Assignment 4
CSE 446: Machine Learning

University of Washington

0 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.

EXTRA CREDIT POLICY: Before you do the extra credit problems, do all the regular questions. Extra credit points will only be awarded if there are answers provided on all the regular questions. Also, credit will be given for an extra credit question only if you honestly attempt all of the extra credit question (e.g. selectively answering only the easy parts of an extra credit problem will not result in credit.).

Readings: Read the required material.

Submission format: Submit your report as a *single* pdf file and submit all your code in separate files, corresponding to separate problems, in a gzipped tarball named `code_problem#.tgz`. For example, `code_problem2.tgz` should contain all the code you used to solve problem 2. The report (in a single pdf file) must include all the plots and explanations for programming questions (if required). We highly recommend typesetting your scientific writing using LaTeX. 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`). Writing solutions by hand will be accepted provided they are neat; written solutions need to be scanned and included into a single pdf.

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. You may **not**, however, use any machine learning libraries such as Scikit-Learn or TensorFlow. 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.

1 Convexity: Linear and Logistic Regression (8 points)

Let us understand a few properties of linear regression (under the square loss) and logistic regression (under the log loss). Suppose we use $\text{sign}(w \cdot x)$ for predicting a label. Here, assume $y_n \in \{-1, 1\}$.

1. (3 points) Suppose our training data are linearly separable. Suppose we run gradient descent for the case of the linear regression: is our squared error converging to 0? Suppose we run gradient descent for the case of the logistic regression: is our log loss converging to 0? Why?
2. (2 points) Suppose now that $d \geq n$, $\lambda = 0$, and our n data points are all linearly independent. Suppose we run gradient descent for the case of the linear regression: is our squared error converging to 0? Suppose we run gradient descent for the case of the logistic regression: is our log loss converging to 0? Why?
3. (3 points) Suppose we are running gradient descent (for the logistic regression problem under the log loss). Suppose at some iteration our misclassification error hits exactly 0 on our training set. If we continue to run gradient descent, do we expect that we will continue to update our parameters? Why or why not?

2 Neural Networks and Non-convex optimization (42 points)

Assume that we have a fully interconnected neural network with L hidden layers and k output nodes (e.g. $k = 10$ for mnist), so $\widehat{y}(x)$ is a k dimensional vector. Suppose that the output at the last layer is just the activation of the k output nodes, so there is no transfer function applied at the last layer.

Assume that our input x is a d-dimensional vector and that we are working with the square loss.

2.1 Computational Complexity (and a little more linear algebra...) (14 points)

Suppose that multiplying a matrix of size $n \times k$ by a matrix of size $k \times d$ — resulting in a matrix of size $n \times d$ — takes time $O(nkd)$ [1].

Let us now examine some computational issues and see how to speed up our code (when using libraries like PyTorch and TensorFlow) by using matrix multiplications appropriately.

Assume that the time to evaluate the transfer function $h(\cdot)$ is C_h. Also, although $h(\cdot)$ is defined as a mapping on a scalar variable, let us overload notation and allow us to apply $h(\cdot)$ to a vector (or a matrix), where we apply $h(\cdot)$ componentwise. So if v is a vector of length d_v then $h(v)$ will be a vector of length d_v, and the computational time for this is $d_v C_h$. This overloading makes it relatively easy (and fast) to write code for the forward pass in a neural net.

Assume that $w^{(1)}, w^{(2)}, \ldots w^{(L+1)}$ are matrices of appropriate size (e.g. $w^{(1)}$ is a matrix of size $d^{(1)} \times d^{(0)}$ where $d^{(0)}$ equals the input dimension d).

1. (3 points) Given the input vector x, write out pseudocode for the forward pass in terms of matrix multiplications, where you apply the transfer function to the vector of activations. The activations and outputs at any layer should be a vector. Specify the dimensions of all the activations and outputs.
2. (4 points) Now suppose we actually want to run the forward pass on a mini-batch of inputs, of size m. Suppose that X is an $m \times d$ matrix (so each row of X corresponds to an input vector now. This is the default use in the community). Now write out the forward pass, where you utilize matrix operations to whatever extent possible (and you apply $h(\cdot)$ componentwise): here you should have matrices of activations and matrices of outputs, each of which has m rows. The output of the network will be the matrix $\widehat{y}(X)$, a matrix of size $m \times k$. Specify the size of all the activations and outputs. Understanding how to do this is critical for writing efficient code in PyTorch and TensorFlow. (You should also understand how to compute the squared error efficiently without "for loops").
3. (3 points) What is the computational time to compute the square loss on the mini-batch X in terms of d, the mini-batch size m, the number of nodes per level $d^{(0)}, d^{(1)}, \ldots, d^{(L+1)}$ (note $d^{(0)} = d$ and $d^{(L+1)} = k$), and the transfer function evaluation cost C_h?

