Stat 6910, Section 002
Deep Learning: Theory and Applications
Spring 2019

Homework 2

Jared Hansen

A-number: A01439768

e-mail: jrdhansen@gmail.com

Due: 5:00 PM, Friday 02/01/2019

Homework II

STAT 6910/7810-002 - Spring semester 2019

Due: Friday, February 1, 2019 - 5:00 PM

Please put all relevant files & solutions into a single folder titled `<lastname and initials>_assignment2` and then zip that folder into a single zip file titled `<lastname and initials>_assignment2.zip`, e.g. for a student named Tom Marvolo Riddle, `riddletm_assignment2.zip`. Include a single PDF titled `<lastname and initals>_assignment2.pdf` and any Python scripts specified. Any requested plots should be sufficiently labeled for full points.

Unless otherwise stated, programming assignments should use built-in functions in Python, Tensorflow, and PyTorch. In general, you may use the `scipy` stack [**?**]; however, exercises are designed to emphasize the nuances of machine learning and deep learning algorithms - if a function exists that trivially solves an entire problem, please consult with the TA before using it.

Problem 1

Consider the setting that we had in problem 5 in Homework 1. In this problem, you will reuse the code you wrote in problem 5.2 to generate data from the same model. Fit a 9-degree polynomial model with L2 norm regularization to the cases with $\sigma = 0.05$ and $N \in \{15, 100\}$, and include your code in `prob1.py`. Vary the parameter λ, and choose three values of λ that result in the following scenarios: underfitting, overfitting, and an appropriate fit. Report the fitted weights and the MSE in each of these scenarios.
Hint: Linear regression with L2 norm regularization is also referred to as ridge regression. You may find the equation for this in the machine learning slides.
Include your answers and plots in a PDF titled `<lastname and initials>_assignment2.pdf`. Include all your code in `<lastname and initials>_assignment2.zip`.

****NOTE:** Although the prompt doesn't ask for plots, I felt that it was helpful to include them to visually illustrate what is happening in terms of model fitting. They're also useful for verifying that my numerical results match up with what we'd expect for the various values of λ and each data set.

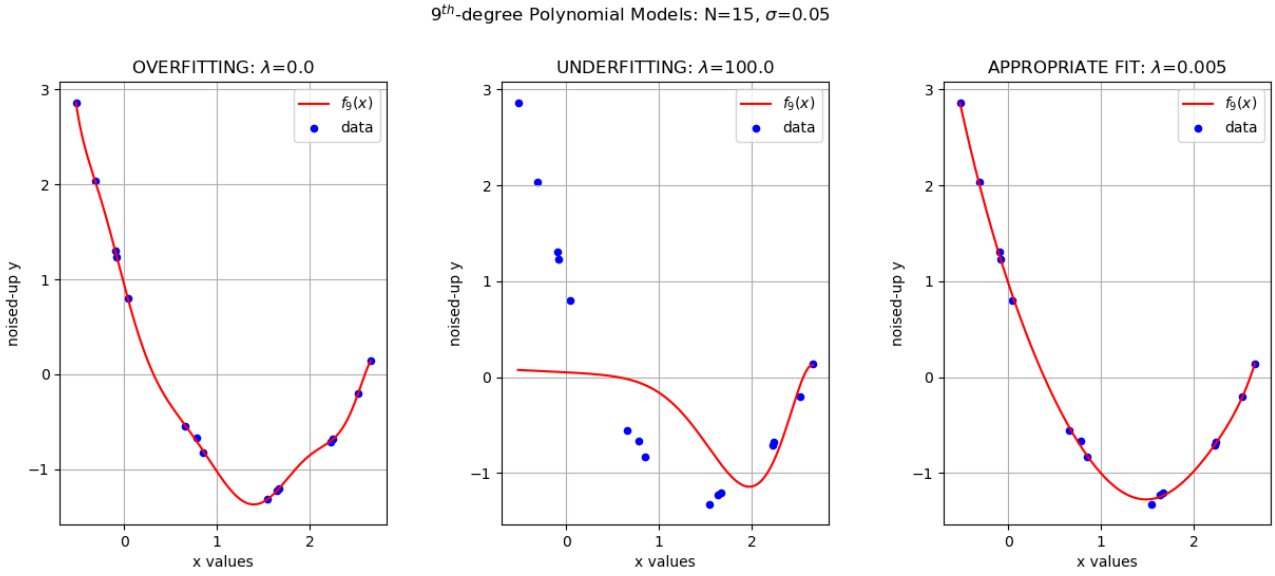

Figure 1: For the N = 15 and $\sigma = 0.05$ data, I landed on the λ parameters specified in the graphic above.

- Choosing $\lambda = 0.0$ is the same as fitting a typical 9^{th} degree polynomial function with no regularization, hence it leads to overfitting since we have relatively few data points with some added noise. Also, the data generating mechanism is only a 2^{nd} degree polynomial, so we'd expect overfitting using a higher order function with no regularization.

- Choosing $\lambda = 100$ leads to underfitting. In essence, we're putting too great a penalty on the sum of the squared weights (regularization term) by scaling it with such a large value. This pushes all of the weights toward 0, leading to an underfit curve.

