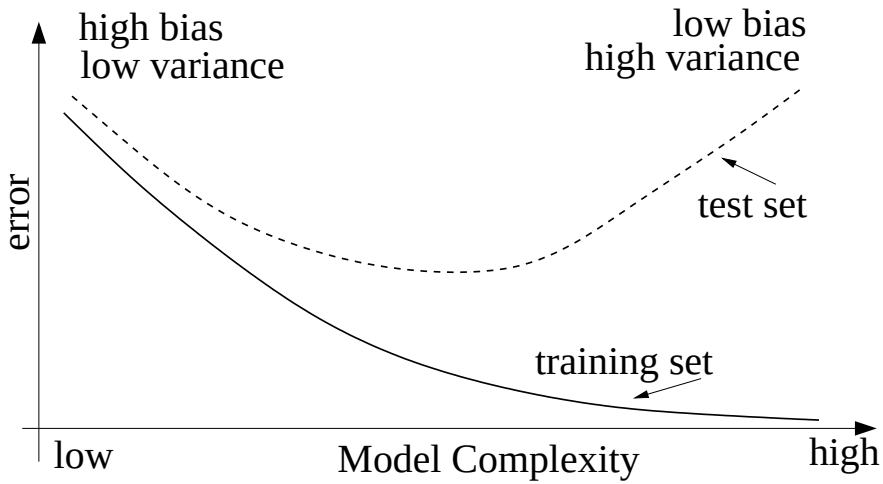
Lecture 11:

Overfitting and Capacity Control

Sam Roweis

November 20, 2002

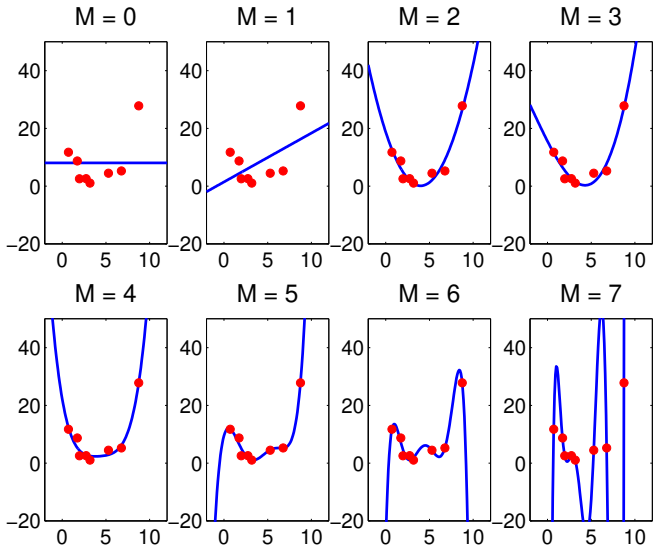
Typical Behaviour



Generalization, Overfitting, Underfitting

- The *generalization* of a machine learning is the performance on test data, not used for training, but drawn from the same (joint) distribution as the training data.
  Our ultimate goal is to get good generalization.
- When our model is too complex for the amount of training data we have, it learns (or memorizes) parts of the noise as well as 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*.

Example: polynomial fitting



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 properly
  - we have too much training data to run the algorithm in a reasonable amount of time or memory
  - 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
- Sounds hopeless! What can we do?

Two goals

- 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.

Practical Solutions

- There are several things we can do to get good generalization in practice.
- Use model classes with continuous control over their complexity.
- Employ regularization or capacity control to match this complexity with the amount of data available.
- Build in as much reliable prior knowledge as possible so the algorithm doesn't have to waste data learning things we already know.
- Use cross validation to make efficient use of limited data.
- Use subsampling or sparse methods to speed up algorithms on huge training sets.

Data Scarcity

- 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:
  - Analytic methods: derive complexity based estimators which try to approximate the test error.
    Examples: BIC, AIC, MDL, VC-dimension.
  - Sample-recycling methods: try to estimate the test error using the same data that we trained on.
    Examples: cross-validation, bootstrap.

