CP8318/CPS803: Machine Learning (Fall 2021)
Assignment 3
Due: Wednesday December 1, 2021 (midnight 11:59PM)
Instructor: Nariman Farsad

Please Read Carefully:

- Submit your assignments on D2L before the deadline. The submission will be closed after the deadline.
- Show your work and write your solutions legibly. You can use applications such as Word or Latex to type up your solutions for the written portion of the assignment or use applications such as CamScanner to create a PDF file from your handwritten solutions. For coding problems you need to submit your **python source files in the format given**, and have the main plots and discussion in the PDF written portion of the solution.
- Do the assignment on your your own, i.e., do not copy. If two assignments are found to be copies of each other, they BOTH receive 0.
- These questions require thought, but do not require long answers. Please be as concise as possible.
- If you have a question about this homework, we encourage you to post your question on our D2L discussion board.
- Students in CP8318 are required to answer all questions in the assignment. CPS803 students are not required to answer the questions marked for CP8318 students. These questions will not be graded for CPS803 students and they can just attempt them for practice.
- For instructions on setting up the python environment for the assignments read the "README.md" file.
- For the coding problems, you may not use any libraries except those defined in the provided "environment.yml" file. In particular, ML-specific libraries such as scikit-learn are not permitted.

1. (50 points) **Neural Networks: MNIST image classification**

In this problem, you will implement a simple convolutional neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size with only a single channel. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image.

The following shows some example images from the MNIST dataset: [1]

The data and starter code for this problem can be found in

- `src/mnist/nn.py`
- `src/mnist/images_train.csv.gz`
- `src/mnist/labels_train.csv.gz`
- `src/mnist/images_test.csv.gz`
- `src/mnist/labels_test.csv.gz`

The starter code splits the set of 60,000 training images and labels into a sets of 50,000 examples as the training set and 10,000 examples for dev set.

To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. Use the sigmoid function as activation for the hidden layer, and softmax function for the output layer. Recall that for a single example (x, y), the cross entropy loss is:

$$CE(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y_k},$$

where $\hat{y} \in \mathbb{R}^K$ is the vector of softmax outputs from the model for the training example x, and $y \in \mathbb{R}^K$ is the ground-truth vector for the training example x such that $y = [0, ..., 0, 1, 0, ..., 0]^\top$ contains a single 1 at the position of the correct class (also called a "one-hot" representation).

For n training examples, we average the cross entropy loss over the n examples.

[1]https://commons.wikimedia.org/wiki/File:MnistExamples.png

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n}\sum_{i=1}^{n} CE(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y_k}^{(i)}.$$

The starter code already converts labels into one hot representations for you.

Instead of batch gradient descent or stochastic gradient descent, the common practice is to use mini-batch gradient descent for deep learning tasks. In this case, the cost function is defined as follows:

$$J_{MB} = \frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})$$

where B is the batch size, i.e. the number of training example in each mini-batch.

1. [20 points] **Unregularized Model**

   Implement both forward-propagation and back-propagation for the above loss function. Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set $B = 1{,}000$ (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch, we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially. Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the loss function averaged over the dev set, and plot it against the number of epochs. Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the dev set versus number of epochs.

   Submit the two plots (one for loss vs epoch, another for accuracy vs epoch) in your writeup.

   Also, at the end of 30 epochs, save the learnt parameters (i.e all the weights and biases) into a file, so that next time you can directly initialize the parameters with these values from the file, rather than re-training all over. You do NOT need to submit these parameters.

   Hint: Be sure to vectorize your code as much as possible! Training can be very slow otherwise.

2. [20 points] **Regularized Model**

   Now add a regularization term to your cross entropy loss. The loss function will become:

$$J_{MB} = \left(\frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})\right) + \lambda\left(\left\|W^{[1]}\right\|_2^2 + \left\|W^{[2]}\right\|_2^2\right)$$

   Be careful not to regularize the bias/intercept term. Set λ to be 0.0001. Implement the regularized version and plot the same figures as part 1.1. Be careful NOT to include the regularization term to measure the loss value for plotting (i.e., regularization should only be used for gradient calculation for the purpose of training).

Submit the two new plots obtained with regularized training (i.e loss (without regularization term) vs epoch, and accuracy vs epoch) in your writeup. Compare the plots obtained from the regularized model with the plots obtained from the non-regularized model, and summarize your observations in a couple of sentences.

As in the previous part, save the learnt parameters (weights and biases) into a different file so that we can initialize from them next time.

3. [10 points] **Final Test**

   All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e, the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it (whatever value it may be), and NOT go back and refine the model any further.

