CS 189 Introduction to Machine Learning
Fall 2019 Jennifer Listgarten & Stella Yu

HW 06

Due: November 7th, 2019

0 Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to assignment on Gradescope, "HW6 Write-Up". If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results, "HW6 Code".
3. If there is a test set, submit your test set evaluation results, "HW6 Test Set".

1 Backpropagation Algorithm for Neural Networks

Learning goal: Understand backpropagation and associated hyperparameters of training neural networks

In this problem, we will be implementing the backpropagation algorithm to train a neural network to classify the difference between two handwritten digits (specifically the digits 3 and 9). The dataset for this problem consists of over 10k black and white images of size 28 x 28 pixels, each with a label corresponding to the digit 3 or 9.

Before we start we will install the library `mnist` so we can load our data properly.
Run `pip install mnist` in your terminal so the library is properly installed.

To establish notation for this problem, we define:

$$\mathbf{a}_{i+1} = \sigma(\mathbf{z}_i) = \sigma(\mathbf{W}_i\mathbf{a}_i + \mathbf{b}_i).$$

In this equation, $\mathbf{W}_i$ is a $n_{i+1} \times n_i$ matrix that maps the input $\mathbf{a}_i$ of dimension n_i to a vector of dimension n_{i+1}, where n_{i+1} is the size of layer $i+1$. The vector $\mathbf{b}_i$ is the bias vector added after the matrix multiplication, and σ is the nonlinear function applied element-wise to the result of the matrix multiplication and addition. $\mathbf{z}_i = \mathbf{W}_i\mathbf{a}_i + \mathbf{b}_i$ is a shorthand for the intermediate result within layer i before applying the nonlinear activation function σ. Each layer is computed sequentially where the output of one layer is used as the input to the next. To compute the derivatives with respect to the weights $\mathbf{W}_i$ and the biases $\mathbf{b}_i$ of each layer, we use the chain rule starting with

the output of the network and work our way backwards through the layers, which is where the backprop algorithm gets its name.

You are given starter code with incomplete function implementations. For this problem, you will fill in the missing code so that we can train a neural network to learn your nonlinear classifier. The code currently trains a network with one hidden layer with 4 nodes.

(a) **Start by drawing a small example network with** 1 **hidden layer, where the last layer has a single scalar output. The width of the input, hidden, and output layers should be** 784 **(number of pixels),** 16**, and** 1**, respectively. The nonlinearities for the hidden and output layers are relu and linear respectively. Label all the n_i as well as all the $\mathbf{a}_i$ and $\mathbf{W}_i$ and $\mathbf{b}_i$ weights. Consider the bias terms to be weights connected to a dummy unit whose output is always** 1 **for ease of labeling. Also draw and label the loss function — use a squared-error loss.**

Here, the important thing is for you to understand your own clear way to illustrate neural nets. You can follow conventions seen online, from class, or develop your own. The important thing is to have your illustration be unambiguous so you can use it to help understand the forward flow of information during evaluation and the backward flow during gradient computations.

(b) Let's start by implementing the least squares loss function of the network. We'll refer to the class that's encoded as 1 as the 'positive class' and the class that's encoded as -1 as the 'negative class.' This convention is arbitrary but convenient for discussions, especially since it matches the usual conventions in binary classification. For this question, let's treat the digit 3 as the positive class and the digit 9 as the negative class. The sign of the predicted value will be the predicted class label. This function is used to assign an error for each prediction made by the network during training.

The error we actually care about is the misclassification error (MCE) which will be:

$$\text{MCE}(\hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{I}\{\text{sign}(y_i) \neq \text{sign}(\hat{y}_i)\}$$

where y_i is the observation that we want the neural network to output and $\hat{y}_i$ is the prediction from the network. However this function is hard to to optimize so the implementation will be optimizing the mean squared error cost (MSE) function, which is given by

$$\text{MSE}(\hat{\mathbf{y}}) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

Write the derivative of the mean squared error cost function with respect to the predicted outputs $\hat{\mathbf{y}}$. In `backprop.py` implement the functions `QuadraticCost.fx` and `QuadraticCost.dx`. Include a screenshot of your function implementations.

