Assignment 2
CSE 547/Stat 548: ML for Big Data

University of Washington

0 Certify you read the HW Policies [0 points]

Please explicitly answer the three questions below and include your answers marked in a "problem 0" in your solution set. *If you have not included these answers, your assignment will not be graded.*

Readings: Read the required material.

Submission format: Submit your HW as a *single* typed pdf document; this document must contain all plots. Handwritten solutions (even if they are scanned in to the pdf) will *not* be accepted. Submit all your code in a gzipped tarball named `code.tgz`. It is encouraged that students use latex, though another comparable typesetting method is also acceptable. Some free tools that might help: ShareLaTeX (`www.sharelatex.com`), TexStudio (Windows), MacTex (Mac), TexMaker (cross-platform), and Detexify[2] (online). If you want to type, but don't know (and don't want to learn) LaTeX, consider using a markdown editor with real-time preview and equation editing (e.g., `stackedit.io`, `marxi.co`).

Written work: Please provide succinct answers *along with succinct reasoning for all your answers*. Points may be deducted if long answers demonstrate a lack of clarity. Similarly, when discussing the experimental results, concisely create tables and figures to organize the experimental results. In other words, all your explanations, tables, and figures for any particular part of a question must be grouped together.

Python source code: for the programming assignment. Please note that we will not accept Jupyter notebooks. Submit your code together with a neatly written README file to instruct how to run your code with different settings (if applicable). We assume that you always follow good practice of coding (commenting, structuring); these factors are not central to your grade.

Coding policies: You must write your own code. You are welcome to use any Python libraries for data munging, visualization, and numerical linear algebra. Examples includes Numpy, Pandas, and Matplotlib. If you use TensorFlow or PyTorch, you may *not* use any functions which define a neural network for you, e.g. no Keras is allowed. In PyTorch, you may *not* use the torch.nn.module (or any of the inherited functions). Basically, this means that you are not allowed to define any sort of neural network through the library: you must just write out the "forward pass" yourself; you may not use any built in functions/libraries which specify the number of nodes/layers in your network; the ML library you are using should not ever know you are coding up a neural net (it should be building computation graphs for the computations you specify). You are, however, allowed to use the torch.nn.functional interface provided by PyTorch.

On some questions, we will explicitly not allow ML packages, like Scikit-Learn, TensorFlow, or PyTorch. If in doubt, post to the message boards or email the instructors.

Collaboration: It is acceptable for you to discuss problems with other students; it is not acceptable for students to look at another students written answers. It is acceptable for you to discuss coding questions with others; it is not acceptable for students to look at another students code. Each student must understand, write, and hand in their own answers. In addition, each student must write and submit their own code in the programming part of the assignment.

Acknowledgments: We expect the students not to refer to or seek out solutions in published material from previous years, on the web, or from other textbooks. Students are certainly encouraged to read extra material for a deeper understanding.

0.1 List of Collaborators

List the names of all people you have collaborated with and for which question(s).

0.2 List of Acknowledgements

If you do inadvertently find an assignment's answer, acknowledge for which question and provide an appropriate citation (there is no penalty, provided you include the acknowledgement). If not, then write "none".

0.3 Certify that you have read the instructions

Please make sure to read and follow these instructions. Write "I have read and understood these policies" to certify this. If you do not yet understand what it means not to use the PyTorch "torch.nn.module", please state the you will figure out how to avoid using the nn.module class (e.g. by making sure you post questions on the discussion board/going to office hours/websearch/etc if in doubt). It is fair game to use the "torch.nn.functional" mode.

1 Generalization, Streaming, and SGD (15 points)

In class, we examined using SGD for empirical loss minimization where we have an N sized training set $\mathcal{T}$. The empirical loss considered was:

$$F(w) = \frac{1}{N} \sum_{(x,y)\in\mathcal{T}} \ell(w, (x, y)) \,.$$

Here, gradient descent for the function F is the algorithm:

1. Initialize at some $w^{(0)}$.
2. Sample (x, y) uniformly at random from the set $\mathcal{T}$
3. Update the parameters:

$$w^{(k+1)} = w^{(k)} - \eta_k \cdot \nabla \ell(w^{(k)}, (x, y))$$

   and go back to 2.

