Machine Learning Fundamentals

Assignment 1

ENGG 6500 | Intro to Machine Learning

Dr. Graham Taylor

Jesse Knight

2015-10-14

1 Theory

1.1 Summary: Curse of Dimensionality

See attached file.

1.2 Polynomial Least Squares Regression

A well-designed machine learning algorithm addresses the challenge of tediously classifying large amounts of data, often based on an equally unwieldly set of features. In neural networks, specifically, the classification features themselves are generated from layered nonlinear combinations of an original set of features. The new task becomes designing an appropriate algorithm for classifying the data, which should match the complexity of the problem. Models which are not complex enough, will have low fidelity to the system, and models which are too complex will fail to generalize.

Here, we consider a one-dimensional toy example showing the effects of model complexity on the solution. We construct a set of 20 training points in $\mathbb{R}^2 = \{x, y\}$, randomly spaced in x, following:

$$y(x) = 1.8\,x - 0.45\,x^2 + \eta, \qquad \eta \sim N(\mu = 0, \sigma^2 = 1) \tag{1.1}$$

and attempt to estimate the polynomial function $y(x)$ through least squares regression. Note that we have two methods for constraining solution complexity: inherent polynomial order and regularization. For a polynomial of order M, and N training points, the linear system we wish to solve becomes:

$$y = X\omega, \qquad \begin{cases} X_{i,j} = x^i \\ i \in \mathbb{Z}\,[0, M) \\ j \in \mathbb{Z}\,[1, N] \end{cases} \tag{1.2}$$

Without regularization, our solution aims to minimize the cost function J_2 (1.3), arriving at (1.4). Figure 1.1 demonstrates the results of this method for polynomial orders $M = \{1, 2, 9\}$.

$$J_2(\omega) = \| X^T\omega - y \|_2^2 \tag{1.3}$$

$$(X^T X)\hat{\omega} = X^T y \tag{1.4}$$

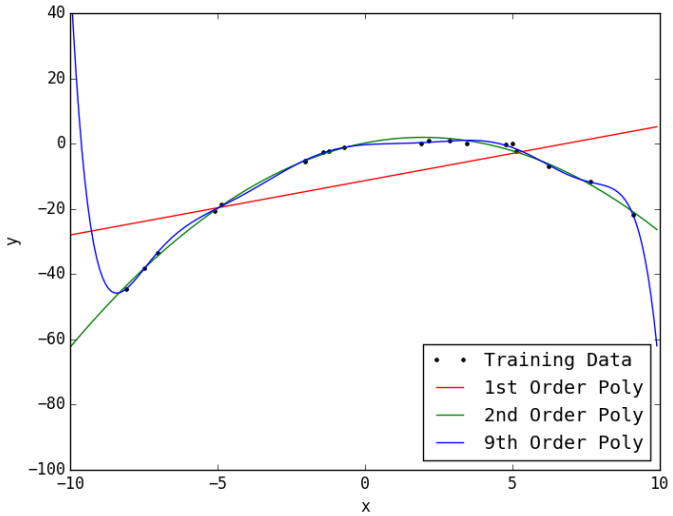

Figure 1.1: Polynomial least squares estimation with varied order showing underestimation of model complexity (1st Order), accurate modeling of complexity (2nd Order), and model overfitting (9th Order)

If we wish to implement regularization – to consider models of higher complexity but punish the solution as it becomes more complex, we extend our cost function to yield (1.5), and augment our linear system with a sparse operation of the regularization parameter λ, as in (1.6). The results of this method are presented, in Figure 1.2, where it can be seen that regularization may not be as effective as correct model complexity estimation.

$$J_{2,\lambda}(\omega) = \| X^T\omega - y \|_2^2 + \lambda\, \omega^T\omega \tag{1.5}$$

$$(A^TA)\hat{\omega} = A^Ty^*, \qquad A = \begin{bmatrix} X \\ \lambda I \end{bmatrix}, \qquad y^* = \begin{bmatrix} y \\ 0_{M,M} \end{bmatrix}, \tag{1.6}$$

