USC CSCI 467
Intro to Machine Learning

Assignment 3

Due: April 9, 2024, 11:59pm

Spring 2024
Instructor: Robin Jia

This assignment has 3 questions, for a total of 100 points. Make sure you also download the `hw3.zip` file from the course website.

When submitting on Gradescope, note that **you must make a submission both for the written portion and programming portion.** For the programming portion, upload the files `backprop.py` and `kmeans.py` with your completed solutions. (There is an "autograder" which will not actually grade your code but it will run it, and should return 0 if it encountered an error and 100 otherwise.) **Please still include the output of your code in the PDF report when requested in the problems.**

Question 1: Implementing Backpropagation (51 points)

In the last assignment, you implemented neural networks in Pytorch. This only required writing code for the forward pass, since Pytorch does automatic differentiation by backpropagation for you. In this assignment, you will delve deeper by writing your own backpropagation code and using it to train both a logistic regression classifier and a neural network classifier.

Let's look at the starter code in `backprop.py`. This contains the code from the backpropagation lecture, with a couple small modifications. Before we talk about the modifications, let's review how the code works. Each `Node` object represents a node in the computation graph. In `Node.__init__(*args)`, a node is told what its child nodes are. It should:

- Store these child nodes, so that it can update the child nodes' gradients in the backward pass.
- Compute and store the value of this node given the values of the children. In other words, this is performing the forward pass.
- Initialize the gradient of this node to be 0.
- Save a pointer to the topological order list for the current computation graph. This can be gotten from any of the child nodes.
- Add the current node to this topological order.

Then, in `Node.backward()`, the node should perform the backward pass by updating the gradient of each child node. The code assumes that `self.grad`, the gradient of the output with respect to the current node, has been computed correctly. That gradient information must get propagated back to the child nodes.

The modifications I've made to the lecture code are:

- I've changed all `Node.value` and `Node.grad` attributes to be numpy datatypes (e.g. `np.float64`) rather than standard python floating point numbers. This prevents some annoying type mismatch errors. **In your implementation, remember to use numpy datatypes and not python floats/ints.**
- I've changed `InputNode` to accept a numpy array as an input, in addition to a normal float. This will be useful when we start defining nodes that takes vectors and matrices as inputs. Note that if the input is a vector or matrix, `self.grad` should be a vector or matrix of the same shape, initialized to all zeros. This is because `self.grad` stores the gradient of the output of the computation graph (which is always a scalar) with respect

to the current node. The gradient is just the vector or matrix of partial derivatives with respect to each entry of the node.

- I've changed `ReluNode` and `AddNode` to work even if the input(s) and output are vectors. In both cases, the operations are element-wise, so the backpropagation rules don't really change; we just have to allow for the gradient objects to be vectors.
- I've added a `ConstantMulNode` class, which is a slight variation on `MulNode`. You may recall that `MulNode` takes in two nodes and multiplies them. `ConstantMulNode` takes in one node and multiplies it by a fixed constant. For example, if you have a node called n, you can compute $5n$ by calling `ConstantMulNode(n, 5)` (in this way, it is similar to `PowerNode`). If you wanted to do the same thing with `MulNode`, you would have to write `MulNode(n, InputNode(5))` which is a bit uglier.

(a) (2 points) We will start by writing a node for the function $\log(x)$ (the base of the log is e). In the forward pass, our node will take in an input x and produce output $u = \log(x)$. In the backward pass, our node gets told that $\frac{\partial y}{\partial u} = g$ for some value g, where y is the final output of the computation graph. We must then figure out how to update $\frac{\partial y}{\partial x}$, the partial derivative of our child with respect to the final output.

Write the formula for the backpropagation update rule to $\frac{\partial y}{\partial x}$ in terms of g and x. It should have the form:

$$\frac{\partial y}{\partial x} += [\cdots]$$

Review the lecture slides on backpropagation if you don't remember why we increment the partial derivative, instead of just setting it to a value.

(b) (3 points) Using your formula from the previous part, implement the `LogNode` class. Your implementation only needs to handle the case where the input and output are scalars.

