CSE 446: Machine Learning
Assignment 4

Due: March 13th, 2020 9:30am

Instructions

Read all instructions in this section thoroughly.

Collaboration: Make certain that you understand the course collaboration policy, described on the course website. You must complete this assignment **individually**; you are **not** allowed to collaborate with anyone else. You may *discuss* the homework to understand the problems and the mathematics behind the various learning algorithms, but **you are not allowed to share problem solutions or your code with any other students.** You must also not consult code on the internet that is directly related to the programming exercise.

You are also prohibited from posting any part of your solution to the internet, even after the course is complete. Similarly, please don't post this PDF file or the homework skeleton code to the internet.

Formatting: This assignment consists of two parts: a problem set and programming exercises.

For the problem set, you must write up your solutions electronically and submit it as a single PDF document. Your problem set solutions must use proper mathematical formatting. For this reason, we **strongly** encourage you to write up your responses using LaTeX.

Your solutions to the programming exercises must be implemented in python, following the precise instructions included in Part 2. Several parts of the programming exercise ask you to create plots or describe results; these should be included in the same PDF document that you create for the problem set.

Homework Template and Files to Get You Started: The homework zip file contains the skeleton code and data sets that you will require for this assignment. **Please read through the documentation provided in ALL files before starting the assignment.**

Citing Your Sources: Any sources of help that you consult while completing this assignment (other students, textbooks, websites, etc.) ***MUST*** be noted in the your PDF document. This includes anyone you briefly discussed the homework with. If you received help from the following sources, you do not need to cite it: course instructor, course teaching assistants, course lecture notes, course textbooks or other course materials.

Submitting Your Solution: You will be submitting only the following files, which you created or modified as part of homework 4:

- `hw4-UWNETID.pdf` (a PDF of your homework 4 writeup)
- `nn.py`
- `imageSegmentation.py`
- `momentum.py` (optional)

Please follow the naming conventions exactly, and do not submit additional files including the test scripts or data sets. Your PDF writeup of Homework w should be named `hw4-UWNETID.pdf`, where "UWNETID" is your own UW netID (for example, my file would be named "hw4-bboots.pdf"). Please submit both the PDF and the `.py` files through Gradescope.

PART I: PROBLEM SET

Your solutions to the problems will be submitted as a single PDF document. Be certain that your problems are well-numbered and that it is clear what your answers are.

1 Logical Functions with Neural Networks (15pts)

For each of the logical functions below, draw the neural network that computes the function, and give a truth table showing the inputs, the value of the logical function, and the output of the neural network verifying that the neural network is correct. Show your work for the computations of the neural network's output.

(a) The NAND of two binary inputs, where $\text{NAND}(x, y) = \neg(x \wedge y)$.

(b) The parity of three binary inputs. The parity is a boolean function with value 1 *iff* the input has an odd number of ones. (Hint: Use your solution to (a) as a subnetwork.)

2 Backpropagation with Momentum (25pts)

[Adapted from Mitchell Sect. 4.5.2.1 and Ex. 4.7] One of the most common modifications to backpropagation is to alter the weight update rule to make the weight update on the nth iteration partially dependent on the update during the previous iteration. The modified weight update rule for the nth epoch is given by:

$$\underbrace{\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}(n)}_{\text{Standard Update Rule}} - \underbrace{\mu D_{ij}^{(l)}(n-1)}_{\text{Momentum Term}} \quad ,$$

where $D_{ij}^{(l)}(n)$ is the average gradient computed at the nth epoch. The first few terms are exactly the same as the standard weight update rule, the last term is new and is governed by a parameter $0 \leq \mu < 1$ called the *momentum*. Essentially, the momentum term includes some fraction of the update from the previous epoch, which will enable the update to bounce out of small local minima or keep moving through flat regions where the search would stop if there were no momentum. It also has the effect of gradually increasing the step size of the search in regions where the gradient does not change, thereby speeding convergence.

