Assigned: 25 September

Homework #3

EE 541: Fall 2023

Due: Sunday, 08 October at 23:59. Submission instructions will follow separately on canvas.

1. An MLP has two input nodes, one hidden layer, and two outputs. Recall that the output for layer l is given by $a^{(l)} = h_l\left(W_l a^{(l-1)} + b_l\right)$. The two sets of weights and biases are given by:

$$W_1 = \begin{bmatrix} 1 & -2 \\ 3 & 4 \end{bmatrix} \qquad b_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 2 & 2 \\ 2 & -3 \end{bmatrix} \qquad b_2 = \begin{bmatrix} 0 \\ -4 \end{bmatrix}$$

   The non-linear activation for the hidden layer is ReLU (rectified linear unit) – that is $h(x) = \max(x, 0)$. The output layer is linear (*i.e.*, identity activation function). What is the output activation for input $x = [+1; -1]^T$?

2. The hd5 format can store multiple data objects in a single file each keyed by object name – *e.g.*, you can store a numpy float array called regressor and a numpy integer array called labels in the same file. Hd5 also allows fast non-sequential access to objects without scanning the entire file. This means you can efficiently access objects and data such as `x[idxs]` with non-consecutive indexes *e.g.*, `idxs = [2, 234, 512]`. This random-access property is useful when extracting a random subset from a larger training database.

   In this problem you will create an hd5 file containing a numpy array of binary random sequences that you generate yourself. Follow these steps:

   (1) Run the provided template python file (`random_binary_collection.py`). The script is set to `DEBUG` mode by default.

   (2) Experiment with the assert statements to trap errors and understand what they are doing by using the shape method on numpy arrays, etc.

   (3) Set the `DEBUG` flag to `False`. Manually create 25 binary sequences each with length 20. It is important that you do this by hand, *i.e.*, , **do not use a coin, computer, or random number generator**.

   (4) Verify that your hd5 file was written properly by checking that it can be read-back.

   (5) Submit your hd5 file as directed.

3. Logistic regression

The MNIST dataset of handwritten digits is one of the earliest and most used datasets to benchmark machine learning classifiers. Each datapoint contains 784 input features – the pixel values from a 28×28 image – and belongs to one of 10 output classes – represented by the numbers 0-9.

In this problem you will use numpy to classify input images using a logistic-regression. Use only Python standard library modules, `numpy`, and `mathplotlib` for this problem.

(a) Logistic "2" detector

In this part you will use the provided MNIST handwritten-digit data to build and train a logistic "2" detector:

$$y = \begin{cases} 1 & \mathbf{x} \text{ is a "2"} \\ 0 & \text{else.} \end{cases}$$

A logistic classifier takes learned weight vector $\mathbf{w} = [w_1, w_2, \ldots w_L]^T$ and the unregularized offset bias $b \triangleq w_0$ to estimate a probability that an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_L]^T$ is "2":

$$p(\mathbf{x}) = P\left[Y = 1|\mathbf{x}, \mathbf{w}\right] = \frac{1}{1 + \exp\left(-\left(\sum_{i=1}^{L} w_k \cdot x_k + w_0\right)\right)} = \frac{1}{1 + \exp\left(-\left(w^T x + w_0\right)\right)}.$$

Train a logistic classifier to find weights that minimize the binary log-loss (also called the binary cross entropy loss):

$$l(w) = -\frac{1}{N} \sum_{i=1}^{N} \left(y_i \log p(x)\right) + \left(1 - y_i\right) \log\left(1 - p(x)\right)$$

where the sum is over the N samples in the training set. Train your model until convergence according to some metric you choose. Experiment with variations of ℓ_1- and/or ℓ_2-regularization to stabilize training and improve generalization.

Submit answers to the following.

i. How did you determine a learning rate? What values did you try? What was your final value?

ii. Describe the method you used to establish model convergence.

iii. What regularizers did you try? Specifically, how did each impact your model or improve its performance?

iv. Plot log-loss (*i.e.,* learning curve) of the training set and test set on the same figure. On a separate figure plot the accuracy against iteration number of your model on the training set and test set. Plot each as a function of the iteration number.

v. Clasify each input to the binary output "`digit is a 2`" using a 0.5 threshold. Compute the final loss and final accuracy for both your training set and test set.