(c) (2 points) Next we will write a node for $\sigma(x)$, the sigmoid function used in logistic regression. Our node takes in input x and produces output $u = \sigma(x)$, where

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

The derivative of $\sigma(x)$ can be written in terms of $\sigma(x)$ itself. Write the formula for the backpropagation update rule to $\frac{\partial y}{\partial x}$ in terms of g and u, where g is the already-computed value of $\frac{\partial y}{\partial u}$.

(d) (2 points) I could have also asked you to write the formula in terms of g and x. What is the computational advantage to writing the formula in terms of u instead? (If you're not sure, try implementing `SigmoidNode` first and then think about what would change if you used a formula in terms of g and x.)

(e) (3 points) Using the formula you wrote, implement the `SigmoidNode` class. To compute $\sigma(x)$, you should use the `sigmoid` function imported at the top of the file. (This will give numerically stable results when x is very large or very small). Again, your implementation only needs to handle the case where the input and output are scalars.

(f) (2 points) To implement logistic regression, we will need one more type of node: a dot product. The node will take in two *vectors*, x and v, and produce output $u = x^\top v$.

Write the formula for the backpropagation update rules to $\nabla_x y$ and $\nabla_v y$ in terms of x, v, u, and g, where g is the already-computed value of $\frac{\partial y}{\partial u}$. I'm using the notation $\nabla_x y$ instead of $\frac{\partial y}{\partial x}$ just because x is now a vector and not a scalar; conceptually these are analogous, since the gradient is just the vector of partial derivatives with respect to each component of x. (Note: your expressions might only use a subset of x, v, and u, but you're free to use any of them in your expression.)

(g) (3 points) Using your formulas from the previous part, implement the `DotNode` class.

(h) (6 points) We now have all the components we need to implement logistic regression! But before we do, it's a good idea to implement a gradient check to make sure our backpropagation code is correct. To test our `LogNode`, `SigmoidNode`, and `DotNode` implementations simultaneously, let's use the function

$$f(a, b, c) = \log(a^\top b) - 3\sigma(a^\top c),$$

where a, b, and c are all vectors in $\mathbb{R}^2$.

Recall from lecture that computing the numerical gradient means you perturb each coordinate of the input by a small number ϵ (you should use `EPSILON = 1e-8` as defined at the top of `backprop.py`), measure the change in f, and use that to approximate the partial derivative with respect to that coordinate. You can look at the lecture demo code as a reference.

Fill in the missing code inside `gradient_check_1` to:

- Compute $f(a, b, c)$ using the new `Node` subclasses you just implemented
- Numerically the gradient of f with respect to a, b, and c

When you're ready, run

```
python3 backprop.py grad_check_1
```

This will compare the numerical gradient to the gradient computed via backpropagation on the inputs $a = [3, 2]$, $b = [-2, 7]$, $c = [1, -4]$ (these were chosen arbitrarily). If you have implemented the forward pass correctly, the program should print that $f(a, b, c) \approx 2.059$. The gradients computed numerically and with backprop should be very similar; in particular, the maximum difference between any of the partial derivatives computed should be less than 10^{-7}. If any of the differences are bigger, you probably have a bug somewhere in your implementation.

(i) (7 points) We are now ready to implement logistic regression! In this problem, we will assume the inputs x are in $\mathbb{R}^2$, and our parameters are a weight vector $w \in \mathbb{R}^2$ and bias $b \in \mathbb{R}$. The predictions of the model are given by

$$P(y = 1 \mid x) = \sigma(w^\top x + b),$$

and the loss function given a dataset of n examples $\{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$ is

$$L(w, b) = \sum_{i=1}^{n} -\log \sigma(y^{(i)} \cdot (w^\top x^{(i)} + b)).$$

Now, implement the following:

- The function `make_logistic_regression`: This function takes in a dataset, formatted as a list of pairs `(x, y)` where `x` is a numpy array of length 2, and `y` is either 1 or -1. It then **returns a function**, which we are calling `compute_loss`, that takes in a choice of w (as a numpy array) and b (as a numpy float) and creates a computation graph that computes the loss on the dataset.[1]

[1] More precisely, we create a **closure**. For a primer on closures in Python, see https://www.geeksforgeeks.org/python-closures/.

