CS 189 / 289A Introduction to Machine Learning
Fall 2022 Jennifer Listgarten, Jitendra Malik **HW2**

Due 10/04/21 at 11:59pm

- Homework 2 consists entirely of coding questions.
- We prefer that you typeset your answers using LaTeX or other word processing software. If you haven't yet learned LaTeX, one of the crown jewels of computer science, now is a good time! Neatly handwritten and scanned solutions will also be accepted.

Deliverables:

Submit a PDF of your homework to the Gradescope assignment entitled "HW2 Write-Up". **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.

- In your write-up, please state with whom you worked on the homework. If you worked by yourself, state you worked by yourself. This should be on its own page and should be the first page that you submit.
- In your write-up, please copy the following statement and sign your signature underneath. If you are using LaTeX, you must type your full name underneath instead. We want to make it *extra* clear so that no one inadvertently cheats. *"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."*
- **Replicate all your code in an appendix**.
- **See the end of the assignment for the deliverables for each part!**

1 Implementing A Neural Network in NumPy (60 points)

In this assignment, we will be implementing a neural network and the backpropagation algorithm in NumPy. Later on in this course, we will introduce you to PyTorch, which is extremely useful for quickly and correctly building neural networks. However, libraries such as PyTorch also hide many of the key algorithmic details behind implementing and training neural networks. To really understand neural networks and the backpropagation algorithm, it is very helpful to implement everything yourself. After this homework assignment, we hope that you gain this deeper understanding!

Our goal will be to train a neural network to classify a handwritten digit as either a 3 or a 9. We will be using a subset of the famous MNIST dataset. The neural network will be trained using mini-batch stochastic gradient descent (SGD). Recall that in standard SGD, at each timestep t, we compute the loss for a *single* data point and update the weights based on the gradient of the loss. In mini-batch SGD, we instead compute an average loss over a randomly sampled *batch* of n data points at each timestep. Note that your code should be general enough to handle SGD (by setting $n = 1$) as well as traditional gradient descent (by setting n equal to the total number of data points).

1.1 Getting started

Install the MNIST Python library by running `pip install mnist` in your terminal. The `get_mnist_threes_nines` function in `hw2.py` provides the training and test splits that we will use in this assignment. Familiarize yourself with the data and look at a few examples using the `display_image` function. Check for yourself that you can answer the following questions: What are the labels for an image of a 3? For a 9? What are the the dimensions of a single image? If we want to treat each image as a vector how could we do that and how many features would that image have?

Neural networks will take longer than other assignments to train. Therefore, it is necessary that you *vectorize* your computations in order to take advantage of the highly optimized operations provided to you by NumPy. Vectorize your code when possible to save you hours of time. A simple rule-of-thumb is to avoid explicit for-loops whenever possible.

A quirk of neural network programming frameworks is that inputs are usually shaped as row vectors being multiplied by matrices on the right, contrary to mathematical frameworks, in which inputs are usually column vectors multiplied by matrices from the left. One reason for formatting inputs as row vectors is that given an array `X` of inputs, we can retrieve the ith input with `X[i]`, i.e. by indexing the first dimension.

A particularly useful function that you should learn is `numpy.einsum`. The syntax is tricky to learn at first and takes a while to get used to. However, many machine learning practitioners find this function incredibly useful, and for this assignment, you will too.

1.2 Questions

(a) (0 points) Backpropagation is quite tricky to implement correctly. You will verify that the gradients computed by your backpropagation code are correct by comparing its output to gradients computed by finite differences. Let $f : \mathbb{R}^d \to \mathbb{R}$ be an arbitrary function (for example, the loss of a neural network with d weights and biases), and let a be a d-dimensional vector. You can approximately evaluate the partial derivatives of f as:

$$\frac{\partial f}{\partial x_k}(a) \approx \frac{f(\ldots, a_k + \epsilon, \ldots) - f(\ldots, a_k - \epsilon, \ldots)}{2\epsilon}$$

where ϵ is a small constant. As ϵ approaches 0 , this approximation theoretically becomes exact, but becomes prone to numerical precision issues. $\epsilon = 10^{-5}$ usually works well. Implement a finite differences checker, which you will use later to verify the correctness of your backpropagation implementation.

(b) (8 points) Here, we will implement several helper functions that we will combine in the next two parts.

(i) To perform binary classification, we will have the last layer of our neural network consist of a single neuron with the sigmoid activation function, $\sigma(s) = \frac{1}{1+e^{-s}}$. Implement a function `sigmoid_activation` which takes in a NumPy array of arbitrary shape and applies the sigmoid function elementwise. The function should return both the output and the gradient of the activation function with respect to the input (as this will be needed for the backward pass). Both returned objects will be arrays of the same shape as the input.

