b and w.

2.3 Backpropogation

Finding optimal w and b is done by using gradient descent, as discussed in Section 1. We select random values of (w, b) as a starting point, and then go "downhill" until we (hopefully) reach a global minimum.

For a neural network, computing derivatives efficiently is more complicated than the procedure described in Section 1. 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 a node as:

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

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

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

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

$$\delta_j^l = \frac{\partial C}{\partial a_j^l} \sigma'(z_j^l)$$

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_j^l. For our choice of quadratic cost function, $\nabla_a C = (a^L - y)$, and so

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

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:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

To summarize, the equations for backpropogation are:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{12}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{13}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{14}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{15}$$

2.4 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 $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^l)$
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 $b^l \implies b^l - \frac{\alpha}{m} \sum_x \delta^{x,l}$

2.5 Using Machine Learning to Identify Rasterized Digits

The MNIST database of handwritten digits (`http://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 and CUDA, implement a 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.), but we will make them all the same size for this assignemnt.
- The last layer should consist of 10 units with sigmoid activations, roughly corresponding to the (unnormalized) predicted probabilities of each digit class.
- 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. It should output to stdout the average cost function and grind rate (samples/second) of each batch.
- Optional: For the output layer, replace the sigmodid activation function with the *softmax activation*:

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

  where $K = 10$ here. Instead of the MSE loss, use the categorical cross-entropy loss function, where for a single sample:

  $$-\sum_{c=1}^{K} y_c \log(p_c)$$

  where $p_c = \sigma(\mathbf{z})_i$ output layer values.

  Note, the backpropagation formulas will be different from those derived in section 2.3 for MSE loss with sigmoid activations. Derive (or lookup) the gradient calculations for this network architecture. See footnote for a good discussion of why cross-entropy is preferable to MSE loss for this classification problem.[1]
- Optional: Data parallel training using multiple GPUs on a single node via MPI syncing of gradient updates. Although it is certainly overkill for the MNIST problem, data parallel training involves dividing each training batch equally among N workers (GPUs). During each training step in every epoch, after completing forward and backward propagation (but before applying the gradients in SGD), the workers synchronize their local approximations to the overall batch gradient for all the tunable weights. First use `MPI_Allreduce` to sum the gradients, then divide by the number of workers (2 to 4 for distributed training on a single node of Midway 3) to get the average gradient.
- Optional: use the cuDNN library to implement a convolutional neural network (CNN) of your choosing. Briefly compare the accuracy of this CNN model on the testing set to the model consisting only of fully connected layers. Use the TensorCore-enabled functions in the cuDNN library `https://docs.nvidia.`

[1] `https://bit.ly/3wLUjvb`

com/deeplearning/cudnn/developer-guide/index.html#tensor-ops-conv-functions. Note, the GPU nodes on Midway 3 all have Tensor Cores (Volta, Turing, Ampere generations). The Midway 2 GPU nodes do not have Tensor Cores.

3 Submission and Documentation

Your repository should include:

- A single PDF writeup containing all analysis and plots for both the gradient descent curve fitting problem and the neural network training code.
- Your source code for both parts.
- Basic documentation on how to run your code (for instance, a `README.txt` in the same directory as your code). If you are using a compiled language, include a Makefile.
- If you are using a Jupyter notebook friendly language like Python or Julia (for the first section only), you are welcome to in-line your writeup along with your source code into a single (nicely formatted) workbook and turn that in. Be sure to include a printed out PDF of the notebook along with your `.ipynb` file! If submitting in notebook format, there is no need to include a README.