# Homework 4

**Deadline: 2018.05.29 (Tuesday) 23:59**

You can EITHER create your own hw4.ipynb to write the code (meet the requirements) [BONUS] from the scratch OR use the hw4.ipynb we provide to complete the required parts.

## Linear Regression with Gradient Descent and Regularization [ hw4.ipynb ]

This homework consists of three parts. First, your task is to implement regularized linear regression to predict the amount of water flowing out of a dam using the change of water level in a reservoir. Second, you will go through some diagnostics of debugging learning algorithms and examine the effects of underfitting vs. overfitting.

## Problem 1: Linear Regression

### 1.1 Dataset Visualization

You are asked to begin by visualizing the dataset containing historical records on the change in the water level $x$, and the amount of water flowing out of the dam $y$.

The provided data consists of three files, corresponding to the follows:

- A training data that your model will learn on: $X, y$
- A validation data for determining the regularization parameter: $X_{val}, y_{val}$
- A test data for evaluating performance. These are "unseen" examples that your model did not see during training: $X_{test}, y_{test}$

You task is write a function `plotScatter(x,y,title,x_title,y_title)` to be called, to plot three scatter-plot figures for the training, validation, and test sets like Figure 1 (title="Training Set", x_title="Dam level change" and y_title="Water out"). In the following parts, you will implement linear regression and use that to fit a straight line to the data and plot learning curves. Following that, you will implement polynomial regression to find a better fit to the data.

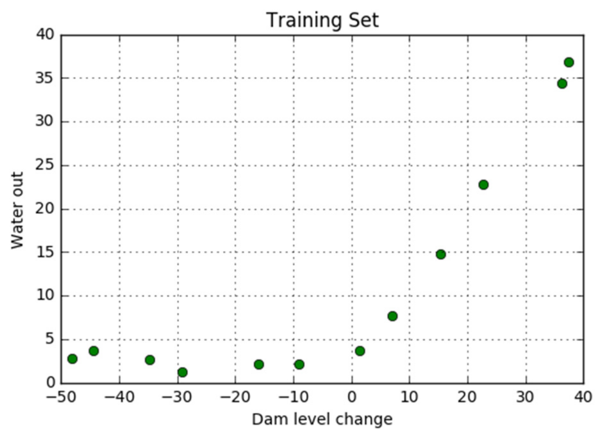


Figure 1. Visualization of training data.

## 1.2 Regularized Linear Regression Cost Function

Recall that regularized linear regression has the following cost function:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

where $\lambda$ is a regularization parameter which controls the degree of regularization (thus, help preventing overfitting). The regularization term puts a penalty on the overall cost $J$. As the magnitudes of the model parameters $\theta_j$ increase, the penalty increases as well. Note that you should not regularize the $\theta_0$ term. Your task is to write a function `computeCost(theta, X, y, lambda)` that takes the initialized $\theta_j$ training $X$ and $y$, and learning rate $\lambda$ as inputs, to calculate the regularized linear regression cost function. The returned value of computeCost function is the cost value.

## 1.3 Regularized Linear Regression Gradient

The partial derivative of regularized linear regression's cost for $\theta_j$ is defined as

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{for } j = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1$$

Your task is to write a function `computeGradient(theta, X, y, lambda)` that takes the initialized $\theta_j$ training $X$ and $y$, and regularization parameter $\lambda$ as inputs, to calculate the gradient for each $\theta_j$. The returned value of computeGradient function is a numpy array of gradient. When you are finished, you can run your gradient function using `theta` initialized at `[0; 0]`. Your task is also to call computeCost and computeGradient functions using training data and initialized theta, and the default regularization parameter $\lambda = 0$.

## 1.4 Fitting Linear Regression

Once your cost function and gradient are working correctly, you can take advantage of the function `optimizeTheta(X, y, theta, lmbda)` provided by us to compute the optimal values of $\theta$. In optimizeTheta, this training function uses `fmin_tnc` in scipy's `optimize` module to minimize the cost function. Note that you may need to use pip to install scipy. Also you can refer to scipy.optimize.fmin_tnc (click it) to understand what is done by fmin_tnc.

Currently we set regularization parameter $\lambda$ to zero. Because our current implementation of linear regression is trying to fit a 2-dimensional $\theta$, regularization will not be incredibly helpful for a $\theta$ of such low dimension. In the later parts of this homework, you will be using polynomial regression with regularization.