   Initialize your model from the parameters saved in part (a) (i.e, the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e, the regularized model).

   Report your test accuracy for both regularized model and non-regularized model. You should have accuracy close to 0.9318 without regularization, and with 0.9670 regularization. Briefly (in one sentence) explain why this outcome makes sense.

2. (10 points) **CP8318 Only Question: A Simple Neural Network**

Let $X = \{x^{(1)}, \cdots, x^{(m)}\}$ be a dataset of m samples with 2 features, i.e $x^{(i)} \in \mathbb{R}^2$. The samples are classified into 2 categories with labels $y^{(i)} \in \{0, 1\}$. A scatter plot of the dataset is shown in Figure 1:

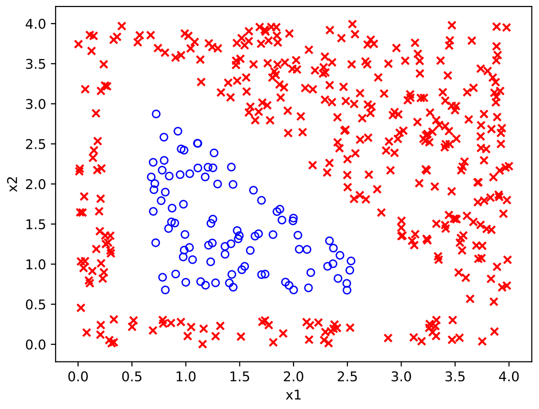

Figure 1: Plot of dataset X.

The examples in class 1 are marked as as "$\times$" and examples in class 0 are marked as "$\circ$". We want to perform binary classification using a simple neural network with the architecture shown in Figure 2:

Denote the two features x_1 and x_2, the three neurons in the hidden layer h_1, h_2, and h_3, and the output neuron as o. Let the weight from x_i to h_j be $w_{i,j}^{[1]}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w_j^{[2]}$. Finally, denote the intercept weight for h_j as $w_{0,j}^{[1]}$, and the intercept weight for o as $w_0^{[2]}$.

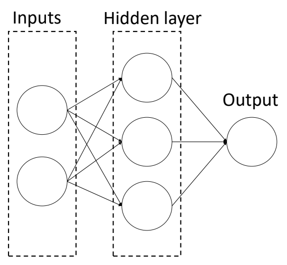

Figure 2: Architecture for our simple neural network.

For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

$$l = \frac{1}{m}\sum_{i=1}^{m}(o^{(i)} - y^{(i)})^2,$$

where $o^{(i)}$ is the result of the output neuron for example i.

1. [5 points] **CP8318 Only Question:** Suppose we use the sigmoid function as the activation function for h_1, h_2, h_3 and o. What is the gradient descent update to $w_{1,2}^{[1]}$, assuming we use a learning rate of α? Your answer should be written in terms of $x^{(i)}$, $o^{(i)}$, $y^{(i)}$, and the weights.

2. [5 points] **CP8318 Only Question:** Let the activation functions for h_1, h_2, h_3 be the linear function $f(x) = x$ and the activation function for o be the same sigmoid function as before.

   Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

   If it is possible, please provide a set of weights that enable 100% accuracy by completing `optimal_linear_weights` within `src/p01_nn.py` and explain your reasoning for those weights in your PDF.

   If it is not possible, please explain your reasoning in your PDF. (There is no need to modify `optimal_linear_weights` if it is not possible.)

3. (0 points) **OPTIONAL PRACTICE QUESTION**

   KL divergence and Maximum Likelihood

   This Question is an optional practice question and will not be graded. The Kullback-Leibler (KL) divergence is a measure of how much one probability distribution is different from a second one. It is a concept that originated in Information Theory, but has made its way into several other fields, including Statistics, Machine Learning, Information Geometry, and many more. In Machine Learning, the KL divergence plays a crucial role, connecting various concepts that might otherwise seem unrelated.

   In this problem, we will introduce KL divergence over discrete distributions, practice some simple manipulations, and see its connection to Maximum Likelihood Estimation.

   The *KL divergence* between two discrete-valued distributions $P(X), Q(X)$ over the outcome space $\mathcal{X}$ is defined as follows[2]:

[2]If P and Q are densities for continuous-valued random variables, then the sum is replaced by an integral, and everything

$$D_{KL}(P\|Q) = \sum_{x\in\mathcal{X}} P(x)\log\frac{P(x)}{Q(x)}$$

For notational convenience, we assume $P(x) > 0, \forall x$. (One other standard thing to do is to adopt the convention that "$0 \log 0 = 0$.") Sometimes, we also write the KL divergence more explicitly as $D_{KL}(P||Q) = D_{KL}(P(X)||Q(X))$.

