IST 597 Foundations of Deep Learning
Homework 1: Regression & Gradient Descent

This assignment is worth 15% of your grade for this class.

1 Introduction

In this assignment, you will implement several forms of regression and fit these models to several datasets. Before starting, we strongly encourage you to read chapters 2 through 5 of the class textbook, *Deep Learning* [3]. Pay particular attention to the end of Chapter 4 (the section on gradient descent). With respect to good practices for conducting machine learning experiments, we recommend that you read the very approachable paper "A Few Useful Things to Know About Machine Learning" [2] (though this is not required).

To get you to think in terms of neural architectures, we will approach the problem of estimating good regression models from the perspective of incremental learning. In other words, instead of using the normal equations to directly solve for a closed-form solution, we will search for optima (particularly minima) by iteratively calculating partial derivatives (of some cost function with respect to parameters) and taking steps down the resultant error landscape. The ideas you will develop and implement in this assignment will apply to learning the parameters of more complex computation graphs, including those that define convolutional neural networks and recurrent neural networks.

To complete the assignment, you will need to have Python installed on your computer, along with `numpy`, `pandas`, and `matplotlib`. To save yourself the headache of installing separate libraries (through a tool such as *pip*), we recommend you install the *Ananconda* package manager [1], which conveniently includes many of the libraries you will require for this and future assignments.[2]

The starter code for this assignment can be found at the following Github repository:

`https://github.com/ago109/IST597-Deep-Learning-Foundations.git`

Specifically, you will want to download the files found under `/problems/HW1/` directory. The data to be used for this assignment can be found inside the `/problems/HW1/data/` folder.

Make sure to add a comment to each script (or modify the file's header comment) that contains your name. You will notice that there is an empty folder, `/problems/HW1/out/`, which is where you will need to store your program outputs for each part of this assignment (i.e., plots).

For the questions you will be asked to answer/comment on in each section, please maintain a separate document (e.g., such as Microsoft Word) and record yours answers/thoughts there. Furthermore, while the Python scripts will allow you to save your plots to disk, please copy/insert the generated plots into your document again mapped to the correct problem number. For answers to various questions throughout, please copy the question(s) being asked and write your answer underneath (as well as the problem number).

Make sure you look for all of the `WRITEME` comments, as these will be the places you will need to write code in order to complete the assignment successfully.

[1]Please visit https://docs.continuum.io/ to download and install Anaconda. You might want to consider *Miniconda*, which is a light-weiht version of *Anaconda*.

[2]Note that the code for this very assignment was written with Python 2.7.8 using Anaconda 2.1.0 (64-bit) on a Cygwin environment (which emulates Bash) on a windows machine.

2 Problem #1: Univariate Linear Regression (25 points)

For this part of the assignment, you will build a simple univariate linear regression model to predict profits for a food truck. The data you will want to examine is in a file called `data/prob1.dat`. The base Python script you will need to modify and submit is `prob1_fit.py`.

For this part, we will restrict ourselves to a linear hypothesis space, constructing a model that adheres to the following form:

$$f_\Theta(x) = \theta_0 + \theta_1 x \tag{1}$$

Given data, your goal will be to estimate the parameters of this model using the method of steepest gradient descent. The parameters are defined within the construct $\Theta = \{\theta_0, \theta_1\}$ (where $\theta_{j>0}$–or w in the code–is the vector of learnable coefficients that weight the observed variables, and θ_0–or b in the Python code–is a single bias coefficient) and can be found by minimizing the negative Gaussian log likelihood (since the data is real-valued). For simplicity, we assume a fixed variance of 1 which leads to the well-known mean squared error. Thus, the cost function you will want to implement is:

$$\mathcal{J}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i)^2 \tag{2}$$

The gradient of the negative log likelihood with respect to the model parameters $\Theta = \{\theta_0, \theta_1\}$, after application of the chain rule, takes the general form:

$$\frac{\partial \mathcal{J}(\Theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i)x^i_j, j = 0, 1, 2, \cdots, n \tag{3}$$

where j indexes a particular parameter, noting that $x_0 = 1$.[3] In the univariate (single-variable) case, this leads us to utilize the following two specific gradients:

$$\frac{\partial \mathcal{J}(\Theta)}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i)x^i_1 \tag{4}$$

$$\frac{\partial \mathcal{J}(\Theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i)(x^i_o = 1) = \frac{1}{m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i) \tag{5}$$

where we see that the partial derivative of the loss with respect to θ_0 (the bias b) takes a simpler form given that the feature it weights is simply $x_0 = 1$ (we are essentially augmenting the pattern x with a bias of one, which allows us to model the mean μ of the data's distribution, assuming that it is Gaussian distributed).