Consider a two-layer feed-forward neural network with two inputs x_1 and x_2, one hidden unit h, and one output unit y. This network has five weights $\left(\Theta_{x_1,h}^{(1)}, \Theta_{x_2,h}^{(1)}, \Theta_{0,h}^{(1)}, \Theta_{h,y}^{(2)}, \Theta_{0,y}^{(2)}\right)$, where $\Theta_{0,z}^{(l)}$ represents the threshold weight for unit z at level l. Initialize these weights to be $(0.1, 0.1, 0.1, 0.1, 0.1)$ and give their values after each of the first two training epochs of backpropagation with momentum.

Assume a learning rate of $\alpha = 0.3$, momentum $\mu = 0.9$, a square loss function (i.e., loss for each example is $(y - \hat{y})^2$), and the following two training examples: $(x_1 = 1, x_2 = 0, y = 1)$ and $(x_1 = 0, x_2 = 1, y = 0)$.

Initialize these weights to be $(0.1, 0.1, 0.1, 0.1, 0.1)$ and give their values after each of the first two training epochs of backpropagation with momentum and the gradients you have calculated to do the update. You should ignore the momentum term when performing the first epoch gradient update, but should use it for the second epoch. You should update the parameters twice in total.

Be sure to include your intermediate computations and all your calculations for the partial derivatives for full credit. You are encouraged to use Python to perform the numerical calculations, but if you choose to do so, the only additional package you may choose to use is NumPy and you should include the code you used for this problem in a file named as `momentum.py` and turn it in along with your programming part files.

3 Hyperbolic Tangent Neural Networks (15pts)

[Adapted from Bishop, Exercise 5.1] In a two-layer neural network (one hidden layer) with sigmoid activations, the outputs are given by:

$$y_k(\boldsymbol{x}, \boldsymbol{\theta}) = \sigma\left(\sum_{j=1}^{M} \Theta_{jk}^{(2)} \sigma\left(\sum_{i=1}^{d} \Theta_{ij}^{(1)} x_i + \Theta_{0j}^{(1)}\right) + \Theta_{0k}^{(2)}\right) \quad ,$$

where $\sigma(a) = \dfrac{1}{1+\exp(-a)}$. This equation simply combines all the stages of the network into a single equation. Instead of the sigmoid function, we could use hyperbolic tangent functions $\tanh(a) = \dfrac{e^a - e^{-a}}{e^a + e^{-a}}$ as an activation function.

Consider the two-layer neural network with sigmoid activations described above. Show that there exists an equivalent network, which computes exactly the same function, but with *hidden unit activation functions* given by $\tanh(a)$. Define the parameters of your newly constructed network explicitly. You are allowed to use an additional affine transformation on the output of your tanh network, since the range of tanh and σ are different.

Hint: begin by re-writing the equation above with tanh hidden unit activations, then find the relation between $\sigma(a)$ and $\tanh(a)$, and show that the parameters of the two networks differ by linear transformations.

4 K-Means (12 pts)

Show 2 iterations of the k-means algorithm ($k = 2$) on the following one-dimensional data set:

Data: [4, 1, 9, 12, 6, 10, 2, 3, 9]

First iteration: cluster centers (randomly selected): 1, 6

Data assignment:

- Cluster 1: { 1, 2, 3 }
- Cluster 2: { 4, 9, 12, 6, 10, 9 }

(a) What are the cluster centers, and then the data assignments, that would be obtained for each of two more iterations? Show your work.

(b) After your iterations, has the algorithm converged to a solution at this point, or not? How can you tell?

5 K-Means and Variance (6 pts)

(a) When using the K-Means clustering algorithm, we seek to minimize the variance of the solution. In general, what happens to the variance of a partition as you increase the value of k (the number of clusters) and why? State your answer in one sentence.

(b) For a dataset with n instances, what value of k can you always get a variance of 0? Why? State your answer in one sentence.