[1] Do you see why $O(nkd)$ is the naive runtime? It is worthwhile understanding why, as it is a straightforward argument. Also, this is the runtime that is actually observed in practice. The theoretically faster algorithms, mentioned in a previous footnote are not actually used in practice.

4. (3 points) Assume that the time to evaluate the gradient of the transfer function is (within a constant of) C_h. What is the computational run time (using BackProp) to compute the gradient the square loss, again on the mini-batch X, with respect to all the parameters $w^{(1)}, w^{(2)}, \ldots w^{(L+1)}$? Why?
5. (1 points) Suppose we just wanted to compute the gradient of the parameters at the top level of the neural network, e.g. we just wanted to compute the $\frac{\partial \ell(y,\widehat{y}(X))}{\partial w^{(L+1)}}$. This partial derivative has far fewer parameters than computing the full gradient. Do you see a procedure whose runtime complexity is less than what you wrote above? If you see a procedure, write it out along with the runtime complexity. If you don't see one, explain why you think it may be unreasonable to obtain a faster runtime.

Remark: The above shows how it is easy to use linear algebra operations to write out the forward pass. Feel free to think about how you might write out the backward pass efficiently in terms of basic matrix operations! (This is a helpful exercise). If you think about it, you will see that there is no "direct" way to write out the backward pass in terms of matrix multiplications; we have to use *Hadamard products*, a componentwise multiplication operation. This does not necessarily mean the backward pass is any less efficient to compute (than what we discussed in class), provided one takes care in how one writes it in code. It does mean that writing your own algorithm to compute the backward pass takes a little care to make it fast (to avoid "for loops"), while the forward pass is relatively straightforward to code.

2.2 Saddle points and Symmetries (12 points)

Here, the underlying function of interest is the square loss on the training set.

1. (2 point) Suppose our MLP has 0 hidden layers (e.g. we are working with linear regression). Is setting all the weights to 0 a saddle point? Why or why not?
2. (3 points) Assume now our MLP has one or more hidden layers. Consider the $\tanh(\cdot)$ transfer function. Suppose we set all our weights to 0. Is this point a saddle point? If so, give a precise proof that this is a saddle point. Else, give a counterexample.
3. (5 points) Again, assume our MLP has one or more hidden layers. Now let us consider the sigmoid transfer function. Consider initializing all the weights to 0. For this case, do we start at a saddle point? If we run gradient descent from this initialization, what can we say about how the weights in any given layer relate to each other?
4. (2 points) Consider the $\tanh(\cdot)$ transfer function. Suppose we have a one hidden layer neural network. Suppose we flip the sign of all the weights. What happens to $\widehat{y}(x)$? What happens in an L-hidden layer network?

2.3 Representation and Non-linear Decision Boundaries (16 points)

Assume our target y is a scalar. Here, when dealing with an MLP, assume we just have one output node (where this node is linear in its activation).

1. Let us consider a linear and a quadratic transfer function:

(a) (4 points) Suppose we have an L-layer neural network and the transfer function is the identity function. Provide an alternative model to using this network in which: 1) all local optima are global optima (and there are no saddle points) and 2) every function that can be represented in this neural network can be represented in your model. Provide concise reasoning for your answer.

(b) (5 points) Suppose we have a one hidden layer neural network and the transfer function is $h(a) = a^2$. Provide an alternative model to using this network which: 1) all local optima are global optima (and there are no saddle points) and 2) every function that can represented in this neural network can be represented in your model. Provide concise reasoning for your answer. (Hint: think about an appropriate feature mapping).

2. Let us consider the parity problem (sometimes referred to as the XOR problem) with two variables. We have seen this problem as it is our usual example of a non-separable dataset. It also has historical significance, due to Minksy and Papert [1] pointing out that the perceptron algorithm is not able to learn this function, which subsequently lead to a decreased interest in neural learning models. For two dimensional inputs, the XOR function is specified as follows: $x = (x[1], x[2])$, where each coordinate of x is in $\{-1, 1\}$ and $y \in \{-1, 1\}$. For a given input x, the XOR function assigns y to be 1 if and only if only both coordinates of x are equal, i.e. $y = 1$ if and only if $x = (-1, -1)$ or $x = (1, 1)$.