We provided guarantees assuming that F was smooth and the gradients in our training set were uniformly bounded, $\|\nabla \ell(w, (x, y))\| \leq B$.

However, in practice, we care about generalization, i.e. statements on how well we do on the underlying distribution. Define:

$$\mathcal{L}(w) = \mathbb{E}_{(x,y)\in\mathcal{D}} \ell(w, (x, y)) \,,$$

where $\mathcal{D}$ is the underlying distribution.

Broadly, speaking generalization is about making statements on the underlying distribution given only samples. It is often (technically) challenging to translate statements on our empirical loss $F(w)$ to the true loss $\mathcal{L}(w)$ due overfitting issues, a property on $F(w)$ does not necessarily imply the same property on $\mathcal{L}(w)$ (to make this leap, we often have to take into account some measure of the "complexity" of function class like the VC dimension/Rademacher complexity/metric entropy, etc). Using our understanding from class, we will see a remarkably simple (and powerful) "algorithmic" procedure to make this leap.

Suppose we sought to find a point where $||\nabla \mathcal{L}(w)||^2$ was small. Obtaining this quantity to be small even in expectation would be acceptable for this problem. Assume that $\mathcal{L}$ is smooth and that the gradients are uniformly bounded, $\|\nabla \ell(w, (x, y))\| \leq B$ for all parameters and all possible points (x, y) (under $\mathcal{D}$).

1. (8 points) Assume we have sampling access to our underlying distribution $\mathcal{D}$. Explain how we can make $||\nabla \mathcal{L}(w)||^2$ small in expectation. What can you guarantee if you obtain m samples and how you would do this? To make a precise claim, how are you appealing to the theory developed in class? Specify how the argument and how the proof change here. (Hint: think carefully about the guarantee of SGD and our discussion in class).

2. (7 points) Suppose we construct an N sized training set $\mathcal{T}$, where each point is sampled under $\mathcal{D}$; then we construct the empirical loss function $F(w)$; then we run SGD on F for K steps (suppose $K \geq N$). Do you see an argument on this procedure that implies something non-trivial (and technically correct) about $||\nabla \mathcal{L}(w)||^2$, even in expectation? If so, what is this argument? If not, comment on what was special about the argument in the first part of this question?

Remark: In the second part of this question, trying to go from $F(w)$ to $\mathcal{L}(w)$ is subtle. One influential paper related to the second part of this question makes the important connection between "Algorithmic Stability" and generalization. See [Bousquet and Elisseeff(2002)]. This is beyond the scope of the class, and we should only be appealing to using very direct (and general) arguments in this question.

1.1 Extra credit: Decaying the stepsize for an "anytime" algorithm (15 points)

In class, we set the stepsize as function of K, which was the known stopping time. In particular, we used a constant stepsize of $O(1/\sqrt{K})$. However, if we do not know K in advance and simply wanted to run SGD with a decaying stepsize, we would use $\eta_k = O(1/\sqrt{k})$. Modify the argument so using that using this decaying η_k (also specify the dependencies of B and $F(w_0) - F*$), we can obtain the same bound presented in class and the notes (up to constant factors).

This observation is convenient in that it provide an "anytime" algorithms.

