CSE 547 - Assignment 1

Philip Pham

April 18, 2018

Problem 0

List of collaborators: I have not collaborated with anyone.

List of acknowledgements: None.

Certify that you have read the instructions: Yes.

Terms and Conditions to use the dataset: I accept the terms and conditions to use the COCO dataset.

Problem 1

Read the course website, up until "Lecture Notes and Readings", so that you understand the course policies on grading, late policies, projects, requirements to pass etc. Write "I have read and understood these policies" to certify this. If you have questions, please contact the instructors.

Solution

I have read and understood these policies.

Problem 2

Consider the function from class (and the notes):

$$f(w_1, w_2) = \left[\sin\left(2\pi\frac{w_1}{w_2}\right) + 3\frac{w_1}{w_2} - \exp(2w_2)\right]\left[3\frac{w_1}{w_2} - \exp(2w_2)\right]. \tag{1}$$

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

The Forward Mode of AutoDiff (AD)

The forward mode for auto-differentiation is a conceptually simpler way to compute the derivative. Let us examine the forward mode to compute the derivative of one variable, $\frac{df}{dw_1}$. In the forward mode, we sequentially compute both z_t and its derviative $\frac{dz_t}{dw_1}$ using the previous variables $z_1, \ldots, z_{t-1}$ and the previous derivatives $\frac{dz_1}{dw_1}, \ldots, \frac{dz_{t-1}}{dw_1}$.

Explicitly write out the forward mode in our example.

Solution

In the forward mode, we sequentially compute the values z_t and derivates of z_t with respect to w_1 in order for $t = 1, 2, \ldots, 6$.

Suppose we want to calculate $\frac{df}{dw_1}(u, v)$. Fix $w_1 = u$ and $w_2 = v$.

1. Compute $z_1 = u/v$.
2. Compute $\frac{dz_1}{w_1} = 1/v$.
3. Compute $z_2 = \sin(2\pi z_1)$.
4. Compute $\dfrac{dz_2}{dw_1} = \dfrac{\partial z_2}{\partial z_1}\dfrac{dz_1}{w_1} = 2\pi\cos(2\pi z_1)\dfrac{dz_1}{w_1}$.
5. Compute $z_3 = \exp(2v)$.
6. Compute $\dfrac{dz_3}{dw_1} = 0$.
7. Compute $z_4 = 3z_1 - z_3$.
8. Compute $\dfrac{dz_4}{dw_1} = \dfrac{\partial z_4}{\partial z_1}\dfrac{dz_1}{dw_1} + \dfrac{\partial z_4}{\partial z_3}\dfrac{dz_3}{dw_1} = 3\dfrac{dz_1}{dw_1} - \dfrac{dz_3}{dw_1}$.
9. Compute $z_5 = z_2 + z_4$.
10. Compute $\dfrac{dz_5}{dw_1} = \dfrac{\partial z_5}{\partial z_2}\dfrac{dz_2}{dw_1} + \dfrac{\partial z_5}{\partial z_4}\dfrac{dz_4}{dw_1} = \dfrac{dz_2}{dw_1} + \dfrac{dz_4}{dw_1}$.
11. Compute $z_6 = z_4 z_5$.
12. Compute $\dfrac{dz_6}{dw_1} = \dfrac{\partial z_6}{\partial z_4}\dfrac{dz_4}{dw_1} + \dfrac{\partial z_6}{\partial z_5}\dfrac{dz_5}{dw_1} = z_5\dfrac{dz_4}{dw_1} + z_4\dfrac{dz_5}{dw_1}$.

Each step can be computed by substituting values from the previous steps. The output is $\boxed{\dfrac{df}{dw_1}(u, v) = \dfrac{dz_6}{dw_1}.}$

The reverse mode of AD

Now let use consider the reverse mode to compute the derivative $\frac{df}{dw}$, which is a two-dimensional vector.

Explicitly write out the reverse mode in our example with the assumption that you have evaluated the trace and already stored all the z_ts in memory already.

Solution

In the reverse mode, we compute $\frac{df}{dz_t} = \frac{dz_6}{dz_t}$ in order $t = 6, 5, \ldots, 0$. The general algorithm is

$$\frac{dz_6}{dz_t} = \sum_{c \text{ is a child of } t} \frac{dz_6}{dz_c}\frac{\partial z_c}{\partial z_t}. \tag{2}$$

1. Seed $\dfrac{dz_6}{dz_6} = 1$.
2. Compute $\dfrac{dz_6}{dz_5} = \dfrac{dz_6}{dz_6}\dfrac{\partial z_6}{\partial z_5} = z_4$.
3. Compute $\dfrac{dz_6}{dz_4} = \dfrac{dz_6}{dz_5}\dfrac{\partial z_5}{\partial z_4} + \dfrac{dz_6}{dz_6}\dfrac{\partial z_6}{\partial z_4} = \dfrac{dz_6}{dz_5} + \dfrac{dz_6}{dz_6}z_5 = z_4 + z_5$.
4. Compute $\dfrac{dz_6}{dz_3} = \dfrac{dz_6}{dz_4}\dfrac{\partial z_4}{\partial z_3} = -\dfrac{dz_6}{dz_4} = -z_4 - z_5$.
5. Compute $\dfrac{dz_6}{dz_2} = \dfrac{dz_6}{dz_5}\dfrac{\partial z_5}{\partial z_2} = \dfrac{dz_6}{dz_5} = z_4$.
6. Compute $\dfrac{dz_6}{dz_1} = \dfrac{dz_6}{dz_2}\dfrac{\partial z_2}{\partial z_1} + \dfrac{dz_6}{dz_4}\dfrac{\partial z_4}{\partial z_1} = 2\pi\cos(2\pi z_1)\dfrac{dz_6}{dz_2} + 3\dfrac{dz_6}{dz_4}$.
7. Compute $\dfrac{dz_6}{dw_1} = \dfrac{dz_6}{dz_1}\dfrac{\partial z_2}{\partial w_1} = \dfrac{1}{w_2}\dfrac{dz_6}{dz_1}$.
8. Compute $\dfrac{dz_6}{dw_2} = \dfrac{dz_6}{dz_1}\dfrac{\partial z_1}{\partial w_2} + \dfrac{dz_6}{dz_3}\dfrac{\partial z_3}{\partial w_2} = -\dfrac{w_1}{w_2^2}\dfrac{dz_6}{dz_1} + 2\exp(2w_2)\dfrac{dz_6}{dz_3}$.