To vectorize the numerically stable version of this function (this will be elaborated in the deliverables below), it may help to use the `numpy.where` function in combination with the optional `where` and `out` arguments inside of `numpy.exp`. Also note that floating point limitations may cause probabilities equal to 0 for very negative inputs, which will cause errors in the next part. To correct this, we can clip the output of the sigmoid function to be between $(\epsilon, 1 - \epsilon)$ where ϵ is a small number such as 10^{-15}. Apply this fix to your implementation of `sigmoid_activation` before proceeding.

(ii) Now, implement a function `logistic_loss(g, y)` where g is the output of the neural network and y is array of the true labels. The function should return two arrays: `loss` and `dL_dg` where `loss[i]` is the negative of the log-likelihood defined in lecture for binary classification, i.e.

$l(g[i], y[i]) = -\log\{g[i])^{y[i]}(1 - g[i])^{1-y[i]})\}$ and `dL_dg[i]` is the derivative of `loss[i]` with respect to `g[i]`.

Note: be careful about NumPy broadcasting rules. It may help to include assertions in your code about the dimensions of certain inputs and outputs to prevent unintended silent broadcasting from introducing bugs. See the NumPy documentation for more information.

(iii) For the intermediate layers, we will utilize the ReLU activation function, $\sigma(x) = \max\{0, x\}$. Write a function `relu_activation` which takes in an $n \times d$ array, s, and returns both the elementwise activation of s (an $n \times d$ array) and the partial derivatives of the activation function with respect to the elements of s (an $n \times d$ array).

(iv) Finally, implement a function `layer_forward(x, W, b, activation_fn)` where `x` is the $n \times d^{(l-1)}$ dimensional matrix consisting of the mini-batch neurons for layer $l-1$, `W` is the $d^{(l-1)} \times d^{(l)}$ weight matrix for layer l, `b` is $1 \times d^{(l)}$ bias row vector for layer l, and `activation_fn` is a function that applies the elementwise activation to $S^{(l)}$ (i.e. this should be either `relu_activation` or `sigmoid_activation`). The function should return two things, `out` and `cache`. `out` should be the $n \times d^{(l)}$ dimensional matrix corresponding to the mini-batch neurons for layer l and `cache` is a tuple consisting of any information that we need to compute $\frac{\partial l(W,b)}{\partial W_{ij}}$ and $\frac{\partial l(W,b)}{\partial b_j}$ in the backward pass. By looking at the induction step that we derived in lecture, you should figure out what exactly `cache` needs to contain.

(c) (8 points) We are now ready to fully implement a forward pass of the neural network! We will specify the neural network by a list of the layer dimensions. For example, to implement a two layer neural network where the first layer maps the 784-dimensional image to 200-dimensional space and the second layer maps to a 1-dimensional output, we would use `layer_dims = [784, 200, 1]`. Similarly, we must specify the activation functions used at each layer, e.g., `activations = [relu_activation, sigmoid_activation]`. Write helper functions `create_weight_matrices(layer_dims)` and `create_bias_vectors(layer_dims)` that return a list of weight matrices and bias vectors, respectively, for each layer specified in `layer_dims`. You should initialize the weights by sampling each weight from a normal distribution with mean 0 and standard deviation 0.01. Finally, write a function `forward_pass(X_batch, weight_matrices, biases, activations)` which takes in an $n \times 784$ dimensional mini-batch of flattened images, a list of weight matrices, a list of bias vectors, and a list of activation functions, and returns a vector of outputs and a list of layer caches (defined previously).

You should now be able to compute the predictions, loss and the gradient with respect to the output for a mini-batch as follows:

```
# run forward pass
output, layer_caches = forward_pass(X_batch,
                                    weight_matrices,
                                    biases,
                                    activations)
# compute loss and derivative with respect to logits
loss, dL_dg = logistic_loss(output, y_batch)
print("Average Loss Across Batch", loss.mean())
```

(d) (6 points) We are now at the most difficult part of this problem: implementing the backward pass. Write a function called `backward_pass` that uses the derivative computed in `logistic_loss` and the layer caches to compute the gradients of each of the weight matrices and bias vectors. Implementing this correctly will require both a strong understanding of the backpropagation algorithm, as discussed in lecture, and careful coding/debugging to make sure each step is implemented appropriately. Your finite differences implementation will come in handy here to make sure that your backpropagation implementation is working correctly.

