Machine Learning Notes - Coursera Stanford Course

August 24, 2016

1 Week 1

1.1 Introduction

Supervised Learning: We are told the correct answer, and we base predictions off of those. Example: predicting house prices.

Unsupervised Learning: We are not told what each data point represents. The algorithm tries to find structure in the data. (Think clustering algorithms). Example: auto-clustering of news into sections regarding certain topics.

Classification Problems: predict discrete-valued outputs.
Regression Problems: predict continuous ie. real-valued outputs.

1.2 Linear Regression - Model and Cost Function

Model representation:

1. m = number of training examples
2. x = input variables
3. y = output variables

1.2.1 Univariate Linear Regression Hypothesis

$$h_\theta = \theta_0 + \theta_1 x$$

Note that this is a function of input variable x.

1.2.2 Cost Function

Cost function ie. squared error function:

$$J(\theta_1, \theta_2) = \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Note that this is a function of the parameters θ_1, θ_2. The goal is to minimize the cost function by tuning parameters θ_1, θ_2. We have in the denominator a $2m$ term to make the math cleaner, but having J without the 2 would yield the same result, as we are simply performing minimization. This cost function is the most common for regression problems.

1.2.3 Cost Function Intuition

We can use contour plots and surface plots in order to determine which values of θ_1 and θ_2 will cause a minimum of $J(\theta_1, \theta_2)$.

1.3 Gradient Descent

Algorithm, given $J(\theta_0, \theta_1)$:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a (local) minimum

Mathematically, we repeat the following until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

For linear regression:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}$$

"Batch" Gradient Descent - each step of the gradient descent uses all the training examples. Not the only form.

where α is the learning rate, (the size of the step taken).

Updates must occur simultaneously (do not use latest value of θ_0 to find θ_1).

1.3.1 Intuition

1. small α makes process slow
2. large α makes gradient descent overshoot the minimum - it may fail to converge or diverge
3. the learning rate does not need to be changed during gradient descent (smaller steps taken automatically by smaller derivative)

2 Week 2

2.1 Multivariate Linear Regression

2.1.1 Multiple Features

- n = number of features
- $x^{(i)}$ = input (features) of i^{th} training example
- $x_j^{(i)}$ = value of feature j in the i^{th} training example

$$h_\theta = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n = \theta^T x$$

For convenience, $x_0 = 1$, such that x can be treated as a vector of size n + 1. The parameters are stored in a vector θ which is also of that size.

2.1.2 Gradient Descent for Multiple Variables ($n \geq 1$)

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

2.1.3 Feature Scaling

Get every feature into approximately a $-1 \leq x_i \leq 1$ range; makes gradient descent converge faster (makes contour plots less skewed)
Mean normalization: replace x_i with $x_i - \mu_i$ to make features have approximately zero mean (do not apply to x_0)

$$x_j = \frac{x_j - \mu}{\sigma}$$

Instead of using σ, can use range instead.

2.1.4 Learning Rate

There is a learning rate at which $J(\theta)$ will monotonically decrease (ie decrease for each iteration). Debug algorithm by plotting $J(\theta)$ vs. number of iterations. Symptoms of a learning rate that is too large are: divergence, $J(\theta)$ that decreases and then increases repeatedly (looks like $-|sin(x)|+c$).

2.1.5 Polynomial Regression

Nonlinear polynomial regression can be done via substitution of a x^n term to a x term. Feature scaling becomes increasingly important. The range used in feature scaling must change accordingly (eg. range from 1-100 for an x term becomes 1-10,000 for an x^2 term).

2.2 Computing Parameters Analytically - Normal Equation

Using calculus, we take partial derivative of J with respect to the parameters θ allows for minimization of J with respect to the parameters.

$$\theta = (X^T X)^{-1} X^T y$$

X is the 'design matrix' and is of size $m \times (n+1)$. The first column of X is a vector of 1's. Each column thereafter contains the j^{th} feature of a given data point x^i. In other words, each row is just a 1 followed by the x-values associated with a given data point. The column vector y contains the output values, and is $m \times 1$ in size.