To learn model parameters using batch gradient descent, we will need to implement the following update rule (for each θ_j in Θ):

$$\theta_j = \theta_j - \alpha\frac{\partial \mathcal{J}(\Theta)}{\partial \theta_j} = \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i)x^i_j, j = 0, 1, 2, \cdots, n. \tag{6}$$

First, implement the relevant components needed to construct a linear regression model. This means you will need to complete the `computeCost(.)` and `computeGrad(.)` routines. Furthermore, you will need to write code for the `regress(.)` function, which will produce the model's output given an input data point (or batch of data points). In the interest of modularity, you should break up your work in writing the computation of the loss by first completing the provided sub-routine `gaussian_log_likelihood(.)` and then using that inside the more general `computeCost(.)`. Again, make sure you look for the `WRITEME` annotations inside the comments throughout the code–please replace these particular comments with your own code.

[3] If x has only a single dimension ($x = \{x_1\}$), or a sample from a univariate distribution, then we would have two parameters, the bias θ_0 and the weight on x_1, θ_1.

Next, you will need to implement the update equations inside the main processing loop in order to search for values of Θ using gradient descent. You will also want to write some code to track the cost throughout training. (There is a "cost" list variable that you can update incrementally if you like.)

Finally, fit your linear regression model on the food truck data and save a plot showing your linear model against the data samples. Furthermore, write code to produce a plot that shows the cost you are minimizing as a function of epoch (or full pass through the dataset). Save both generated plots and paste them into your answer document (either manually using the pop-up windows or by writing a call to `matplotlib` that will save to disk the plot as an image).

Since learning a linear model is a convex optimization problem, you should see convergence to a low mean squared error. However, you will need to tune the learning rate/step size α (bear in mind that values that are too big will result in divergence). You should record the settings you tried and what you found that worked also in the answer sheet. Write a few sentences describing what you learned from the training/model fitting process. Things to discuss: What happens when you change the step-size α? How many epochs did you need to converge to a reasonable solution (for any given step size?

3 Problem #2: Polynomial Regression & Regularization (25 points)

For this section, you will need to use the data file `data/prob2.dat`. The samples found in this file were generated from a modified cosine function corrupted with Gaussian noise. The base Python script you must modify and submit is `prob2_fit.py`.

In the previous problem, you implemented a simple linear regressor, which, as discussed in lecture, means we have restricted our hypothesis space to the set of linear functions. To increase our model capacity, we can add additional terms to Equation 1 and build a non-linear model (or a *p*-th order polynomial model). This means our hypothesis now takes the form:

$$f_\Theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_n x^n. \tag{7}$$

Modify the relevant routines within `prob2_fit.py` to implement a flexible polynomial regression model of arbitrary order (as in, you can enter any positive value for *p* and your code should construct the correct number of terms). However, we can still directly use the equations we defined for previous problem if we choose to view our non-linear model as a multivariate linear regression over a set of "created" features. Specifically, we apply a fixed function that will transform the single input variable into a higher dimensional vector of scaled features (or rather, the polynomial terms we desire). For a $p = 6$ model, this would mean we have a feature map defined as follows:

$$Map_{feature}(x) = [x_1 = x_1, x_2 = x_1^2, x_3 = x_1^3, x_4 = x_1^4, x_5 = x_1^5, x_6 = x_1^6] \tag{8}$$

where x is mapped to full vector (of 7 elements) with each new feature indexed at j (reserving $j = 0$ for the bias). The goal is to now learn the set of parameters $\Theta = \{\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$, related through the following linear hypothesis:

$$f_\Theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5 + \theta_6 x_6. \tag{9}$$

Adding these additional features will enhance our model's expressivity, but at the increased risk of chasing spurious correlations.

In addition to completing the `computeCost(.)`, `gaussian_log_likelihood(.)`, `computeGrad(.)`, and `regress(.)` routines/sub-routines, you will also need to implement the feature mapping procedure sketched above in Equation 8. As you write code for the necessary routines that define your linear model, you will now need to make sure that you use the correct linear algebraic operators (such as those discussed in class, including inner and outer products, hadamard products, transpose, etc.) to ensure correct shaping of inputs and outputs. Also, do not forget to write the gradient descent update rules into the main training loop.