Each step can be computed by substituting the output of one of the previous steps. From the last two steps, we obtain our desired result

$$\boxed{\frac{df}{dw} = \begin{pmatrix} \dfrac{dz_6}{dw_1} \\ \dfrac{dz_6}{dw_2} \end{pmatrix}.} \tag{3}$$

Problem 3: Computation and Memory in AD

Suppose we seek to compute the derivative with respect to a real valued function $f(w) : \mathbb{R}^d \to \mathbb{R}$. Let us examine some of the computational and memory issues involved in AD.

Computation

Let T be the computation time to compute $f(w)$ using our program.

1. Suppose we want to find the derivative of one variable $\frac{df(w)}{dw_1}$ with respect to the variable w_1. In order notation, how does the computational complexity (the runtime) of the forward mode compare to the reverse mode $\frac{df(w)}{dw_1}$?

Solution

For the forward mode, we just do one sweep through our computation graph to compute $\frac{df(w)}{dw_1}$, so the running time is $O(T)$.

In the reverse mode, we also just do one sweep through the graph, so the running time is also $O(T)$.

Both running times are of the same complexity in this case.

2. 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 and, in order notation, what is the computational complexity? Again, in order notation, how does the computational complexity (the runtime) of the forward mode compare to the reverse mode to compute $\frac{df(w)}{dw}$.

Solution

For the forward mode, we have to do a sweep for each $\frac{df(w)}{dw_1}, \ldots, \frac{df(w)}{dw_d}$ that we want to compute, so the runtime complexity is $O(Td)$.

In the reverse mode, we have to do one sweep. Then, we have enough information to compute each $\frac{df(w)}{dw_j}$, so the runtime complexity is $O(T + d)$.

The computational complexity of the reverse mode is smaller than that of the forward mode.

3. If we could easily parallelize our computation (not worrying about communication), do you see a way to speed up the reverse mode? If so, what would be the new serial runtime? If not, why?

Solution

Only one sweep needs to be done, and sweeping though the graph must be done in a certain order, so there is no easy way to parallelize that part. Once the sweep is done, one could use the computed intermediate derivatives to to calculate each $\frac{df(w)}{dw_j}$ in a parallel manner. In that case the new runtime complexity is $O(T)$.

4. If we could easily parallelize our computation (not worrying about communication), do you see a way to speed up the forward mode? If so, how so and what would be the new serial runtime? If not, why?

Solution

Yes. The obvious way is to parallize the forward mode is to have d workers each doing a separate sweep and computing a separate $\frac{df(w)}{dw_j}$. If we do that, the runtime complexity is also $O(T)$.

Memory

Memory is often a bottleneck in practice (e.g. we load as much as we can on a GPU when doing our computation in batches of data). Let us understand some of these issues. Suppose our input w is d-dimensional and our evaluation trace is T steps. Assume one unit of memory is required to store a real number. Also, assume that we are free to delete variables at any time to free up memory (in practice, this often occurs by overwriting variables). In this question, when memory is used to refer to how much "scratch space" we need to utilize in order to run our program. In this question, order notation is sufficient. When stating your answers do not include the memory required to store the parameter w or the program itself. Also, when in your answers, do not include the memory required to write out the programs output (assume we have already allocated this memory). We are interested in how much excess memory the program needs to have free in order to store its intermediate variables and do all its computations.

1. Suppose that we were just interested in computing $f(w)$. Is it often the case that we can get away with less than T units of memory? How so?

   Solution

   Yes, we can get away with less than T units of memory by discarding unnecessary intermediate variables.

   For example suppose the computation graph was a perfectly balanced tree and $z_T = z_{T-1} + z_{T-2}$ where the computation traces of $z_{T-1} + z_{T-2}$ are each $(T-1)/2$. After computing z_{T-1}, we can free any memory used to compute it.

2. Suppose we only need to use m units of memory to compute $f(w)$. If we only wanted to compute $\frac{df}{dw_1}$, how much memory would we require to compute this using the forward mode? Explain.

   Solution

   We also only need $O(m)$ units of memory to compute $\frac{df}{dw_1}$. Suppose that we are trying to compute z_t. We only need to know the value of its parents. To compute $\frac{dz_t}{dw_1}$, we only need to know the value of its parents and the derivatives of the of the parents with respect to w_1.

3. Suppose we only need to use m units of memory to compute $f(w)$. If we only wanted to compute $\frac{df}{dw_1}$, how much memory would we require to compute this using the reverse mode? Explain.

   Solution

   Since we want to avoid blowing up the computation, we need $O(T)$ units of memory. One may think that since the computation of $\frac{dz_T}{dz_t}$ only requires the children of z_t, less units of memory are needed. But since we're working backwards, we may be deep in the graph, so those children have to be computed ahead of time. The first steps require knowing the values near the leaves for instance. If we discarded the intermediate

values, we would may to do expensive calculations to recover them when deep in the graph.

4. Suppose we only need to use m units of memory to compute $f(w)$ and suppose we want to compute $\frac{df}{dw}$. If we don't care about runtime, what is the most memory efficient algorithm you can come up with? How much memory is needed?

Solution

We could just repeat the forward pass to compute each $\frac{df}{dw_j}$. Each pass takes $O(m)$. After we're done the pass, we would discard any intermediate variables and just keep $\frac{df}{dw_j}$. If w is d-dimensional, we only need $O(m+d)$ units of memory.

Problem 4: PyTorch Can Give Us Some Crazy Answers

Now you will construct an example in which PyTorch provides derivatives that make no sense. The issue is in understanding when it is ok and when it is not ok to use dynamic computation graphs. You are going to find a way code up the same function in two different ways so that PyTorch will return different derivatives at the same point. The purpose of this exercise is to better understand how dynamic computation graphs work and to understand what you are doing when you using various AD softwares.