- The function `gradient_descent`: This function takes as input a loss function (such as the one returned by `make_logistic_regression`), a list of initial values for parameters (i.e., the inputs to the loss function), a learning rate, and number of iterations. It runs gradient descent on the loss function, storing the updated values of the parameters inside the provided list of parameter values.

When you're done, run your logistic regression implementation on a simple dataset:

```
python3 backprop.py logreg -d simple
```

This trains on a 4-example linearly separable dataset. You should get a final loss of roughly 0.014.

(j) (2 points) While logistic regression perfectly classifies this dataset, it cannot perfectly classify a dataset that is not linearly separable. Try the following:

```
python3 backprop.py logreg -d xor
```

This runs logistic regression on `XOR_DATASET` defined at the top of `backprop.py`. What is the final loss that logistic regression achieves? Why is it much higher than the loss on the simple dataset?

(k) (4 points) To fit this XOR dataset, we will need a neural network. To implement a neural network, we will need one more operation: multiplying a matrix by a vector. Thus, we will implement `MVMulNode`, which takes as input a matrix M of dimension $p \times d$ and vector v of length d, and computes the matrix-vector product $u = Mv$ as output.

Write the formulas for the backpropagation update rules to $\nabla_M y$ and $\nabla_v y$ in terms of M, v, u, and g, where g is the already-computed value of $\nabla_u y$. Note that since the output u is a vector of length p, g is a vector of length p as well. $\nabla_M y$ is a matrix the same shape as M, where each entry is the partial derivative of y with respect to the corresponding entry of M:

$$\nabla_M y = \begin{pmatrix} \frac{\partial y}{\partial M_{11}} & \frac{\partial y}{\partial M_{12}} & \cdots \frac{\partial y}{\partial M_{1d}} \\ \frac{\partial y}{\partial M_{21}} & \frac{\partial y}{\partial M_{22}} & \cdots \frac{\partial y}{\partial M_{2d}} \\ \cdots & & \\ \frac{\partial y}{\partial M_{p1}} & \frac{\partial y}{\partial M_{p2}} & \cdots \frac{\partial y}{\partial M_{pd}} \end{pmatrix}$$

Hint: This is the trickiest gradient problem, but you can derive it from first principles. Write out the formula for Mv and analyze the partial derivative with respect to M_{ij} and v_j. Also, you can look at the notes of Section 3 Calculus and Gradients on course website for reference. (Note: your expressions might only use a subset of M, v, and u, but you're free to use any of them in your expression.)

(l) (3 points) Using your formulas from the previous part, implement the `MVMulNode` class.

(m) (6 points) Let's do another gradient check for `MVMulNode`. Our inputs will be a matrix `M` and vector `v`, and we will compute the function

$$f(M, v) = v^\top \operatorname{ReLU}(Mv),$$

where $\operatorname{ReLU}(z) = \max(z, 0)$ is the element-wise ReLU function.

Fill in the missing code inside `gradient_check_2`. When you're ready, run

```
python3 backprop.py grad_check_2
```

This will compare the numerical gradient to the gradient computed via backpropagation on the inputs

$$M = \begin{pmatrix} 2 & -1 \\ 0 & 3 \end{pmatrix}$$

$$v = \begin{pmatrix} 1 \\ -3 \end{pmatrix}$$

If you have implemented the forward pass correctly, the program should print that $f(M, v) = 5$. The gradients computed numerically and with backprop should be very similar; in particular, the maximum difference between any of the partial derivatives computed should be less than 10^{-7}. If the differences are bigger, you probably have a bug somewhere in your implementation.