- Choosing $\lambda = 0.005$ in this case gives an appropriate fit. The ideal situation is to have $\boldsymbol{w}^* = [w_1 \quad w_2 \quad w_3 \quad \dots \quad w_9 \quad w_0]^T = [-3 \quad 1 \quad 0 \quad \dots \quad 0 \quad 1]^T$ so I just tinkered around with the λ value until I got a weights vector very close to this, giving an appropriate fit (since the most appropriate fit is the data-generating function $f(x) = -3x + 1x^2 + 0x^3 + \ldots + 0x^9 + 1$).

- See the table below for the respective weights and MSE values for each model.

	N = 15, sigma = 0.05		
	Underfitting: λ = 100.0	**Appropriate Fit: λ = 0.005**	**Overfitting: λ = 0.0**
MSE	1.1278934726540402	0.00041692291994991477	0.0011010875405049812
Optimal Weights $\boldsymbol{w}^* = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_9 \\ w_0 \end{bmatrix}$	[-0.05135718] [-0.02219891] [-0.04111097] [-0.03931655] [-0.04379213] [-0.03248614] [-0.0051303] [0.03392408] [-0.00896058] [0.05077133]	[-2.92574921e+00] [1.10837264e+00] [-2.78989360e-01] [1.24049152e-01] [-6.30153893e-02] [8.04918901e-02] [-2.88580517e-02] [-2.26428163e-03] [1.63192205e-03] [9.85002863e-01]	[-3.66442977] [1.02450683] [5.32299861] [-3.93622573] [-9.3273313] [15.69909188] [-9.33704883] [2.5058537] [-0.25546781] [0.94653902]

Figure 2: Here are the weights and MSE values for the models fit on the N = 15 and $\sigma = 0.05$ data. As we would expect, the appropriately fit model has the lowest MSE, followed closely by the overfit model. Unsurprisingly, the underfit model has the highest MSE by far.
A couple of things to note:

- As shown on on the left side, these weight vectors are given in the form $\boldsymbol{w}^* = [w_1 \quad w_2 \quad w_3 \quad \dots \quad w_9 \quad w_0]^T$ where w_1 is the x^1 coefficient, w_2 is the x^2 coefficient, and so on through w_9, and w_0 is the offset term.
- Python gave some values in scientific notation. Rather than re-type these it was much easier to take screenshots and paste them.

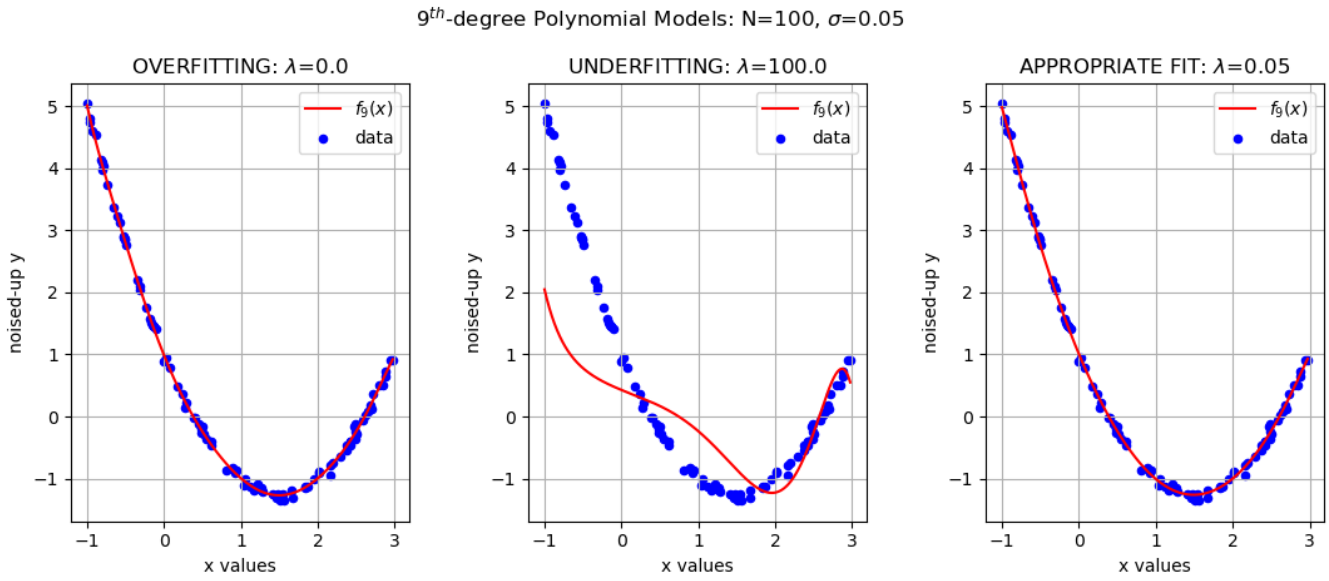

Figure 3: For the N = 100 and $\sigma = 0.05$ data, I landed on the λ parameters specified in the graphic above.

- Choosing $\lambda = 0.0$ is the same as fitting a typical 9^{th} degree polynomial function with no regularization. We would expect this to overfit the data (which it did for N = 15), but in this case it actually gives quite a good fit. I'd attribute this good fit to the fact that we have more data points, and their individual "noisiness" cancel each other out collectively. Therefore, we really can't overfit these data. If it were possible to overfit these data, $\lambda = 0.0$ would be the value to pick since it makes the regularization term 0 (and hence I chose it for my answer).

