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

HW 07

Due: Tuesday, November 19, 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, "HW7 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, "HW7 Code".
3. If there is a test set, submit your test set evaluation results, "HW7 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 three computational layers, where the last layer has a single scalar output.** The first layer has a single external input x. The computational layers should have widths of 5, 3, 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 the purpose of labeling. Also draw and label the loss function that will be important during training — 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`. Remember to attach your function implementations in the writeup.

(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. **Remember to attach your function implementations in the writeup.**

(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

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

Remember to attach your function implementations in the writeup.

(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. (Just for reference, staff's solution is able to achieve a error below 0.1 in general.) **Remember to attach your function implementations in the writeup.**

(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 Regularized and Kernel k-Means

Learning Goal: Combine previous ideas such as regularization and kernels to produce commonly-used variants of K-means

Recall that in k-means clustering we attempt to minimize the objective

$$\min_{C_1,C_2,\ldots,C_k} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2, \text{ where}$$

$$\mu_i = \text{argmin}_{\mu_i \in \mathbb{R}^d} \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j, \quad i = 1, 2, \ldots, k.$$

The samples are $\{x_1, \ldots, x_n\}$, where $x_j \in \mathbb{R}^d$. C_i is the set of sample points assigned to cluster i and $|C_i|$ is its cardinality. Each sample point is assigned to exactly one cluster.

(a) What is the minimum value of the objective when $k = n$ (the number of clusters equals the number of sample points)?

(b) (Regularized k-means) Suppose we add a regularization term to the above objective. The objective is now

$$\sum_{i=1}^{k} \left(\lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2 \right).$$

Show that the optimum of

$$\min_{\mu_i \in \mathbb{R}^d} \lambda \|\mu_i\|_2^2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2^2$$

is obtained at $\mu_i = \frac{1}{|C_i|+\lambda} \sum_{x_j \in C_i} x_j$.

(c) Here is an example where we would want to regularize clusters. Suppose there are n students who live in a $\mathbb{R}^2$ Euclidean world and who wish to share rides efficiently to Berkeley for their CS189 final exam. The university permits k vehicles for shuttling students to the exam location. The students need to figure out k good locations to meet up. The students will then walk to the closest meet up point and then the shuttles will deliver them to the exam location. Let x_j be the location of student j, and let the exam location be at $(0, 0)$. Assume that we can drive as the crow flies, i.e., by taking the shortest path between two points. Write down an appropriate objective function to minimize the total distance that the students and vehicles need to travel. *Hint: your result should be similar to the regularized k-means objective.*

(d) (Kernel k-means) Suppose we have a dataset $\{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^\ell$ that we want to split into K clusters, i.e., finding the best K-means clustering *without regularization*. Furthermore, suppose we know *a priori* that this data is best clustered in an impractically high-dimensional feature space $\mathbb{R}^m$, $l \ll m$ with an appropriate metric. Fortunately, instead of having to deal with the (implicit) feature map $\Phi : \mathbb{R}^\ell \to \mathbb{R}^m$ and (implicit) distance metric[1], we have a kernel function $\kappa(x_1, x_2) = \Phi(x_1) \cdot \Phi(x_2)$ that we can compute easily on the raw samples. How should we perform the kernelized counterpart of k-means clustering?

[1]Just as how the interpretation of kernels in kernelized ridge regression involves an implicit prior/regularizer as well as an implicit feature space, we can think of kernels as generally inducing an implicit distance metric as well. Think of how you would represent the squared distance between two points in terms of pairwise inner products and operations on them.

Derive the underlined portion of this algorithm, and show your work in deriving it. The main issue is that although we define the means μ_i in the usual way, we can't ever compute Φ explicitly because it's way too big. Therefore, in the step where we determine which cluster each sample point is assigned to, we must use the kernel function κ to obtain the right result. (Review the lecture on kernels if you don't remember how that's done.)

Algorithm 1: Kernel k-means

Require: Data matrix $X \in \mathbb{R}^{n \times l}$; Number of clusters K; kernel function $\kappa(x_1, x_2)$
Ensure: Cluster class class(j) for each sample x_j.
function KERNEL-K-MEANS(X, c)
  Randomly initialize class(j) to be an integer in $1, 2, \ldots, K$ for each x_j.
  while *not converged* **do**
    for $i \leftarrow 1$ ***to*** K **do**
      Set $S_i = \{j \in \{1, 2, \ldots, n\} : \text{class}(j) = i\}$.
    for $j \leftarrow 1$ ***to*** n **do**
      Set class(j) = argmin_k _____
  Return S_i for $i = 1, 2, \ldots, c$.
end function

(e) The expression you derived may have unnecessary terms or redundant kernel computations. Explain how to eliminate them; that is, how to perform the computation quickly without doing irrelevant computations or redoing computations already done.

