Homework 4
CSE 446: Machine Learning

University of Washington

1 Policies *[0 points]*

Please read these policies. **Please answer the three questions below and include your answers marked in a "problem 0" in your solution set.** Homeworks which do not include these answers will not be graded.

PyTorch policies: As before, you must write your own code. You may use PyTorch in addition to Numpy (however, for consistency, *you may not use TensorFlow*). 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 these built in library functions: 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.

The purpose of this is for a better understanding of what is going on under the hood, and see that you are really just access to an auto-differentiation library. Furthermore, all the built in 'neural net' libraries are usually pretty trivial; it is often helpful to just code networks up yourself.

If in doubt, post to the message boards or email the instructors.

Gradescope submission: When submitting your HW, please tag your pages correctly as is requested in gradescope. Untagged homeworks will not be graded, until the tagging is fixed.

Readings: Read the required material.

Submission format: Submit your report as a *single* pdf file. Also, please include all your code in the PDF file in a section at the end of your document, marked "Code"; also specify which problem(s) the code corresponds to. The report (in a single pdf file) must include all the plots and explanations for programming questions (if required). Homework solutions must be organized in order, with all plots arranged in the correct location in your submitted solutions. We highly recommend typesetting your scientific writing using LaTeX(see the website for references for free tools). 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.

Including your Python source code: For the programming assignments, submit your code in the pdf file along with a neatly written README file that instructs us how you ran your code with different settings (if applicable). Please note that we will not accept screenshots of Jupyter notebooks. If you do use Jupyter, you must export your code to a text file and put the text of your code in the submitted pdf file (in the last section) in a manner that can be executed in that order (without any extraneous or missing code).

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 in doubt, post to the message boards.

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.

Extra Credit Policy: In order to get extra credit, you must do all the regular problems. Extra credit points will only be awarded if there are (honest attempts at) answers to *all* the regular questions. This is because they are not designed to be alternative questions to the regular questions.

1.1 List of Collaborators

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

1.2 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".

1.3 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.

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 assignment, 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 $w^{(1)}, w^{(2)}, \ldots w^{(L+1)}$?
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 Stationary points, Saddle points and Symmetries *[12 points]*

Here, the underlying function of interest is the square loss on the training set. Recall that a stationary point is a point with 0 gradient, and that a saddle point is not local minima (e.g. crudely speaking, a saddle point is an unstable stationary point).

Assume that we say the derivative of the $\text{Relu}(x)$ function is 0 when $x = 0$ (any number between 0 and 1 is reasonably assignment. This choice of 0 does not alter any of yor answers.)

1. ***[2 points]*** Suppose our MLP has 0 hidden layers (e.g. we are working with linear regression). Is setting all the weights to 0 a stationary 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 stationary point? If so, give a precise argument as to why. Else, give a counterexample.
3. ***[5 points]*** Now assume our MLP has one hidden layer and the output of the network is just one number. Let us consider the sigmoid transfer function. Suppose we now initialize our weights randomly, except that all the weights in our hidden layer are all identical to each other; specifically, suppose $w^{(1)}_{j,i} = w^{(1)}_{k,i}$ and $w^{(2)}_j = w^{(2)}_k$ for all nodes j and k in the hidden layer (recall that there is only one output node in this network). Suppose we run gradient descent from this initialization and reach a stationary point, will the point we reached be a saddle point or a local extrema? (give the best answer of the two). What can we say about how the weights at each node in the hidden layer relate to each other at this stationary point? Explain your reasoning.
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 when we flip the sign of all the weights?

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 mappings).

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. Show 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 PyTorch warmup *[10 points]*

Let us redo our softmax problem, but now using PyTorch to handle the derivatives for us. You are free to use PyTorch's built in cross entropy loss function; and you must read the document to make sure you understand how you are using it. Note that this function is poorly named in that you do not feed in a probability distribution.

Recall the softmax classifier. Here, y takes values in the set $\{1, \ldots k\}$. The model is as follows: we have k weight vectors, $w^{(1)}, w^{(2)}, \ldots w^{(k)}$. Here, we can view these parameters as columns in a matrix of size $W \in \mathbb{R}^{d \times 10}$ matrix. For $\ell \in \{1, \ldots k\}$,

$$p_W(y = \ell | x) = \frac{\exp(w^{(\ell)} \cdot x)}{\sum_{i=1}^{k} \exp(w^{(i)} \cdot x)}$$

Again, note that this is a valid probability distribution (the probabilities are positive and they sum to 1).

As before, the (negative) likelihood function on an N size training set is:

$$\mathcal{L}_\lambda(W) = \frac{-1}{N} \sum_{n=1}^{N} \log p_W(y = y_n \mid \mathbf{x}_n) + \frac{\lambda}{2} \|W\|^2 .$$

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

For classification, one can choose the class with the largest predicted probability.