- Choosing $\lambda = 100$ leads to underfitting. In essence, we're putting too great a penalty on the sum of the squared weights (regularization term) by scaling it with such a large value. This pushes all of the weights toward 0, leading to an underfit curve.

- Choosing $\lambda = 0.05$ in this case gives an appropriate fit. The ideal situation is to have $\boldsymbol{w^*} = [w_1 \quad w_2 \quad w_3 \quad \dots \quad w_9 \quad w_0]^T = [-3 \quad 1 \quad 0 \quad \dots \quad 0 \quad 1]^T$ so I just tinkered around with the λ value until I got a weights vector very close to this, giving an appropriate fit (since the most appropriate fit is the data-generating function $f(x) = -3x + 1x^2 + 0x^3 + \ldots + 0x^9 + 1$).

- See the table below for the respective weights and MSE values for each model.

	N = 100, sigma = 0.05		
	Underfitting: λ = 100.0	**Appropriate Fit: λ = 0.05**	**Overfitting: λ = 0.0**
MSE	1.4231042599822248	0.00016520816763340048	0.0003429380044503575
Optimal Weights $\boldsymbol{w}^* = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_9 \\ w_0 \end{bmatrix}$	[-0.51117986] [0.14545615] [-0.34163943] [0.11069515] [-0.19114603] [0.18945467] [-0.0930455] [0.03199852] [-0.00496096] [0.42735977]	[-2.96631573e+00] [9.89028028e-01] [-1.18166516e-01] [5.42871381e-02] [7.75495702e-02] [-6.77427613e-02] [1.49989976e-02] [7.97678063e-04] [-4.42912316e-04] [1.00992235e+00]	[-3.06855009] [1.28063593] [0.06153413] [-0.58264499] [0.25857015] [0.24692825] [-0.24543721] [0.07537783] [-0.00799781] [0.9868962]

Figure 4: Here are the weights and MSE values for the models fit on the N = 100 and $\sigma = 0.05$ data. As we would expect, the appropriately fit model has the lowest MSE, followed closely by the overfit model. Unsurprisingly, the underfit model has the highest MSE by far.
A couple of things to note:

- As shown on on the left side, these weight vectors are given in the form $\boldsymbol{w}^* = [w_1 \quad w_2 \quad w_3 \quad \dots \quad w_9 \quad w_0]^T$ where w_1 is the x^1 coefficient, w_2 is the x^2 coefficient, and so on through w_9, and w_0 is the offset term.
- Python gave some values in scientific notation. Rather than re-type these it was much easier to take screenshots and paste them.

Problem 2

1. Load the dataset from file assignment2.zip

   NOTE:
   I was having some issues with the .dat file format for part 2 of this problem. As such, I used Excel to turn the .dat file into a .csv file for greater ease in reading in and manipulating the data. I've zipped the .csv file (data_seed.csv) with the rest of my assignment materials so that my code can for parts 2 and 3 of this problem can be run easily.
   I hope this is alright: I figure in an application scenario we often have to do what works. In my case, this was the most straightforward solution to my problems. Additionally, I feel that the focus is on implementing the classifiers and not on reading in a certain file format.

2. Write a program that applies a k-nn classifier to the data with $k \in \{1, 5, 10, 15\}$. Calculate the test error using both leave-one-out validation and 5-fold cross validation. Plot the test error as a function of k. You may use the existing methods in scikit-learn or other libraries for finding the k-nearest neighbors, but do not use any built- in k-nn classifiers. Also, do not use any existing libraries or methods for cross validation. Do any values of k result in underfitting or overfitting?

