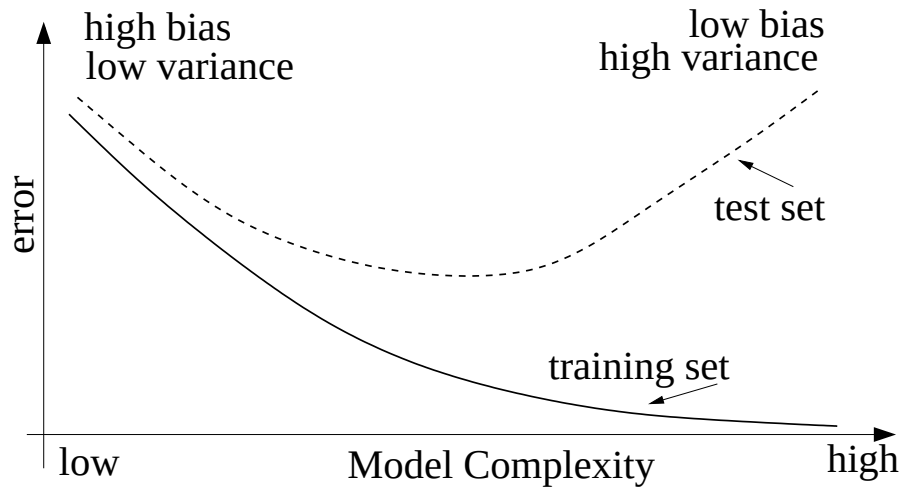
Lecture 11:

Overfitting and Capacity Control

November 21, 2006

Typical Behaviour



Generalization, Overfitting, Underfitting

- The *generalization* of a machine learning method is the performance (classification, regression, density estimation) on test data, not used for training, but drawn from the same (joint) distribution as the training data. Often, our real goal is to get good generalization.
- When our model is too complex for the amount of training data we have, it memorizes parts of the noise as well as learning the true problem structure. This is called *overfitting* or *model variance*.
- When our model is not complex enough, it cannot capture the structure in our data, no matter how much data we give it. This is called *underfitting* or *model bias*.
- An *unbiased* model is one which given enough data will eventually learn the correct model. But if all we care about is generalization and we only have a finite amount of data then we should be happy to introduce a little bit of bias if it reduces the variance a lot.

Model Selection: Bias-Variance Tradeoff

- Amongst all the models we can think of (and can train), we need to select one to use for making predictions. (For now, at least. Later we'll see how to combine the predictions of several models.)
- What basic problems are we trying to avoid with model selection?
- Overfitting: if we chose a model that is too complex, it will overfit to the noise in our training set. Another way of saying this is that the machine we end up with is very sensitive to the particular training sample we use. The model has a lot of *variance* across training samples of a fixed size.
- Underfitting: if we chose a model that is not complex enough, it cannot fit the true structure, and so no matter what training sample we use there is some error between the true function and our model approximation. The model has a lot of *bias*.
- Intuitively, we need the right balance. How can we formalize this?

Bias-Variance Decomposition of Squared Error

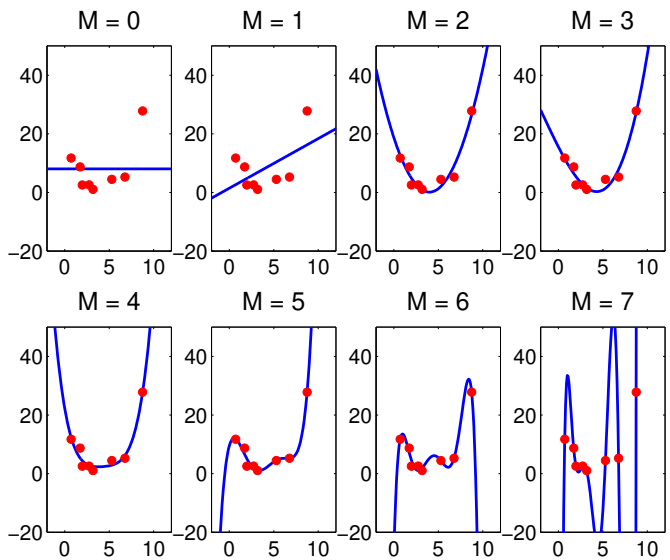
- Let us consider a supervised learning setup (scalar for now), with random noise (uncorrelated to inputs/outputs) and squared error:

$$y = g(x) + \text{noise} \qquad \text{true function}$$
$$\hat{y} = f(x) \qquad \text{our prediction}$$
$$\text{error} = (y - \hat{y})^2$$