3 Expectation Maximization (EM) Algorithm: In Action!

Learning Goal: Compare and contrast the behavior of various clustering algorithms on a mixture of Gaussians.

Suppose we have the following general mixture of Gaussians. We describe the model by a pair of random variables $(\mathbf{X}, Z)$ where $\mathbf{X}$ takes values in $\mathbb{R}^d$ and Z takes value in the set $[K] = \{1, \ldots, K\}$. The joint-distribution of the pair $(\mathbf{X}, Z)$ is given to us as follows:

$$Z \sim \text{Multinomial}(\boldsymbol{\pi}),$$
$$(\mathbf{X}|Z = k) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), \quad k \in [K],$$

where $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)^\top$ and $\sum_{k=1}^K \pi_k = 1$. Note that we can also write

$$\mathbf{X} \sim \sum_{k=1}^K \pi_k \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k).$$

Suppose we are given a dataset $\{\mathbf{x}_i\}_{i=1}^n$ without their labels. Our goal is to identify the K-clusters of the data. To do this, we are going to estimate the parameters $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$ using this dataset. We are going to use the following three algorithms for this clustering task.

K-Means: For each data-point for iteration t we find its cluster by computing:

$$y_i^{(t)} = \arg\min_{j \in [K]} \|\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)}\|^2$$

$$C_j^{(t)} = \{\mathbf{x}_i : y_i^{(t)} == j\}_{i=1}^n$$

where $\boldsymbol{\mu}_j^{(t-1)}$ denotes the mean of $C_j^{(t-1)}$, the j-th cluster in iteration $t-1$. The cluster means are then recomputed as:

$$\boldsymbol{\mu}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} \mathbf{x}_i.$$

We can run K-means till convergence (that is we stop when the cluster memberships do not change anymore). Let us denote the final iteration as T, then the estimate of the covariances Σ_k from the final clusters can be computed as:

$$\boldsymbol{\Sigma}_j = \frac{1}{|C_j^{(T)}|} \sum_{\mathbf{x}_i \in C_j^{(T)}} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(T)})(\mathbf{x}_i - \boldsymbol{\mu}_j^{(T)})^\top.$$

Notice that this method can be viewed as a "hard" version of EM.

K-Means-with-Covariance-Updates: Given that we also estimate the covariance, we may consider a version of K-means where the covariances keep getting updated at every iteration and also play a role in determining cluster membership. The objective at the assignment-step would be given by

$$y_i^{(t)} = \arg\min_{j \in [K]} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)})^\top (\boldsymbol{\Sigma}_j^{(t-1)})^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t-1)}).$$
$$C_j^{(t)} = \{x_i : y_i^{(t)} == j\}_{i=1}^n$$

We could then use $C_j^{(t)}$ to recompute the parameters as follows:

$$\boldsymbol{\mu}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} \mathbf{x}_i, \quad \text{and}$$
$$\boldsymbol{\Sigma}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x}_i \in C_j^{(t)}} (\mathbf{x}_i - \boldsymbol{\mu}_j^{(t)})(\mathbf{x}_i - \boldsymbol{\mu}_j^{(t)})^\top.$$

We again run K-means-with-covar until convergence (that is we stop when the cluster memberships do not change anymore). Notice that, again, this method can be viewed as another variant for the "hard" EM method.

EM: The EM updates are given by

- E-step: For $k = 1, \ldots, K$ and $i = 1 \ldots, n$, we have

$$q_i^{(t)}(Z_i = k) = p(Z = k | \mathbf{X} = \mathbf{x}_i; \boldsymbol{\theta}^{(t-1)}).$$

- M-step: For $k = 1, \ldots, K$, we have

$$\pi_k^{(t)} = \frac{1}{n} \sum_{i=1}^n q_i^{(t)}(Z_i = k) = \frac{1}{n} \sum_{i=1}^n p(Z = k | \mathbf{X} = \mathbf{x}_i; \boldsymbol{\theta}^{(t-1)}),$$

$$\boldsymbol{\mu}_k^{(t)} = \frac{\sum_{i=1}^n q_i^{(t)}(Z_i = k)\mathbf{x}_i}{\sum_{i=1}^n q_i^{(t)}(Z_i = k)}, \quad \text{and}$$
$$\boldsymbol{\Sigma}_k{}^{(t)} = \frac{\sum_{i=1}^n q_i^{(t)}(Z_i = k)(\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})(\mathbf{x}_i - \boldsymbol{\mu}_k^{(t)})^\top}{\sum_{i=1}^n q_i^{(t)}(Z_i = k)}.$$

Notice that unlike previous two methods, in the EM updates, each data point contributes in determining the mean and covariance for each cluster.

