HOMEWORK SET 1

CSCI 5980 Think Deep Learning (Spring 2020)

Due 11:59 pm, Mar 12 2020

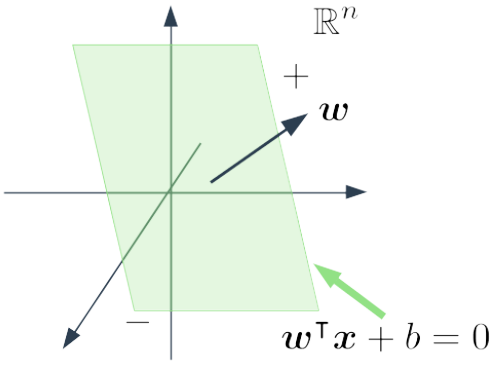
Instruction Please typeset your homework in LaTeX and submit it as a single PDF file in Canvas. No late submission will be accepted. For each problem, your should acknowledge your collaborators if any. For problems containing multiple subproblems, there are often close logic connections between the subproblems. So always remember to build on previous ones, rather than work from scratch.

Notation We will use small letters (e.g., u) for scalars, small boldface letters (e.g., $\boldsymbol{a}$) for vectors, and capital boldface letters (e.g., $\boldsymbol{A}$) for matrices. For a matrix $\boldsymbol{A}$, $\boldsymbol{a}^i$ (supscripting) means its i-th row as a *row vector*, and $\boldsymbol{a}_j$ (subscripting) means the j-the column as a column vector, and a_{ij} means its (i,j)-th element. $\mathbb{R}$ is the set of real numbers. $\mathbb{R}^n$ is the space of n-dimensional real vectors, and similarly $\mathbb{R}^{m\times n}$ is the space of $m \times n$ real matrices. The dotted equal sign $\doteq$ means defining.

Problem 1 (Neural networks can represent all Boolean functions) The standard perceptron is a single-layer neural network with the step function as the activation, i.e.,

$$f(\boldsymbol{x}) = \begin{cases} 1 & \boldsymbol{w}^\intercal \boldsymbol{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

Geometrically, f is a $\{0,1\}$-valued function with the hyperplane $\{\boldsymbol{x} : \boldsymbol{w}^\intercal \boldsymbol{x} + b = 0\}$ as the separating boundary between the 0- and the 1-region; see Fig. 1 (left).



X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

Figure 1: (left) Geometric illustration of the perceptron. (right) An example truth table.

Consider Boolean functions $\{0,1\}^n \to \{0,1\}$. We will work out how arbitrary Boolean functions can be represented by two-layer or deep neural networks.

(a) Consider $n = 1$ first. Show that the `NOT` function can be implemented using a single-input perceptron by setting the weight w and the bias b appropriately. (0.5/12)

(b) Consider the case $n = 2$ first. Show that the `AND`, `OR` functions can be implemented using a two-input perceptron. Hint: the geometric view might help. For example, for the `AND` function, we are effectively trying to separate the point $(1,1)$ from $(1,0)$, $(0,1)$ and $(0,0)$. The hint applies to all subsequent questions in this problem. (1/12)

(c) Can we encode the XOR function (https://en.wikipedia.org/wiki/Exclusive_or) using a two-input perceptron? How if yes? Why if not? (1/12)

(d) For general $n \geq 2$, we consider general AND functions that take n inputs. A typical such function looks like $x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_{n-1} \cdot x_n$, i.e., negation in input variable is allowed. Show that all general n-input AND function can be implemented using an n-input perceptron. (1/12)

(e) Similar to (d), show that all n-input general OR function can be implemented using an n-input perceptron. (1/12)

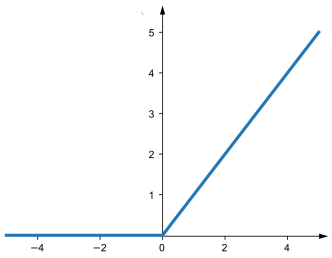
(f) Boolean functions are fully specified by a list of all variable combinations leading to output 1. This list is often tabulated and called the truth table. For example, in the truth table of Fig. 1 (right), the Boolean function represented reads

$$\overline{x_1}\,\overline{x_2}x_3x_4\overline{x_5} + \overline{x_1}x_2\overline{x_3}x_4x_5 + \overline{x_1}x_2x_3\overline{x_4}\,\overline{x_5} + x_1\overline{x_2}\,\overline{x_3}\,\overline{x_4}\,x_5 + x_1\overline{x_2}x_3x_4x_5 + x_1x_2\overline{x_3}\,\overline{x_4}x_5,$$

where product $\cdot$ (which is omitted) means AND and summation $+$ means OR. In Boolean logic, this is called the disjunctive normal form (https://en.wikipedia.org/wiki/Disjunctive_normal_form). All Boolean functions can be represented in the disjunctive normal form.

Based on these, show that all n-input Boolean functions can be represented by a two-layer neural network. (1/12) In the worst case, how many hidden nodes are needed? (0.5/12)

Problem 2 (Universal approximation property of ReLU networks) Recall how we argued that two-layer neural networks with the sigmoid activation function (i.e., $\sigma(z) = \frac{1}{1+e^{-z}}$) can approximate any functions that map $\mathbb{R}$ to $\mathbb{R}$. We constructed the step function and then the bump function, and finally we sum up the bumps to form the approximation.