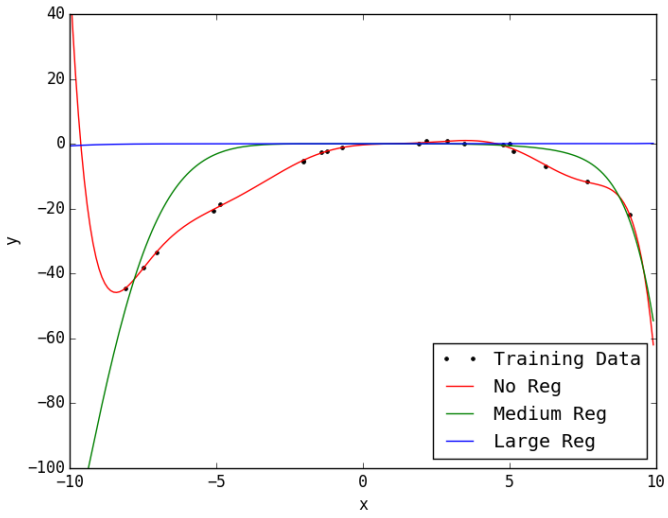

Figure 1.2: Polynomial least squares estimation with constant order M = 9, but varied regularization parameter λ, showing propensity for overfitting (red), but reasonable solution achieved with appropriate regularization (green)

Python code for this least squares regression model can be found in Appendix A.

2 Practice

2.1 Back-Propagation Gradients

Neural networks represent an extremely powerful tool for computing a decision surface for a labeled dataset. At the crux of the neural network is the concept of back-propagation: adjusting network parameters based on the performance or some other criteria. In practice, back propagation is computed recursively for many layers of a network, ideally using low level (e.g. CPU) matrix kernels for efficiency. However, it is a useful exercise to describe the parameter updates explicitly to understand the relationships between layers. Below is an implementation of back-propagation in a 2 layer model, used to calculate the gradient of the cross entropy error with respect to layer weights and biases.

```
def my_grads(X, y, W_hid, b_hid, W_out, b_out):

    # FPROP - redunant but necessary here with specified inputs
    # -----
    z_h = np.dot(X, W_hid) + b_hid   # hidden pre
    y_h = logistic(z_h)              # hidden act
    z_o = np.dot(y_h, W_out) + b_out # output pre
    y_o = logistic(z_o)              # output act
```

```
    # BPROP - where the magic happens
    # -----
    # partial d error WRT output pre (simplification of chain rule)
    dE_zo = (y_o - (t * np.ones(y_o.shape)))
    # partial d error WRT output b
    dE_bo = dE_zo.mean(axis=0)
    # partial d error WRT output W
    dE_Wo = np.multiply(np.atleast_2d(y_h).T, dE_zo).mean(axis=1)

    # partial d error WRT hidden act (another chain rule addition)
    dE_yh = np.multiply(np.atleast_2d(W_out).T, dE_zo).T
    # partial d error WRT hidden pre (last generalized chain rule)
    dE_zh = np.multiply(dE_yh, (y_h * (np.ones(y_h.shape) - y_h)))
    # partial d error WRT hidden b
    dE_bh = dE_zh.mean(axis=0)
    # partial d error WRT hidden W
    dE_Wh = np.multiply(np.atleast_3d(X), np.transpose(np.atleast_3d(dE_zh),[0,2,1])).mean(axis=0)

    return (dE_Wh, dE_bh, dE_Wo, dE_bo)
```

The 4 output gradients were validated against results from a symbolic gradient calculation algorithm using `pyautodiff` and `theano`. The Frobenius (Euclidean) norms of the comparison are shown below:

```
Checking gradients:
W_hid: OK   - norm of gradient difference with pyautodiff: 4.76752891685e-08
b_hid: OK   - norm of gradient difference with pyautodiff: 1.09809879829e-08
W_out: OK   - norm of gradient difference with pyautodiff: 3.03633887141e-07
b_out: OK   - norm of gradient difference with pyautodiff: 7.0353167736e-12
All gradients OK!
```

2.2 Optimizing Learning Rate

Learning rate has the potential to affect two aspects of the model training. First, it will almost always affect the speed of convergence: exceedingly low learning rates will take a long time to converge, and not necessarily escape local minima. However, learning rates which are too high risk missing good local minima, and may get caught 'bouncing around' on the solution surface.