Background on Information Theory

Before we dive deeper, we give a brief (optional) Information Theoretic background on KL divergence. While this introduction is not necessary to answer the assignment question, it may help you better understand and appreciate why we study KL divergence, and how Information Theory can be relevant to Machine Learning.

We start with the *entropy* $H(P)$ of a probability distribution $P(X)$, which is defined as

$$H(P) = -\sum_{x\in\mathcal{X}} P(x)\log P(x).$$

Intuitively, entropy measures how dispersed a probability distribution is. For example, a uniform distribution is considered to have very high entropy (i.e. a lot of uncertainty), whereas a distribution that assigns all its mass on a single point is considered to have zero entropy (i.e. no uncertainty). Notably, it can be shown that among continuous distributions over $\mathbb{R}$, the Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$ has the highest entropy (highest uncertainty) among all possible distributions that have the given mean μ and variance σ^2.

To further solidify our intuition, we present motivation from communication theory. Suppose we want to communicate from a source to a destination, and our messages are always (a sequence of) discrete symbols over space $\mathcal{X}$ (for example, $\mathcal{X}$ could be letters $\{\text{a}, \text{b}, \dots, \text{z}\}$). We want to construct an encoding scheme for our symbols in the form of sequences of binary bits that are transmitted over the channel. Further, suppose that in the long run the frequency of occurrence of symbols follow a probability distribution $P(X)$. This means, in the long run, the fraction of times the symbol x gets transmitted is $P(x)$.

A common desire is to construct an encoding scheme such that the average number of bits per symbol transmitted remains as small as possible. Intuitively, this means we want very frequent symbols to be assigned to a bit pattern having a small number of bits. Likewise, because we are interested in reducing the average number of bits per symbol in the long term, it is tolerable for infrequent words to be assigned to bit patterns having a large number of bits, since their low frequency has little effect on the long term average. The encoding scheme can be as complex as we desire, for example, a single bit could possibly represent a long sequence of multiple symbols (if that specific pattern of symbols is very common). The entropy of a probability distribution $P(X)$ is its optimal bit rate, i.e., the lowest average bits per message that can possibly be achieved if the symbols $x \in \mathcal{X}$ occur according to $P(X)$. It does not specifically tell us *how* to construct that optimal encoding scheme. It only tells us that no encoding can possibly give us a lower long term bits per message than $H(P)$.

stated in this problem works fine as well. But for the sake of simplicity, in this problem we'll just work with this form of KL divergence for probability mass functions/discrete-valued distributions.

To see a concrete example, suppose our messages have a vocabulary of $K = 32$ symbols, and each symbol has an equal probability of transmission in the long term (i.e, uniform probability distribution). An encoding scheme that would work well for this scenario would be to have $\log_2 K$ bits per symbol, and assign each symbol some unique combination of the $\log_2 K$ bits. In fact, it turns out that this is the most efficient encoding one can come up with for the uniform distribution scenario.

It may have occurred to you by now that the long term average number of bits per message depends only on the frequency of occurrence of symbols. The encoding scheme of scenario A can in theory be reused in scenario B with a different set of symbols (assume equal vocabulary size for simplicity), with the same long term efficiency, as long as the symbols of scenario B follow the same probability distribution as the symbols of scenario A. It might also have occured to you, that reusing the encoding scheme designed to be optimal for scenario A, for messages in scenario B having a *different probability* of symbols, will always be suboptimal for scenario B. To be clear, we do not need know *what* the specific optimal schemes are in either scenarios. As long as we know the distributions of their symbols, we can say that the optimal scheme designed for scenario A will be suboptimal for scenario B if the distributions are different.

Concretely, if we reuse the optimal scheme designed for a scenario having symbol distribution $Q(X)$, into a scenario that has symbol distribution $P(X)$, the long term average number of bits per symbol achieved is called the *cross entropy*, denoted by $H(P, Q)$:

$$H(P, Q) = -\sum_{x \in \mathcal{X}} P(x) \log Q(x).$$

To recap, the entropy $H(P)$ is the best possible long term average bits per message (optimal) that can be achived under a symbol distribution $P(X)$ by using an encoding scheme (possibly unknown) specifically designed for $P(X)$. The cross entropy $H(P, Q)$ is the long term average bits per message (suboptimal) that results under a symbol distribution $P(X)$, by reusing an encoding scheme (possibly unknown) designed to be optimal for a scenario with symbol distribution $Q(X)$.