Feature scaling is not necessary for the normal equation method.

2.2.1 Gradient Descent vs Normal Equation

Gradient descent requires iterations and a learning rate, and works well even when n (the number of features) is large. The normal equation method becomes slow when n is very large, as $X^T X$ is an $n \times n$ matrix whose inversion is slow. For $n \geq 10,000$, opt for gradient descent.

2.2.2 Normal Equation Non-invertibility

Two causes:

1. Redundant features (linearly dependent features)
2. Too many features ie. $m \leq n$ such that too little data is available to fit all those features (results in non-invertible/singular matrix)
3. (solution is to delete some features or to use regularization)

2.3 Vectorization of Implementation

User-defined iterating routines are likely slower than library calls which are vectorized implementations. Example: performing dot product as matrix multiplication is faster than performing it via a summing for-loop. A vectorized implementation of gradient descent:

$$\theta := \theta - \alpha\delta$$

where θ and δ are size $n+1$ column vectors, and

$$\delta = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$$

where $x^{(i)}$ is a size $n+1$ column vector and $(h_\theta(x^{(i)}) - y^{(i)})$ is a real number.

3 Week 3

3.1 Classification

$y\epsilon\{0,1\}$, where 0 is the negative class, and 1 is the positive class.
Logistic Regression: classification algorithm that enforces $0 \leq h_\theta(x) \leq 1$

3.1.1 Hypothesis Representation

$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}} = P(y = 1|x;\theta)$
It is based on the Sigmoid/logistic function: $g(z) = \frac{1}{1+e^{-z}}$ The output of the hypothesis is the probability that y = 1 given an input x that is parameterized by θ.

3.1.2 Decision Boundary

Given a boundary (like predict 1 when $h(x)$ is greater than 0.5 and otherwise 0), it is easy to solve for the boundary value of $\theta^T x$. From there, it is easy to determine the values of x_i that result in a given outcome in terms of an inequality.

3.2 Logistic Regression Model

3.2.1 Cost Function

Since $h(x)$ is now a nonlinear function, using the cost function as previously defined would result in several local minimia, therefore it is unlikely for gradient descent to achieve the global minimum, which is only guaranteed to occur for convex functions. The following cost function gives us a convex cost function that is local minimum free, and is derived from the principle of maximum likelihood estimation (stats concept):

$$J(\theta) = \frac{1}{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases} = -ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$$

3.2.2 Gradient Descent

Again, use gradient descent to minimize cost function, and the update rule is identical to linear regression once partial derivatives of J are taken.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

However, the definition of $h_\theta(x)$ is now based on the sigmoid function rather than a polynomial. Use vectorized implementation using θ as a vector of the parameters and update them simultaneously:

$$\theta := \theta - \alpha \frac{1}{m} [\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}]$$

Can also use feature scaling make logistic regression run faster.

3.3 Advanced Optimization

Optimization algorithms to minimize $J(\theta)$:

1. Gradient Descent
2. Conjugate Gradient
3. BFGS
4. L-BFGS

The latter algorithms require no manual picking of α and often arrive at a solution much faster than gradient descent would. However, they are more complex. Use the fminunc (unconstrained minimization) function in MATLAB. You must write a function that computes the cost function given some θ vector and computes a vector known as the 'gradient' which stores the partial derivatives of J with respect to the parameters θ_i.

3.4 Multi-class Classification (any number of outputs)

One-vs-all (one-vs-rest) classification: given k classes, train k logistic regression classifiers where all data not in the k^{th} class is considered as being in the negative class. $h_\theta^{(i)}(x) = P(y = i|x;\theta)$ for each class $(i = 1, 2..k)$. To classify a unknown input x, pick the class i that maximizes $h_\theta^{(i)}(x)$. In other words, pick the classifier which thinks most enthusiastically that the new data point fits into that class.