Selecting an optimal learning rate may be done using a grid search, but this is best done with a logarithmic grid base for good coverage. We may iterate through the learning rates, re-training the model and observing convergence of the test data error – in this case, cross entropy. However, we must maintain constant model seeding to control for effects of random initialization. Moreover, perhaps one random set of initialization parameters biases the convergence accuracy towards a certain learning rate. Therefore, it is best practice to average a number of results for each rate.

Figure 2.1 summarizes the results of our learning rate optimization, using rates in the set e^{-a}, for $a \, \epsilon \, \mathbb{Z} \, [1, 7]$, and an average of 5 results from random initializations. Increasing learning rate improved convergence speed almost monotonically throughout the set; however, the wobble visible at right corresponding to the largest learning rate suggests some 'bounding' in rates higher than this. Therefore, we selected an 'optimal' learning rate of $e^{-2} \approx 0.135$.

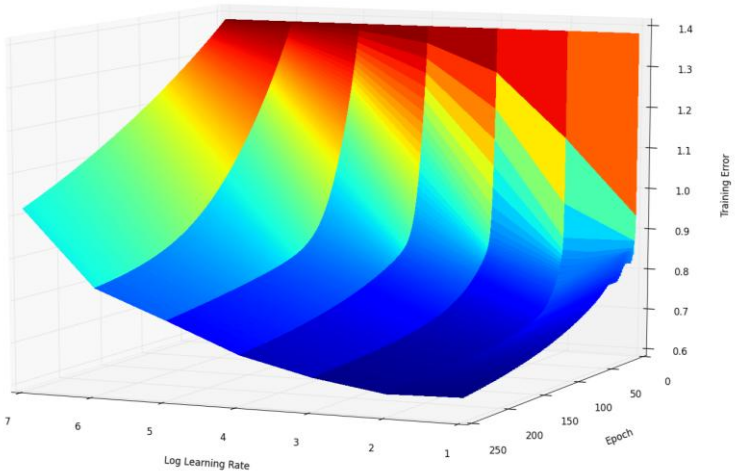

Figure 2.1: Surface plot of Test Data cross entropy (error) versus training epoch for a range of 7 learning rates; rates were e^{-a}, for $a \in \mathbb{Z}[1, 7]$, displayed in log scale.

2.3 Other Hyper-Parameter Optimization

There are a number of other hyper-parameters which we may wish to optimize. The most obvious is the number of training epochs. This is trivial to consider, as we simply run the model for a large number of epochs and select a reasonable point of stopping given the trade-off in efficiency. (FIG) shows the very minimal improvements in test data performance associated with this model past 500 epochs, when training with 100 hidden units and the optimal learning rate from 2.2.

Another parameter suitable for optimization is the number of hidden units. This ties back to our earlier investigation into model complexity. With too many hidden units, the training may over fit the data, while too few hidden units will not be able to model the system adequately. With the same learning rate and 250 epochs, we considered the number of hidden units in the set e^a, for $a \in \mathbb{Z}_{0.5}\ [1, 6]$, and examined the performance over epochs. Figure 2.3 shows the results, averaged over 10 trials to control for randomized initializations (which cannot be shared as the number of hidden units is a variable). Increasing complexity improves performance marginally, but significantly increases computation time.

