Programming Assignment 2: Learning with Gradients

CS4787/5777 — Principles of Large-Scale ML — Fall 2023

Project Due: October 4, 2023 at 11:59pm

Late Policy: Up to two slip days can be used for the final submission.

Please submit all required documents to **Gradescope**. There are two separate assignments there, one for the code and one for the report. You need to submit **both**.

This is a group project. You can either work alone, or work **ONLY** with the other members of your group, which may contain AT MOST three people in total. Failure to adhere to this rule (by e.g. copying code) may result in an Academic Integrity Violation.

Overview and Background on Gradient Descent: In this project, you will be implementing and evaluating gradient descent and stochastic gradient descent for empirical risk minimization on a multinomial logistic regression task on the MNIST dataset. We'll start by reviewing multinomial logistic regression as a method. Consider a multi-class classification task with c classes in which the predictions $\hat{y}$ are distributions over the c classes. That is,

$$\hat{y} \in \mathbb{R}^c, \qquad \hat{y}_j \geq 0 \text{ for all } j \in \{1, \ldots, c\} \qquad \text{and} \qquad \sum_{j=1}^{c} \hat{y} = 1.$$

The example labels y are one-hot vectors identifying a single class. That is, $y \in \mathbb{R}^c$, each $y_j \in \{0, 1\}$, and there exists exactly one $j \in \{1, \ldots, c\}$ such that $y_j = 1$. The examples themselves are vectors $x \in \mathbb{R}^d$. Multinomial logistic regression uses a linear model hypothesis with a softmax function. Specifically, given a parameter matrix $W \in \mathbb{R}^{c \times d}$, the hypothesis is

$$h_W(x) = \text{softmax}(Wx)$$

where the softmax function maps from $\mathbb{R}^c$ to $\mathbb{R}^c$ and is defined by

$$\left(\text{softmax}(u)\right)_j = \frac{\exp(u_j)}{\sum_{k=1}^{c} \exp(u_k)}.$$

We want to perform empirical risk minimization using the cross entropy loss function

$$L(\hat{y}, y) = -\sum_{j=1}^{c} y_j \cdot \log(\hat{y}_j),$$

where here $\log$ refers to the natural logarithm, and using ℓ_2 regularization. Our regularized empirical risk for some dataset $\mathcal{D}$, using regularization parameter $\gamma > 0$ is defined as

$$R(h_W) = \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} L(h_W(x), y) + \frac{\gamma}{2} \|W\|_F^2,$$

where $\|W\|_F$ denotes the Frobenius norm (also known as Euclidean norm), which is just the square root of the sum of the squares of the entries of W.

$$\|W\|_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} |w_{i,j}|^2.$$

Part 1: The gradient of the loss function. A key part of gradient descent is computing the gradient of the loss. For this problem, the gradient of the (unregularized) loss for a single example with respect to the parameters W is

$$\nabla_W L(h_W(x), y) = (\text{softmax}(Wx) - y)\, x^T.$$

Because we want to evaluate the systems efficiency of various methods, and we don't want to be confused by the overhead of your automatic differentiation engine from PA1, we'll just compute the gradient directly (i.e. you'll write a function that computes the gradient for this loss). Derive an expression for the gradient of the total loss with regularization over a whole dataset with respect to the weights W. Also write out an expression for a single update step of gradient descent. **Include these expressions (and the derivation) in your report.**

Implement a function `multinomial_logreg_loss_i` to compute the loss of a single example (x, y) given parameters W.

Implement a function `multinomial_logreg_grad_i` to compute the gradient of a single example (x, y) given parameters W. (Hint: you may want to use numerical differentiation, your backprop engine from PA1, or PyTorch to check your work.)

Part 2: Gradient descent.

Implement the function `gradient_descent`, the gradient descent algorithm. Your implementation should use the function `multinomial_logreg_batch_grad` which computes the total gradient of the loss function for whole batch (by default, the whole dataset), using your implementation of `multinomial_logreg_grad_i`.