With respect to the feature mapping, what is a potential problem that the scheme we have designed above might create? What is one solution to fixing any potential problems created by using this scheme (and what other problems might that solution induce)? Write your responses to these questions in your external answer document.

Fit a 1st order (i.e., linear), 3rd order, 7th order, 11th order, and 15th order polynomial regressor to the data in `prob2_data.txt`. What do you observe as the capacity of the model is increased? Why does this happen? Record your responses to these questions in your answer sheet.

One way to combat the effects of over-parametrization is to introduce a second term to our cost function that penalizes the magnitude of the model parameters, commonly known as the L2 penalty. This can also be seen, from a Bayesian perspective, as imposing a Gaussian prior distribution over the weights of the regressor. This constraint (or prior) we impose over the weights we ultimately learn is an additional bias we introduce into the learning process. However, this assumption often proves to useful in practice since the penalty forces the optimization to favor smaller weights/parameters and thus simpler models (this is a quantitative embodiment of Occam's Razor).[4]

The regularized loss function takes the following form:

$$\mathcal{J}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i)^2 + \frac{\beta}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{10}$$

where β is another meta-parameter for us to set (through trial and error, or, in practice, through proper cross-fold validation). The gradient of this regularized form of the loss with respect to parameters Θ is straightforward:

$$\frac{\partial \mathcal{J}(\Theta)}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(f_\Theta(x^i) - y^i)x_j^i + \frac{1}{m}\theta_j \tag{11}$$

except for the case of $j = 0$ (which indexes the bias parameter), on which the penalty is not applied.

Modify the loss function code inside `computeCost(.)` (and the corresponding `computeGrad(.)` to include the L2 penalty term. Refit your 15th order polynomial regressor to the same data but this time vary the β meta-parameter using the specific values $\{0.001, 0.01, 0.1, 1.0\}$ and produce a corresponding plot for each trial run of your model. What do you observe as you increase the value of β? How does this interact with the general model fitting process (such as the step size α and number of epochs needed to reach convergence)? Record this in your answer document.

During the model fitting procedure, you might notice that the cost "flat-lines" or hardly changes after so many epochs. This is, when measured on the training subset of your data, an indicator your model has converged to an optima. In this scenario, we would like to cut back on useless computation especially if we have found a good linear hypothesis earlier on in training. One way to do this would be to compare the current cost (at time t) with the one immediately before ($t - 1$) and compare the decrease in the cost against some minimum value (or ϵ). Modify your main training loop to include a check for convergence as well as a means to exit the loop the moment convergence is reached.[5] Comment (in your answer sheet) as to how many steps it then took with this early halting scheme to reach convergence. What might be a problem with a convergence check that compares the current cost with the previous cost (i.e., looks at the deltas between costs at time t and $t - 1$), especially for a more complicated model? How can we fix this?

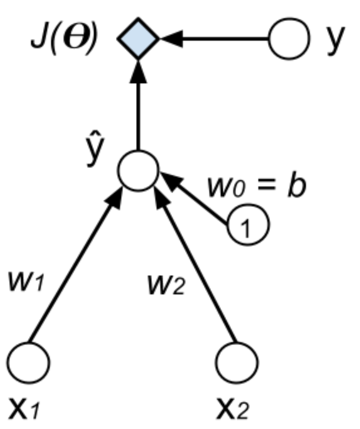
4 Problem #3: Multivariate Regression & Decision Boundaries (50 points)

To complete this problem, you will make use of the data found inside the file `data/prob3.dat`. The base Python script you will need to modify and submit is `prob3_fit.py`.

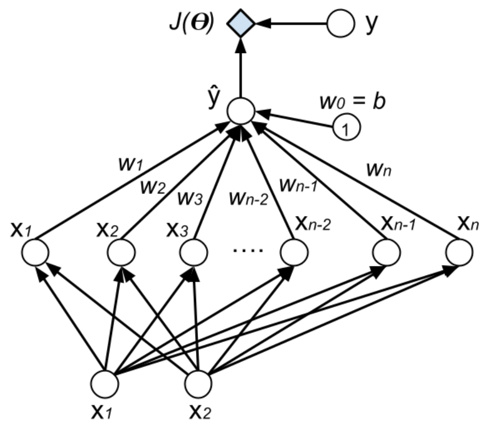
As we saw in Problem #2, in the case of artificially constructed features, the linear model you have implemented in the first part can be easily extended to the multivariate regime by simply assigning a

[4] L2-regularized regression is also known as ridge regression and Tikhonov regularization.