Submit your trained weights to Autolab. Save your weights and bias to an hdf5 file. Use keys `w` and `b` for the weights and bias, respectively. `w` should be a length-784 numpy vector/array and `b` should be a numpy scalar. Use the following as guidance:

```
with h5py.File(outFile, 'w') as hf:
  hf.create_dataset('w', data = np.asarray(weights))
  hf.create_dataset('b', data = np.asarray(bias))
```

Note: you will **not** be scored on your models overall accuracy. But a low-score may indicate errors in training or poor optimization.

(b) Softmax classification: gradient descent (GD)

In this part you will use soft-max to peform multi-class classification instead of distinct "one against all" detectors. The target **vector**

$$[\mathbf{Y}]_l = \begin{cases} 1 & \mathbf{x} \text{ is an "}l\text{"} \\ 0 & \text{else.} \end{cases}$$

for $l = 0, \ldots, K-1$. You can alternatively consider a scalar output Y equal to the value in $\{0, 1, \ldots, K-1\}$ corresponding to the class of input $\mathbf{x}$. Construct a logistic classifier that uses K seperate linear weight vectors $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{K-1}$. Compute estimated probabilities for each class given input $\mathbf{x}$ and select the class with the largest score among your K predictors:

$$P\left[Y = l|\mathbf{x}, \mathbf{w}\right] = \frac{\exp(\mathbf{w}_l^T \mathbf{x})}{\sum_{i=0}^{K} \exp(\mathbf{w}_i^T \mathbf{x})}$$

$$\hat{Y} = \arg\max_l P\left[Y = l|\mathbf{x}, \mathbf{w}\right].$$

Note that the probabilities sum to 1. Use log-loss and optimize with batch gradient descent. The (negative) likelihood function on an N sampling training set is:

$$L(w) = -\frac{1}{N} \sum_{i=1}^{N} \log P\left[Y = y^{(i)}|x^{(i)}, w\right]$$

where the sum is over the N points in our training set.

Submit answers to the following.

i. Compute (by-hand) the derivative of the log-likelihood of the soft-max function. Write the derivative in terms of conditional probabilities, the vector $\mathbf{x}$, and indicator functions (*i.e.,* do not write this expression in terms of exponentials). You need this gradient in subsequent parts of this problem.

ii. Implement batch gradient descent. What learning rate did you use?

iii. Plot log-loss (*i.e.,* learning curve) of the training set and test set on the same figure. On a separate figure plot the accuracy against iteration number of your model on the training set and test set. Plot each as a function of the iteration number.

iv. Compute the final loss and final accuracy for both your training set and test set.

(c) Softmax classification: stochastic gradient descent

In this part you will use stochastic gradient descent (SGD) in place of (deterministic) gradient descent above. Test your SGD implmentation using single-point updates and a mini-batch size of 100. You may need to adjust the learning rate to improve performance. You can either: modify the rate by hand or according to some decay scheme or you may choose a single learning rate. You should get a final predictor comparable to that in the previous question.

Submit answers to the following.

i. Implement SGD with mini-batch size of 1 (*i.e.,* compute the gradient and update weights after each sample). Record the log-loss and accuracy of the training set and test set every 5,000 samples. Plot the sampled log-loss and accuracy values on the same (respective) figures against the batch number. Your plots should start at iteration 0 (*i.e.,* include initial log-loss and accuracy). Your curves should show performance comparable to batch gradient descent. How many iterations did it take to acheive comparable performance with batch gradient descent? How does this number depend on the learning rate? (or learning rate decay schedule if you have a non-constant learning rate).

ii. Compare (to batch gradient descent) the total computational complexity to reach a comparable accuracy on your training set. Note that each iteration of batch gradient descent costs an extra factor of N operations where N is the number data points.

iii. Implement SGD with mini-batch size of 100 (*i.e.,* compute the gradient and update weights with accumulated average after every 100 samples). Record the log-loss and accuracies as above (every 5,000 **samples** – not 5,000 batches) and create similar plots. Your curves should show performance comparable to batch gradient descent. How many iterations did it take to acheive comparable performance with batch gradient descent? How does this number depend on the learning rate? (or learning rate decay schedule if you have a non-constant learning rate).

iv. Compare the computational complexity to reach comparable perforamnce between the 100 sample mini-batch algorithm, the single-point mini-batch, and batch gradient descent.

Submit your trained weights to Autolab. Save your weights and bias to an hdf5 file. Use keys `W` and `b` for the weights and bias, respectively. `W` should be a 10×784 numpy array and `b` should be 10×1 – shape: `(10,)` – numpy array. The code to save the weights is the same as (a) – substituting `W` for `w`.

Note: you will **not** be scored on your models overall accuracy. But a low-score may indicate errors in training or poor optimization.