(a) (2 points) Give a feature mapping $\phi(x)$ using only linear and quadratic terms so that a linear classifier, i.e. $\text{sign}(w \cdot \phi(x))$, can represent this function with 0 misclassification error.

(b) (5 points) Provide a one hidden layer neural network, with a $\tanh(\cdot)$ transfer function and which has no more than two hidden nodes, which can represent this function (with 0 misclassification error). You may assume your prediction is $\text{sign}(\widehat{y}(x))$ (you are not allowed break ties in your favor if $\widehat{y}(x) = 0$). Specifically, you must provide your weights, and you are free to use a bias term in your activations (see Bishop or the lecture notes for the definition of a bias term in a neural network). Prove that your network's predictions matches the XOR function.

Remark: The XOR function with d dimensional inputs is more tricky to represent and far more problematic to learn from samples.

3 MLPs: Let's try them out on MNIST (40 points)

You will use the full dataset (the same as we used in assignment 2), with all 10 classes.

You are free to use PyTorch, TensorFlow, or directly write your own backprop code. Please note which method you used.

You will have 10 output nodes, one for each class. The loss function you should use is the square loss. You must use (mini-batch) SGD for this problem. We do expect you find a way to get reasonable performance (compared to what is achievable for the given architecture). Training neural nets can be a bit of an art, until you get used to it. Use regularization if you find it helpful.

All your plots must have the x-axis correspond to the "effective" epoch number, so 1 unit on the x-axis corresponds to when $50K$ training points have been "touched" (when you algorithm has

used $50K$ points). You should evaluate (and plot) your train, dev, and test set errors every "half epoch", i.e. after $25K$ training points have been "touched"; e.g. if your mini-batch size is 10, then you should be evaluating your train, dev, and test set every 2500 iterations. If your batch size is 200, then these evaluations should be every 125 iterations.

Remark: Note that a unit of 5 on your x-axis is comparable to a unit of 5 on one of your fellow students x-axis (even if you both used different mini-batch sizes) in the following sense: at this point on both of your curves, you both touched the same number points (and performed the same number of basic computations) though you may have used a different number of iterations (if you had differing mini-batch sizes). Note that "wall-clock time" to get to 1-epoch will be determined by your mini-batch size due to that "mini-batch computations" can be done more quickly (provided you utilize matrix multiplications appropriately). This does not necessarily mean you want to use larger mini-batch sizes.

Things you can try (please state which one you use):

- You are free to try out/use "momentum" in the optimizer. The usual setting of the momentum parameter is 0.9. Almost everyone finds it is a pretty handy to keep momentum on (and just use 0.9 as the parameter).
- You are free to try using the "softmax" at the top layer. In PyTorch, this is a one line change going to the "cross-entropy" loss.

3.1 Try to learn a one hidden layer network.

1. (20 points) Use a one hidden layer network with $d^{(1)} = 100$ hidden nodes. The transfer function should be a sigmoid. *Also, for each of the* 10 *output nodes, let us also use a sigmoid activation (so each output is bounded between* 0 *and* 1*).* This means that the 10-dimensional output is $\widehat{y}(x) = h(w^{(2)}z^{(1)})$, where $h(\cdot)$ is the sigmoid function (applied componentwise as $w^{(2)}$ is a $10 \times d^{(1)}$ sized matrix).

   (a) (2 points) Now run stochastic (mini-batch) gradient descent. Specify all your parameter choices: your mini-batch size, your regularization parameter, and your learning rates (if you alter the learning rates, make sure you precisely state when it is decreased). Also, specify how you initialize your parameters.

   (b) (3 points) For what learning rate do you start observing a non-trivial decrease in your error. If you learning rate is larger than this, do you diverge and do you get "NaN's"? Why or why not?

   (c) (5 points) Make a plot showing your training average squared error, your development average squared error, and your average test squared error on the y-axis and the iteration on the x-axis. All three curves should be on the same plot.

   (d) (5 points) Make this plot again (with all three curves), except use the misclassification error, as a percentage, instead of average squared error. Here, make sure to start your x-axis at a slightly later iteration, so that your error starts below 20%, which makes the behavior more easy to view (it is difficult to view the long run behavior if the y-axis is over too large a range). Again, it is expected that you obtain a good test error (meaning you train long enough and you regularize appropriately, if needed). Report the lowest test error.