[5] When this form of conditional halting is performed using the loss measured on a separate validation subset (in order to estimate your model's generalization ability), this is known as *early stopping*. This "trick" is often applied to the training of more complex models, such as artificial neural networks and can be viewed as another form of model regularization where simpler models with smaller weights are strongly favored.



(a) Linear regressor architecture.



(b) Non-linear regression architecture.

Figure 1: By using feature mapping, we are building a somewhat "deeper" architecture–the first layer is effectively a fixed function (i.e., there are no learnable weights along these connections from the bottom layer to the first "hidden" layer). On the left, in Figure (a), we see a high-level view of the computation graph of the multivariate linear regressor applied to two observed variables. On the right, Figure (b), we see that a non-linaer regressor is really just a multivariate linear model applied to the features (still two features, like model in Panel (a)) created by our feature mapping function. This highlights the intuition behind projecting input variables to higher dimensional spaces–the projected space might allow data points to be more easily separated under a linear hypothesis. Note that this is one of the key intuitions behind kernel-based Support Vector Machines [1].

separate coefficient to each dimension of the input and encapsulating these parameters into a single vector. The architecture of our extended model is depicted in Figure 1 (left panel). Furthermore, the feature mapping scheme we employed in the previous problem can be applied to multi-dimensional inputs as well, even allowing us to model multiplicative interactions between pairs of covariates. Such a scheme, say for a third order polynomial model applied to two-dimensional input, looks like:

$$Map_f(x) = [x_1 = x_1, x_2 = x_2, x_3 = x_1^2, x_4 = x_1 x_2, x_5 = x_2^2, x_6 = x_1^3, \cdots, x_9 = x_2^3]. \tag{12}$$

Note that we have implemented the general form of this feature mapping function for you. The architecture of multivariate regression applied to a feature mapping stage is depicted in Figure 1 (right panel).

For the data you will work in this part of the assignment, you will notice that the targets take the form of discrete variables (y could either be a zero or a one). This means we will be performing a binary classification task and need to separate data points as to whether they fall under category 0 or 1. To model this type of target variable, we will adapt our linear model (which before was being used to estimate the mean of a continuous, Gaussian distribution) to estimate the probability p a data point falling under category 1 (with the probability for the alternative naturally taking the form $1-p$). This means that we are trying to estimate the parameter of a Bernoulli distribution, $Bern(1,p)$. Only one real change needs to be made to our linear model:

$$f_\Theta(x) = \phi(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5 + \theta_6 x_6 + \cdots + \theta_9 x_9). \tag{13}$$

where $\phi(v) = \frac{1}{1+e^{-v}}$ is the logistic link function (or logistic sigmoid), which will squash input values to the range $[0,1]$, effectively modeling probability values.

While some of what you implemented for the previous problems can be re-appropriated for this section, in order to perform logistic regression we will need to change the cost function. The regularized loss function now takes the following form:

$$\mathcal{J}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m} -y^i \log(f_\Theta(x^i)) - (1-y^i)\log(1-f_\Theta(x^i)) + \frac{\beta}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{14}$$

which is also known as binary cross-entropy (and can be derived from the Bernoulli likelihood function). β, again, controls the strength of the regularization (or penalty on the weights). Interestingly enough, the gradient of this loss with respect to parameters is identical to that of the Gaussian log likelihood, meaning we can simply reuse Equation 11 (also recall that we do not regularize the bias parameter θ_0). [Show yourself this by simply finding the partial derivative with respect to any x_j. You will see, as emphasized in your book, why the logistic sigmoid function is so special when you examine its derivatives.]

As was done in Problems #1 and #2, complete the `computeCost(.)`, `computeGrad(.)`, and `regress(.)` routines, but this time from the perspective of Bernoulli log likelihood, developed above. This will also mean that you will need to complete `sigmoid(.)` which we will need as part of the estimator of our of the Bernoulli parameter p. Much like the case for Gaussian log likelihood, we have separated the negative log likelihood calculation from the `computeCost(.)` routine for you, encouraging you to first implement `bernoulli_log_likelihood(.)`. Furthermore, you will need to write the gradient descent update rules for Θ as well as an early-stopping mechanism based on the training loss.