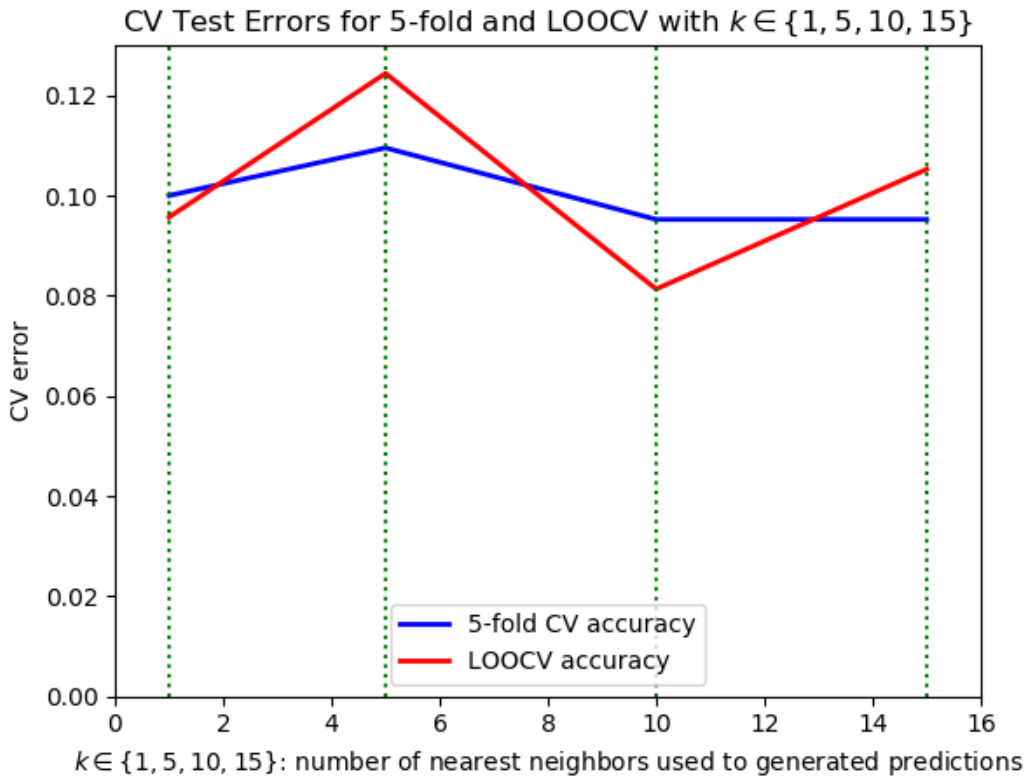

Figure 5: Based on this plot (and other iterations I ran with different random seeds), it looks like the "true" error rate of a k-nn classifier for this data is roughly 0.095. If we assume this to be true, k = 5 tends to underfit for LOOCV since it has a relatively higher error rate than 0.095. Also, k = 10 tends to overfit for LOOCV since it has a relatively lower error rate than 0.095. We see little perceptible overfitting/underfitting for 5-fold CV. Most notably, a value of k = 5 underfits just a bit.
See below for the CV test errors.

	Test Error Rates for k-nn Classification	
k value	**5-fold CV**	**LOOCV**
k = 1	[0.0999999999999998,	[0.0956937799043062 8,
k = 5	0.1095238095238094 7,	0.1244019138755980 7,
k = 10	0.0952380952380952 3,	0.0813397129186602 7,
k = 15	0.0952380952380952 3]	0.1052631578947368 4]

Figure 6: Table of CV test error rates for *k-nn* classification. Corresponds to graphic above in problem 2.2.

3. Apply two other classifiers of your choice to the same data. Possible algorithms include (but are not limited to) logistic regression, QDA, naive Bayes, SVM, and decision trees. You may use any existing libraries. Use 5-fold cross validation to calculate the test error. Report the training and test errors. If any tuning parameters need to be selected, use cross-validation and report the training and test error for several values of the tuning parameters. Which of the classifiers performed best? Did any of them underfit or overfit the data? How do they compare to the k-nn classifiers in terms of performance?
Hint: You may want to check out the `scikit-learn` library.

Seed Data Error Rates of Other Classifiers		
	Logistic Regression	**Random Forests**
Training Error	0.07142857142857142	0.009523809523809525
5-fold CV Test Error	0.07619047619047625	0.08571428571428574

- **NOTE:** this comment also appears in my code, but I'll make it here as well. Per https://scikit-learn.org/stable/modules/multiclass.html, *"All classifiers in scikit-learn do multiclass classification out-of-the-box."* Hence, there's no need to do anything like one-V-all classification to use logistic regression like we might in another language (since it's a binary method.)
- **NOTE:** I understood training error to mean the error obtained by training a model on all of the data and then using the trained model to predict back onto all of the data.
- Since I used logistic regression and random forests, I had no parameters to tune. Although random forests could be tuned with parameters like number of trees, maximum tree depth, etc., my practical experience has shown me that very minimal gains are obtained through tuning these parameters. As such, I left the defaults and achieved good results.
- We can see that the random forest predicting back onto the whole dataset is very overfit. It is far lower than the 5-fold CV error for random forests, and is also far lower than any other error rates obtained by any other method. A training error of 0.0095 is only $\frac{2}{210}$ observations misclassified.
- On the other hand, the training error rate of logistic regression is very near it's 5-fold CV error rate. Logistic regression slightly outperforms random forests and *k-nn* when comparing with 5-fold CV test error, and wins the best classifier award for this problem.
My guess as to why its training error is close to its test error: a linear boundary might be less likely to overfit and these data might be well-separated by a linear boundary, OR whatever built-in machinery scikit-learn uses to implement multiclass classification with binary classifiers (like logistic regression) has some kind of cross-validating effect even when generating training error.

Problem 3

1. Suppose we have random variables $X_1, \ldots, X_n$ that are independent and identically distributed each with mean μ and variance σ^2. In this setting, the variables X_i can be viewed as data points sampled in such a way that we expect each data point to be drawn from the same distribution. In general, we do not know this distribution including its properties such as the mean and variance. Thus if we want to know these properties, we need to estimate them from the data.
   Suppose that we wish to estimate some parameter c with an arbitrary estimator $\hat{c}$. Since the estimator depends on the data, it is also a random variable. The ***bias*** of this estimator is equal to $E[\hat{c}] - c$.
   Define the following:
   $$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i.$$
   $\bar{X}$ is an estimator of μ. Calculate the bias of $\bar{X}$.