3.5 The Problem of Over-fitting

Fitting data that appears to follow a square root function with a straight line is said to produce an output that is 'underfit' and has 'high bias'. The algorithm has a preconception that the data should be linear.
Fitting the same data with a quartic polynomial may result in all training examples being passed through well, but the algorithm has 'overfit' the data and has 'high variance'. We don't have enough data to constrain this high order polynomial.
Overfitting: the algorithm makes accurate predictions for examples in the training set (the cost function approaches 0), but it does not generalize well to make accurate predictions on new, previously unseen examples due to too many features and not enough data.

3.5.1 Addressing Overfitting

1. Reduce number of features (either manually or through model selection algorithm)

2. Regularization

3.5.2 Regularization

Penalize parameters $\theta_1 ... \theta_n$ to make them very small - modify cost function as follows:

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

where λ is the regularization parameter.

The result is a simpler hypothesis that is less prone to overfitting, as minimizing this function results in much smaller θ values. The left half satisfies the goal of fitting the training set, and the right half satisfies the goal of keeping parameters small. If λ is too large, then under-fitting will occur as essentially θ_0 will remain in the hypothesis, with very small contributions from the other parameters.

3.5.3 Regularized Linear Regression

For gradient descent, repeat:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\theta_j := \theta_j(1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \text{ for } (j = 0, 1, 2, 3, ..., n)$$

For the normal equation:

$$\theta = (X^T X + \lambda I')^{-1} X^T y$$

where I' is the identity matrix with size $(n + 1) \times (n + 1)$ where the top left entry is 0. Using regularization with λ>0 makes $X^T X$ always invertible.

3.5.4 Regularized Logistic Regression

For gradient descent, repeat (same as linear regression) :

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\theta_j := \theta_j(1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \text{ for } (j = 0, 1, 2, 3, ..., n)$$

The partial derivatives are now:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j$$

4 Week 4

4.1 Neural Networks

4.1.1 Model Representation

Neuron model - logistic unit: based on sigmoid (logistic) activation function. ”Parameters” θ can also be known as ”weights”. We often omit the x_0 (bias unit) because its value is always 1.
Neuron network: neurons strung together.

Layers:

1. Layer 1: input layer - x value
2. Layer 2: hidden layer - values you don't observe in the training set (could be more than jut one layer; there can also be a bias unit here)
3. Layer 3: output layer - y value

Terminology:

1. $a_i^{(j)}$: ”activation” of unit i in layer j, where ”activation” means the output value
2. $\theta^{(j)}$: matrix of weights controlling function mapping from layer j to layer $j + 1$.
3. L = total number of layers in network
4. K = number of output units (for multi-class classification problems this is the number of classes)

If a network has s_j units in layer j, s_{j+1} units in layer $j + 1$, then $\theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$. These units do not include the bias unit.

Forward propagation (vectorized implementation): You're basically just using logistic regression to get an output from the last hidden layer to the output layer, but instead of using the inputs you're using the activations. The activations (in the hidden layers) are learned as a function of the input. You're no longer constrained to just using the features provided, and can have complex nonlinear hypotheses.

4.2 Applications

Digital logic can be computed given that the sigmoid function reaches 0.99 at 4.6 and 0.01 at -4.6. These can be rounded to binary values of 1 and 0. You can write mathematical functions that then just output values greater than these tolerances to make pseudo digital logic. XNOR is computed by ORing the AND of two inputs with the NAND of two inputs. This requires one hidden layer.

4.2.1 Multiclass Classification

Example: Handwriting detection
Extension of one-vs-all method. In the training set $x^{(m)}, y^{(m)}$, represent $y^{(m)}$ as a column vector of size equivalent to the number of classes. The values in the vector are 0 for all classes except the class to which the data point belongs. Likewise, the output $h_\theta(x)$ is a vector of similar size and characteristics. However, the values will be approximately 0 or 1, not necessarily exact due to the asymptotic nature of the sigmoid function.