Two Identical Non-differentiable Functions with Different "Derivatives"

1. Define *and* plot a one dimensional, real valued function which is not continuous.

Solution

Consider the function

$$f(x) = \begin{cases} 2x, -1 & x < 0; \\ 0, & x = 0; \\ 2x+1, & x > 0. \end{cases} \tag{4}$$

It is implemented in Listings 1 and 2 and plotted in Figure 1.

2. Now write out your function in PyTorch. You should be able to define this function so that PyTorch returns a derivative of 0 at some point x_0.

Solution

Equation 4 is implemented as f in Listing 1. In Figure 1, you can see that PyTorch calculates the derivative as 0 at $x_0 = 0$ (left plot, blue `x`).

3. Now find another way to write out your function in PyTorch; do this so that it is exactly the same function. Do this in a way so that PyTorch now returns a derivative of 2 at exactly the same point x_0 that you obtained a derivative of 0 in the previous question.

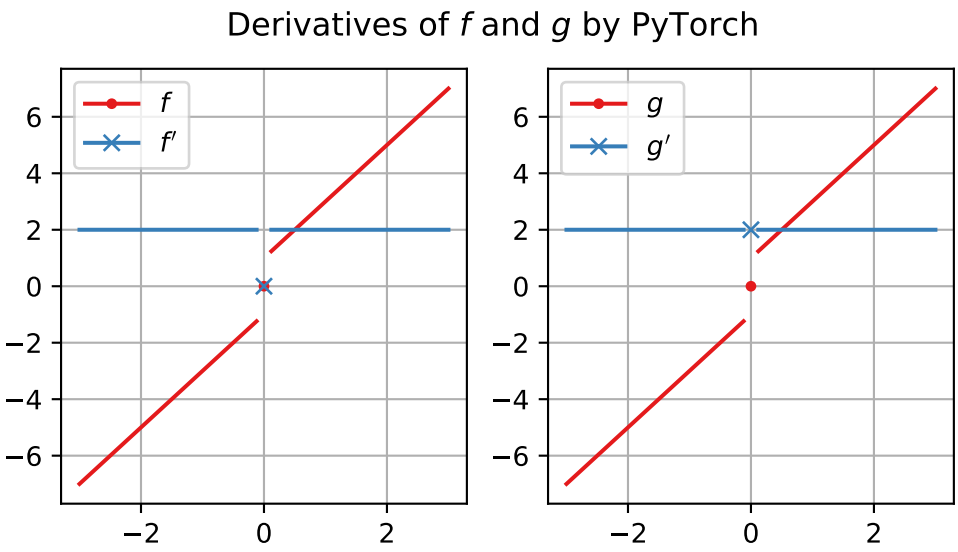

Figure 1: The functions from Listings 1 and 2 are plotted along with their PyTorch-computed derivatives.

```
def f(x: Variable) -> Variable:
    assert x.requires_grad
    return (2*x*torch.sign(x) + 1)*torch.sign(x)
```

Listing 1: Equation 4 defined with the sign function factored out.

Solution

Equation 4 is implemented as g in Listing 2. In Figure 1, you can see that PyTorch calculates the derivative as 2 at $x_0 = 0$ (right plot, blue `x`).

4. There is a no sane definition of the derivative for your function. Yet, you should not only found a way to get PyTorch to provide a derivative, you should have also found a way to give you two *different* derivatives at exactly the same point (on the same function). What went wrong?

Solution

PyTorch is just keeping track of the operations and does a pointwise calculation. It's not paying attention to any local behvaior like continuity. In Listing 2, by changing the `else` arm, we can have it return any arbitrary number for the derivative. For instance, having it return `-3*x` would have PyTorch calculating the derivative as -3 at $x_0 = 0$.

Extra Credit: Differentiable Functions with Different "Derivatives"

Provide a differentiable function, where you can code it up in PyTorch in two different ways and where you can get two different derivatives at the same point x_0.

```
def g(x: Variable) -> Variable:
    def g1d(x: Variable) -> Variable:
        if x.data[0] > 0:
            return 2*x + 1
        elif x.data[0] < 0:
            return 2*x - 1
        else:
            return 2*x

    if x.dim() == 0:
        return 1*x
    if x.size() == torch.Size([1]):
        return g1d(x)

    return torch.stack([g(sub_x) for sub_x in x])
```

Listing 2: Equation 4 defined element-wise by recursing into the tensor.

Solution

We can reuse the same idea. PyTorch doesn't correctly compute the derivative when squaring the sign function, even though, the sign function squared is just the identity.

Using this, I implement the simple, differentiable line $f(x) = 2x$ in two ways: (1) verbosely using the sign function as f and (2) the canonical way as g in Listing 3.

```
def f(x: Variable) -> Variable:
    assert x.requires_grad
    return 2*x*torch.sign(x)*torch.sign(x)

def g(x: Variable) -> Variable:
    assert x.requires_grad
    return 2*x
```

Listing 3: The function $f(x) = 2x$ defined in two different ways.

This results in Figure 2. PyTorch computes $f'(0) = 0$ despite the true value being 2.

The code for this exercise can be found at `https://gitlab.cs.washington.edu/pmp10/cse547/tree/master/hw1/problem4`.

Problem 5: Elementary properties of l_2 regularized logisitic regression

The binary case

Consider minimizing

$$J(\mathbf{w}) = -l\left(\mathbf{w}, \mathcal{D}_{\text{train}}\right) + \lambda \left\|\mathbf{w}\right\|_2^2, \tag{5}$$