PART II: PROGRAMMING EXERCISES

For the programming part, you are only allowed to use NumPy and packages mentioned in this spec. If you wish to use a particular function from some other package, please consult the course staff before you do so.

1 Neural Networks (50 pts)

Relevant Files in the Homework Skeleton

- **nn.py**
- `digitsVisualization.tiff`
- `data/digitsX.dat`
- `data/digitsY.dat`

In this problem, you will implement an artificial neural network (NN) classifier that learns via back-propagation. Instead of choosing the architecture of the network at implementation time, you should implement your neural net in a more general manner; the specific architecture of the network will be given to the constructor. Recall from class that we can specify the architecture of a basic neural net via a vector $\mathbf{s}$, where each entry in the vector gives the number of nodes in that layer and the length of the vector is the total number of layers in the network. We will assume that within each layer, each node receives input from every other node in the previous layer.

Implement the neural network in a class called `NeuralNet` in `nn.py`, following the API given below:

- `__init__(layers, learningRate, regParam, epsilon=0.12, numEpochs=100)` : constructor

  `layers` – a vector of $L-2$ positive integers, where the number of layers in the network is L. The value contained in `layers[i]` specifies the number of nodes in the ith hidden layer. Note that the specification of `layers` is a bit different from the $\mathbf{s}$ vector; in particular, it does not include s_0 and s_{L-1}, which are initialized from the training data in `fit()`.

  `epsilon` – one half the interval around zero for the initial weights (defaults to 0.12)

  `regParam` – the regularization parameter

  `learningRate` – the learning rate for back-propagation

  `numEpochs` – the number of epochs for backpropagation

- `fit(X,Y)`: train the neural network model via backpropagation
- `predict(X)`: use the trained neural network model for prediction via forward propagation
- `visualizeHiddenNodes(filename)`: (Extra Credit Only) outputs an image representing hidden layers

While implementing the neural net, I recommend you follow the steps outlined below.

1.1 Network Structure and Initialization

Complete the constructor, which simply saves the arguments for use later. Since the number of units in the first and last layer are specified by the training data, we cannot initialize the network architecture and weight matrices until `fit()`. After finishing `__init()__`, start writing `fit()`.

The first thing you should do in `fit()` is to create all of the weight matrices $\Theta^{(1)}, \ldots, \Theta^{(L-1)}$. Initialize all of the weights to be uniformly chosen from $[-\epsilon, \epsilon]$, where ϵ is specified by the `epsilon` argument to the constructor.[1] Then, unroll the weight matrices $\Theta^{(1)}, \ldots, \Theta^{(L-1)}$ into a single long vector $\boldsymbol{\theta}$ that contains all parameters for the neural net.

1.2 Forward-propagation

Implement a private function to perform forward-propagation. This method will be useful for both back-propagation and the `predict()` method. This private function should take in a vector of parameters (e.g., $\boldsymbol{\theta}$) for the neural network and an instance (or instances) and return the neural network's outputs.

[1] According to Andrew Ng, an alternative (and effective) strategy for choosing ϵ is to choose a different value for each layer's weights, with the ϵ for $\Theta^{(l)}$ as $\frac{\sqrt{6}}{\sqrt{s_l + s_{l+1}}}$, where $\mathbf{s}$ is the vector containing the number of nodes in each layer.

1.3 Backpropagation

Finish the `fit()` method to use backpropagation to minimize $J(\boldsymbol{\theta})$. Details for implementing backpropagation are in the lecture slides. At a minimum, you'll need to compute $J(\boldsymbol{\theta})$ and $\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta})$.