Run the gradient descent algorithm with L2 regularization parameter $\gamma = 0.0001$ and step size $\alpha = 1.0$ for 10 iterations, and record the value of the parameters every 10 iterations (i.e. only at the beginning and end). Time how long it took to run the algorithm. How long do you expect it would take to run 1000 iterations with this method? **Report these results in your report.**

Part 3: Computing more efficiently

Re-implement the functions `multinomial_logreg_batch_loss` and `multinomial_logreg_batch_grad` to use NumPy matrix arithmetic rather than Python for-loops to express computations of the loss and gradients of the whole dataset more efficiently. You should be able to implement this using no Python for-loops at all.

Re-run the gradient descent algorithm with the same parameters as in Part 2, for 10 epochs as before. Time how long it took to run the algorithm. How does this compare to your measurements from Part 2? Which approach is faster? Can you explain why? **Report these results in your report.**

Part 4: Evaluating gradient descent

Implement a function `multinomial_logreg_error` to compute the error of the classifier given parameters W.

Run your gradient descent algorithm with L2 regularization parameter $\gamma = 0.0001$ and step size $\alpha = 1.0$ for 1000 iterations, and record the value of the parameters every 10 iterations (i.e. a total of 101 records).

Evaluate the training error, test error, training loss, and test loss of each recorded model exactly, using the entire MNIST training or test dataset. Plot the resulting error and loss with your results in part in four figures, one for Training error, one for Test error, one for Training loss, and one for Test loss. Here, the x-axis of your figures should report iterations. **Include these plots in your report.**

Background on Stochastic Gradient Descent: So far in class, we've been talking about stochastic gradient descent implemented as the following

Algorithm 1 Minibatch SGD with Fixed α and With-Replacement Random Sampling

procedure SGD($(f_1, \ldots, f_n), \alpha, w_0, B, T$)
  initialize $w \leftarrow w_0$
  for $t = 1$ **to** T **do**
    for $b = 1$ **to** B **do**
      sample $\tilde{i}_b$ **uniformly at random from** $\{1, \ldots, n\}$
    end for
    $w \leftarrow w - \alpha \cdot \frac{1}{B} \sum_{b=1}^{B} \nabla f_{\tilde{i}_b}(w)$
  end for
  return w
end procedure

(Possibly we may want to use some averaging or random sampling of the iterates, but we'll ignore that for now.) Here f_i denotes the component loss function for the ith training example in the dataset, α denotes the step size, and B denotes the minibatch size. One problem with **Algorithm 1** is that since the training examples are chosen at random, they are unpredictable (or to use a more technical term, they have poor memory locality). This can make our results difficult to replicate, which can be bad when we want to scale our systems and test them automatically. Sampling at random can also be bad for the memory subsystem of the processor we are running on, especially if it depends on caches that try to predict which data to prefetch and make available for quick access (e.g. a CPU). We will discuss this more in detail later in the course, when we talk about hardware for machine learning. But for the moment, one (naive) way to address this problem is to just sample the training examples in a predictable order. And what order is more predictable than the order in which the training examples appear in memory? This motivates the development of the following **sequential-scan-order** version of SGD.

Algorithm 2 Minibatch SGD with Fixed α and Sequential Sampling

procedure SGD($(f_1, \ldots, f_n), \alpha, w_0, B,$ `num_epochs`)
  require that B divides n
  initialize $w \leftarrow w_0$
  for $t = 1$ **to** `num_epochs` **do**
    for $i = 0$ **to** $(n/B) - 1$ **do**
      $w \leftarrow w - \alpha \cdot \frac{1}{B} \sum_{b=1}^{B} \nabla f_{i \cdot B + b}(w)$
    end for
  end for
  return w
end procedure

Here, `num_epochs` denotes the number of **epochs**, or passes through the dataset, used in training. The total number of iterations here will be $T =$ `num_epochs` $\cdot n$. In theory, **Algorithm 2** can run individual iterations faster than **Algorithm 1** because its memory accesses are more predictable. (Although note that our convergence proofs from class do not apply to this modified sequential-sampling algorithm, and more care is needed to get convergence guarantees here.)

Another example selection method that is very popular is without-replacement sampling, also known as random reshuffling. Here, the examples are permuted ("shuffled") every epoch. This method tends to yield faster convergence in practice, compared to with-replacement random sampling.