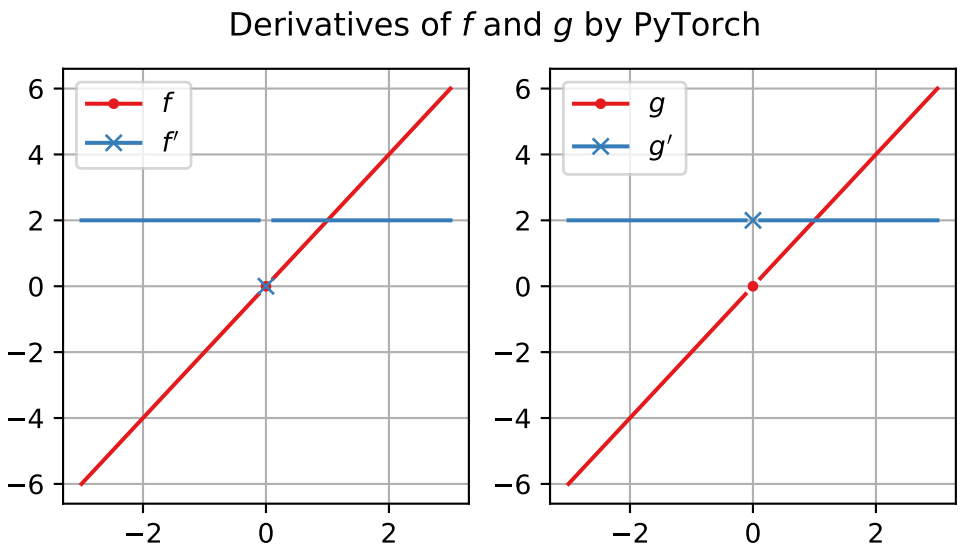

Figure 2: f and g from Listing 3 plotted along with their PyTorch-computed derivatives.

where

$$l(\mathbf{w}, \mathcal{D}) = \sum_j \log \mathbf{P}\left(y^j \mid \mathbf{x}^j, \mathbf{w}\right) \tag{6}$$

is the log-likelihood on the data set $\mathcal{D}$ for $y^j \in \{\pm 1\}$

State if the following are true or false. Briefly explain your reasoning.

1. With $\lambda > 0$ and the features x_k^j linearly separable, $J(\mathbf{w})$ has multiple locally optimal solution.

   Solution

   False. When the features are linearly separable, we can push loss unregularized loss arbitrarily close to 0 but the the loss is still convex. The sum of two convex functions is convex, so there will be global optimum.

2. Let $\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} J(\mathbf{w})$ be a global optimum. $\hat{\mathbf{w}}$ is typically sparse.

   Solution

   False. This may be true with l_1 regularization, but is not usually the case with l_2 regularization. If one considers the dual Lagrangian problem, $\hat{\mathbf{w}}$ will lie on some hypersphere at a point which will not generally have 0 values.

3. If the training data is linearly separable, then some weights w_j might become infinite if $\lambda = 0$.

Solution

True. By making the weights larger and larger we can push the loss to be arbitrarily close to 0 if $\lambda = 0$.

4. $l(\hat{\mathbf{w}}, \mathcal{D}_{\text{train}})$ always increases as we increase λ.

Solution

True. If one thinks in term of the Lagrangian dual problem increasing λ is contraining the weights further away from the global optimum by restricting them to a smaller hypersphere.

5. $l(\hat{\mathbf{w}}, \mathcal{D}_{\text{test}})$ always increases as we increase λ.

Solution

False. While the training loss may increase, we could be overfitting. Thus, sometimes increasing λ may decrease test loss.

Multi-class Logistic Regression

In multi-class logistic regression, suppose $Y \in \{y_1, \ldots, y_R\}$. A simplified version (with no bias term) is as follows. When $k < R$ the posterior probability is given by:

$$P(Y = y_k \mid X) = \frac{\exp(\langle w_k, X \rangle)}{1 + \sum_{j=1}^{R-1} \exp(\langle w_j, X \rangle)}. \tag{7}$$

For $k = R$, the posterior is

$$P(Y = y_R \mid X) = \frac{1}{1 + \sum_{j=1}^{R-1} \exp(\langle w_j, X \rangle)}. \tag{8}$$

To simplify notation, we can define $w_R = \mathbf{0}$ as a vector of all 0s. This gives us Equation 7 for all k.

1. How many parameters do we need to estimate? What are these parameters?

Solution

Assume the data is D-dimensional. Our parameters are the weights w_k each which is a D-dimensional vector. There are $\boxed{(R-1)D}$ parameters to estimate.

2. Given N training samples $\left\{(x^1, y^1), (x^2, y^2), \ldots, \left(x^N, y^N\right)\right\}$, write down explicitly the log-likelihood function and simplify it as much as you can:

$$L(w_1, \ldots, w_{R-1}) = \sum_{j=1}^{N} \log\left(P\left(y^j \mid x^j, w\right)\right). \tag{9}$$

Solution

Let $y^j = y_{l^j}$, that is, let l^j be the class label of observation j. If we use the posterior probability in Equation 7, then, we have that

$$\log\left(P\left(y^j \mid x^j, w\right)\right) = \left\langle w_{l^j}, x^j \right\rangle - \log\left(1 + \sum_{k=1}^{R-1} \exp\left(\left\langle w_k, x^j \right\rangle\right)\right). \tag{10}$$

Substituting Equation 10 into Equation 9, we have

$$L\left(w_1, \ldots, w_{R-1}\right) = \sum_{j=1}^{N}\left(\left\langle w_{l^j}, x^j \right\rangle - \log\left(1 + \sum_{k=1}^{R-1} \exp\left(\left\langle w_k, x^j \right\rangle\right)\right)\right). \tag{11}$$

3. Compute the gradient of L with respect to each w_k and simplify it.

Solution

w_k is a D-dimensional vector so, $\frac{\partial L}{\partial w_k}$ will also be D-dimensional. Denote the features for the observations with class label k by $\mathcal{X}_k = \{x^j : l^j = k\}$.

We have that for $k = 1, 2, \ldots, R-1$:

$$\begin{aligned} \frac{\partial L}{\partial w_k} &= \sum_{x \in \mathcal{X}_k} x - \sum_{j=1}^{N} x^j \frac{\exp\left(\langle w_k, x^j \rangle\right)}{1 + \sum_{m=1}^{R-1} \exp\left(\langle w_m, x^j \rangle\right)} \\ &= \sum_{x \in \mathcal{X}_k} x - \sum_{j=1}^{N} x^j P\left(Y = y_k \mid X = x^j\right). \end{aligned} \tag{12}$$

4. Now add the regularization term λ and define a new objective function:

$$L\left(w_1, \ldots, w_{R-1}\right) = \sum_{j=1}^{N} \log\left(P\left(y^j \mid x^j, w\right)\right) + \frac{\lambda}{2} \sum_{l=1}^{R-1} \|w_l\|_2^2. \tag{13}$$

Compute the gradient of this new L with respect to each w_k.