2 GD vs. Adaptive GD (20 points)

Let us compare GD, with a constant stepsize η, to Adagrad on two simple one dimensional objective functions. Adagrad will use the stepsize at iteration k where:

$$\eta_k = \frac{C}{\sqrt{\sum_{j=0}^{k} \|\nabla F(w^{(j)})\|^2}}$$

1. (10 points) Consider $F(w) = \frac{1}{2}w^2$. Let us start at $w_0 = 1$. For this problem, a constant stepsize certainly makes sense for GD. As GD will converge in one step if our stepsize is 1; let us, for the sake of comparison, use a constant stepsize of $3/4$ for GD (still a very good choice for GD). Similarly, for Adagrad, it will converge in one step if we use $C = 1$. Hence, for the sake of comparison, let us make the constant $C = 3/4$ (also a good choice for Adagrad). Obviously, we do not expect algorithms to converge in one step, so this is a reasonable comparison. Do either one of the two questions:

   (a) Analytically, characterize the convergence rates of GD and Adagrad with these parameter settings. Your analysis must provide sharp convergence rates to make this comparison fair; a loose upper bound on either algorithm would not make a fair comparison. Do they differ substantially, say by more than a constant factor, in the time it takes $w^{(k)}$ to become small, say less than some ε?

(b) Empirically, plot the learning curves, where the iteration is on the x-axis and the log of $F(w^{(k)}) - F_*$ on the y-axis. For this question, you must code the algorithm yourself (without using any AD or any solvers), which is easy to do. This so that you know that your algorithm is being implemented as specified. (You don't need to answer this: Why do we use the log for the y-axis?) Do these rates differ "substantially"? Note if the function value of one algorithm at iteration k, is the same as another algorithm at $O(k)$, this is only a constant factor difference; a difference that looks like $O(k^2)$ is definitely "substantial" and even $O(k \log k)$ is noteworthy. Your plots should be clean and reflect the behavior that you are arguing for.

2. (10 points) Consider our favorite "non-smooth" and convex function, namely, the absolute value function, $F(w) = |w|$. Let us start at $w_0 = 1$. For non-smooth problems, a constant stepsize for GD is clearly not a great idea. Regardless let us compare the behavior of GD and Adagrad, just as before. For, GD use a stepsize of $3/4$ and for adagrad, use $C = 3/4$ (for the same reason as before). Again choose one of the two to answer:

   (a) Analytically, just as before, sharply characterize the behavior of these algorithms. A sloppy bound (loosing more than constants) would not provide a fair comparison. Do they different substantially?

   (b) Empirically, compare the two and see if the curves look substantially different. In particular, plot the iteration on the x-axis and $F(w^{(k)}) - F_*$ on the y-axis (Do not use the log for the y-axis this time. You are free to think about why.) Do these rates differ substantially?

3 Understanding non-convexity a little better (15 points)

3.1 Newton's method

The least squares problem can be written as:

$$\min_w L(w) \,, \text{where } L(w) := \frac{1}{2N} \|Xw - Y\|^2$$

where X is our $N \times d$ data matrix and Y is our $N \times 1$ output vector. If we define:

$$\Sigma := \frac{1}{N} X^\top X, \quad u = \frac{1}{N} X^\top Y$$

then it straightforward to see that this expression can be written as: **SK: Fixed defintion of u above.**

$$L(w) = \frac{1}{2} w^\top \Sigma w - u^\top w + \frac{1}{2N} \|Y\|^2 \,.$$

Note that Σ is a positive definite matrix. For this question, we assume that Σ is full rank.

Newton's method is typically thought of as being better than gradient descent due to it using second order information. At some w_0, the method first constructs a second order Taylor's approximation around w_0, and then it makes the new iterate to be the point at which the gradient of this approximation is 0 [1]. In other words, the update in Newton's method is:

$$w \leftarrow w - [\nabla^2 L(w)]^{-1} \nabla L(w)$$

If Σ were not full rank, we could use a pseudo-inverse.

1. (3 points) Starting at some w_0 write out one step of Newton's method (in terms of Σ and u). If you take the "Newton step", what point do you get to?

2. (1 points) Comment on the loss of this point and how it compares to the minimal function value?

3.2 A simple, non-convex case (11 points)

While our favorite non-convex problem is the eigenvector problem, the following problem is arguably a close second. It is particularly nice due to its simplicity.