Finally, your ipynb script should be able to plot the best fit line, resulting in an image similar to Figure 2. The best fit line tells us that the model is not a good fit to the training data because the data has a non-linear pattern. While visualizing the best fit as shown is one possible way to debug your learning algorithm, it is not always easy to visualize the data and model. Later, you will implement a function to generate learning curves that can help you debug your learning algorithm even if it is not easy to visualize the data. Your task here is to make sure the ipynb can generate figure like Figure 2.

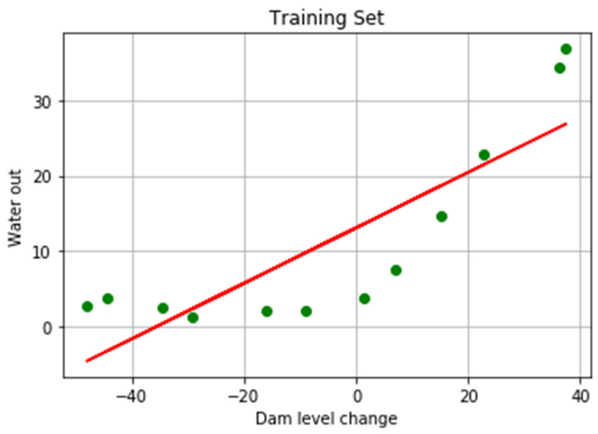


Figure 2. Linear fit.

## Problem 2: Underfitting vs. Overfitting

An important concept in machine learning is the underfitting-overfitting tradeoff. Models with high bias are not complex enough for the data and tend to underfit, while models with high variance overfit to the training data. In this part, you will plot training and test errors on a learning curve to diagnose underfitting-overfitting problems.

### 2.1 Learning Curve

You task now is to implement code to generate the learning curve that will be useful in debugging learning algorithms. A learning curve plots training and validation error as a function of training set size. Your job is to write a code section (in a for loop) so that it returns a vector of errors for the training set and validation set.

To plot the learning curve, we need a training and validation set error for different training set sizes. To obtain different training set sizes, you should use different subsets of the original training set $X$. Specifically, for a training set size of $i$, you should use the first $i$ examples (i.e., $X(1{:}i,:)$ and $y(1{:}i)$).

You can use the `optimizeTheta` function to find the $\theta$ parameters. After learning the $\theta$ parameters, you should compute the error on the training and validation sets. Recall that the training error for a dataset is defined as:

$$J_{\text{train}}(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2\right]$$

Note that the training error does not include the regularization term. One way to compute the training error is to use your existing cost function and set $\lambda$ to 0 only when using it to compute the training error and validation error. When you are computing the training error, make sure you compute it on the training subset (i.e., $X(1{:}\,i, :)$ and $y(1{:}\,i)$) (instead of the entire training set). However, for the validation error, you should compute it over the entire validation set. You should store the computed errors in the vectors `error_train` and `error_val`.

When you are finished this part, the ipynb will be able to print the learning curves and produce a plot similar to Figure 3.

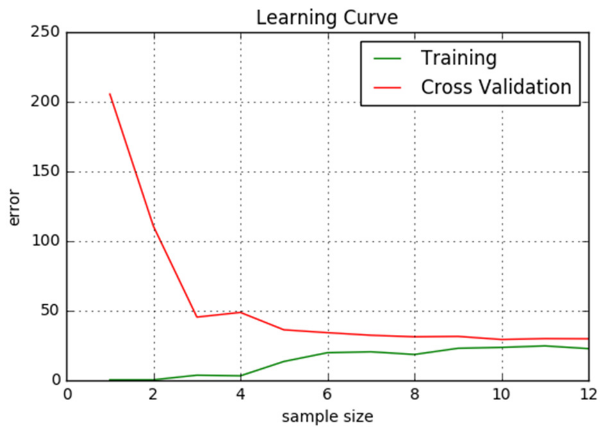


Figure 3. Linear regression learning curve

In Figure 3, you can observe that both the train error and validation error are high when the number of training examples is increased. This reflects a high bias problem in the model – the linear regression model is too simple and is unable to fit the validation data well. In the next part, you will implement polynomial regression to fit a better model for this data.

## Problem 3: Polynomial Regression with Regularization

The problem with our linear model was that it was too simple for the data and led to underfitting. In this part, you will address this problem by adding more features.

For using polynomial regression, the hypothesis has this form:

$$\begin{aligned} h_\theta(x) &= \theta_0 + \theta_1 * (\text{waterLevel}) + \theta_2 * (\text{waterLevel})^2 + \cdots + \theta_p * (\text{waterLevel})^p \\ &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_p x_p. \end{aligned}$$

By defining $x_1$ = (waterLevel), $x_2$ = (waterLevel)$^2$, ..., $x_p$ = (waterLevel)$^p$, we obtain a linear regression model where the features are the various powers of the original value (waterLevel).