(e) (5 points) Note that the (input) weights to any hidden node correspond to a weighting over the image. This means you can try to visualize the (input) weights corresponding to any hidden node! Provide 8 plots corresponding to 8 different nodes (see Canvas for a function which shifts an image to be between 0 and 1 which could helpful in plotting). Provide a brief interpretation.

2. (20 points) Now replace all the transfer functions in the previous network with the "ReLu" transfer function. Again, provide answers to the previous 5 questions for this network.

3.2 EXTRA CREDIT: Be free with your MLP! (10 points)

Now be creative and try to get good performance with an MLP. Merge your training set and the dev set (sigh...), as you may be interested in comparing your numbers to the MNIST table on http://yann.lecun.com/exdb/mnist/. You should try a network with more than one hidden layer.

Again, provide answers to the previous 5 questions for this network (and specify the network architecture in the answer to the first question). Here, note that there is no dev set. Also, it is difficult to visualize the parameters in any layer beyond the first hidden layer, so only make images out of the first hidden layer.

If you did Q5 from the previous HW (and got a low error), then I challenge you to beat it! (This is not easy to do. It is pretty hard for the instructor to do better than Q5 with an MLP.)

4 EXTRA CREDIT: Convolutional Neural Nets on MNIST (25 points)

Try out a convolutional neural network. This is a considerably more expensive procedure (though, with a little search, you might find your error drops very quickly with an architecture that is not too costly for you to compute.) You will find that your 'architecture' choices are governed by your computational resources. This is not an unreasonable practical lesson: many of the choices we make in practice are largely due to our computational and/or memory constraints (when we have GPUs, we often run on large enough problems so that we are at the limits of our computational resources).

Again, provide answers to the previous 5 questions. For this problem, in the first question, specify your architecture, including the size of your "filters", your "pooling region" and your "stride" (I would use average pooling). As before, merge your dev set in the training set. Also, note that your visualized weights are going to be the learned filters (so they will be a smaller size than the image), which will depend on the size of the filters that you use.

5 EXTRA CREDIT: Non-convex optimization and convergence rates to stationary points (25 points)

(To obtain credit on this problem, you must do the first part of this question.)

Let us say a function $F : \mathbb{R}^d \to \mathbb{R}$ is L *-smooth* if

$$\|\nabla F(w) - \nabla F(w')\| \leq L\|w - w'\|\,,$$

where the norm is the Euclidean norm. In other words, the derivatives of F do not change too quickly.

Gradient descent, with a constant learning rate, is the algorithm:

$$w^{(k+1)} = w^{(k)} - \eta \cdot \nabla F(w^{(k)})$$

In this question, we do *not* assume that F is convex. If you find it helpful, you can assume that F is twice differentiable.

1. (15 points) Now let us bound the function value decrease at every step. In particular, show that the following holds (for all η):

   $$F(w^{(k+1)}) \leq F(w^{(k)}) - \eta\|\nabla F(w^{(k)})\|^2 + \frac{1}{2}\eta^2 L\|\nabla F(w^{(k)})\|^2$$

   It is fine to prove a looser version of this bound, where the factor of $1/2$ is replaced by 1. Brownie points if you get the factor of $1/2$. (Hint: Taylor's theorem is the natural starting point. You may also want to consider what smoothness implies about the second derivative. If you think about the intermediate value theorem, you can actually get the factor of $1/2$).
2. (3 points) Let us now show that if the gradient is large, then it is possible to substantially decrease the function value. Precisely, show that with an appropriate setting of η, we have that:

   $$F(w^{(k+1)}) \leq F(w^{(k)}) - \frac{1}{2L}\|\nabla F(w^{(k)})\|^2$$

3. (7 points) Now let $F(w_*)$ be the minimal function value (i.e. the value at the global minima). Argue that gradient descent will find a $w^{(k)}$ that is "almost" a stationary point in a bounded (and polynomial) number of steps. Precisely, show that there exist some k where:

   $$k \leq \frac{2L(F(w^{(0)}) - F(w_*))}{\epsilon}$$

   such that

   $$\|\nabla F(w^{(k)})\|^2 \leq \epsilon\,.$$

   (Hint: you could consider a proof by contradiction. Also, note that $\|\nabla F(w^{(k)})\|$ may not be decreasing at every step.)

References

[1] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969.