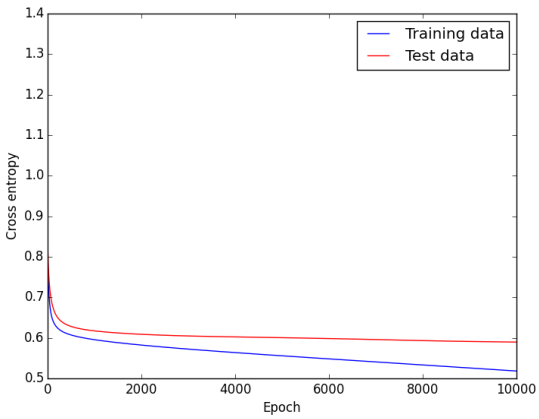

Figure 2.2: Model training performance using a large number of epochs

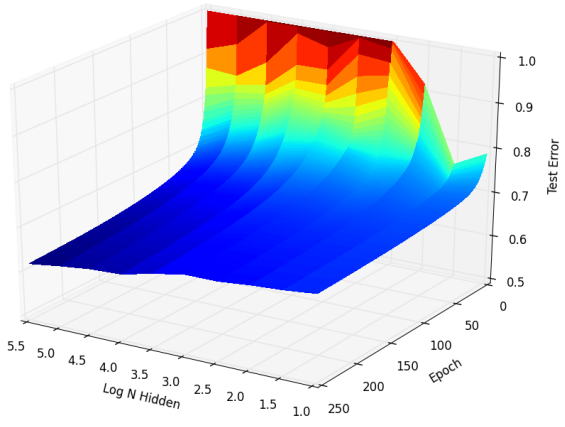

Figure 2.3: Model training performance with respect to the number of hidden units

Appendix A. Least Squares Regression Code

```
import numpy as np
import pylab
from numpy.random import uniform

true_coeffs = np.array([0.0, +1.8, -0.45])
xrng = (-10.0, +10.0)
n_points = 20

def x_mat(n, x):
    pwrs = np.arange(n)
    X = np.zeros((n, x.size))
    for n in np.arange(n):
        for i in np.arange(x.size):
            X[n][i] = np.power(x[i], pwrs[n])
    return X

def eval_poly(coeffs, x):
    return np.dot(coeffs.T, x_mat(coeffs.size, x))

def poly_fit(x, y, order, reg):
    order += 1
    X = x_mat(order, x)
    A = np.concatenate((X.T, reg * np.identity(order)), axis=0)
    Y = np.concatenate((y, np.zeros(order)), axis=0)
    W, res, rank, s = np.asarray(np.linalg.lstsq(A, Y))
    return W

def gen_plot(x, y0, xs, y1, y2, y3, lab1, lab2, lab3):
    pylab.plot(x, y0, '.k', label="Training Data")
    pylab.plot(xs, y1, '-r', label=lab1)
    pylab.plot(xs, y2, '-g', label=lab2)
    pylab.plot(xs, y3, '-b', label=lab3)
    pylab.legend(loc='lower right')
    pylab.xlim(xrng)
    pylab.ylim((-100, 40))
    pylab.xlabel("x")
    pylab.ylabel("y")
    pylab.show()

if __name__ == "__main__":
    x = np.sort(uniform(low=xrng[0], high=xrng[1], size=n_points))
    rnd = np.random.randn(n_points)
    y0 = eval_poly(true_coeffs, x) + rnd
    xs = np.arange(xrng[0], xrng[1], 0.1)
    pylab.rc('font',**{'family':'monospace'})

    wp1 = poly_fit(x, y0, 1, 0)
    wp2 = poly_fit(x, y0, 2, 0)
    wp3 = poly_fit(x, y0, 9, 0)
    wr1 = poly_fit(x, y0, 9, 0)
    wr2 = poly_fit(x, y0, 9, 10**5)
    wr3 = poly_fit(x, y0, 9, 10**9)
    y1 = eval_poly(wp1, xs)
    y2 = eval_poly(wp2, xs)
    y3 = eval_poly(wp3, xs)
    gen_plot(x, y0, xs, y1, y2, y3, "1st Order Poly", "2nd Order Poly", "9th Order Poly")
    y1 = eval_poly(wr1, xs)
    y2 = eval_poly(wr2, xs)
    y3 = eval_poly(wr3, xs)
    gen_plot(x, y0, xs, y1, y2, y3, "No Reg", "Medium Reg", "Large Reg")
```