## 3.1 Generating Polynomial Features

Now you are able to add more features using the higher powers of the existing feature $X$ in the dataset. Your task in this part is to write a function `generatePolynomialFeatures(X, p)` that takes training data $X$ and the polynomial $p$ as inputs, so that the function maps the original training set $X$ of size $m \times 1$ into its higher powers. Specifically, when a training set $X$ of size $m \times 1$ is passed into the function, the function should return a $m \times p$ matrix X_`poly`, where column 1 holds the original values of $x$, column 2 holds the values of $x^2$, column 3 holds the values of $x^3$, and so on. Note that you do not have to account for the zero-th power in this function.

Note that now you have a function that will map features to a higher dimension. In the following, you will apply it to the training and test sets, and the validation set that you haven't used yet.

## 3.2 Learning Polynomial Regression

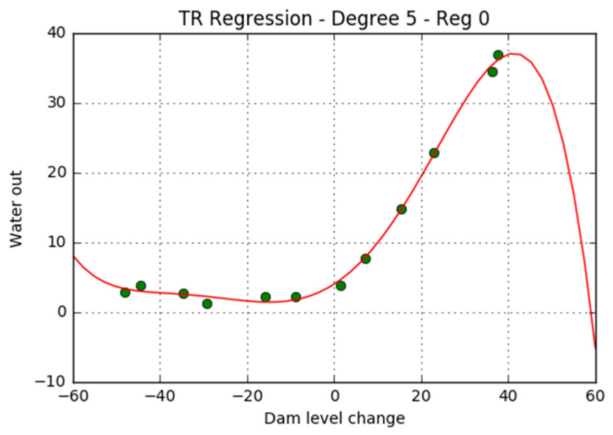
You proceed to train polynomial regression using your linear regression cost function. Keep in mind that even though we have polynomial terms in our feature vector, we are still solving a linear regression optimization problem. The polynomial terms have simply turned into features that we can use for linear regression. We are using the same cost function and gradient that you wrote for the earlier part of the prediction.

In this homework, you are asked to use a polynomial of degree 5 (`poly_degree=5`). It turns out that if we run the training directly on the data, will not work well as the features would be badly scaled (e.g. an example with $x = 40$ will now have a feature $x^5 = 40^5$). Therefore, you will need to use feature normalization.

Before learning the parameters $\theta$ for the polynomial regression, the ipynb needs to first call the `normalizeFeatures` function to normalize the features of the training set, storing the `mu` (means) and `sigma` (standard deviations) parameters separately. We have already implemented this function for you. Your task is to write the `createPolynomialFeaturesAndTrain(X, y, poly_degree, lmbda)` function that takes training $X$ and $y$, `poly_degree`, and lambda as inputs, to create the normalized polynomial features, initialized theta values $\theta$, and train the model. In this function, you can use the `generatePolynomialFeatures` function that you just wrote to create polynomial features, use the `normalizeFeatures` function we provided to normalize the created polynomial features, use np.zeros to initialize theta, and use the `optimizeTheta` function we provided to train the model and obtain the final theta $\theta$ and the cost. The returned values of this function include the normalized features X_`poly`, normalization parameters `original_means` and `original_std_devs`, and `theta_final` that is learned after training.

After learning the parameters $\theta$ by `createPolynomialFeaturesAndTrain`, you should see two plots (Figure 4 and Figure 5) generated for polynomial regression with $\lambda = 0$ by calling `plotScatter`, `plotPolynomialFit`, and `PlotPolynomialLearningCurvesForDifferentSampleSizes` (we have provided you). From Figure 4, you should see that the polynomial fit is able to follow the data points very well – thus, obtaining a low training error. However, the polynomial fit is very complex and even drops off at the extremes. This is an indicator that the polynomial regression model is overfitting the training data and will not generalize well.

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve (Figure 5) shows the same effect where the low training error is low, but the validation error is high. There is a gap between the training and validation errors, indicating a high variance problem.



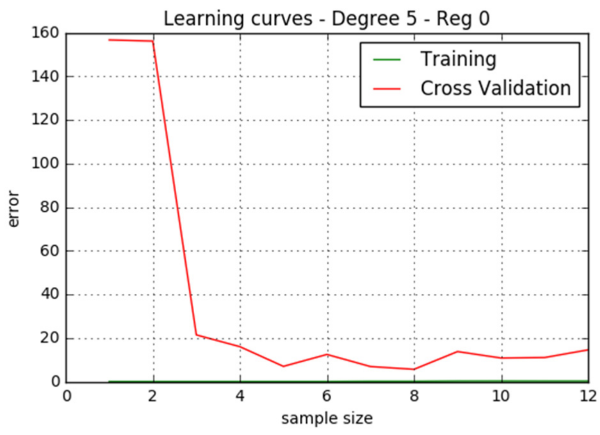


Figure 4. Polynomial fit, $\lambda = 0$.