Now, KL divergence is the penalty we pay, as measured in average number of bits, for using the optimal scheme for $Q(X)$, under the scenario where symbols are actually distributed as $P(X)$. It is straightforward to see this

$$\begin{aligned} D_{KL}(P, Q) &= \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)} \\ &= \sum_{x \in \mathcal{X}} P(x) \log P(x) - \sum_{x \in \mathcal{X}} P(x) \log Q(x) \\ &= H(P, Q) - H(P). \quad \text{(difference in average number of bits.)} \end{aligned}$$

If the cross entropy between P and Q is zero (and hence $D_{KL}(P||Q) = 0$) then it necessarily means $P = Q$. In Machine Learning, it is a common task to find a distribution Q that is "close" to another distribution P. To achieve this, we use $D_{KL}(Q||P)$ to be the loss function to be optimized. As we will see in this question below, Maximum Likelihood Estimation, which is a commonly used optimization objective, turns out to be equivalent minimizing KL divergence between the training data (i.e. the empirical distribution over the data) and the model.

Now, we get back to showing some simple properties of KL divergence.

1. [0 points] **Nonnegativity.** Prove the following:

$$\forall P, Q \quad D_{KL}(P \| Q) \geq 0$$

and

$$D_{KL}(P \| Q) = 0 \quad \text{if and only if } P = Q.$$

[Hint: You may use the following result, called **Jensen's inequality**. If f is a convex function, and X is a random variable, then $E[f(X)] \geq f(E[X])$. Moreover, if f is strictly convex (f is convex if its Hessian satisfies $H \geq 0$; it is *strictly* convex if $H > 0$; for instance $f(x) = -\log x$ is strictly convex), then $E[f(X)] = f(E[X])$ implies that $X = E[X]$ with probability 1; i.e., X is actually a constant.]

2. [0 points] **Chain rule for KL divergence.** The KL divergence between 2 conditional distributions $P(X|Y), Q(X|Y)$ is defined as follows:

$$D_{KL}(P(X|Y) \| Q(X|Y)) = \sum_y P(y) \left(\sum_x P(x|y) \log \frac{P(x|y)}{Q(x|y)} \right)$$

This can be thought of as the expected KL divergence between the corresponding conditional distributions on x (that is, between $P(X|Y=y)$ and $Q(X|Y=y)$), where the expectation is taken over the random y.

Prove the following chain rule for KL divergence:

$$D_{KL}(P(X,Y) \| Q(X,Y)) = D_{KL}(P(X) \| Q(X)) + D_{KL}(P(Y|X) \| Q(Y|X)).$$

3. [0 points] **KL and maximum likelihood.** Consider a density estimation problem, and suppose we are given a training set $\{x^{(i)}; i = 1, \ldots, m\}$. Let the empirical distribution be $\hat{P}(x) = \frac{1}{m}\sum_{i=1}^m 1\{x^{(i)} = x\}$. ($\hat{P}$ is just the uniform distribution over the training set; i.e., sampling from the empirical distribution is the same as picking a random example from the training set.)

Suppose we have some family of distributions P_θ parameterized by θ. (If you like, think of $P_\theta(x)$ as an alternative notation for $P(x;\theta)$.) Prove that finding the maximum likelihood estimate for the parameter θ is equivalent to finding P_θ with minimal KL divergence from $\hat{P}$. I.e. prove:

$$\arg\min_\theta D_{KL}(\hat{P} \| P_\theta) = \arg\max_\theta \sum_{i=1}^m \log P_\theta(x^{(i)})$$

Remark. Consider the relationship between parts (b-c) and multi-variate Bernoulli Naive Bayes parameter estimation. In the Naive Bayes model we assumed P_θ is of the following form: $P_\theta(x,y) = p(y)\prod_{i=1}^n p(x_i|y)$. By the chain rule for KL divergence, we therefore have:

$$D_{KL}(\hat{P} \| P_\theta) = D_{KL}(\hat{P}(y) \| p(y)) + \sum_{i=1}^n D_{KL}(\hat{P}(x_i|y) \| p(x_i|y)).$$

This shows that finding the maximum likelihood/minimum KL-divergence estimate of the parameters decomposes into $2n+1$ independent optimization problems: One for the class priors $p(y)$, and one for each of the conditional distributions $p(x_i|y)$ for each feature x_i given each of the two possible labels for y. Specifically, finding the maximum likelihood estimates for each of these problems individually results in also maximizing the likelihood of the joint distribution. (If you know what Bayesian networks are, a similar remark applies to parameter estimation for them.)

Congratulations! You made it to the end of the assignments of the machine learning class!