- Consider the expected error at a single test point x_0, averaged over all possible training sets of size N, drawn from the joint distribution over inputs and outputs $p(x, y) = p(x)p(y|x)$.

$$\begin{aligned} e(x_0) &= \langle (y_0 - \hat{y}_0)^2 \rangle \\ &= \langle (g(x_0) + \epsilon_0 - \hat{y}_0)^2 \rangle \\ &= \langle \epsilon_0^2 \rangle + (\langle f(x_0) \rangle - g(x_0))^2 + \langle f(x_0) - \langle f(x_0) \rangle \rangle^2 \\ &= \sigma^2 + (\mathrm{mean}[f(x_0)] - g(x_0))^2 + \mathrm{var}[f(x_0)] \\ &= \text{Unavoidable Error} + \text{Bias}^2 + \text{Variance} \end{aligned}$$

Example: polynomial fitting



Model Selection & Performance Estimation

- Model Selection: out of a set of models (or continuum of model complexity), choose the model which will perform the best on future test data.
- Model Assessment: for the selected model, estimate its generalization error on new data.
- If we have lots of data, these two problems can be solved by dividing our data into 3 parts:
  - Training Data – used to train each model
  - Validation Data – used to measure performance of each trained model in order to select the best model
  - Assessment Data – used only once, on the final selected model, to estimate performance on future test data
- Typical split is 60% training, 20% validation, 20% assessment. So that's it, are we done?

Approximations when Data is Limited

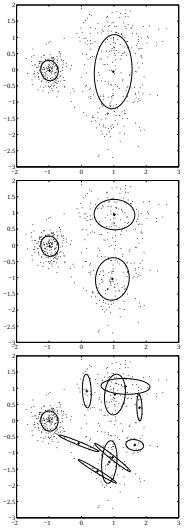
- Often, we don't have enough data to make 3 separate and reasonably sized training, validation and assessment sets.
- If we don't have very much data, we can try to *approximate* the results of validation and assessment.
- Two basic approaches for finite datasets:
  - Analytic methods: derive algebraic expressions which try to approximate the test error, e.g. using a complexity penalty which scales as the ratio between the number of parameters in the model and the number of training cases.
    Examples: BIC, AIC, MDL, VC-dimension.
  - Sample-recycling methods: try to estimate the test error computationally, using the same data that we trained on.
    Examples: jackknife,cross-validation, bootstrap.

Regularization and Capacity Control

- How can we improve generalization? Reduce either bias or variance!
- One obvious way: use more training data, and commensurately more complex models. If we scale up model complexity slowly enough, using more data reduces *both* bias and variance.
- But what if we can't get more data?
  Our goal should be to reduce variance (by using simpler models) while not increasing our bias too much (by not using *too* simple a model). We should not force ourselves to use unbiased (Bias=0) models, because we only really care about the sum Bias^2+Variance.
- We need a knob to control this tradeoff (e.g. by discretely constraining model *structure* or by continuously *regularizing* model complexity or smoothness) and a way to set the knob (i.e. decide on the right tradeoff balance).

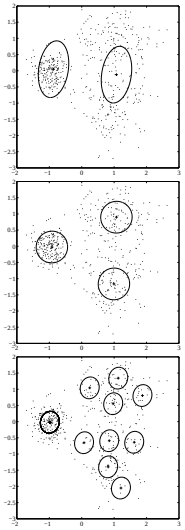
One Kind of Knob: Model Structure

- We can control the structure of our model as a way of determining its complexity.
- This includes the number of hidden units in a multilayer perceptron neural network, the number of clusters in a mixture of Gaussians or K-Means model, the number of experts in a mixture of experts, the number of latent factors in FA/PCA.
- Model structure also includes sparsity, i.e. specifying which weights are zero and which are nonzero. This is useful in diagonal-covariance Gaussian noise estimates, constrained HMM transition matrices, variable subset selection for regression, local-receptive field vision networks, etc.



Another Regularizer: Parameter Sharing

- Another way to control model complexity is to *tie together* or *share* various parameters. This allows us to have a complete model structure but not have to estimate a huge number of free parameters.
- This is used in mixtures of factor analyzers, to jointly estimate the sensor noises, in mixtures of Gaussians to jointly estimate cluster covariances (e.g. Fisher's discriminant is a class-conditional Gaussian model with shared covariances), in vision neural networks to learn translation-invariant receptive fields, etc.