(e) (6 points) Now it is time to bring it all together. Using a mini-batch size n of 100 and a step size of 0.1 for SGD, implement a training loop that iterates over the mini-batches of data, computes the forward pass, computes the loss, computes the backward pass, and updates the weights using SGD. Note that typically, instead of randomly sampling each mini-batch, the training data is shuffled and then divided up into mini-batches for efficiency. Each loop over the entire dataset is commonly referred to as an *epoch*, and the training data is reshuffled for each epoch. Train a two layer network with a hidden dimension of 200 (i.e. `layer_dims = [784, 200, 1]`) for 5 epochs. For each timestep, record the average loss across the training mini-batch, the average accuracy across the training mini-batch, the average loss across the entire test set, and the average accuracy across the entire test set. You should be able to achieve a test accuracy of around 98%.

1.3 Deliverables

We have provided inputs, weight matrices, and biases for a neural network with input layer of dimension 4, hidden layer of dimension 2 and output layer of dimension 1. The activation functions are again ReLU and sigmoid.

The toy test input and parameters can be loaded using the `pickle` module as follows:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)
```

Parts (a) through (d) of the deliverables use the toy test data and neural network provided in the pickle file. Part (e) of the deliverables uses the MNIST data and the full-size neural network trained on the MNIST data in 1.2 (e).

(a) (6 points) Compute the finite difference approximation to the gradients of the loss with respect to the weights and biases of the neural network. The code to produce the result may look like:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)

grad_Ws, grad_bs = my_nn_finite_difference_checker(X_batch,
                                                    y_batch,
                                                    weight_matrices,
                                                    biases,
                                                    activations)


with np.printoptions(precision=2):
    print(grad_Ws[0])
    print()
    print(grad_Ws[1])
    print()
    print(grad_bs[0])
    print()
    print(grad_bs[1])
```

Your answer should be the result of the printout of the gradients.

(b) (4 points) Answer the following questions:

(i) What does `sigmoid_activation` return for `s = np.asarray([1., 0., -1.])`?

(ii) What does `sigmoid_activation` return for `s = np.asarray([-1000, 1000])`? If you observe an overflow warning from NumPy: can you figure which input, -1000 or 1000, is causing it? Ensure that your implementation of `sigmoid_activation` is numerically stable and does not cause overflows, and explain how you achieved this.
Hint: Consider the equivalent definitions of the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$. Can we use both of these definitions to avoid overflow errors?

(iii) What is the derivative of the negative log-likelihood loss with respect to g?

(iv) Explain what is returned in `cache` in your `layer_forward` implementation.

(c) (4 points) Report the average loss for the test data, weight, and biases provided. The code needed to produce the response may look like:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)

activations = [relu_activation, sigmoid_activation]
output, _ = forward_pass(X_batch, weight_matrices, biases,
                         activations)
loss, dL_dg = logistic_loss(output, y_batch)
print(loss.mean())
```

(d) (6 points) Report the gradients of the weights and biases computed from the test data using backpropagation. Compare these gradients to the finite differences approximation and make sure the two gradients are close. The code to produce the response might look like:

```
with open("test_batch_weights_biases.pkl", "rb") as fn:
    (X_batch, y_batch, weight_matrices, biases) = pickle.load(fn)

activations = [relu_activation, sigmoid_activation]
output, layer_caches = forward_pass(X_batch, weight_matrices, biases,
                                    activations)
loss, dL_dg = logistic_loss(output, y_batch)
grad_Ws, grad_bs = backward_pass(dL_dg, layer_caches)

with np.printoptions(precision=2):
    print(grad_Ws[0])
    print()
    print(grad_Ws[1])
    print()
    print(grad_bs[0])
    print()
    print(grad_bs[1])
```

(e) (12 points) Now turning to the full-size neural network trained according to 1.2 (e), answer the following questions:

   (i) Make a plot of the training and test losses per timestep.

   (ii) Make another plot of the training and test accuracies per timestep.

   (iii) Examine the images that your network guesses incorrectly, and explain at a high level what patterns you see in those images.

   (iv) Rerun the neural network training but now increase the step size to 10.0. What happens? You do not need to include plots here.

(f) (Optional) Neural networks are often referred to as *universal function approximators*, which means given any continuous function on a bounded domain and a desired approximation accuracy, if the hidden layer in neural network you coded has large enough dimension, there is a set of weights which approximate that function to that accuracy. Colloquially, this means that a big enough neural network can be trained to approximate any function. Adding layers is another way of increasing a neural network's approximation power.

Generate a new training set `X_train_rand` consisting of 100 inputs of dimension 784 sampled using `numpy.random.rand`, which samples each entry from the Uniform([0, 1]) distribution. Train your network to fit `X_train_rand` to the first 100 labels of the original MNIST training data. Of course, the network isn't really learning anything, because there is no actual relationship between the input and the label. Fitting the training data without capturing a general pattern is called *memorization*. Try playing with your network and training configuration so that your neural network memorizes the training set, i.e. achieves 100% training accuracy, in as few iterations as possible.