(c) Now, let's take the derivatives of the nonlinear activation functions used in the network. **Write the derivatives and implement the following nonlinear functions in the code and their derivatives:**

$$\sigma_{\text{linear}}(z) = z$$

$$\sigma_{\text{ReLU}}(z) = \begin{cases} 0 & z < 0 \\ z & \text{otherwise} \end{cases}$$

$$\sigma_{\tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

For the tanh function, feel free to use the tanh function in `numpy`. We have provided the sigmoid function as an example activation function. **Include a screenshot of your function implementations.**

(d) We have implemented the forward propagation part of the network for you (see `Model.evaluate` in the code). We now need to compute the derivative of the cost function with respect to the weights $\mathbf{W}$ and the biases $\mathbf{b}$ of each layer in the network. We will be using all of the code we previously implemented to help us compute these gradients. **Assume that $\frac{\partial \text{MSE}}{\partial \mathbf{a}_{i+1}}$ is given, where $\mathbf{a}_{i+1}$ is the input to layer $i+1$. Write the expression for $\frac{\partial \text{MSE}}{\partial \mathbf{a}_i}$ in terms of $\frac{\partial \text{MSE}}{\partial \mathbf{a}_{i+1}}$. Then implement these derivative calculations in the function `Model.compute_grad`.** Recall, $\mathbf{a}_{i+1}$ is given by

$$\mathbf{a}_{i+1} = \sigma(\mathbf{z}_i) = \sigma(\mathbf{W}_i \mathbf{a}_i + \mathbf{b}_i) \, .$$

Include a screenshot of your function implementations.

(e) We use gradients to update the model parameters using batched stochastic gradient descent. **Implement the function `GDOptimizer.update` to update the parameters in each layer of the network.** You will need to use the derivatives $\frac{\partial \text{MSE}}{\partial \mathbf{z}_i}$ and the outputs of each layer $\mathbf{a}_i$ to compute the derivates $\frac{\partial \text{MSE}}{\partial \mathbf{W}_i}$ and $\frac{\partial \text{MSE}}{\partial \mathbf{b}_i}$. Use the learning rate η, given by `self.eta` in the function, to scale the gradients when using them to update the model parameters. **Train with batch sizes [10, 50, 100, 200] and number of training epochs [10, 20, 40]. Report the final error on the training set given the various batch sizes and training epochs** Does the result match your expectation, and why? What can you conclude from it. **Include a screenshot of your function implementations.**

(f) Let's now explore how the number of hidden nodes per layer affects the approximation. **Train a models using the tanh and the ReLU activation functions with** 2**,** 4**,** 8**,** 16**, and** 32 **hidden nodes per layer (width)**. Use the same training iterations and learning rate from the starter code. **Report the resulting error on the training set after training for each combination of parameters.** Does the result match your expectation, and why? What can you conclude from it.

2 Linear Neural Networks

Learning Goal: Understand the effect of depth and overparameterization on the loss surface in the simpler linear case.

In this problem, we will consider neural networks with the the identity function as the activation. These are known as *linear neural networks*. Formally, consider a set of input data $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, and outputs $\mathbf{y}_1, \ldots, \mathbf{y}_n \in \mathbb{R}^p$. For a depth $L \geq 1$, and weight dimensions $d_0, \ldots, d_L$, where $d_0 = d$ and $d_L = p$, an L-layer neural network with weights $\mathbf{W}_{1:L} := (\mathbf{W}_1, \ldots, \mathbf{W}_L)$, where $\mathbf{W}_i \in \mathbb{R}^{d_i \times d_{i-1}}$, is defined as the predictor function

$$\mathbf{f}_{\mathbf{W}_{1:L}}(\mathbf{x}) := (\mathbf{W}_L \cdot \mathbf{W}_{L-1} \cdot \ldots \mathbf{W}_1)\mathbf{x}.$$

Note then that $\mathbf{f}_{\mathbf{W}_{1:L}}(\mathbf{x}) = \overline{\mathbf{W}}\mathbf{x}$, where $\overline{\mathbf{W}} := \mathbf{W}_L \cdot \mathbf{W}_{L-1} \cdot \ldots \mathbf{W}_1 \in \mathbb{R}^{p \times d}$. Thus, $\mathbf{f}_{\mathbf{W}_{1:L}}(\mathbf{x})$ is a *linear predictor*.