Solution

We can just differentiate term by term. The gradeint of the first term comes from Equation 12. We can write $\|w_l\|_2^2 = w_{l1}^2 + w_{l2}^2 + \cdots + w_{lD}^2$, so we have that

$$\frac{\partial L}{\partial w_l} = \sum_{x \in \mathcal{X}_l} x - \sum_{j=1}^{N} x^j \frac{\exp\left(\langle w_l, x^j \rangle\right)}{1 + \sum_{m=1}^{R-1} \exp\left(\langle w_m, x^j \rangle\right)} + \lambda w_l. \tag{14}$$

Problem 6: Getting Familiar with Our Dataset

Let us consider solving a binary classification problem (labels being $\{-1, 1\}$) with the dataset provided. We will use the square loss and consider training two models, namely (i) a linear model and (ii) a multi-layer perceptron. Note that this the same dataset that we will start branching out on, for the purposes of later assignments and the (default) course project. The dataset contains two supercategories: vehicle and animal, and a number of categories for each supercategory. In the small dataset provided, each image contains objects of a single supercategory, say vehicle, and potentially multiple objects from the supercategory, such as car, boat, etc. In this exercise we shall build a classifier that learns to classify between these supercategories, by optimizing a square loss objective with the above models. For the purposes of learning these classifiers, we shall use features from a convolutional neural network (as opposed to the raw pixels from these images). We have provided starter code to read these features.

SGD and Linear Regression

Here, the objective function we choose to optimize is:

$$L(w) = \frac{\lambda}{2} \|w\|_2^2 + \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(y_i - \langle w, x_i \rangle\right)^2, \tag{15}$$

where, $y_i \in \{-1, 1\}$ is the label, $x_i \in \mathbb{R}^d$ are the features, $w \in \mathbb{R}^d$ is the linear model that we wish to optimize for and $\lambda > 0$ is the strength of l_2 regularization. Now consider running stochastic gradient descent on $L(w)$, where the stochastic gradient is computed using a single sample (i.e. a batch size of 1).

1. Report the stepsize at which SGD starts to diverge (specified up to, say a factor of 2 from the actual value). Why would you expect the algorithm to diverge at too large a learning rate?

   Solution

   I found that my model began to diverge with a learning rate of 2×10^{-4}. The misclassification became 50% with is essentially a coin flip.

   I'd expect the algorithm to diverge since it's taking too large steps in the parameter space and missing the optima.

2. After every 500 updates (index the first update at 0), compute $L(w)$ evaluated over the training/development/test dataset and make a plot with these values in the y-axis and the iteration on the x-axis. All three curves should be on one same plot. What value of λ did you use?

   Solution

   This is plotted in Figure 3. I used $\lambda = 128$.

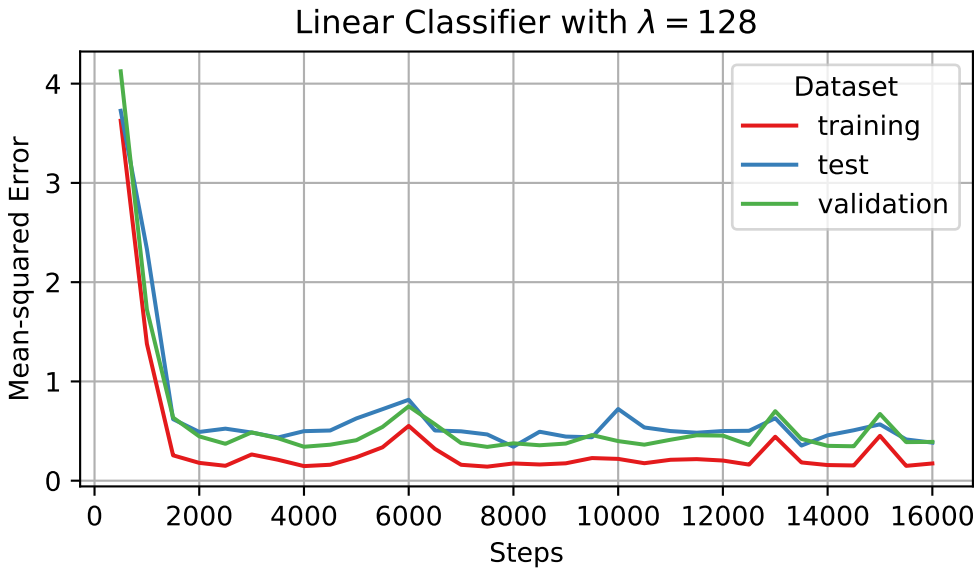

Figure 3: Loss for the linear classifier.

3. Compute the misclassification error every 500 updates and plot these quantities (in a single plot) evaluated over the training, development and test dataset. Here, make sure to start your x-axis at a slightly later iteration to make the behavior of the 0/1 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 test error.

   Solution

   This is plotted in Figure 4. The linear model performed the best, better than the multi-layer perceptron. From Table 1, the lowest misclassification rate was 7.2% for the test data and 6.4% for the validation data.

Implement a Multi-Layer Perceptron (MLP)

Here, the objective function we choose to optimize is:

$$L(w) = \frac{\lambda}{2}\|w\|_2^2 + \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(y_i - f_i(w)\right)^2, \text{ where } f_i(w) = \langle w_2, \text{relu}\left(w_1^\intercal x_i\right)\rangle . \tag{16}$$

$y_i \in \{-1, 1\}$ is the label, $x_i \in \mathbb{R}^d$ are the features, $w_1 \in \mathbb{R}^d \times \mathbb{R}^h$, $w_2 \in \mathbb{R}^h$, and $\text{relu}(x) = \max\{x, 0\}$, where the max is applied element wise, and h is the number of hidden nodes in our MLP. Now consider running stochastic gradient descent on $L(w)$, where the stochastic gradient is computed using a single sample. In this exercise, answer the questions below with the number of hidden nodes being (a) 10, (b) 100, (c) 500.