More Regularization: Method with Local Support

- Yet another way to control model complexity is to restrict the amount of training data that can be used to predict the output on any new test case.
- Each test case prediction is only allowed to use a small fraction of the training data, typically the training points whose inputs are close to the input of the test case.
- This is known as a *locally weighted* method, e.g. nearest neighbour classification, Parzen density estimation, locally weighted regression.
- Local methods are related to "semi-parametric" models, which try to use the reservoir of training data to store most of the bits of their capacity, and only have a few "metaparameters" which control how that reservoir is used at test time.
  Examples: K-NN classifier, locally-weighted regression, Parzen window density estimators

Regularization by adding a penalty term

- Instead of discrete complexity controls, it is often useful to have a continuous range of complexity, set by one or more real valued "fudge-factors" or "hyperparameters".
- The most common way to achieve this is to add a "penalty term" to the cost function (error, log likelihood, etc) which measures in a quantitative and continuous way how complex/simple our model is:

$$\text{cost}(\theta) = \text{error}(\text{data}, \theta) + \lambda \text{penalty}(\theta)$$

- We can then weight this penalty term relative to the original error (or likelihood) and minimize the resulting penalized cost.
- The larger the penalty weight λ, the simpler our model will be.
- How can we set λ? If we have lots of data, we can use the performance on a held out set of validation examples to determine the correct value of the penalty weight.

Example Penalty: Weight Decay

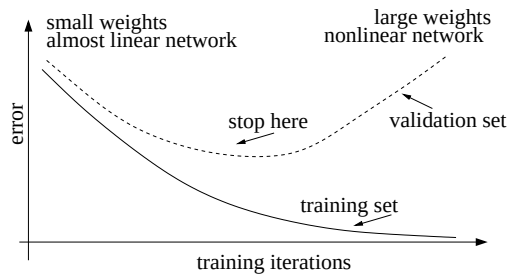
- The most common regularization is the ridge regression penalty (weight decay) which discourages large parameter values in generalized linear models:

$$\text{cost}(\theta) = \text{error}(\text{data}, \theta) + \lambda \sum_k \theta_k^2$$

- This says: "don't use big weights unless they really help to reduce your error a lot". Otherwise, there is nothing to stop the model from using enormous positive and negative weights to gain a tiny benefit in error.
- Remember: on a finite training set, there will always be some tiny, accidental correlation between the noise in the inputs and the target values.

Example Penalty: Early Stopping

- Another approach to regularization in models whose complexity grows with training time is to stop training early.
- This works quite well in neural networks, since small weights mean that the network is mostly linear (low complexity) and it takes a while for the weights to get bigger, giving nonlinear networks (high complexity). Essentially a penalty equal to # training iterations.
- A validation set can be used to detect stopping point.



Example Penalty: AIC/BIC

- There are also asymptotic "parsimony criteria" derived from theoretical assumptions which attempt to penalize model complexity in a way that would result in a good estimate of test error if we had an infinite amount of data.
- The two most popular are the *Akaike Information Criterion* (AIC) and the *Bayesian Information Criterion* (BIC) both of which apply to probabilistic models:

$$AIC = \bar{L} - P/N$$

$$BIC = \bar{L} - \frac{1}{2} P \log N/N$$

where $\bar{L}$ is the average log likelihood of the training points under the model, P is the number of "free" parameters, and N is the number of "independent" datapoints.

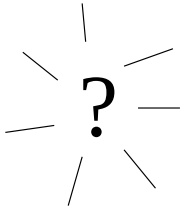
- Asymptotically, AIC performs like leave-one-out cross validation and BIC performs like carefully chosen K-fold cross validation.

Issues in High Dimensions

- As the dimensionality of the input and output variables in a learning problem grows, the naive approach to most problems requires *exponentially* more training data to get good generalization.
- This is known as the "curse of dimensionality". In general, it affects density estimation, regression, clustering and classification.
- Example: if we divide pixel intensities into L levels, and we examine d pixels, there are L^d possible images. A learning algorithm which just memorizes (finds all exact occurrences of a test image in its training set) needs an exponential amount of data to have even one example of each image.
- We must always regularize to generalize, say by constraining our models (e.g. Naive Bayes) or by doing dimensionality reduction on the input (e.g. with FA/PCA). But we can also try to adjust our training procedure (as opposed to our models) in order to improve performance.