Now let's work out a similar property for two-layer neural networks with the ReLU activation. The ReLU function is $\sigma(z) = \max(0, z)$, as shown in the right figure. We now follow the step-bump-sum roadmap to work out the argument.

(a) Show that we can use two ReLU's, probably shifted versions of the standard, to construct an arbitrary good approximation to any step function. Please draw the construction as a neural network. (1/12)

(b) Show that we can further construct "bump" functions out of the step functions. (0.5/12)

(c) Now we are ready to sum up shifted copies of bumps to approximate arbitrary functions. How may (weight) layers do we have in our current construction? If it is not two, can you show how to reduce it to two without sacrificing the approximation capacity? (1/12)

Problem 3 (Autoencoder, factorization, and PCA; gradient descent and back-tracking line search) Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ be a collection of points in $\mathbb{R}^n$ and suppose they are zero-centered, i.e., $\sum_{i=1}^m \boldsymbol{x}_i = \boldsymbol{0}$. We write $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m] \in \mathbb{R}^{n\times m}$, i.e., stacking the data points columnwise into a data matrix. Recall that PCA extracts the first r (with $r \leq n$) eigenvectors of $\boldsymbol{X}\boldsymbol{X}^\intercal$, collects them

columnwise into a matrix $\boldsymbol{U} \in \mathbb{R}^{n\times r}$, and obtains a new representation of each data point $\boldsymbol{x}_i$ as $\boldsymbol{U}^\intercal \boldsymbol{x}_i \in \mathbb{R}^r$. Geometrically, PCA amounts to deriving the best rank-r linear subspace approximation to the point set $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$:

$$\min_{\substack{\boldsymbol{U}\in\mathbb{R}^{n\times r}:\, \boldsymbol{U}^\intercal\boldsymbol{U}=\boldsymbol{I} \\ \boldsymbol{Z}\in\mathbb{R}^{r\times m}}} \|\boldsymbol{X} - \boldsymbol{U}\boldsymbol{Z}\|_F^2 .$$

So a crucial step in PCA is to compute the subspace basis. Let's now generate a synthetic point set as follows (please do NOT change the random seed and the dimensions),

```python
import numpy as np
import random

random.seed(59802020)

n = 50
m = 200
r = 10

# generate a random dataset
A = np.random.randn(n, r)
B = np.random.randn(r, m)
X = np.matmul(A, B) + 0.01*np.random.randn(n, m)

# zero centering
X -= np.mean(X, axis=1).reshape(n, 1)
```

and complete the following tasks. **Please submit your code for this problem as a separate .ipynb file, not to be combined with the codes for other problems. No auto-differentiation is allowed for solving this problem.**

(a) Continue the above Python code to compute the basis for the best rank-10 subspace approximation to $\boldsymbol{X}$, i.e., a matrix $\boldsymbol{A}_1 \in \mathbb{R}^{n\times 10}$ that contains the first 10 PCA basis vectors. (0.5/12)

(b) A classic unsupervised learning technique in deep learning is the autoencoder (we'll cover it later in the course). The mathematical formulation specialized to our case is

$$\min_{\boldsymbol{A}\in\mathbb{R}^{n\times 10}} \; f(\boldsymbol{A}) \doteq \|\boldsymbol{X} - \boldsymbol{A}\boldsymbol{A}^\intercal\boldsymbol{X}\|_F^2 .$$

(i) Derive the gradient of the objective, i.e., $\nabla f(\boldsymbol{A})$ (hint: it is much easier to use the Taylor expansion method rather than the chain rule method). (0.5/12)

(ii) Implement the gradient descent method to solve the optimization problem, with **backtracking line search** for the step size. (0.5/12)

(iii) Let's say the solution computed from the last step is $\boldsymbol{A}_2$. Now we want to compare the subspaces represented by $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$. We cannot directly do $\boldsymbol{A}_1 - \boldsymbol{A}_2$, as from linear algebra we know that $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ can have the same column/range space, but take very different forms. Instead, a reasonable metric here is the difference between the subspace projectors induced by them, i.e., $\left\|\boldsymbol{A}_1\boldsymbol{A}_1^\intercal - \boldsymbol{A}_2\boldsymbol{A}_2^\dagger\right\|_F$, where $\boldsymbol{A}^\dagger$ denotes the matrix pseudoinverse (https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse; in Numpy, you can call this function `numpy.linalg.pinv`). Report your result here. Is it close to 0 or not? (0.5/12)

(c) Consider another formulation, which is normally called **factorization**:

$$\min_{\boldsymbol{A}\in\mathbb{R}^{n\times 10},\boldsymbol{Z}\in\mathbb{R}^{10\times m}} \; g\left(\boldsymbol{A},\boldsymbol{Z}\right) \doteq \left\|\boldsymbol{X}-\boldsymbol{A}\boldsymbol{Z}\right\|_F^2 .$$

(i) Derive the gradient of the objective, i.e., $\nabla g\left(\boldsymbol{A},\boldsymbol{Z}\right)$ (hint: it is much easier to use the Taylor expansion method rather than the chain rule method). (0.5/12)

(ii) Implement the gradient descent method to solve the optimization problem, with **backtracking line search** for the step size. (0.5/12)

(iii) Let's say the solution computed from the last step is $\boldsymbol{A}_3$. Please compute the subspace differences between $\boldsymbol{A}_1$, $\boldsymbol{A}_2$ and $\boldsymbol{A}_3$ and report your results here. Are they close to 0 or not? (0.5/12)