   - Here, $c = \mu$ (the true value of the parameter we want to estimate), and $\hat{c} = \bar{X} = \frac{1}{n}\sum\limits_{i=1}^{n} X_i$ since we're using $\bar{X}$ to estimate μ.
   - Using properties of expectation, let's make the following string of manipulations/deductions:
     $E[\hat{c}] = E[\bar{X}] = E\left[\frac{1}{n}\sum\limits_{i=1}^{n} X_i\right] = \frac{1}{n}E\left[\sum\limits_{i=1}^{n} X_i\right] = \frac{1}{n}\left[\sum\limits_{i=1}^{n} E[X_i]\right]$
     From the prompt we know that the X_i are iid with mean μ and variance $\sigma^2 \implies E[X_i] = \mu$.
     Thus our expression for $E[\hat{c}]$ becomes: $\frac{1}{n}\left[\sum\limits_{i=1}^{n} E[X_i]\right] = \frac{1}{n}\left[\sum\limits_{i=1}^{n} \mu\right] = \frac{1}{n}\frac{n\mu}{1} = \mu \implies E[\hat{c}] = \mu$
   - So $bias(\bar{X}, \mu) = E[\bar{x}] - \mu = \mu - \mu = 0 \rightarrow \boxed{bias(\bar{X}, \mu) = 0}$

2. Calculate the variance of the estimator $\bar{X}$.
   Hint: The variance of the sum of INDEPENDENT random variables is equal to the sum of the variances. Also, if a is a constant scalar, then $Var[aX] = a^2 Var[X]$.

   - Using properties of variance we can make the following string of manipulations/deductions:
     $Var[\bar{X}] = Var\left[\frac{1}{n}\sum\limits_{i=1}^{n} X_i\right] = \left(\frac{1}{n}\right)^2 Var\left[\sum\limits_{i=1}^{n} X_i\right] = \frac{1}{n^2}\sum\limits_{i=1}^{n} Var\left[X_i\right]$.
   - From the prompt we know that the X_i are iid with mean μ and variance $\sigma^2 \implies Var[\bar{X}_i] = \sigma^2$.
     Thus our expression for $Var[\bar{X}]$ becomes: $\frac{1}{n^2}\sum\limits_{i=1}^{n} Var\left[X_i\right] = \frac{1}{n^2}\left(n\sigma^2\right) = \frac{n\sigma^2}{n^2} \rightarrow \boxed{Var\left[\bar{X}\right] = \frac{\sigma^2}{n}}$

3. The mean squared error (MSE) of an estimator is defined as

$$MSE(\hat{c}) = E[(\hat{c} - c)^2].$$

It can be shown that the MSE of an estimator is equal to the square of its bias plus the variance. What is the MSE of the estimator $\bar{X}$?

<u>NOTE:**</u> the way the question is phrased, I understand it to mean that I'm not required to show that $MSE(\hat{c}) = E\big[(\hat{c}-c)^2\big] = Var(\hat{c}) + \big[Bias(\hat{c}, c)\big]^2$, but I am allowed to use this fact for solving the problem.

- Assuming that the information discussed in **<u>NOTE</u>** is true, we can simply use the given formula for $MSE(\hat{c})$ and the answers from problems 3.1 and 3.2 to solve for $MSE(\bar{X})$.
- From 3.1 we know that $Bias(\bar{X}, \mu) = 0 \implies \big[Bias(\bar{X}, \mu)\big]^2 = 0$.
- From 3.2 we know that $Var(\bar{X}) = \dfrac{\sigma^2}{n}$.
- Therefore, $MSE(\bar{X}) = Var(\bar{X}) + \big[Bias(\bar{X}, \mu)\big]^2 = \dfrac{\sigma^2}{n} + 0 \rightarrow \boxed{MSE(\bar{X}) = \dfrac{\sigma^2}{n}}$

4. Consider the following estimator of σ^2:

$$\hat{s}^2 = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2.$$

Calculate the bias of $\hat{s}^2$. If this estimator is biased, is there a way to define a new estimator of σ^2 that is unbiased?

- Since $Bias(\hat{c}, c) = E[\hat{c}] - c$ and we're estimating σ^2 using $\hat{s}^2 \implies Bias(\hat{s}^2, \sigma^2) = E[\hat{s}^2] - \sigma^2$.
- Using properties of expectation we can make the following string of manipulations/deductions:
  $E[\hat{s}^2] = E\left[\dfrac{1}{n}\sum\limits_{i=1}^{n}\left(X_i - \bar{X}\right)^2\right] = E\left[\dfrac{1}{n}\sum\limits_{i=1}^{n}({X_i}^2 - 2X_i\bar{X} + \bar{X}^2\right]$
  $= E\left[\dfrac{1}{n}\sum\limits_{i=1}^{n}{X_i}^2 - 2\dfrac{1}{n}\sum\limits_{i=1}^{n}X_i\bar{X} + \dfrac{1}{n}\sum\limits_{i=1}^{n}\bar{X}^2\right]$
- Recall that $\bar{X}$ does not depend on the index i in the sums, so we can rewrite the expression as:
  $E\left[\dfrac{1}{n}\sum\limits_{i=1}^{n}{X_i}^2 - 2\bar{X}\left(\dfrac{1}{n}\sum\limits_{i=1}^{n}X_i\right) + \dfrac{1}{n}\dfrac{n\bar{X}^2}{1}\right] = E\left[\dfrac{1}{n}\sum\limits_{i=1}^{n}{X_i}^2 - 2\bar{X}(\bar{X}) + \bar{X}^2\right] = E\left[\dfrac{1}{n}\sum\limits_{i=1}^{n}{X_i}^2 - \bar{X}^2\right]$
  $= E\left[\dfrac{1}{n}\sum\limits_{i=1}^{n}{X_i}^2\right] - E\left[\bar{X}^2\right] = E\big[{X_i}^2\big] - E\big[\bar{X}^2\big]$
- <u>Aside</u>: what is $E\big[{X_i}^2\big] = ?$ Recall that $Var[X_i] = E[{X_i}^2] - ([E[X_i])^2 \implies \sigma^2 = E[{X_i}^2] - (\mu)^2 \implies E[{X_i}^2] = \sigma^2 + \mu^2$
- <u>Aside</u>: what is $E\big[\bar{X}^2\big] = ?$ Recall that $Var[\bar{X}] = E[\bar{X}^2] - (E[\bar{X}])^2 \implies \dfrac{\sigma^2}{n} = E[\bar{X}^2] - (\mu)^2 \implies E[\bar{X}^2] = \dfrac{\sigma^2}{n} + \mu^2$
- So now $E[\hat{s}^2] = E\big[{X_i}^2\big] - E\big[\bar{X}^2\big] = (\sigma^2 + \mu^2) - \left(\dfrac{\sigma^2}{n} + \mu^2\right) = (\mu^2 - \mu^2) + \left(\dfrac{n\sigma^2}{n} - \dfrac{\sigma^2}{n}\right) = \dfrac{\sigma^2(n-1)}{n}$