(n) (6 points) Finally, we are ready to implement a neural network! We will use a neural network with a single hidden layer that has 3 hidden units, and the ReLU activation function. As before with logistic regression, we will assume that the inputs are of dimension 2. This means that the parameters are a matrix $W \in \mathbb{R}^{3\times 2}$, vector $b \in \mathbb{R}^3$, vector $v \in \mathbb{R}^3$, and scalar c. We will use a logistic regression-style loss function, so the output of the neural network can be interpreted as a probability:

$$p(y = 1 \mid x) = \sigma(v^\top \operatorname{ReLU}(Wx^{(i)} + b) + c).$$

Thus, the loss function given a dataset of n examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ is

$$L(w, b) = \sum_{i=1}^{n} -\log \sigma(y^{(i)} \cdot (v^\top \operatorname{ReLU}(Wx + b) + c)).$$

Implement `make_neural_network`, which is analogous to `make_logistic_regression` but will instead compute the neural network loss function described above. It will take as input values for W, b, v, and c.

When you're done, run your neural network implementation on the xor dataset:

```
python3 backprop.py neural -d xor
```

This will use the same `gradient_descent` function you completed earlier, but with the loss function produced by `make_neural_network`. You should get a final loss of roughly 0.019, much lower than the loss of logistic regression on the same problem. Congratulations! You have written code that trains a neural network from scratch.

Question 2: k-Means and Image Compression (24 points)

In this problem, you will implement k-Means clustering and use it to compress an image.

(a) (10 points) Implement the `kmeans` function inside `kmeans.py`. This function should return a tuple containing the cluster assignments, the means of each cluster, and the k-means loss (also known as the reconstruction error), defined as

$$\sum_{i=1}^{n} \|x^{(i)} - \mu_{z_i}\|^2$$

where $x^{(i)}$ is the i-th example, z_i is the ID of the cluster assigned to the i-th example, and μ_c is the centroid of the c-th cluster.

The starter code already initializes the cluster centroids to randomly chosen examples in the dataset. You should add code that keeps updating the cluster assignments and centroids until the assignments stop changing. **Your code should not contain any for loops over the number of examples.** Hint: it may be helpful to use the fact that

$$\|x - y\|^2 = \|x\|^2 + \|y\|^2 - 2x^\top y.$$

The function `np.argmin()` will also be useful. (It is OK to have a for loop over the number of clusters)

When you're ready, run your code with $k = 2$:

```
python3 kmeans.py basic -k 2
```

This will run your code on simple 2-dimensional dataset. You should get a reconstruction error of roughly 6698.

(b) (4 points) Now try $k = 3, \ldots, 10$. Make a plot of the reconstruction error for each value of k. Based on the elbow criterion, which value(s) of k seem to be the best? Explain your reasoning.

(c) (3 points) The code automatically generates plots of your clusters and the cluster centroids in files called `clusters_k*.png`. Paste the images for $k = 2$, $k = 10$, and the best k chosen in the previous part. Describe all three plots, and describe what makes the k you chose better than both $k = 2$ and $k = 10$.

(d) (3 points) Now we will use k-means to "compress" an image. The original image is at `original.png`. Our compression strategy will be to view the color of each pixel as a 3-dimensional vector of (red, green, blue) intensity values. We can then cluster this set of 3-dimensional vectors to find k colors that are representative of the overall color scheme of the image (i.e., the cluster centroids). Then, we will replace each pixel's color with the color of the cluster centroid it was assigned to. This allows us to simply store the cluster assignment for each pixel, rather than the color of each pixel.

Run the following command:

```
python3 kmeans.py image -k 2
```

Try it with $k = 2, 4, 8, 16$ and show the resulting images (written to `compressed_k*.png`). Describe the resulting images and how the choice of k affects the appearance of the compressed image. (Note: This should only take a few seconds to run. If it is taking much longer, you may need to speed up your k-means implementation.)

(e) (4 points) Assume each pixel of an image is normally stored as three 8-bit integers, for the red, green, and blue channels. When you compress the image by running k-Means, by what factor does the number of bits needed to store the image go down? You may assume that k is a power of two, and ignore the fact that to store the image, you would also have to store the mapping from each cluster ID to the corresponding color. Finally, use the formula you derived to compute the *factor* of memory saved for $k = 2, 4, 8, 16$.