1. Now use stochastic gradient descent (using one point at a time) with PyTorch:

   (a) ***[1 point]*** Specify your parameter choices (your stepsize, λ, and and your step size schedule, if you choose to decay it or what you did.)

   (b) ***[5 points]*** Show your log loss on the training set and the development set (where the y-axis is the log loss and x-axis is the iteration). Do this after every 500 updates (starting at your first update at 0).

   (c) ***[4 points]*** Make this plot again (with both curves), except use the misclassification error, as a percentage, instead of the average log loss. Here, make sure to start your x-axis at a slightly later iteration, so that your error starts below 15%, 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 the spacing on the x-axis is 500 updates.

   (d) ***[1 points]*** Again, it is expected that you obtain a good test error (meaning you train long enough and you regularize appropriately, if needed). Report your lowest test error.

4 MLPs: Let's try them out on MNIST *[32 points]*

Make sure to read the policies for PyTorch at the start of this assignment. These policies apply for the entire HW set.

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

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

You will have 10 output nodes, one for each class. 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 and dev set errors every "quarter epoch", i.e. after $50K/4 = 12500$ training points have been "touched"; e.g. if your mini-batch size is 5, then you should be evaluating your train and dev set every 2500 iterations. If your batch size is 100, 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).

4.1 A one hidden layer network with sigmoids and the softmax. *[16 points]*

Use a one hidden layer network with $d^{(1)} = 200$ hidden nodes. The transfer function should be a sigmoid. The output of the neural net will then be a linear transformation of the outputs of layer 1, i.e. $\widehat{y}(x) = w^{(2)}z^{(1)}$ $w^{(2)}$ is a $10 \times d^{(1)}$ sized matrix. To obtain our prediction and loss, we will use the multi-class logistic regression and use softmax loss function.

1. ***[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.
2. ***[3 points]*** For what learning rate do you start observing a non-trivial decrease in your cross entropy error.
3. ***[3 points]*** Make a plot showing your average cross entropy error for both your training and your development sets on the y-axis and have the iteration on the x-axis. Both curves should be on the same plot.
4. ***[3 points]*** Make this plot again (with both curves), except use the misclassification error, as a percentage. 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).
5. ***[4 points]*** Again, it is expected that you obtain a good test error (meaning you train long enough and you regularize appropriately, if needed). Report your lowest test error.
6. ***[1 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.

4.2 A one hidden layer network with ReLu's. *[16 points]*

Now let use the "ReLu" transfer function for *both* the hidden learner in the previous network and on the output layer of the network *Specifically, for each of the* 10 *output nodes, let us also use the ReLu activation (so each output is bounded below by* 0*).* This means that the 10-dimensional output is $\widehat{y}(x) = h(w^{(2)}z^{(1)})$, where $h(\cdot)$ is the ReLu function (applied componentwise as $w^{(2)}$ is a $10 \times d^{(1)}$ sized matrix).

Here you will use the average squared error on the output of this network (where you average by both the number of samples in your training/test/dev set and you average by the number of classes).

Make sure to use the same mini-batch size as before.

1. ***[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.
2. ***[3 points]*** For what learning rate do you start observing a non-trivial decrease in your squared error.
3. ***[3 points]*** Make a plot showing your average squared error for both your training and your development sets on the y-axis and have the iteration on the x-axis. Both curves should be on the same plot.
4. ***[3 points]*** Make this plot again (with both curves), except use the misclassification error, as a percentage. 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).
5. ***[4 points]*** Again, it is expected that you obtain a good test error (meaning you train long enough and you regularize appropriately, if needed). Report your lowest test error.
6. ***[1 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.

4.3 EXTRA CREDIT: Be free with your MLP! *[25 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 must try to get good results with two different networks: 1) you should experiment with trying a sufficiently wide one hidden layer neural network and 2) you should try a network that has at least two hidden layers.

You must try both architectures to receive any credit in this HW. Only trying one architecture will result in 0 *credit. The performance you obtain with each of these networks is important for full credit and we will give partial credit.*

1. *Specify both of your architectures and be sure to label which plots are for which architectures.* You should have plots for both of these architectures.
2. Now run stochastic (mini-batch) gradient descent. Specify all your parameter choices: your mini-batch size, your regularization parameter, *your loss function*, 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.
3. Make plots showing your average error (what error did you used?) on both your training and your test sets on the y-axis and the iteration on the x-axis. Both curves should be on the same plot. You should have two plots (for each of your two architectures).
4. Make these plot again (with both 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). You should have two plots (for each of your two architectures).
5. ***[4 points]*** Again, it is expected that you obtain a good test error (meaning you train long enough and you regularize appropriately, if needed). Report your lowest test error.