5 Week 5

5.1 Neural Network Cost Function

Expands on regularized logistic regression cost function.

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_\theta(x^{(i)}))_k + (1-y_k^{(i)}) log(1-h_\theta(x^{(i)}))_k + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ji}^{(l)})^2]$$

The hypothesis $h_\theta(x)\epsilon\mathbb{R}^K$ and $(h_\theta(x))_i = i^{th} output$.

We do not sum over the θ terms where $i = 0$ because those multiply into the bias units - we don't want to regularize them.

5.1.1 Gradient Computation

In order to minimize the cost function using an algorithm like gradient descent, we need to compute the gradient of J. Given one training example (x, y), we start with forward propagation. A vectorized implementation where there are 2 hidden layers (total 4 layers):

$$a^{(1)} = x$$
$$z^{(2)} = \theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)})$$
$$\text{add } a_0^{(2)} \text{ bias unit}$$
$$z^{(3)} = \theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)})$$
$$\text{add } a_0^{(3)} \text{ bias unit}$$
$$z^{(4)} = \theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\theta(x) = g(z^{(4)})$$

Now we do back propagation:

Define $\delta_j^{(l)} = =$ "error" of node j in layer l (based on its activation). There is no $\delta^{(1)}$ because the input layer has no error associated with it. Depending on the implementation, the δ values corresponding to the bias units ($\delta_0^{(l)}$) can be meaningless as we generally don't change the value of +1 assigned to the bias units. Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i)$

$$\delta^{(4)} = a^{(4)} - y$$

where each value is a vector of size equal to the number of outputs.

$$\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)}. \times g'(z^{(3)})$$
$$==$$
$$\delta^{(3)} = a^{(3)}. \times (1 - a^{(3)})$$

$$\delta^{(3)} = (\theta^{(2)})^T \delta^{(3)}. \times g'(z^{(2)})$$
$$==$$
$$\delta^{(2)} = a^{(2)}. \times (1 - a^{(2)})$$

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = a_j^{(l)} \delta_i^{(l+1)}$$

if we set $\lambda = 0$.

Formally, given training set $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$:

1. Set $\triangle_{ij}^{(l)} = 0$ for all l, i, j.
2. For i = 1 to m:
   (a) Set $a^{(1)} = x^{(i)}$
   (b) Perform forward prop to compute $a^{(l)} forl = 2, 3, ...L$
   (c) Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
   (d) Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$
   (e) $\triangle_{ij}^{(l)} = \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} == \triangle^{(l)} = \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
3. $D_{ij}^{(l)} = \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \theta_{ij}^{(l)}$ if $j \neq 0$
4. $D_{ij}^{(l)} = \frac{1}{m} \triangle_{ij}^{(l)}$ if $j = 0$

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)}$$

5.1.2 Intuition on Back Propagation

https://www.coursera.org/learn/machine-learning/lecture/du981/backpropagation-intuition

5.2 Back Propagation in Practice

Using advanced optimization algorithms like fminunc in MATLAB requires use of initial guess and gradient in the form of size $n + 1$ vectors. With a neural network, we now deal with θ and gradient D matrices that must be unrolled into vectors.

Learning algorithm (given L = 4):

1. Have initial parameters $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$
2. Unroll to get initialTheta to pass to fminunc(@costFunction, initialTheta, options) using [Theta1(:); Theta2(:), Theta3(:)]
3. function [jval, gradientVec] = costFunction(thetaVec)
   (a) From thetaVec, get $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$ using reshape command
   (b) Use forward/backward propagation to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\theta)$
   (c) Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\theta)$ to get gradientVec

5.2.1 Gradient Checking

Even with subtle bugs, neural networks can seem like they're working in that the cost function continues to be minimized. But in reality, they do not necessarily find the minimum. Gradient checking is a means of detecting such bugs.

Use a central difference to numerically approximate the gradient, using a step of 10^{-4}.