1. Report the stepsize at which SGD starts to diverge (specified up to, say a factor of 2 from the actual value).

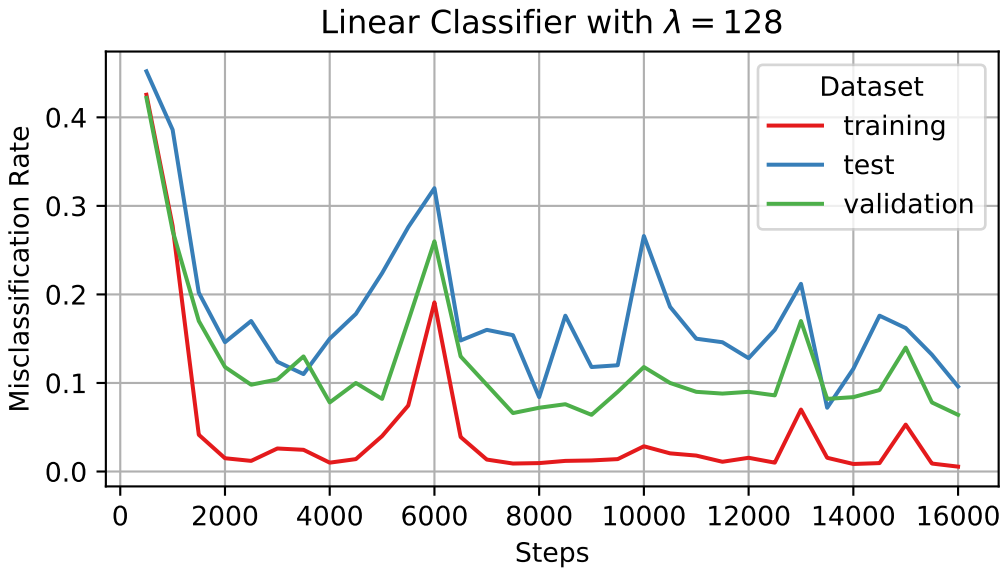

Figure 4: Misclassification rate for the linear classifier.

	Loss			Accuracy		
Model	Training	Test	Validation	Training	Test	Validation
Linear	0.141587	0.342109	0.340544	0.9945	0.928	0.936
MLP (10 units)	0.202833	0.421297	0.331022	0.9620	0.862	0.928
MLP (100 units)	0.182172	0.394072	0.289352	0.9725	0.876	0.926
MLP (500 units)	0.160434	0.417597	0.313770	0.9810	0.876	0.936

Table 1: The smallest loss and highest accuracy obtained by each model.

Solution

I started seeing divergence with a learning rate of 0.001. The algorithm didn't make any progress and got stuck with a 50% misclassification rate.

2. Compute $L(w)$ evaluated over the training, development, and, test dataset every 500 updates and plot these values in a single plot. Specify your learning rate scheme if you chose to decay your learning rate.

Solution

The plots are shown in Figures 5, 6, and 7. $\lambda = 0.0004$ was used as the l_2 penalty. There appears to overfitting as the test and validation loss are much higher than the training loss. A fixed learning rate was used.

3. Compute the misclassification error every 500 updates and plot these quantities (in a single plot) evaluated over the training, development and test dataset. As in the case of the linear model, make sure to start your x-axis at a slightly later iteration so as to make the behavior of the 0/1 error more easy to view. Report the lowest test error.

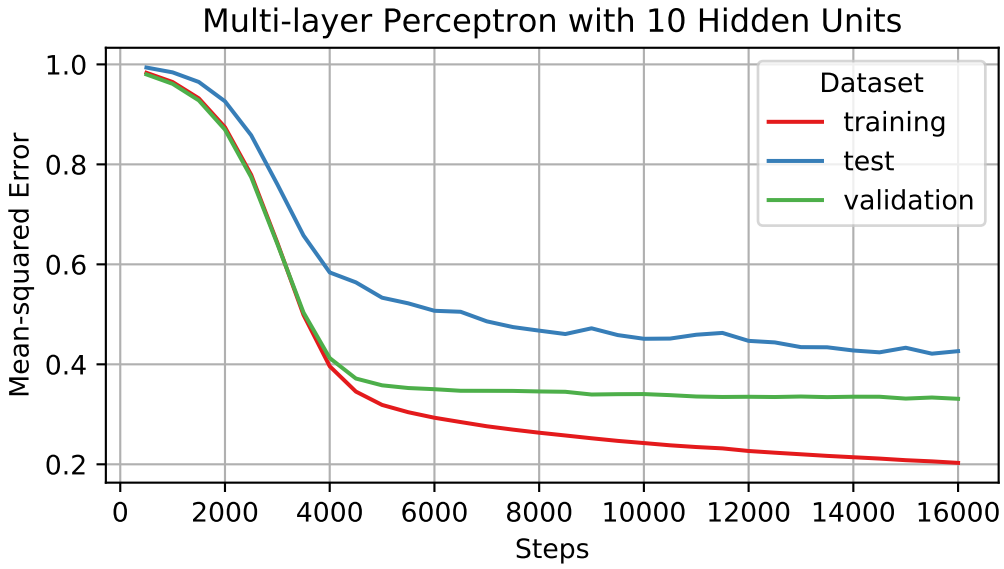

Figure 5: Loss for the multi-layer perceptron with 10 hidden units.

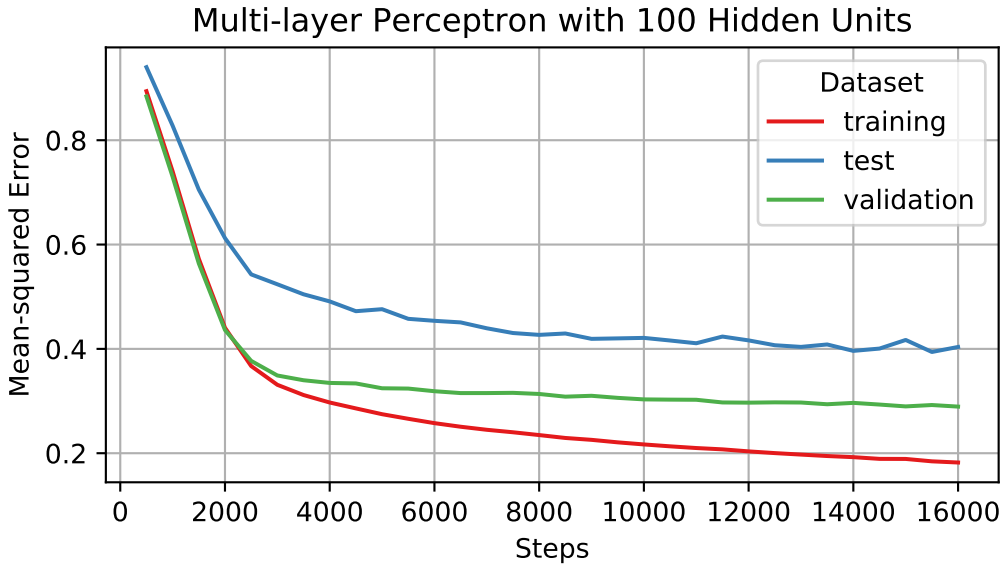

Figure 6: Loss for the multi-layer perceptron with 100 hidden units.