Question 3: Clustering Binary Vectors with EM (25 points)

In class, we saw how to use the Expectation-Maximization (EM) algorithm for Gaussian Mixture Models (GMMs). In this problem, we will apply EM to a different clustering problem, in which our data consists not of real-valued vectors but of binary vectors.

Suppose we have a dataset $D = \{x^{(1)}, \ldots, x^{(n)}\}$ where each $x^{(i)} \in \{0,1\}^d$. That is, each $x^{(i)}$ is a d-dimensional vector where every entry is either 0 or 1. For instance, maybe each example represents a genome, d is the number of SNPs (genomic variants) we have measured, and the j-th component of $x^{(i)}$ is 1 if the i-th person has a substitution in the j-th position relative to the reference genome, and 0 if they have the same DNA base as the reference genome.

As we did with GMMs, we will posit a latent variable probabilistic model for this data. Let X_i be the random variable representing the i-th example. We assume that the X_i's are independently and identically distributed (iid). Each X_i is generated by first sampling a cluster ID $Z_i \in \{1, \ldots, k\}$ from some distribution π. π is represented by a k-dimensional vector of probabilities; π_c is the probability that $Z_i = c$.

Now, we make a Naive Bayes-like assumption. Conditioned on Z_i, the probability of X_i is given as

$$P(X_i = x^{(i)} \mid Z_i = c) = \prod_{j=1}^{d} P(X_{ij} = x_j^{(i)} \mid Z_i = c).$$

This says that each component of X_i is sampled independently at random from a distribution governed by the cluster ID c. Thus, the second set of parameters is a matrix $W \in \mathbb{R}^{k \times d}$, where each W_{cj} is the probability that the j-th component of X_i is 1 conditioned on $Z_i = c$:

$$P(X_{ij} = 1 \mid Z_i = c) = W_{cj}.$$

Now, we will derive an EM algorithm to both infer the latent cluster assignments and learn the parameters of this probabilistic model.

(a) (7 points) **E step**. In the E step, we use our current guesses of the parameters π and τ to infer the posterior distribution over the Z_i's. In particular, for each i, and each cluster c, we want to compute

$$r_{ic} = P(Z_i = c \mid X_i = x^{(i)}; \pi, W).$$

Derive the formula in terms of π and W to compute this quantity. Show all of your steps. (Hint: It may be helpful to use the notational trick that for a random variable Y where $P(Y = 1) = p$ and $P(Y = 0) = 1 - p$, the expression $p^y(1-p)^{1-y}$ always evaluates to $P(Y = y)$ both when $y = 0$ and $y = 1$.)

(b) (5 points) **M step: Expected Complete Log-likelihood**. In the M step, we take our current guesses for the distribution of the Z_i's and use those to update our guesses of the parameters π and W. Recall that we do this by maximizing the expected complete log-likelihood (ECLL), a quantity that is similar to the log-likelihood but that takes an expectation with respect to the distribution of Z_i's computed in the E step. Write the expected complete log-likelihood of the parameters π and W in terms of π, W, the data $\{x^{(1)}, \ldots, x^{(n)}\}$ and the r_{ic} values computed in the E step.

(c) (10 points) **M step for W**. Now, let's derive the M step update for W. We'll do this by deriving the general rule to update W_{cj} for any choice of c and j. You should take the partial derivative of the ECLL with respect to W_{cj} and set this equal to 0. Then, solve for W_{cj} to derive the update rule for W_{cj}. Show your work. You should get an expression for W_{cj} in terms of the r_{ic}'s computed in the E-step and the data $x^{(i)}$. Finally, explain in a couple sentences why the formula you derived makes sense intuitively (Hint: think about what the r_{ic} values represent).

(d) (3 points) **M step for** π. Finally, write down the M step update for π. Again, explain in a couple sentences why this formula makes sense intuitively. (Note: This will be very similar to the GMM update rule from lecture. You do not need to provide a derivation, just give the update rule.)