In what follows, we will examine the optimization landscape to understand the relationship between critical points and global minima.

(a) We begin by understanding the *capacity* of the linear network. Fix a matrix $\mathbf{W}_0 \in \mathbb{R}^{p \times d}$. **Show that there exists weights** $(\mathbf{W}_1, \mathbf{W}_2 \ldots \mathbf{W}_L)$ of associated dimensions $d_1 \times d_0, \ldots, d_L \times d_{L-1}$ **such that** $\overline{\mathbf{W}} = \mathbf{W}_0$ **if and only if** $\text{rank}(\mathbf{W}_0) \leq \min_{i=0}^{L} d_i$**.**

Hint: For the "if" direction, start with the case where

$$\mathbf{W}_0 = \begin{bmatrix} I_{d_*} & \mathbf{0}_{\mathbf{d}_* \times (\mathbf{d} - \mathbf{d}_*)} \\ \mathbf{0}_{(p-d_*) \times d_*} & \mathbf{0}_{(p-d_*) \times (d_* - d)}, \end{bmatrix}$$

(b) Next, we consider the convexity of the training loss. For a function $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^p$, and data $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$, where $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}^p$, we introduce the training loss

$$\widehat{\mathcal{R}}(\mathbf{f}) := \frac{1}{2} \sum_{i=1}^{n} \|\mathbf{f}(\mathbf{x}_i) - \mathbf{y}_i\|_2^2.$$

Show that for $\mathbf{f}_{\mathbf{W}_{1:L}}$ **defined above**,

$$\widehat{\mathcal{R}}(f_{\mathbf{W}_{1:L}}) = Q(\overline{\mathbf{W}}) := \frac{1}{2} \|\overline{\mathbf{W}}\mathbf{X}^\top - \mathbf{Y}^\top\|_F^2$$

where we define the data matrices $\mathbf{X}$ and $\mathbf{Y}$ to have rows given by $\mathbf{x}_i$ and $\mathbf{y}_i$ respectively.

Then, argue that $Q(\overline{\mathbf{W}})$ **is a convex function of** $\overline{\mathbf{W}}$.

Hint: you may use the following fact for convexity. For a matrix $\mathbf{W} \in \mathbb{R}^{p \times d}$*, let* $\mathbf{W}[j] \in \mathbb{R}^d$ *denote the j-th row of* $\mathbf{W}$*. Then if* $F(\mathbf{W}) = \sum_{j=1}^{p} F_j(\mathbf{W}[j])$*, and each* $F_j(\cdot)$ *is convex, then* F *is convex.*

(c) Now, we will consider derivatives of the training loss. First, we will compute the derivative of the loss with respect to the *matrix product* $\overline{\mathbf{W}}$. To do so, we may want to make use of our knowledge of backpropagation. Consider the relation $\mathbf{F} = \mathbf{AB}$ and let $\mathbf{Z}$ denote the message backpropagated to the $\mathbf{F}$ node. The recall from lecture that $\mathbf{Z}_1 = \mathbf{Z}\mathbf{B}^\top$ and $\mathbf{Z}_2 = \mathbf{A}^\top\mathbf{Z}$ are the messages that will be passed back from the $\mathbf{A}$ and $\mathbf{B}$ nodes respectively.

Derive the expression for $\frac{\partial Q}{\partial \overline{\mathbf{W}}}$.