Figure 7: Loss for the multi-layer perceptron with 500 hidden units.

Solution

The plots are shown in Figures 8, 9, and 10. These plots mirror the those for loss and show evidence of overfitting. The lowest misclassification rate was was the 12.4% achieved by using 100 and 500 hidden units. On the validation dataset, the 500 hidden units model does the best misclassification rate of 6.4%.

Get Familiar with Amazon Web Services (AWS)

Use Amazon AWS to run the code developed in previous sections. Attach a screenshot showing the progress of the algorithm (and the result) as viewed in the remote machine. The purpose of this exercise is to get familiar with AWS and set yourselves up for future assignments and the course project.

Solution

I trained the models as AWS Batch jobs. You can see the result of a linear model in Figure 11 and the result of the multi-layer perceptron model in Figure 12.

Code to run this exercise can be found at `https://gitlab.cs.washington.edu/pmp10/cse547/blob/master/hw1/run.py`. The Jupyter Notebook for the plots is in `https://gitlab.cs.washington.edu/pmp10/cse547/blob/master/hw1/problem6`.

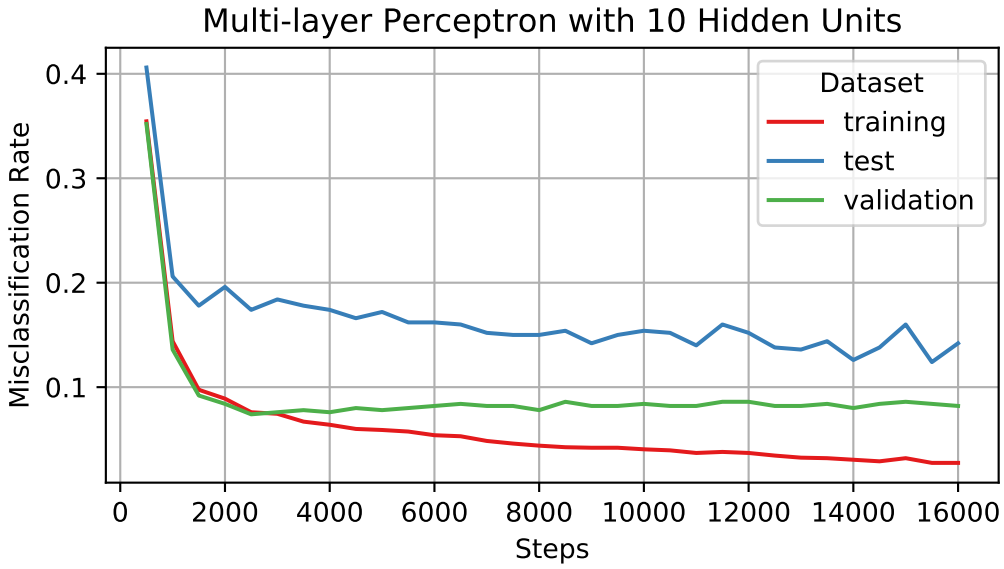

Figure 8: Misclassification rate for the multi-layer perceptron with 10 hidden units.

Figure 9: Misclassification rate for the multi-layer perceptron with 100 hidden units.

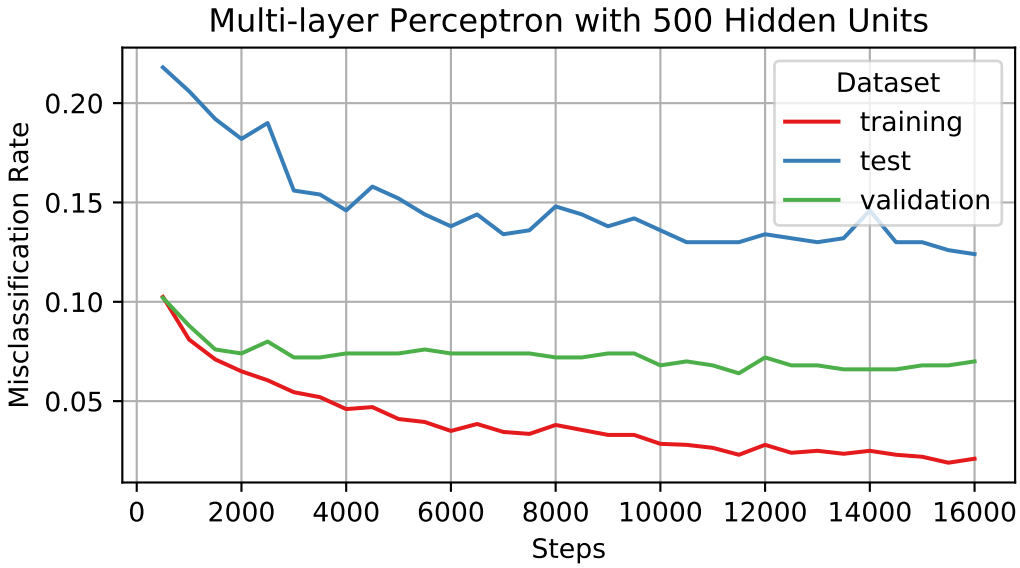

Figure 10: Misclassification rate for the multi-layer perceptron with 500 hidden units.