Consider the objective function:

$$F(w) = \frac{1}{2} w^\top A w - b^\top w + c$$

where A is a symmetric matrix, b is a vector, and c is a scalar. Our goal is to minimize this objective function. Again assume that Σ is full rank.

Assume that A is still symmetric, yet now suppose that it has at least one negative eigenvalue. Hence, the problem is non-convex due to this, and the motivation from regression is no longer valid. One reason to understand this function is that, at least locally, this is how we might expect many reasonable functions to behave, as this $F(w)$ is essentially an arbitrary quadratic. This is a reasonable abstraction to help sharpen our intuition of the local behavior in non-convex problems.

Again, suppose you start at w_0 and take a step of Newton's method. Let this new point be w_1.

1. (2 points) What is the point w_1 and objective value $F(w_1)$.

2. (1 points) What is the gradient at w_1? (Note: you might have expected this answer from Newton's method).

3. (3 points) What is the minimal objective value of $F(\cdot)$? Did we achieve it?

4. (5 points) Suppose you now are at w_1. In terms of the eigendecomposition $A = UDU^\top$ what are the directions of movement which *strictly* decrease the objective value from w_1? Why?

[1] Newton's method is really a root findding algorithm, where it uses a *first order* approximation to find the roots of some equation. In the context of optimization, Newton's method is being used to find the "zeros" (the roots) of the gradient equation. That it is being applied to find the roots of the a first order equation is what makes it look like a second order method in the context of optimization

Remark: First order vs. Second order stationary points. This problem should give you some intuition about first order and second order stationary points. With Newton's method, we hopped to a first order stationary point. It also makes clear why having a zero gradient is a rather weak guarantee.

Remark: Trust region methods. Naively, it may seem like Newton's method is a poor idea. Do you see how to modify Newton's method to make it more robust? Trust region methods are one set of methods which can be considered a variant of Newton's method, that can be very effective (provided you can compute Hessians).

4 Empirical Optimization (50 points)

Cautionary note: You must figure out how to get reasonable performance numbers. If you were not able to train your MLP in HW1, determine what you did incorrectly as you need to get this correct here. Also, remember you have limited cloud resources, so make sure to prototype and parameter tune locally.

We will now consider the multi-label classification problem. In the multi-label problem, there are multiple labels that could be "on" for each input x. You will use either the square loss or the binary logistic loss and consider training two models, namely (i) a linear model and (ii) a multi-layer perceptron with a number of hidden nodes that you will tune.

You will try out three methods in each of the following: 1) SGD with a mini-batch size that you tune. You will use the same minibatch size for the other algorithms. 2) Try out Polyak's "heavy ball method" (aka momentum) or Nesterov's accelerated gradient descent (NAG) 3) either Adagrad or Adam. You must tune all the parameters of these methods.

The dataset contains 18 total categories with a number of categories for each supercategory (vehicle or animal). In the dataset provided, each image contains potentially multiple objects of different supercategories, such as car, boat, etc. In this exercise we shall build a classifier that learns to identify *all the categories of objects* present in each image, by optimizing either a square loss or a logistic loss objective. For the purposes of learning these classifiers, we shall use a larger version of the dataset and features from the first homework - the dataset has been shared with you on Google Drive under the folder data2.

The objective function we choose to optimize is

$$L(w) = \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k}\ell(y_{ij}, f_{ij}(w)),$$

where $f_{ij}(w) = w_j^\top x_i$ and $w_j \in \mathcal{R}^d$ is the jth column of $w \in \mathcal{R}^{d\times k}$. Here, w is the linear model we wish to optimize over $\lambda > 0$ is the strength of ℓ_2 regularization. Here ℓ is the loss function:

- $\ell(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ is the square loss.
- $\ell(y, \hat{y}) = y\log(1 + \exp(-\hat{y})) + (1 - y)\log(1 + \exp(\hat{y}))$ is the logistic loss where the true label $y \in \{0, 1\}$.