To confirm that your gradient computations are correct, I recommend that you implement the following gradient checking procedure to numerically estimate the gradient. Form two vectors:

$$\boldsymbol{\theta}_{i+c} \leftarrow \boldsymbol{\theta};\ \text{then}\ \ \theta_i \leftarrow \theta_i + c \qquad\qquad \boldsymbol{\theta}_{i-c} \leftarrow \boldsymbol{\theta};\ \text{then}\ \ \theta_i \leftarrow \theta_i - c\ . \tag{1}$$

In other words, $\boldsymbol{\theta}_{i+c}$ (or $\boldsymbol{\theta}_{i-c}$) is the same as $\boldsymbol{\theta}$, but the ith element has been incremented (or decremented) by c. Then, we can numerically estimate the gradient and verify that our gradient computations are correct by confirming that

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{J(\boldsymbol{\theta}_{i+c}) - J(\boldsymbol{\theta}_{i-c})}{2c}\ . \tag{2}$$

Setting $c = 10^{-4}$, you should find that the computed gradient and the estimated gradient should agree to at least four significant digits.

This gradient checking procedure is very expensive, and so should only be used for small networks (small numbers of parameters). **Make absolutely certain to disable the gradient checking procedure once you're certain that your gradient implementation is correct.** Use the gradient checking procedure only for debugging purposes; be sure to disable it before running your learning algorithm.

Once your gradient computations are confirmed to be correct, finish `fit()` to run backpropagation for the number of epochs specified in the constructor to train the neural net.

1.4 Complete the Prediction Function

Your prediction function should call the forward-propagation method with the neural net parameters $\boldsymbol{\theta}$ trained via backpropagation in `fit()`.

1.5 Apply Your Neural Network to Digit Recognition

Write a test script named `testNeuralNetDigits.py` to apply your neural network classifier to the problem of digit recognition. The homework skeleton contains a data set of 5,000 20×20 digit images (see `digitsVisualization.tiff` for a visualization). We can represent each image as a 400-dimensional vector of pixel intensities. The features for the digits are provided in the `data/digitsX.dat` file and their corresponding labels are in `data/digitsY.dat`.

Train your neural network on the digits data with one hidden layer of 25 nodes over 100 epochs. Choose a small value for the regularization parameter in the neural network (e.g., start with something like 0.001 and tune it up or down as needed by hand – no need to tune it via cross-validation).

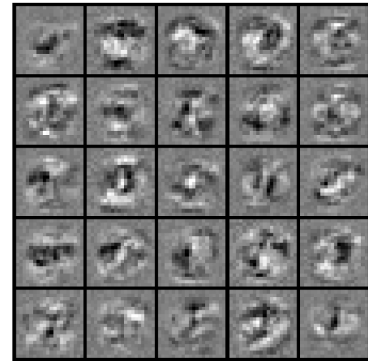
Tune the learning rate for the neural network. You should be able to get a **training** accuracy of approximately 95.3% or higher if your implementation is correct ($\pm 1\%$ due to random initialization). If you're having trouble obtaining this accuracy with only 100 epochs, try using more epochs. It is fine if your implementation requires a few hundred more epochs to obtain higher accuracy. Report your optimal learning rate, regularization parameter, and the maximum training performance you obtained in your PDF writeup.

1.6 Visualizing the Hidden Layers

Complete the `visualizeHiddenNodes(filename)` function to visualize the hidden units in the network. For the neural network you trained above, note that the ith row of $\Theta^{(1)}$ is a 401-dimensional vector that specifies the parameters for the ith hidden unit. Discarding the bias term yields a 400-dimensional vector that we can reshape into an 20×20 matrix via the `numpy.reshape` command. If we remap[2] the values of this matrix to lie in $0 \ldots 255$, we can visualize the weights as a greyscale image.

[2] I suggest you either map -1 to 0 and +1 to 255, or the min value over all units to 0 and the max value over all units to 255 – whichever gives you the nicer picture.