We now see the three methods in action. You are provided with a code for all the above 3 algorithms (`gmm_em_kmean.py`). You can run it by calling the following function from main:

```
experiments(seed, factor, num_samples, num_clusters)
```

We assume that $\mathbf{x} \in \mathbb{R}^2$, and the default settings are number of samples is 500 ($n = 500$), and the number of clusters is 3 ($K = 3$). Notice that `seed` will determine the randomness and `factor` will determine how far apart are the clusters.

(a) Run the following setting:

```
experiments(seed=11, factor=1, num_samples=500, num_clusters=3)
```

Observe the initial guesses for the means and the plots for the 3 algorithms on convergence. **Comment on your observations. Attach the two plots for this case.**

Note that the colors are used to indicate that the points that belong to different clusters, to help you visualize the data and understand the results.

(b) **Comment on the results obtained for the following setting:**

```
experiments(seed=63, factor=10, num_samples=500, num_clusters=3)
```

and attach the two plots for this case as well.

4 One Dimensional Mixture of Two Gaussians

Learning Goal: Derive the EM algorithm for a simple mixture of Gaussians.

Suppose we have a mixtures of two Gaussians in $\mathbb{R}$ that can be described by a pair of random variables (X, Z) where X takes values in $\mathbb{R}$ and Z takes value in the set $1, 2$. The joint-distribution of the pair (X, Z) is given to us as follows:

$$Z \sim \text{Bernoulli}(\beta),$$
$$(X|Z = k) \sim \mathcal{N}(\mu_k, \sigma_k), \quad k \in 1, 2,$$

We use $\boldsymbol{\theta}$ to denote the set of all parameters $\beta, \mu_1, \sigma_1, \mu_2, \sigma_2$.

(a) Write down the expression for the joint likelihood $p_{\boldsymbol{\theta}}(X = x_i, Z_i = 1)$ and $p_{\boldsymbol{\theta}}(X = x_i, Z_i = 2)$. What is the marginal likelihood $p_{\boldsymbol{\theta}}(X = x_i)$?

(b) What is the log-likelihood $\ell_{\boldsymbol{\theta}}(\mathbf{x})$? Why can't we optimize this by taking the derivative of the log-likelihood and setting it to 0?

(c) You'd like to optimize the log likelihood: $\ell_{\boldsymbol{\theta}}(x)$. However, we just saw this can be hard to solve for an MLE closed form solution. Show that a lower bound for the log likelihood is $\ell_{\boldsymbol{\theta}}(x_i) \geq \mathbb{E}_q\left[\log\left(\frac{p_{\boldsymbol{\theta}}(X=x_i, Z_i=k)}{q_{\boldsymbol{\theta}}(Z_i=k|X=x_i)}\right)\right]$.

(d) The EM algorithm initially starts with two randomly placed Gaussians (μ_1, σ_1) and (μ_2, σ_2), which are both particular realizations of $\boldsymbol{\theta}$.

- E-step: $\mathbf{q}_{i,k}^{t+1} = p_{\theta^t}(Z_i = k|X = x_i)$. For each data point, determine which Gaussian generated it, being either (μ_1, σ_1) or (μ_2, σ_2).
- M-step: : $\boldsymbol{\theta}^{t+1} = \text{argmax}_\theta \sum_{i=1}^n \mathbb{E}_{q^{t+1}}\left[\log(p_{\boldsymbol{\theta}}(X = x_i, Z_i = k))\right]$. After labeling all datapoints in the E-step, adjust (μ_1, σ_1) and (μ_2, σ_2).

Prove that alternating between the E-step and the M-step maximizes the lower bound. *Hint: Express the lower bound in two ways such that for the E-step we have the lower bound written in terms of θ^t and q and for the M-step we have the lower bound written in terms of q^{t+1} and θ. Then, isolate the variables being maximized (q for the E-step and θ for the M-step).*

(e) E-step: What are expressions for probabilistically imputing (replacing missing data Z with an estimated value based on observed data X) the classes for all the datapoints, i.e. $q_{i,1}^{t+1}$ and $q_{i,2}^{t+1}$?

(f) What is the expression for μ_1^{t+1} for the M-step?

(g) Compare and contrast k-means, soft k-means, and mixture of Gaussians fit with EM. Some questions to think about (you don't need to answer each exactly) include: What are assumptions do we make about our solutions? What types of data might these models have trouble with? How are they related? When might you choose one over another?