Hint: write $Q(\overline{\mathbf{W}})$ *as a computation graph:* $\mathbf{E} = \overline{\mathbf{W}}\mathbf{X}^\top - \mathbf{Y}^\top$*,* $Q = \frac{1}{2} \sum_{i,j} \mathbf{E}_{ij}^2$

(d) Now, we can reason about the global minimizers of the training loss. **Prove that if** $\min_{i=0}^{L} d_i = \min\{p, d\}$**, then** $\mathbf{W}_1, \ldots, \mathbf{W}_L$ **is a global minimizer of** $(\mathbf{W}_1, \ldots, \mathbf{W}_L) \mapsto \widehat{\mathcal{R}}(f_{\mathbf{W}_{1:L}})$ **if and only if** $\overline{\mathbf{W}}\mathbf{X}^\top\mathbf{X} - \mathbf{Y}^\top\mathbf{X} = 0$**.**

Hint: for this problem, you will need to show an equivalence (i.e. "if and only if") between the global minimizer of $Q(\cdot)$ and the global minimizers of the training loss $\widehat{\mathcal{R}}(f_{\mathbf{W}_{1:L}})$.

(e) Now we consider each layer's weight matrix independently. **Write down** $\frac{\partial \widehat{\mathcal{R}}(f_{\mathbf{W}_{1:L}})}{\partial \mathbf{W}_i}$. You may want to write $\widehat{\mathcal{R}}(f_{\mathbf{W}_{1:L}})$ as an extension of the computation graph in part (c). For this, you can define $\overline{\mathbf{W}}_{(i-1:1)} := \mathbf{W}_{i-1} \cdot \ldots \cdot \mathbf{W}_1$ and $\overline{\mathbf{W}}_{(L:i+1)} := \mathbf{W}_L \cdot \ldots \cdot \mathbf{W}_{i+1}$ (or identity if $i = 1, L$). Then $\mathbf{P} = \mathbf{W}_i\overline{\mathbf{W}}_{(i-1:1)}$ and $\overline{\mathbf{W}} = \overline{\mathbf{W}}_{(L:i+1)}\mathbf{P}$.

Use this answer to explain why a critical point of $\widehat{\mathcal{R}}$ is not necessarily a global minimum.

(f) Suppose that $p = d$ and $d = d_0 = d_1 = \cdots = d_L$, that is, all the the weight matrices are square. Moreover, suppose that $\mathbf{W}^*_{1:L} = (\mathbf{W}^*_1, \ldots, \mathbf{W}^*_L)$ is a critical point of the objective $\mathbf{W}_{1:L} \mapsto \widehat{\mathcal{R}}(f_{\mathbf{W}_{1:L}})$. **Give a sufficient condition on $\mathbf{W}^*_1, \ldots, \mathbf{W}^*_L$ which ensures that $\mathbf{W}^*_{1:L}$ is in fact a global minimum.**

3 Promises and Pitfalls of Neural Networks

Learning Goal: Understand the large capacity of neural networks to approximate functions as well as its inability to necessarily find a good approximation due to optimization

In this problem, we will consider the ability of neural networks to represent and learn functions. Consider the following setting. Features x are drawn uniformly at random from $\{0, 1\}^d$. The labels depend on the parameter v, which is also drawn uniformly at random from $\{0, 1\}^d$. Note that v is drawn independently from x. The labels are defined as $y = (-1)^{v^\top x}$.

In this setting, we can consider the parameter v as a mask, so $v^\top x$ counts the nonzero elements of x at the positions determined by the mask v. Then the output y classifies whether this quantity is even or odd.

In this problem we will show that neural networks of modest size can approximate such a function exactly. This is a positive result on the *capacity* of neural networks. However, we will subsequently explore the failure of a gradient-based training scheme to arrive at these correct parameters, showing a negative result on the ability to *learn*.

We will consider the two layer fully-connected neural network with k hidden units in the hidden layer, ReLU activations applied elementwise to the hidden layer, and a linear activation (with a bias term) on the single output node:

$$f_{W,b,\alpha,\beta}(x) = \sum_{i=1}^{k} \alpha_i \max\{0, W_i^\top x + b_i\} + \beta \,.$$

Here W_i refers to the i-th row of the weight matrix $W \in \mathbb{R}^{k \times d}$.