Given parameter unrolled vector θ:

$$\frac{\partial}{\partial\theta_n}J(\theta) \approx \frac{J(\theta_1, \theta_2, ...\theta_n + \epsilon) - J(\theta_1, \theta_2, ...\theta_n - \epsilon)}{2\epsilon}$$

Code:

```
for i = 1:n
        thetaPlus = theta;
        thetaPlus(i) = thetaPlus(i) + EPSILON;
        thetaMinus = theta;
        thetaMinus(i) = thetaMinus(i) - EPSILON;
        gradientApproximation(i) = (J(thetaPlus)     J(thetaMinus))
end
```

Recall that back propagation generates DVec (which are the same partial derivatives of the cost function with respect to our params). Compare our numerically computed derivatives with the DVec value. If these two ways give similar answers (up to a few decimals), our implementation of back prop is likely correct. When actually using back prop for training the classifiers, turn off gradient checking as it is computationally expensive.

5.2.2 Initialization of θ Parameters

Initializing $\theta_{ij}^{(l)}$ to be 0 does not work like in regression. After each update, the parameters corresponding to the inputs going into each of the hidden units are equal to each other even through iteration of gradient descent. This limits what kind of functions the neural network can compute. This problem is called symmetric weighting.
Use random initialization to allow for symmetry breaking. Initialize each $\theta_{ij}^{(l)}$ to a (different) random value in $[-\epsilon, \epsilon]$ where ϵ is some value.

5.3 Putting it all Together

Training a NN:

1. Pick a network architecture (connectivity pattern)
2. Number of input units = dimension of features $x^{(i)}$
3. Number of output units = number of classes
4. Reasonable Default: use 1 hidden layer (most common), or if you have more than one hidden layer, have the same number of hidden units in every layer (usually the more the better but more expensive) and comparable to the number of inputs or some multiple of it?
5. Randomly initialize weights to small values near 0
6. Implement forward propagation to get $h_\theta(x^{(i)}$ for any $x^{(i)}$
7. Implement code to compute cost function $J(\theta)$
8. Implement back prop to compute partial derivatives $\frac{\partial}{\partial\theta_{jk}^{(l)}}J(\theta)$
9. Use a for loop, iterating of training examples, performing forward prop and backward prop on each training example $(x^{(i)}, y^{(i)}$ to get $a^{(l)}$ and $\delta^{(l)}$ terms for $l = 2, ..., L$. In the for loop, compute $\triangle^{(l)} = \triangle^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
10. Outside the for loop, compute partial derivatives of J, taking into account regularization term λ.
11. Use gradient checking to compare gradients computed using back prop vs. a numerical estimate using central differences of J
12. Disable gradient checking
13. Use gradient descent or advanced optimization method with back prop (which computes the partial derivatives) to try to minimize J as a function of parameters θ. Note that J is non-convex in this case, so you're not guaranteed to get a global optimum. Local optima tend to be good enough though.

Idealized output y is a column vector with a 1 in one row and zeros in all the rest.

6 Week 6

If the hypothesis is not very good at making predictions on new data, there are some options:

1. Getting more training examples (fixes high variance problems)
2. Trying a smaller set of features (choosing them more carefully to prevent overfitting) (fixes high variance)
3. Getting additional features (fixes high bias)
4. Try adding polynomial features (fixes high bias)
5. Decreasing λ for regularization (fixes high bias)
6. Increasing λ for regularization (fixes high variance)

Don't waste time using gut feelings to pick a learning algorithm. Use a machine learning diagnostic: a test you can run to gain insight into what is or isn't working with a learning algorithm, and gain guidance as to how to best improve its performance.