Potential Pitfalls

- Several things can cause us trouble when we are trying to get good generalization from a learning algorithm:
  - we might not have enough training data to learn target concept
  - our testing might not *really* be from the same distribution as our training data
  - our model might not be complex enough, so it underfits
  - our model might be too complex, so it overfits
  - we have too much training data to run the algorithm in a reasonable amount of time or memory
- Sounds hopeless!
  What can we do?



Practical Solutions

- Several simple ways to good generalization in practice.
- Use model classes with flexible control over their complexity. (e.g. ridge regression, mixture models)
- Employ regularization (capacity control) and (cross) validation,to match model complexity with the amount of data available.
- Build in as much reliable prior knowledge as possible, so algorithms don't have to waste data learning things we already know.
- Use cross-validation/bootstrap to make efficient use of limited data.
- Use subsampling or sparse methods to speed up algorithms on huge training sets, and keep them fast and small at test time.

Cross Validation (CV) / Jackknife

- Instead of setting aside a separate validation set, we can leave out part of our data, train on the rest, measure errors on the part we left out, and then repeat, leaving out a different bunch of data.
- If we break our data into K equal groups, and cycle through them, leaving out one at a time, this is known as K-fold cross validation.
- The cost function is the average training error across all folds, and our estimate of the validation error is the average of all validations.
- Our validation error estimates are biased, because the same data is also used to train the model during the other folds of cross validation. But we don't waste any data.
- If we leave out only one data point at a time, this is *leave-one-out cross validation* (LOO), sometimes called the jackknife estimator.
- LOO is my favourite way to set everything!

Potential Problems with Cross-Validation

- CV is awesome and it can be used on clustering, density estimation, classification, regression, etc.
- But intensive use of cross-validation can overfit, if you explore too many models, by finding a model that accidentally predicts the whole training set well (and thus every leave-one-out sample well).
- CV can also be very time consuming if done naively.
- Often there are efficient tricks for computing all possible leave-one-out cross validation folds, which can save you a lot of work over brute-force retraining on all N possible LOO datasets.
- For example, in linear regression, the term $(\sum_{n\neq\ell} \mathbf{x}_n\mathbf{x}_n^\top)^{-1}$ which leaves out datapoint ℓ can be computed using the matrix inversion lemma: $(\sum_n \mathbf{x}_n\mathbf{x}_n^\top - \mathbf{x}_\ell\mathbf{x}_\ell^\top)^{-1}$.
- This is also true of the Generalized Cross Validation (GCV) estimate of Golub and Wahaba. (see extra readings)

Bootstrap (Efron 1979)

- A similar idea to CV, in that it re-uses samples to generate a large number of datasets. Both CV and bootstrap try to use computational power in situations where theoretical calculations are not possible. (e.g. standard error of mean is easy to derive, but what about standard error of median or correlation coefficient?)
- In the bootstrap, we generate datasets by *sampling the original training data with replacement* to get a set the same size as the original. (If we do this B times, this is a B-fold bootstrap.)
- We can then measure our statistic of interest (e.g. classification performance) using the examples left out of each bootstrap sample (if any). The average of these bootstrap estimates is a conservative approximation of our validation estimate. (A better approximation involves blending between this bootstrap average and the training error when trained on the whole dataset.)

Model Averaging

- One last way to reduce variance, while not affecting bias too severely, is to average together the predictions of a bunch of different models.
- These models must be different in some way, either because they were trained on different subsets of the data, or with different regularization parameters, different local optima, or something.
- When we average them together, we would like to weight more strongly the models we believe are fitting the data better.
- Such systems are often called *committee machines*.
- Really, this is just a weak form of Bayesian learning.
  MAP = estimate of mode of posterior over models
  Bagging (next class) = estimate of mean of posterior over models
  BIC/AIC = estimates of correct predictive distribution

Bayesian Programme

- In Bayesian learning, we think of the parameters as random variables, just like the data.
- We have a prior over parameters, $p(\theta)$ and a model of how data can be generated given any particular set of parameters: $p(\text{data}|\theta)$.
- Our goal is to do *parameter inference*, i.e. to infer the posterior over parameters: $p(\theta|\text{data})$.
- When we make predictions, we should *integrate over all possible parameter settings*, weighting each one by its posterior.
- This is the ultimate in model averaging.
- The marginal likelihood, $\log p(\text{data}) = \log \int_{\theta'} p(\text{data}|\theta')p(\theta')$ is what tells us how well our model fits the data.
  It is sometimes called the "evidence".
- Much more to say on this topic.
  See Mackay, Neal, Ghahramani, Bishop.