- So $Bias(\hat{s}^2, \sigma^2) = E[\hat{s}^2] - \sigma^2 = \dfrac{\sigma^2(n-1)}{n} - \dfrac{n\sigma^2}{n} = \dfrac{n\sigma^2 - n\sigma^2 - \sigma^2}{n} \implies \boxed{Bias(\hat{s}^2, \sigma^2) = \dfrac{-\sigma^2}{n}}$
- Since $Bias(\hat{s}^2, \sigma^2) \neq 0 \implies \hat{s}^2$ is a biased estimator of σ^2.
- Let's define $\hat{u}^2$ to be a new estimator of σ^2 such that $Bias(\hat{u}^2, \sigma^2) = 0$.
- Define $\hat{u}^2 = a\hat{s}^2$ where $a \in \mathbb{R}^1$ (a is a scalar constant).
- We want $\left[E[a\hat{s}^2] = \sigma^2\right] \implies \left[aE[\hat{s}^2] = \sigma^2\right] \implies \left[a\left(\dfrac{\sigma^2(n-1)}{n}\right) = \sigma^2\right] \implies \left[a = \dfrac{n}{n-1}\right]$, giving $\left[E\left[\left(\dfrac{n}{n-1}\right)\hat{s}^2\right] = \sigma^2\right] \implies \left[\hat{u}^2 = \left(\dfrac{n}{n-1}\right)\hat{s}^2\right] \implies \left[\hat{u}^2 = \left(\dfrac{n}{n-1}\right)\dfrac{1}{n}\sum_{i=1}^{n}\left(X_i - \bar{X}\right)^2\right]$ giving $\boxed{\dfrac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \bar{X}\right)^2 = \hat{u}^2}$, where $\hat{u}^2$ is an unbiased estimator of σ^2 since $E[\hat{u}^2] = \sigma^2$.

Problem 4

1. Suppose we take all the weights and biases in a network of perceptrons, and multiply them by a positive constant, c > 0. Show that the behavior of the network doesn't change. (Exercise in Ch1 Nielsen book) **Hint**: Consider a single perceptron first. For a fixed input, does multiplying the weights and bias by a positive constant change the output of the perceptron? Now argue that the output of the entire network is unchanged.

   - Let's mathematically verify that the same inputs to any perceptron will yield the same output when the weights and bias are multiplied by a positive constant c.
   - If we let $\boldsymbol{w}$ be the vector of incoming weights to the perceptron, $\boldsymbol{x}$, be the vector of incoming inputs to the perceptron, and b be the bias of that perceptron, we know that the output of that perceptron will be $\begin{cases} 0, & \boldsymbol{w}^T\boldsymbol{x} + b \leq 0 \\ 1, & \boldsymbol{w}^T\boldsymbol{x} + b > 0 \end{cases}$
   - What happens if we multiply the weights $\boldsymbol{w}$ and bias b by c and feed in the same inputs $\boldsymbol{x}$? We will have output $= \begin{cases} 0, & c\boldsymbol{w}^T\boldsymbol{x} + cb \leq 0 \\ 1, & c\boldsymbol{w}^T\boldsymbol{x} + cb > 0 \end{cases} = \begin{cases} 0, & (c)\big(\boldsymbol{w}^T\boldsymbol{x} + b\big) \leq 0 \\ 1, & (c)\big(\boldsymbol{w}^T\boldsymbol{x} + b\big) > 0 \end{cases} = \begin{cases} 0, & \left(\frac{1}{c}\right)(c)\big(\boldsymbol{w}^T\boldsymbol{x} + b\big) \leq \left(\frac{1}{c}\right)(0) \\ 1, & \left(\frac{1}{c}\right)(c)\big(\boldsymbol{w}^T\boldsymbol{x} + b\big) > \left(\frac{1}{c}\right)(0) \end{cases}$
   $= \begin{cases} 0, & \boldsymbol{w}^T\boldsymbol{x} + b \leq 0 \\ 1, & \boldsymbol{w}^T\boldsymbol{x} + b > 0 \end{cases} \implies \begin{cases} 0, & c\boldsymbol{w}^T\boldsymbol{x} + cb \leq 0 \\ 1, & c\boldsymbol{w}^T\boldsymbol{x} + cb > 0 \end{cases} = \begin{cases} 0, & \boldsymbol{w}^T\boldsymbol{x} + b \leq 0 \\ 1, & \boldsymbol{w}^T\boldsymbol{x} + b > 0 \end{cases} \implies$ the output of a perceptron remains mathematically identical when we scale the incoming weights $\boldsymbol{w}$ and bias b by a positive constant c and feed in the same inputs $\boldsymbol{x}$.
   - We have shown that a single perceptron takes the same inputs $\boldsymbol{x}$ and yields the same output when $\boldsymbol{w}$ and b are scaled by c. Since all perceptrons in a network function on this mathematical decision rule we can know with surety that the behavior of all perceptrons is the same in the "scaled by c" network as in the "not scaled by c" network. Therefore, the behavior of the two networks must be identical, showing the desired result.