6.1 Evaluating a Learning Algorithm

6.1.1 Evaluating Hypothesis

You can plot in 2D to check for overfitting, but you can't do this with more features. Solution: split your training data into two sets: the training set, and the test set. The training set should contain a random 70% of your training data, and the test set should contain the balance 30%. Learn the parameters θ from the training set (ie. run minimization on J(θ), then compute the test set error. For linear regression, this means we compute (note no regularization):

$$J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

For logistic regression, we compute:

$$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y_{test}^{(i)} log h_\theta(x_{test}^{(i)}) + (1 - y_{test}^{(i)}) log h_\theta(x_{test}^{(i)})$$

Or, we can use a number that's easier to understand - the 0/1 misclassification error which is just the ratio of how many are classified correctly vs incorrectly.

$$err(h_\theta(x), y) = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0.5 \text{ and } y = 0, \text{ or if } h_\theta(x) < 0.5 \text{ and } y = 1 \\ 0 & \text{otherwise} \end{cases}$$

Test error $= \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\theta(x_{test}^{(i)}), y_{test}^{(i)})$

6.1.2 Model Selection - How to Choose the Degree of Polynomial to Fit

Take another parameter, the degree of the polynomial, d, (in addition to the θ parameters) that we must determine. Take each model, fit it to the training set, and compute the test set error, and pick the one with the lowest test set error. But how well does this model generalize? We picked the value of d that gave us the best possible performance on the test set is likely to be an overly optimistic estimate of generalization error. Because we fit the parameter d based on the test set, it is no longer fair to evaluate the hypothesis on this test set. (You can't report the generalization error accurately by using the test set.)
Solution: use 60% of data as training set, 20% as cross-validation set, and 20% as the test set. Compute $J(\theta)$ for each of the sets. Minimize $J(\theta)$ w.r.t. θ for each degree of model. Compute the cross-validation set error, and pick the hypothesis with the lowest cross-validation error. Now, estimate the generalization error using the test set error.

6.2 Bias vs. Variance

6.2.1 Diagnosing Bias vs Variance

Low degree polynomials result in high training error and high cross validation error ($J_{cv}(theta) \approx J_{train}(\theta)$). This is an underfitting/high bias problem.
High degree polynomials result in low training error and high cross validation error ($J_{cv}(\theta) >> J_{train}(\theta)$). This is a overfitting/high variance problem.
Good fits will have intermediate training error and intermediate cross-validation error.

6.2.2 Auto-selecting λ

Have some range of λ you want to fit, and increment in multiples of 2 (0, 0.01, 0.02, 0.04...). For each, minimize the cost function $J(\theta)$ (which includes

the regularization term) to solve for the parameter vector θ. In this case, we define separately $J_{train}, J_{cv}, J_{test}$ to not include the regularization term. Use the cross-validation set fit with the different parameter vectors, and compute the cross-validation error. Pick the one with the smallest cross-validation error. Report the test set error as J_{test}) as a measure of how well the model will generalize.
Consider J_{train} and J_{cv} (the ones without regularization). Low λ results in overfitting (high variance) and thus low J_{train} and high J_{cv}. Large λ results underfitting (high bias) and thus high J_{train} and high J_{cv}. Intermediate values of λ result in intermediate J_{train} and small J_{cv}. Plotting these allows for confidence in picking λ.

6.2.3 Learning Curves

High bias problems (low degree polynomial): as you increase the training set size, J_{cv} decreases up to a certain point but plateau's relatively quickly. J_{train} will increase up to a certain point and plateau. Both are high values and are approximately the same. Getting more training data will not, by itself, help much.
High variance problems (high degree polynomial with small λ): as you increase the training set size, the training set error increases as it becomes harder to fit lots of data points. Cross-validation error decreases as you increase the training set size. There is a large gap because the training error and cross-validation error. The two curves converge to one another, so getting more training data is likely to help - cross validation error will continue to go down, which is good.

6.2.4 Relating to Neural Networks

Small networks have fewer parameters - they are cheaper but prone to underfitting. Large networks have more parameters (either through more hidden layers or more units in a given hidden layer) - they are more expensive and are more computationally expensive. Use regularization to deal with overfitting. To pick the number of layers, you can use training, cross-validation, and test sets, and see which performs best on the cross-validation set.