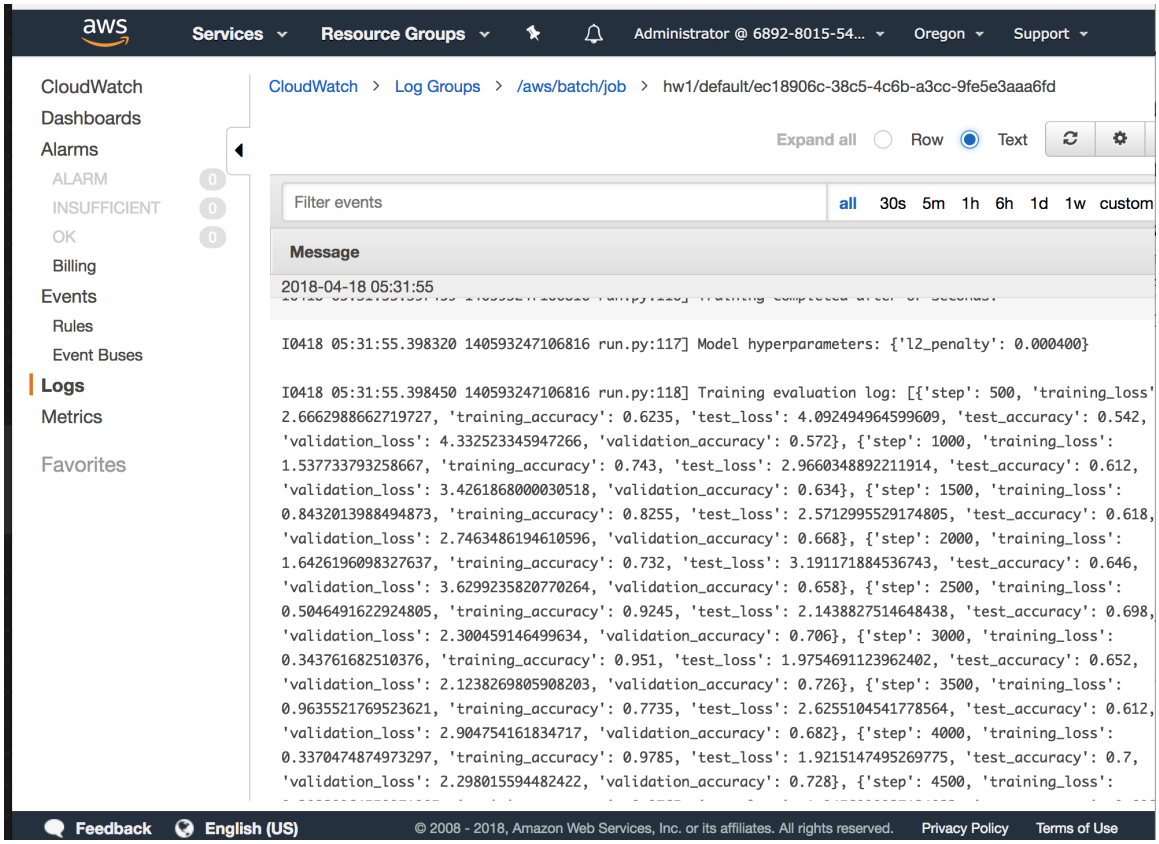

Figure 11: Training the linear classifier as an AWS batch job with $\lambda = 0.0004$.

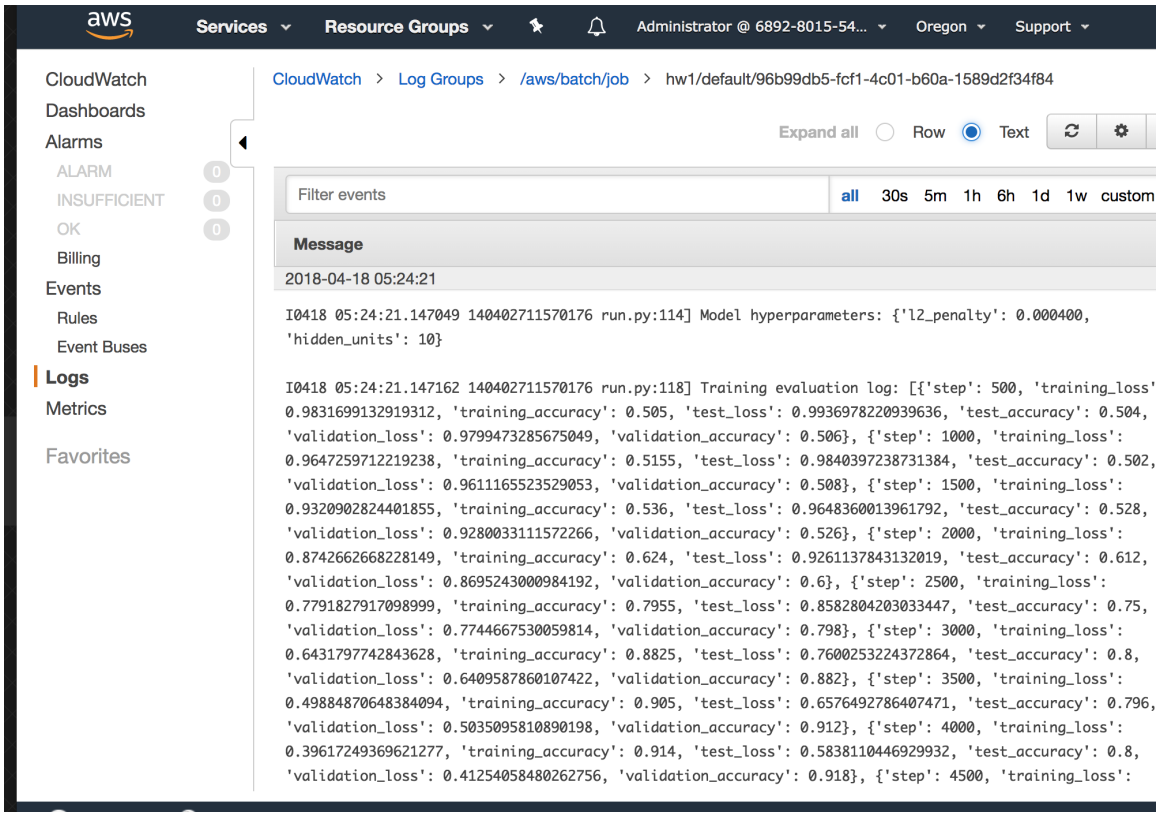

Figure 12: Training the multi-layer perceptron as an AWS batch job with 10 hidden units.