Regularization as Parameter Priors

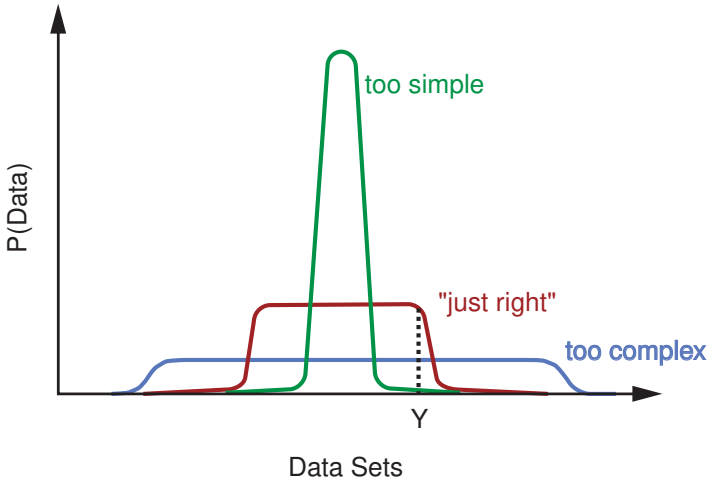
- We can think of the penalty term in regularization as being the logarithm of prior probabilities on our parameters.
- We can think of the error term as being the logarithm of the probability of the data given the parameters.
- Then minimizing the regularized error is equivalent to minimizing the posterior over parameters give the data and our priors.

$$\begin{aligned}\log p(\theta|\text{data}) &= \log p(\text{data}|\theta) + \log p(\theta) - \log p(\text{data}) \\ &= \text{error}(\text{data}, \theta) + \lambda\text{penalty}(\theta) + \text{constant}\end{aligned}$$

where $\log p(\text{data}) = \log \int_{\theta'} p(\text{data}|\theta')p(\theta')$.

- So the Bayesian method includes all regularization methods as a special case, if you chose a prior over parameters which is $p(\theta) \propto \exp(\lambda\text{penalty}(\theta))$.
- Key question: what's the chance of drawing some parameters from the prior and then, using those parameters, generating the data?

Ockham's Razor



We want to use the simplest model which explains the data well.
[A now famous figure, first introduced by Mackay.]

Meta-Learning

- The idea of meta-learning is to come up with some procedure for taking a learning algorithm and a fixed training set, and somehow repeatedly applying the algorithm to *different* subsets (weightings) of the training set or using *different* random choices within the algorithm in order to get a large ensemble of machines.
- The machines in the ensemble are then *combined* in some way to define the final output of the learning algorithm (e.g. classifier)
- The hope of meta-learning is that it can "supercharge" a mediocre learning algorithm into an excellent learning algorithm, without the need for any new ideas! (Details next class.)
- There is, as always, good news and bad news....
  - The Bad News: there is (quite technically) No Free Lunch.
  - The Good News: for many real world datasets, meta learning works well because its implicit assumptions are often reasonable.

No Free Lunch

- David Wolpert and others have proven a series of theorems, known as the "no free lunch" theorems which, roughly speaking, say that *unless you make some assumptions* about the nature of the functions or densities you are modeling, no one learning algorithm can *a priori* be expected to do better than any other algorithm.
- In particular, this lack of clear advantage includes any algorithm and any meta-learning procedure applied to that algorithm. In fact, "anti-cross-validation" (i.e. picking the regularization parameters that give the *worst* performance on the CV samples) is a priori just as likely to do well as cross-validation. Without assumptions, random guessing is no worse than any other algorithm.
- So capacity control, regularlization, validation tricks and meta-learning (next class) cannot *always* be successful.

- A key issue here is the difference between test error on a test set drawn from the same distribution as the training data (may contain duplicates) and *out of sample* test error.
- Remember back to the first class: learning binary functions.
  No assumptions == no generalization on out of sample cases.
- The only way to learn is to wait until you have seen the whole world and memorize it.
- Luckily, we *can* make some progress in real life.
- Why? Because the assumptions we make about function classes are often (partly) true.