Figure 5. Polynomial learning curve, $\lambda = 0$.

One way to combat the overfitting (high-variance) problem is to add regularization to the model. In the next section, you will get to try different $\lambda$ parameters to see how regularization can lead to a better model.

### 3.3 Adjusting the Regularization Parameter [No code need to be written in this part]

In this part, you will get to observe how the regularization parameter affects the underfitting-overfitting of regularized polynomial regression. You should now modify the lambda parameter and try $\lambda = 1$ and $\lambda = 100$. For each of these values, the ipynb should generate a polynomial fit to the data and also a learning curve.

For $\lambda = 1$, you should see a polynomial fit that follows the data trend well (Figure 6) and a learning curve (Figure 7) showing that both the validation and training error converge to a relatively low value.

This shows the $\lambda = 1$ regularized polynomial regression model does not have the high- bias or high-variance problems. In effect, it achieves a good trade-o between underfitting and overfitting. For $\lambda = 100$, you should see a polynomial fit (Figure 8) and a learning curve (Figure 9) that does not follow the data well. In this case, there is too much regularization and the model is unable to fit the training data.

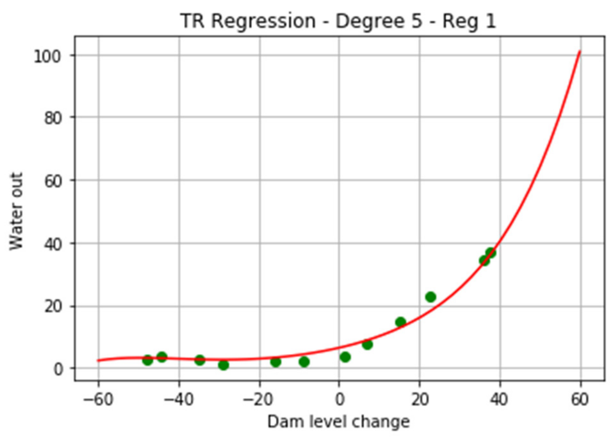


Figure 6. Polynomial fit, $\lambda = 1$.

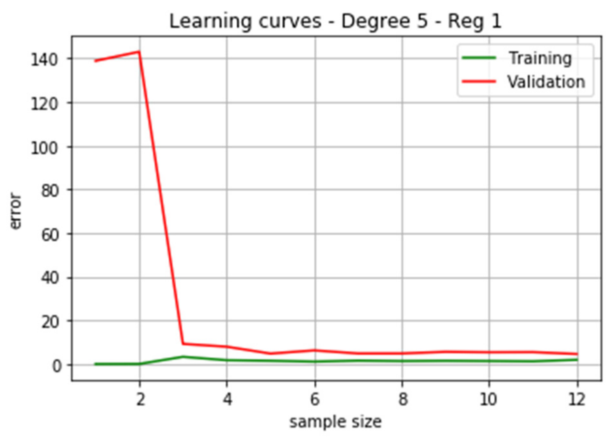


Figure 7. Polynomial learning curve, $\lambda = 1$.

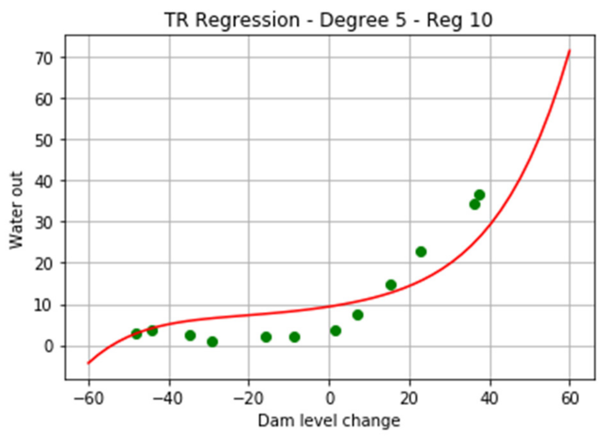


Figure 8. Polynomial fit, $\lambda = 100$.