Notice that we encode y_i as a binary vector of length $k = 18$ (the number of categories) where a 1 indicates the presence of a category and 0 indicates the absence. *Important:* Note that while the loss function has a sum over k, you should be able to avoid a for loop for *both* of these loss functions.

For the square loss (with linear regression), the objective function can be written more concisely as:

$$L(w) = \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \tfrac{1}{2} \cdot \|y_i - w^\top x_i\|^2.$$

Determine which works better for a linear classifier (either the regularized square loss or regularized logistic regression method) and use that loss throughout this question.

When using a MLP instead, the objective function changes as follows: $f_{ij}(w) = \langle w_j^{(2)}, \text{relu}(w^{(1)} x_i) \rangle$, where $w^{(1)} \in \mathcal{R}^{h \times d}$ are the weights in the first layer and h is the number of hidden nodes. Again, $w_j^{(2)} \in \mathcal{R}^h$ is the jth column of $w^{(2)} \in \mathcal{R}^{h \times k}$, the weights of the second layer.

You should have 4 plots for each question (train/validation and linear/mlp). Make sure all the plots are marked appropriately.

4.1 SGD and Linear Regression [10 points]

Now consider running stochastic gradient descent on $L(w)$. Make sure to label your plots clearly and make it clear which questions they are associated with. For reporting your training loss, you may compute it on only a subset of the training loss if you find that is faster (you must still train on the full training set!).

1. What mini-batch size do you use? What stepsize did you use? What value of λ did you use? Specify your stepsize scheme if you chose to decay your stepsize. Which loss function did you find works better?

2. Report your training and development loss (for the loss you use) after a number of iterations that corresponds to half an epoch, so after (# points in training set) / (2 * # mini-batch size) updates. Make a plot where the y-axis has these numbers and the x-axis has the scale of the epoch number. All curves should be on one same plot. Do this plot for both the linear regression case and for your MLP.

3. Compute the AveragePrecision Score (the AP score) in the same manner, and plot these quantities (in a single plot) evaluated over the training and development dataset. Here, make sure to start your x-axis at a slightly later iteration to make the behavior of the error more easy to view (it is difficult to view the long run behavior if the y-axis is over too large a range). Report the lowest validation error. Do this plot for both the linear regression case and for your MLP.

4. Now, report the overall AP score on the *test* set (and *not* on the development set). Also report the AP score for each of the 18 classes separately; report these 18 numbers in a table.

You should choose your model appropriately (e.g. the one with the lowest AP score on the development set is a fine candidate). Comment on how the AP scores relate to the class imbalance.

Note:

- It is expected that you obtain a good test error (meaning you train long enough and you regularize appropriately, if needed).
- For this part, you could either obtain gradients via autograd or code them up yourselves (it is easy for a "single layer model" such as this one). The former is easier but the latter would be faster in terms of wall clock time.
- It is crucial/fundamental to set any (hyper-) parameter of the algorithm (such as the step-size/regularization parameter) based on performance on the training/development set for scientific integrity. Inference based on performance on the test set is equivalent to cheating the system.

4.2 Try out the Heavy Ball or Nesterov's method [20 points]

Make sure your plots are clearly marked with questions they are associated with. You may use either Polyak's "heavy ball method" (aka momentum) or Nesterov's accelerated gradient descent (NAG).

1. Report *all* your parameters, as before. Also, which of the two methods did you choose to use?
2. Same as before
3. Same as before
4. Same as before.

4.3 Try out Adagrad or Adam [20 points]

Make sure your plots are clearly marked with questions they are associated with. You may use either Adagrad or Adam.

1. Report *all* your parameters, as before. Also, which of the two methods did you choose to use?
2. Same as before
3. Same as before
4. Same as before.

References

[Bousquet and Elisseeff(2002)] Olivier Bousquet and André Elisseeff. Stability and generalization. *J. Mach. Learn. Res.*, 2, 2002.