Algorithm 3 Minibatch SGD with Fixed α and With-Replacement Random Sampling

procedure SGD($(f_1, \ldots, f_n), \alpha, w_0, B,$ `num_epochs`)
  require that B divides n
  initialize $w \leftarrow w_0$
  for $t = 1$ **to** `num_epochs` **do**
    sample σ uniformly from the set of permutations of $\{1, \ldots, n\}$

```
    for i = 0 to (n/B) − 1 do
        w ← w − α · (1/B) Σ_{b=1}^{B} ∇f_{σ(i·B+b)}(w)
    end for
  end for
  return w
end procedure
```

In the rest of this programming assignment, we will implement, and explore the performance of all three of these algorithms.

Part 5: Implementing SGD.

1. Implement the three stochastic gradient descent algorithms described above in the Background section.
   - For consistency, the functions in the starter code in `main.py` take in how long the algorithm needs to run for in terms of epochs, i.e. total passes through the dataset. You will need to modify Algorithm 1 from its description above accordingly, using the expression $T = \texttt{num\_epochs} \cdot n/B$ for Algorithm 3. This ensures that the same number of total gradient samples are used for all the algorithms.
   - Also note that due to their use of randomness, the algorithms here that use random sampling will have less stringent autograding.
2. Run all three SGD algorithms (Algorithms 1, 2, and 3) **without minibatching** (i.e. with $B = 1$) with L2 regularization parameter $\gamma = 0.0001$ and step size $\alpha = 0.001$ for 10 epochs. Set the monitor period to 6000 i.e. record the value of the parameters every 6000 update iterations per epoch; ex: n=60000, monitor_period=6000 results in monitoring 10 times per epoch.
3. Now run all three minibatched SGD algorithms with L2 regularization parameter $\gamma = 0.0001$, step size $\alpha = 0.05$, and minibatch size $B = 60$ for 10 epochs, and record the value of the parameters every 100 batch update iterations (i.e. 10 times per epoch).
4. Evaluate the training error and test error of each recorded model exactly, using the entire MNIST training and test dataset.
5. Plot the resulting error against the number of epochs in two figures, one for Training error and one for Test error. How does the performance of the six methods compare? **Include these plots and the answer to this question in your report.**

Part 6: Exploration.

1. For with-replacement stochastic gradient descent (Algorithm 1) evaluate some other values of the step size other than the one given to you in Part 5.
2. For SGD (Algorithm 1), find a value of the step size and minibatch size that allows the algorithm to reach a lower training error after 10 epochs than the step size given to you in Part 5. What effect do these changed hyperparameters have on the final test error?
3. For SGD (Algorithm 1), can you find a value of the step size that allows the algorithm to reach the same (or better) training error after 5 epochs than could be achieved in 10 epochs using the step size given to you in Part 5? (This may be the same step size that you used in

the previous step.) What effect do these changed hyperparameters have on the final test error after 5 epochs?

4. For at least three different hyperparameter configurations you explored in this Part, plot the resulting error against the number of epochs in two figures, one for Training error and one for Test error, just as you did for the evaluation in Part 5. **Include these plots and the answers to the questions from this part in your report.**
    - If you found hyperparameters that improved the performance in Steps 2 and 3 use those hyperparameters for these figures.

Part 7: Systems Evaluation.

1. Measure the runtime of all three algorithms using the configurations prescribed in Parts 5.2 and 5.3. (This will involve measuring six runtimes, two for each algorithm.) How does the runtime of the algorithms compare?
    - To measure the runtime of each algorithm, measure the wall clock time it takes it to run all its epochs, and average your result across five (5) total runs of the algorithm.
2. You should observe that some of these algorithm settings run an epoch in substantially less time than the others do. Can you explain why? **Include your measurements and an answer to the questions from this part in your report.**

What to submit:

1. An implementation of the functions in main.py.
2. A lab report containing:
    - A formula for the gradient of the loss function and a single update step for gradient descent, as described in Part 1.
    - Timing results for gradient descent from Parts 2 and 3, and an explanation for how they compare.
    - Plots of the training and test error of gradient descent, from Part 4.
    - Plots of the training and test error of various versions of stochastic gradient descent, from Part 5, as well as text describing how the six methods compare.
    - Plots for the three hyperparameter configurations you studied in Part 6, and answers to the questions from Part 6.
    - The measured runtime of all six algorithms, as described in Part 7. Text describing how the runtimes of the algorithms compare, and an explanation for why we might expect some setups to run faster than others.

Setup:

1. Run `pip3 install -r requirements.txt` to install the required python packages