2. Given the same setup of `problem 4.1` - a network of perceptrons - suppose that the overall input to the network of perceptrons has been chosen and fixed. Suppose the weights and biases are such that $wx+b \neq 0$ for the input x to any particular perceptron in the network. Now replace all the perceptrons in the network by sigmoid neurons, and multiply the weights and biases by a positive constant $c > 0$. Show that in the limit as $c \to \infty$ the behavior of this network of sigmoid neurons is exactly the same as the network of perceptrons. How can this fail when $wx + b = 0$ for one of the perceptrons? (Exercise in Ch1 Nielsen book)
Hint: Use a similar approach as in problem 4.1 where you first consider the behavior of a single sigmoid neuron and then extend to the entire network.

- **NOTE:** At times in this problem I may use equal signs where they may not be exactly mathematically appropriate. I believe my logic and arguments are sound, but my notation may not be 100% mathematically kosher. If there is room for mercy in this regard I would appreciate it.
- Before beginning let's outline the following rules of limits to be used for computation:
  (a) $\lim_{x\to a}\big[f(x)+g(x)\big] = \lim_{x\to a} f(x) + \lim_{x\to a} g(x)$
  (b) $\lim_{x\to a}\left[\dfrac{f(x)}{g(x)}\right] = \dfrac{\lim_{x\to a} f(x)}{\lim_{x\to a} g(x)}$ provided that $\lim_{x\to a} g(x) \neq 0$
  (c) $\lim_{x\to\infty} e^{x} = \infty$
  (d) $\lim_{x\to\infty} e^{-x} = 0$
  (e) $\lim_{x\to a} c = c$ where $c \in \mathbb{R}$
- Now let's show that in the limit as $c \to \infty$ the the output of each individual sigmoid neuron will be the same as the output of each perceptron.
- For the remainder of this problem we will define $z = \boldsymbol{w}^T\boldsymbol{x} + b$ to save on notation. Note that z is the pre-decision-rule numerical outcome of a perceptron (NOT the final 0 or 1 output).
- As the prompt directs, when we multiply the weights $\boldsymbol{w}$ and bias b by positive constant c we will have $c\boldsymbol{w}^T\boldsymbol{x} + cb = c\big(\boldsymbol{w}^T\boldsymbol{x} + b\big) = cz$. The term cz will be our input to the sigmoid function (this is the scaled-by-c, pre-decision-rule numerical outcome of a perceptron).
- We know that the sigmoid function is defined as follows: $\sigma(x) = \dfrac{1}{1+e^{-x}}$. Since our input will be cz, our sigmoid neuron will have output $\sigma(cz) = \dfrac{1}{1+e^{-cz}}$.
- We are interested in the output behavior of these sigmoid neurons as $c \to \infty$. Therefore we want to know $\lim_{c\to\infty}\left[\dfrac{1}{1+e^{-cz}}\right]$. Using the limit rules defined above we can manipulate this to be: $\lim_{c\to\infty}\left[\dfrac{1}{1+e^{-cz}}\right] = \dfrac{\lim_{c\to\infty} 1}{\lim_{c\to\infty} 1 + \lim_{c\to\infty} e^{-cz}} = \dfrac{1}{1+\lim_{c\to\infty} e^{-cz}}$. By examining this final expression, we can see that the value only depends on $\left[\lim_{c\to\infty} e^{-cz}\right]$. Therefore we will characterize this limit for different values of z.
- We will consider the first two cases implied in the prompt: z > 0 and z < 0.
- **Case z > 0:** We know that $\left[\lim_{c\to\infty} -cz = (-z)\lim_{c\to\infty} c = (-z)(\infty) = -\infty\right] \implies \left[\lim_{c\to\infty} e^{-cz}\right] = \left[e^{-\infty} = 0\right]$ Now we can plug this quantity back into the overall sigmoid function to have: $\left[\lim_{c\to\infty}\sigma(cz)\right] = \lim_{c\to\infty}\left[\dfrac{1}{1+e^{-cz}}\right] = \dfrac{1}{1+\lim_{c\to\infty} e^{-cz}} = \dfrac{1}{1+0} = \dfrac{1}{1} = 1$. This value of 1 is the final output of the sigmoid neuron when $z > 0$ and $c \to \infty$. We know that this behavior is identical to that of a perceptron: when $z > 0$ the output of the perceptron is 1.