For brevity, we will define a vector of parameters as

$$p = \begin{bmatrix} W_1^\top & \dots & W_k^\top & b_1 & \dots & b_k & \alpha_1 & \dots & \alpha_k & \beta \end{bmatrix} \in \mathbb{R}^{dk+2k+1} .$$

In this problem, we will make use of PyTorch. From the previous problems you should already have PyTorch installed.

(a) We claim that if $k \geq \frac{3d}{2}$, then the network can exactly represent the function $y = (-1)^{v^\top x}$ for any fixed v. Specifically, we claim that the following parameter settings achieve this:

- $W_i = v$ for $1 \leq i \leq \frac{3d}{2}$, and 0 otherwise,
- **for** $0 \leq i \leq \frac{d}{2}$
  - $b_{3i+1} = -(2i - \frac{1}{2})$, $b_{3i+2} = -2i$, $b_{3i+3} = -(2i + \frac{1}{2})$,
  - $\alpha_{3i+1} = 4$, $\alpha_{3i+2} = -8$, $\alpha_{3i+3} = 4$,
- $\beta = -1$.

Your task is to verify this statement numerically. To do so, first implement the network structure using the starter code in `pitfalls.py` with $k = 10d$. Please read the starter code comments and examples carefully. Then, manually set the weights of the network as described above. Note that python indexing starts at 0 rather than 1!. Running the script will verify that the network does represent the function. Please include the relevant parts of your code in your write-up.

(b) Instead of specifying weights by hand, we prefer to train a deep classifier based on data $\{x_i, y_i\}_{i=1}^n$. To do so, we will run gradient descent on the hinge loss:

$$\ell(f_p(x), y) = \max\{0, 1 - f_p(x) \cdot y\} .$$

Implement the batch loss in `myLoss()`. Be sure that you use only functions supplied in the `torch` library so that the auto-differentiation functionality will work. Then use the provided code to train the network for 5×10^4 iterations with input dimensions $d = 5, 10, 30$. For each dimension plot the loss over iterations, and comment on what you see. All three curves should be plotted in one plot with a legend indicating the dimension. Please include your plot and the relevant parts of your code in your write-up.

(c) Despite the capacity result in part (a), it appears that, in some cases, our neural network is not learning. We will now show that this failure comes from a lack of informative gradients, i.e. gradients do not give useful information about the key parameter v. A network is trained with gradient descent updates on the *risk*, which for this problem is given by[1]

$$R(p) = \mathbb{E}_{x,y}\left[\ell(f_p(x), y)\right] = \mathbb{E}_x\left[\max\{0, 1 - f_p(x) \cdot (-1)^{v^\top x}\}\right] .$$

[1] In practice, networks are trained on the *empirical risk*, which is $\frac{1}{n}\sum_{i=1}^n \ell(f_p(x_i), y_i)$. In this problem, we analyze the full *population risk* to show that even in the best case, the gradients are uninformative.

To show that gradients are uninformative, we will show that there is little variation in their value for different values of v. Specifically, we will show that

$$\mathbb{E}_v\left[|\nabla R(p) - a|^2\right] \leq \frac{C_p}{2^d},$$

where a is a quantity independent of v and C_p is a constant depending on network parameters. This statement means the amount of information that the gradient contains about the underlying parameter v decreases exponentially in d.

You can use the following two facts in your proof. First, every entry of the gradient is bounded

$$\left|\frac{\partial}{\partial p_j} f_p(x)_j\right| \leq c_p \tag{1}$$

for all j and some constant c_p that only depends on the network architecture. Second,

$$\sum_v \left(\mathbb{E}_x\left[(-1)^{v^\top x} h(x)\right]\right)^2 \leq \mathbb{E}_x\left[h(x)^2\right] \tag{2}$$

for any function $h(x)\colon \{0,\ 1\}^d \to \mathbb{R}$ (i.e. any function mapping d-dimensional binary strings to real numbers).