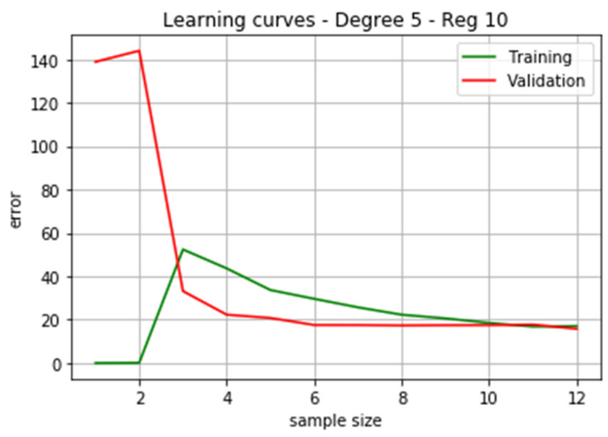


Figure 9. Polynomial learning curve, $\lambda = 100$.

### 3.4 Selecting $\lambda$ Using Validation Data

From the previous parts, you observed that the value of $\lambda$ can significantly affect the results of regularized polynomial regression on the training and validation set. In particular, a model without regularization ($\lambda = 0$) fits the training set well, but does not generalize. Conversely, a model with too much regularization ($\lambda = 100$) does not fit the training set and testing set well. A good choice of $\lambda$ (e.g., $\lambda = 1$) can provide a good fit to the data.

In this part, you will implement an automated method to select the $\lambda$ parameter. Concretely, you will use a validation set to evaluate how good each $\lambda$ value is. After selecting the best $\lambda$ value using the validation set, we can then evaluate the model on the test set to estimate how well the model will perform on actual unseen data. Your task is to a code section (in a for loop) to try different $\lambda$ values by using each $\lambda$ to (a) create the normalized polynomial features and train the model, (b) compute cost of the training subset (no regularization), and (c) compute cost of the validation subset (all samples in validation, not 1:m)(no regularization). Specifically, you should first use the `createPolynomialFeaturesAndTrain` function to train the model using different values of $\lambda$ and compute the training error and validation error using `computeCost`. You are asked to try $\lambda$ in the following range: $\{0.05, 0.10, 0.15, \ldots, 4.95, 5.00\}$.

After you have completed the code, the ipynb will run and plot a validation curve of error vs. $\lambda$ that allows you select which $\lambda$ parameter to use. You should see a plot similar to Figure 10. In this Figure, we can see that the best value of $\lambda$ is around 1.36. Due to randomness in the training and validation splits of the dataset, the validation error can sometimes be lower than the training error.

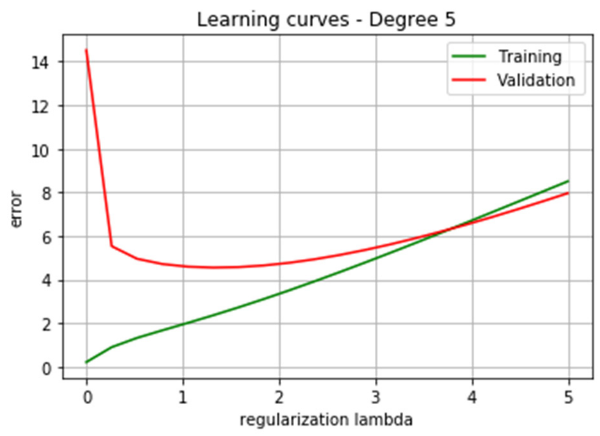


Figure 10. Selecting $\lambda$ using validation set.

### 3.5 Computing Test Error

In the previous part, you implemented code to compute the validation error for various values of the regularization parameter $\lambda$. However, to get a better indication of the model's performance in the real world, it is important to evaluate the "final" model on a test set that was not used in any part of training (that is, it was neither used to select the $\lambda$ parameters, nor to learn the model parameters $\theta$). For this part, your task is to write around three code statements to compute the test error using the best value of $\lambda$ you found. In our validation, we obtained a test error of 8.38 for $\lambda = 1.36$.

Note that in your codes for these problems,
you need to write some comments to describe the meaning of each part.

## How to Submit Your Homework?

**Submission in NCKU Moodle.** Before submitting your homework, please zip the files (**hw4.ipynb** and **hw4_data.mat**) in a zip file, and name the file as "學號_hw4.zip". For example, if your 學號 of your team are H12345678, then your file name is:

"H12345678_hw**4**.zip" or "H12345678_hw**4**.rar"

When you zip your files, please follow the instructions provided by TA's slides to submit your file using NCKU Moodle platform http://moodle.ncku.edu.tw .

## Have Questions about This Homework?

Please feel free to visit TAs, and ask/discuss any questions in their office hours. We will be more than happy to help you.