- **Case z $<$ 0:** For $z < 0$ let's define constant $k = -z \implies k > 0$. Substituting this into $\left[\lim_{c\to\infty} e^{-cz}\right]$ gives $\left[\lim_{c\to\infty} e^{kc}\right]$. We know that $\left[\lim_{c\to\infty} ck = (k)\lim_{c\to\infty} c = (k)(\infty) = \infty\right]$. Use this to get $\left[\lim_{c\to\infty} e^{kc}\right] = e^{\infty} = \infty$. Plugging this quantity back into the overall sigmoid function we have: $\left[\lim_{c\to\infty} \sigma(cz)\right] = \lim_{c\to\infty}\left[\frac{1}{1+e^{-cz}}\right] = \frac{1}{1+\lim_{c\to\infty} e^{-cz}} = \frac{1}{1+\infty} = \frac{1}{\infty} = 0$. This value of 0 is the final output of the sigmoid neuron when $z < 0$ and $c \to \infty$. We know that this behavior is identical to that of a perceptron: when $z < 0$ the output of the perceptron is 0.
- Because the behavior of each individual sigmoid neuron in the limit as $c \to \infty$ is identical to the behavior of each individual perceptron, we know that the networks as a whole will exhibit identical behavior **so long as** $z < 0$ **or** $z > 0$.
- **Case z = 0:** When z = 0 the parity between sigmoid neurons and perceptrons does not hold. $\left[\lim_{c\to\infty} e^{-cz} = \lim_{c\to\infty} e^{(-c)(0)} = \lim_{c\to\infty} e^{0} = \lim_{c\to\infty} 1 = 1\right]$. Plugging this quantity back into the overall sigmoid function we have: $\left[\lim_{c\to\infty} \sigma(cz)\right] = \lim_{c\to\infty}\left[\frac{1}{1+e^{-cz}}\right] = \frac{1}{1+\lim_{c\to\infty} e^{-cz}} = \frac{1}{1+1} = \frac{1}{2} = 0.5$. The behavior of the sigmoid neuron as $c \to \infty$ when $z = 0$ deviates from the behavior of a perceptron since the output of a perceptron when $z = 0$ is 0 (based on the decision rule).
  For z = 0: $\left[\text{ output of perceptron} = 0 \right] \neq \left[\text{ output of sigmoid neuron} = 0.5 \right]$.
  Thus we have shown that the identical behavior of the networks breaks down specifically for the case $z = 0$.

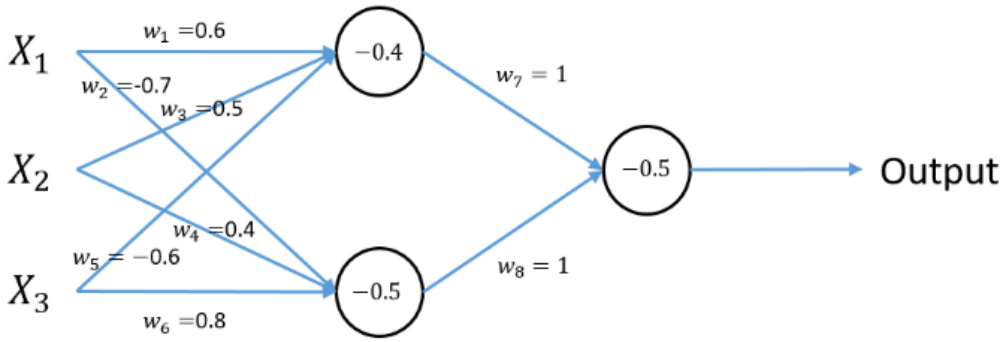

Figure 7: Multilayer perceptron with three inputs and one hidden layer.

3. For each possible input of the MLP in Figure 7, calculate the output. I.e., what is the output if $X = [0, 0, 0]$, $X = [0, 0, 1]$, etc. You should have 8 cases total.

   **__NOTE:__ I wrote Python code to compute these values as well as the values for 4.4. Since no instructions were given to exhibit computation or turn in code, I've only included my answers.

MLP Network Behavior	
Input case	**Output**
[0,0,0]	0
[1,0,0]	1
[0,1,0]	1
[0,0,1]	1
[1,1,0]	1
[1,0,1]	0
[0,1,1]	1
[1,1,1]	1

4. If we change the perceptrons in Figure 7 to sigmoid neurons what are the outputs for the same inputs (e.g., inputs of [0,0,0], [0,0,1], ...)?

   **__NOTE:__ I wrote Python code to compute these values. Since no instructions were given to exhibit computation or turn in code, I've only included my answers. Also, I rounded these values to the 6^{th} decimal place.

Sigmoid Neuron Network Behavior	
Input case	**Output**
[0,0,0]	0.569265
[1,0,0]	0.569867
[0,1,0]	0.622459
[0,0,1]	0.585012
[1,1,0]	0.617326
[1,0,1]	0.575084
[0,1,1]	0.633144
[1,1,1]	0.628311

5. Using perceptrons with appropriate weights and biases, design an adder that does two-bit binary addition. That is, the adder takes as input two two-bit binary numbers (i.e. 4 binary inputs) and adds them together. Don't forget to include the carry bit. The resulting output should be the two-bit sum and the carry bit for a total of three binary outputs.
Hint: You may want to consult the introduction slides for how to design a 1-bit adder using perceptrons.