Issues in High Dimensions

- As the dimensionality of the input and output variables in a learning problem grows, the naive approach to the problem requires *exponentially* more training data to get good generalization.
- This is known as the "curse of dimensionality".
- Example: if we divide pixel intensities into L levels, and we examine d pixels, there are L^d possible images. A learning algorithm which attempts to generalize by finding all occurrences of a test image in its training set needs an exponential amount of data to have even one example of each image.
- In general, this affects density estimation, regression, clustering and classification.
- Two approaches: constrained models (e.g. Naive Bayes) or dimensionality reduction on the input (e.g. PCA).

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:

$$y = g(x) + \text{noise}$$
$$\hat{y} = f(x)$$
$$\text{error} = (y - \hat{y})^2$$

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

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

Bias-Variance Tradeoff

- What's the basic problem 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 to formalize this?

Regularization and Capacity Control

- How can we improve generalization?
- One obvious way: use more training data, and commensurately more complex models. If we scale up model complexity at the right rate, this 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).
- This tradeoff can be implemented in one of two ways: by discretely constraining model *structure* or by continuously *regularizing* model complexity or smoothness.

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.
- 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.

Local Methods

- 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.

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.

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 "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 or 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, the simpler our model will be.

Example: Weight Decay

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

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

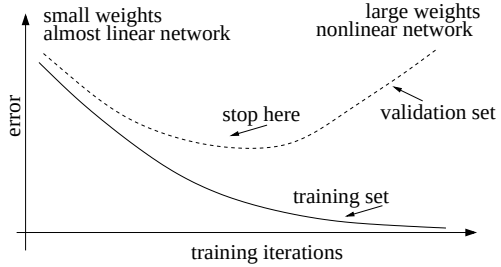
- This says: "don't use big weights unless they really help to reduce your error a lot". Otherwise, there is nothing to stop the model from using enormous positive and negative weights to gain a tiny benefit in error.

Cross Validation (CV)

- 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 all, leaving one out 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*, or LOO-CV.
- My favourite way to set everything!

Example: 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).
- A validation set can be used to detect stopping point.



Efficient Generalized Cross-Validation

- CV is awesome and it can be used on clustering, density estimation, classification, regression, etc.
- But it can 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 reading)

Boostrap

- A similar idea to CV, in that it re-uses samples to generate a large number of datasets.
- But now, 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 the performance on the original training set, to get a validation estimate and also some estimates of the variance of our predictions, parameters, and errors.

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 often called the "evidence".
- Much more to say on this topic.
  See Mackay, Neal, Ghahramani, Bishop.

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, or something.
- Such systems are often called *committee machines*.
- Really, this is just a weak form of Bayesian learning.

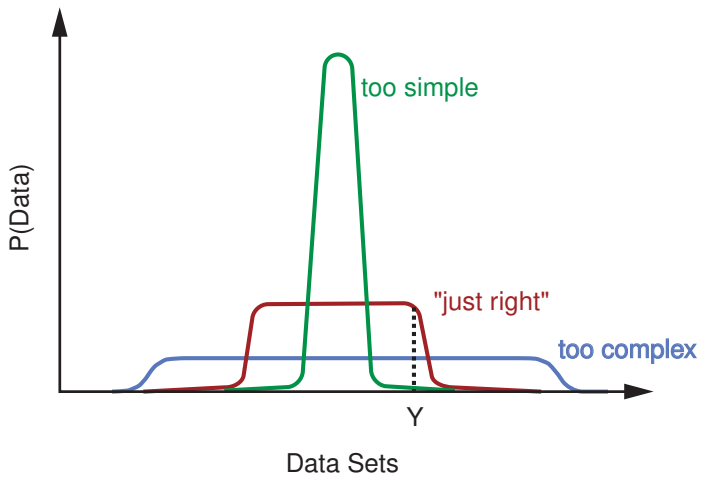
Regularization as Parameter Priors

- 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 $p(\theta) \propto \exp(\lambda\text{penalty}(\theta))$.

Ockham's Razor



We want to use the simplest model which explains the data well.

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 any better than any other learning algorithm.*
- This includes algorithms which employ cross validation, random guessing, whatever.
- 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.