Use the Python Image Library (you might find it useful to consult http://en.wikibooks.org/wiki/Python_Imaging_Library). to create an image that visualizes all of the hidden layers. Separate the layers into blocks (e.g., you can visualize a 25 unit layer as an 5×5 grid, where each grid entry is the 20×20 greyscale image). You might find it useful to consult http://en.wikibooks.org/wiki/Python_Imaging_Library/Editing_Pixels. Save this image to the filename given as an argument to `visualizeHiddenNodes()`.



You should find that the hidden units correspond to different stroke detectors and other patterns in the input. For an example of the hidden unit visualization, see the image to the right. Yours will likely look slightly different from this one due to randomization. Include your output image visualizing the hidden layers of your network in your PDF writeup.

2 Image Segmentation using K-Means (30 pts)

In the problem, you will apply K-Means to image segmentation. Write a program named `imageSegmentation.py` that reads in an image, segments that image using K-Means clustering as described below, and outputs the new segmented image. Your program must support the following command line arguments:

```
python imageSegmentation.py K inputImageFilename outputImageFilename
```

The first argument `K` is an integer greater than 2 that specifies the number of clusters, `inputImageFilename` is the filename of the input image, and `outputImageFilename` is the filename to write the output image. For example, we might call your program via:

```
python imageSegmentation.py 24 newyorkcity.jpg nyc-segmented.jpg
```

Choose several nice natural images, such as a farmhouse against a blue sky, or a city scene. First write code to load the image using the Python Image Library, which supports a wide variety of image file formats, and will automatically determine the filetype based on the file extension. (We will test your program with .jpg and .png files, so make certain to test your program with those types.)

We can think of an image as being represented as a 3-D matrix of size $imageWidth \times imageHeight \times 3$. For each location in the image (i, j), the matrix contains three values for the red, green, and blue components of the pixels. We will use these pixel values for clustering. In addition to the color values (r_p, g_p, b_p) for pixel p, we will also use the x,y coordinates (i_p, j_p) as features. In particular, we can represent each pixel p as a five-dimensional data vector $\mathbf{x}_p = \begin{bmatrix} r_p & g_p & b_p & i_p & j_p \end{bmatrix}$.

Complete the program via the following steps:

- Convert the input image into a data set with five features, as described above. To improve results, you should also standardize the values of each feature in the data set.
- Implement your own version of K-Means and use it to cluster the data (i.e. the features for each pixel) into `K` clusters. If a cluster is ever empty during the procedure, assign a random data point to it. Use random initializations for the cluster centers, and iterate until the centroids converge.
- Use the cluster centers to generate the segmented image by replacing each data point's color values with the closest center. For example, $\mathbf{x}_p$ becomes
$$\hat{\mathbf{x}}_p = \begin{bmatrix} r_{\mathcal{C}(p)} & g_{\mathcal{C}(p)} & b_{\mathcal{C}(p)} & i_p & j_p \end{bmatrix} ,$$
where $\mathcal{C}(p)$ is the cluster to which $\mathbf{x}_p$ belongs and $(r_{\mathcal{C}(p)}, g_{\mathcal{C}(p)}, b_{\mathcal{C}(p)})$ are the corresponding RGB values of that cluster's centroid. Note specifically that we're only replacing the color values of each instance with its centroid's colors, we're **not** changing the (i,j) coordinates of that instance.

- Create an output image the same size as the input image. Then fill in the color values of each pixel of the image based on the $\hat{\mathbf{x}}_p$'s. For example, $\hat{\mathbf{x}}_p$ informs us that the pixel at (i_p, j_p) should have color $(r_{\mathcal{C}(p)}, g_{\mathcal{C}(p)}, b_{\mathcal{C}(p)})$. Note that you also have to undo the feature standardization at this point (just invert the standardization equation by solving for the original value given the standardized value).
- Output the resulting image to the file `outputImageFilename`.
- In your PDF writeup, include three different examples of an original image alongside the resulting segmented image.

The result of this process is called an over-segmented image. It is the first step to building such systems as this: http://make3d.cs.cornell.edu/. Later steps would piece these segments together into objects.