If you did Q7 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 Q7 with an MLP.)

5 Auto-differentiation in our example *[20 points]*

Consider the function from class (and the notes):

$$f(w_1, w_2) = (\sin(2\pi w_1/w_2) + 3w_1/w_2 - \exp(2w_2)) * (3w_1/w_2 - \exp(2w_2))$$

Suppose our program for this function uses the following evaluation trace:

input: $z_0 = (w_1, w_2)$
1. $z_1 = w_1/w_2$
2. $z_2 = \sin(2\pi z_1)$
3. $z_3 = \exp(2w_2)$
4. $z_4 = 3z_1 - z_3$
5. $z_5 = z_2 + z_4$
6. $z_6 = z_4 z_5$

return: z_6

5.1 The reverse mode of AD

Let us consider the reverse mode to compute the gradient $\frac{df}{dw}$, which is a two dimensional vector.

1. ***[8 points]*** Explicitly write out the reverse mode in our example. Write the pseudocode computing all the intermediate derivatives as you would actually compute them in an implementation. You may assume you have evaluated the "trace" and have already stored all the $z_t's$ in memory already. Your pseudocode for computing $\frac{dz_6}{dz_t}$ should *only* be written as a function of $z_1, \ldots z_6$ and the derivatives $\frac{dz_6}{dz_{t+1}}, \ldots \frac{dz_6}{dz_6}$ (as is specified by the reverse mode algorithm). In particular, it must be written in the manner that the reverse mode is implemented (so as to be an efficient algorithm). You should basically have the same number of lines of pseudocode as the original function.

5.2 The forward mode of AD

The "forward mode" of AD has nothing to do with the "forward pass" in backprop.

Let us start with the forward mode for auto-differentiation. This is actually a conceptually much simpler way to compute the derivative than the reverse mode. We use the forward mode to compute the derivative with respect to the first coordinate w_1, i.e. $\frac{df}{dw_1}$, The forward mode computes the derivative $\frac{df}{dw_1}$ in sequential manner, directly using the previous variables and the chain rule. The idea is as follows: at time line t we:

1. compute z_t as normal.
2. compute $\frac{dz_t}{dw_1}$ using our previously computed derivatives $\frac{dz_1}{dw_1}, \ldots \frac{dz_{t-1}}{dw_1}$ using the chain rule:

$$\frac{dz_t}{dw_1} = \sum_{\tau<t} \frac{dz_t}{dz_\tau} \frac{dz_\tau}{dw_1}$$

So in your pseudocode you will write $\frac{dz_t}{dw_1}$ as only a function of the previous derivatives (as this is how you would compute it).

1. ***[8 points]*** Explicitly write out the forward mode in our example, where you will . Write out the pseudocode computing all the intermediate derivatives as you would actually compute them in an implementation. You will be writing out a series of steps (the pseudocode) where you will be computing *both* z_t and its derivative $\frac{dz_t}{dw}$. Your pseudocode for computing $\frac{dz_t}{dw}$ should *only* be written as a function of $z_1, \ldots z_{t-1}$ and the previous derivatives $\frac{dz_1}{dw_1}, \ldots \frac{dz_{t-1}}{dw_1}$ (as is specified by the chain rule above). You should have 12 steps.

You should be able to see that computing $\frac{df}{dw_1}$ is of the same computational cost as is computing the function $f(w)$ itself. Let us explore this a little more in the next question.

5.3 Computation in the forward mode vs the reverse mode *[4 points]*

Suppose we seek to compute the derivative with respect to a real valued function $f(w) : \mathbb{R}^d \to \mathbb{R}$. Recall that in class we stated that we can obtain the (d-dimensional) derivative $\frac{df}{dw}$ within a computational cost that is no more than factor of 5 of computing just the scalar function $f(x)$ itself. This is achievable using the reverse mode. Let T be the computation time to compute $f(w)$ using our program. So the reverse mode computes $\frac{df}{dw}$ in time that is no more that $5T$.

1. ***[2 points]*** Suppose we want to find the derivative $\frac{df(w)}{dw}$, which is a d-dimensional vector. How would we do this with the forward mode?
2. ***[2 points]*** In order notation, what is the computational complexity of computing $\frac{df}{dw}$ using the forward mode, as function of T and, possibly, other relevant factors like the dimensionality d.

Feel free to think about the differences between the forward mode and the reverse mode!

Remark: PyTorch and TensorFlow are literally implementing the reverse mode of AD, just like you did in the above example.

6 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.

7 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.)

8 Code

Please include all your code in the PDF file in this section. Specify which problem(s) the code corresponds to. Re Jupyter: refer to the policies section of the HW.

References

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