(i) Write an expression for $\nabla R(p)$ that is amenable to analysis. Show that $\nabla R(p) = a + \mathbb{E}_x\left[(-1)^{v^\top x} g(x)\right]$ for some vector valued function $g(x)$ (that may also depend on p) and $a = -\mathbb{E}_x\left[\frac{\mathbf{1}_1(f_p(x)) - \mathbf{1}_1(-f_p(x))}{2} \nabla_p f_p(x)\right]$, where we define an indicator function

$$\mathbf{1}_1(u) = \begin{cases} 0 & u \geq 1 \\ 1 & u \leq 1 \end{cases}.$$

Hint: It may be useful to note that $y \cdot \mathbf{1}_1(f_p(x) \cdot y) = \begin{cases} \mathbf{1}_1(f_p(x)) & y = 1 \\ -\mathbf{1}_1(-f_p(x)) & y = -1 \end{cases}$

(ii) Use the results of (i) as well as the facts given to you at the start of the problem to complete the proof.
Hint: Expand $\mathbb{E}_v\left[|\nabla R(p) - a|^2\right]$ *as a double summation, over elements* $v_i \in \{0, 1\}^d$ *and elements of vector valued function* $g_j(x)$*.*

(d) Your ask your friend, who did not take 189, about the training problems that you observed in part (b). Uninterested in your thoughtful analysis in part (c), your friend suggests that you make the network deeper, and switch to sigmoid activation functions. Why might this be a bad idea, even for small d?

4 Image Classification

Learning Goal: Understand how to train a CNN for image classification

In this problem, we will consider the task of image classification. We will use the CIFAR-10 dataset, which contains sixty thousand 32×32 color images in ten different classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is split into $50,000$ training images and $10,000$ validation images.

First, we approach this problem using linear methods studied earlier in the semester. We will visualize the data using principle component analysis (PCA) and canonical correlation analysis (CCA), and then we will try a simple linear regression approach to classification. Finally, we will use a state-of-the-art convolution neural network (CNN) model. There are many images in the CIFAR-10 dataset, and the CNN model we use will be fairly computationally intensive. For this reason, make sure to give yourself plenty of time to complete this problem.

(a) How hard is this image classification problem? We will investigate by visualizing the dataset in two dimensions. Dataset loading code is provided for you in `cifar10.py`, along with methods to compute the PCA and CCA embeddings of the dataset. **Fill in code to visualize the two dimensional PCA and CCA embeddings.** Make sure to use different colors for each class, and label your plots with titles and legends. **Include your plots, and explain the difference between these two visualization methods.**

(b) Overlapping clusters in a two dimensional visualization does not necessarily mean overlap in a higher dimensional space. As a baseline, we will try linear regression. **Set `LINEAR_FLAG=True` and report the training and validation accuracy**. Do you expect linear regression to perform well? What is your reason?

(c) Now, we will use state-of-the-art neural net architectures. Specifically, we will use a ResNet18 model: a residual CNN with 18 layers. Inspect the code in the class `ResNet18` to understand the architecture. You may also want to run the script with `PRINT_NN_FLAG=True`. **Include a diagram summarizing the network architecture.** Your diagram should indicate the number of dimensions for each layer.

(d) For state-of-the-art performance, a common strategy is data augmentation, in which random crops and shifts are added to the dataset. **Modify `transform_train` to include a random crop of size** 32 **and a random horizontal flip**. You should use methods in `torchvision.transform`. Include a screenshot of your code.

(e) We are almost ready to train the network. In this step, we will compare three different learning rate schemes: constant, annealed, and a specially defined scheme. The constant learning rate should always return a constant value. The annealed learning rate should return the previous learning rate multiplied by a decay factor. The special scheme is defined for you and does not need to be modified. **Implement the constant and annealed learning rate functions**. Include a screenshot of you code in your writeup.

Now, train the network with each of these learning rate schemes for one epoch, using the existing code and setting `TRAIN_NN_FLAG=True`. You may want to use the subset flags to test your

code (e.g. `python cifar10_sol.py --subset-train 128 --subset-val 512`), since it will take a nontrivial amount of time to run on a CPU. **Report your results here**.

(f) (**Not required**) Continue running with the best learning rate scheme for several more epochs to increase the accuracy. By running for several epochs, it is possible to get to over 90% accuracy. This will take several hours on a CPU. (If you have access to a GPU, you can speed up the training significantly.)