To actually perform classification under the linear model Θ, you will further need to complete `predict(.)` routine, which will produce a discrete output variable by outputting a one if the probability p exceeds some specified threshold value in the range $(0,1)$. In this assignment, this threshold will be fixed to 0.5. (Feel free to play around with this value and write your observations in the answer document.) Finally, you will need to write some code to calculate the classification error of your model and print that to screen.

Fit your (non-linear) logistic regression[6] to the data found in `data/prob3.dat`, keeping β fixed at zero (which means no regularization), tuning only the number of epochs and your learning rate α. At the end of your trial run, the Python script will produce a contour plot for you that will super-impose your learned model's decision boundary over top the individual data points. Examining the decision boundary of your model can be useful (and simple for two observed variables) when trying to build an understanding of how the model learns to distinguish the categories/classes of your data. More importantly, this is quite useful when investigating if your model has overfit the data and how regularization might change/deform the decision boundary. Note that while the decision boundary you should see in your contour plots is certainly not linear (in the original input space), it is linear in the projected higher-dimensional space created by our feature mapping routine (remember, our model can only embody linear hypotheses). As mentioned in the caption of Figure 1, this forms part of the foundation upon which Support Vector Machines [1] stand on.

For the last part of this assignment, re-fit your logistic regression but this time with $\beta = \{0.1, 1, 10, 100\}$, again, tuning the number of epochs and the learning rate. Copy each of the resultant contour plots to your answer sheet and report your classification error for each scenario. Comment (in the answer sheet) how the regularization changed your model's accuracy as well as the learned decision boundary. Why might regularizing our model be a good idea if it changes what appears to be such a well-fit decision boundary?

4.1 Regression as a Neural Architecture

As already implied throughout this assignment, you can think of logistic regression as a very simple neural network with no internal processing nodes (or "hidden" processing elements).[7] When we employ a feature mapping function, we are effectively creating a deeper model by adding an additional

[6]In this assignment, keep $p = 6$.

[7]Actually, one can view a deep neural network (that uses logistic sigmoid activation functions) as simply a stack of logistic regressors.

processing stage. However, this type of layer is fixed and requires careful consideration on the part of the human designer. In some sense, this is a simpler form of feature engineering.

As we start to develop the picture of representation learning, we will seek ways to learn these mapping functions automatically (ultimately trying to learn multiple layers of these feature processing functions). From this perspective, one can then regard a deep neural architecture as simply one complex, non-linear feature mapping function–we aim to compose multiple levels of feature processing/mapping functions to gradually build up "layers of abstraction". The highest level of our network will then produce a space that a linear classifier (such as logistic regression) can use to easily separate our data points. This, again, is one of the key motivations behind deeper statistical learning architectures.

5 On Your Own (Ungraded): Mini-Batch Gradient Descent

While in this assignment we focus on using batch gradient descent, it is important to note that this form of optimization does not scale well for large datasets (especially with many observed variables and many samples). In the next assignment, you will use a more practical form of gradient descent known as stochastic gradient descent, where we may take only a single random sample of the dataset and calculate a noisy estimate of the true gradient of the (instantaneous) loss with respect to model parameters. Generally, we will use a "block" of randomly selected examples at each iteration to compute a slightly less noisy estimate of the gradient–this is known as mini-batch gradient descent.

Feel free to modify these exercises on your own time to implement mini-batch gradient descent. If you do, make sure that you randomly sample data points *without replacement* to create the mini-batches. Using this form of training will now require your main training loop to be a nested loop–one inner loop that randomly samples the training set until it is empty (representing a full epoch) and one outer loop that will repeatedly initiate the inner loop for so many epochs. Note that in the outer loop, you will check the training (or validation) loss each time the inner loop has terminated.

References

[1] CORTES, C., AND VAPNIK, V. Support-vector networks. *Mach. Learn. 20*, 3 (Sept. 1995), 273–297.

[2] DOMINGOS, P. A few useful things to know about machine learning. *Commun. ACM 55*, 10 (Oct. 2012), 78–87.

[3] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.