Improving Generalization

Neural Computation : Lecture 10

1. Training, Validation and Testing Data Sets
2. Cross-Validation
3. Weight Restriction and Weight Sharing
4, Optimal Network Architectures
5. Stopping Training Early
6. Regularization and Weight Decay
7. Adding Noise / Jittering
8. Which is the Best Approach?

Improving Generalization

We have seen that, for neural networks to generalize well, one needs to avoid both under-fitting of the training data (which corresponds to high statistical bias) and over-fitting of the training data (which corresponds to high statistical variance).

There are a number of approaches for improving generalization – one can:

1. Arrange to have the optimum number of free parameters (independent connection weights) in the model.
2. Stop the gradient descent training at the most appropriate point.
3. Add a regularization term to the error function to smooth out the mappings that are learnt.
4. Add noise to the training patterns to smooth out the data points.

To employ these effectively, a way is needed to *estimate* what the generalization is likely be. Fortunately, we *do not* need to calculate the bias and variance to do this!

Training, Validation and Testing Data Sets

The data available for training our networks is called the ***training data***, and the unseen data that is used to test the network's generalization ability is called the ***testing data***.

One usually wants to optimize the network's training procedures to result in the best generalization, but clearly we cannot use the testing data to do this. What we can do is assume that the training data and testing data are drawn randomly from the same data distribution, and then a random sub-set of the training data that is not used to train the network can be used to estimate the likely performance on the testing set, i.e. predict what the generalization will be. The portion of the data available for training that is withheld from the network training is called the ***validation data set***, and the remainder of the data is called the ***training data set***. This approach is called the ***hold out method***.

The idea is that we split the available data into training and validation sets, train various networks using the training set, test each one on the validation set, and the network which is best is on that is likely to provide the best generalization to the testing set.

Cross Validation

Often the availability of training data is limited, and it is not practical to take a large part of it to be a validation set. An alternative is to use the procedure of ***cross-validation***.

In ***K-fold cross-validation*** the full set of training data is randomly divided into K distinct subsets, and the network is trained using $K-1$ subsets, and tested on the remaining subset. The process of training and testing is then repeated for each of the K possible choices of the subset omitted from the training. The average performance on the K omitted subsets is then the estimate of the generalization performance.

This procedure has the advantage that is allows the use of a high proportion of the available training data (a fraction $1-{}^{1}/_{K}$) for training, while making use of all the data points in estimating the generalization error. The disadvantage is that the network needs to be trained K times. Typically $K \sim 10$ is considered reasonable.

If K is made equal to the full sample size, it is called ***leave-one-out cross validation***.

Weight Restriction and Weight Sharing

Perhaps the most obvious way to prevent over-fitting in any machine learning model (including neural networks) is to restrict the number of free parameters they have.

The simplest way to do this for neural networks is to restrict the number of hidden units, as this will automatically reduce the number of weights. Some form of validation or cross-validation scheme can be used to find the best number for each given problem.

An alternative is to have many weights in the network, but constrain certain groups of them to be equal. If there are obvious symmetries in the problem, it makes sense to enforce ***hard weight sharing*** by building them into the network in advance.

In other problems, it is possible to use ***soft weight sharing*** where appropriate sets of weights are encouraged to have similar values by the learning algorithm. This can be conveniently implemented by adding a suitable term to the error/cost function. This method can then be seen as a particular form of ***regularization***.

Optimal Network Architectures

Identifying a good neural network architecture (e.g., setting an appropriate number of hidden units) is clearly very important, but it is also very problem dependent.

There are various different criteria for what is ***optimal***. Usually it is the generalization ability, but learning time, memory requirements, and so on, may also be important.

In the same way that we can have our networks learn their weights incrementally, we can modify our network architecture, to suit its given task, in an incremental fashion.

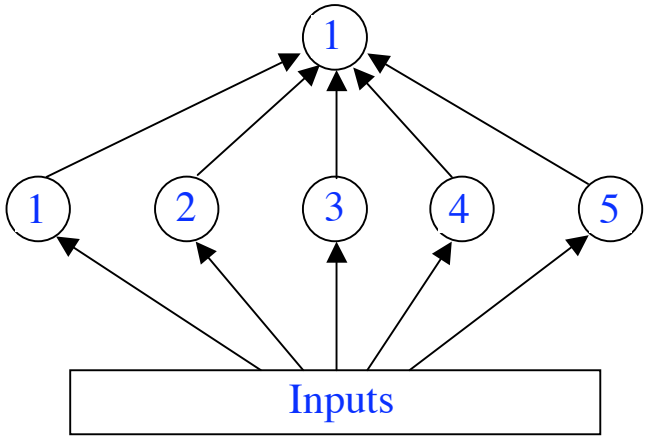
There are two obvious ways to proceed:

1. Start with too few hidden units, and add some more $\Rightarrow$ ***Constructive algorithms***
2. Start with too many hidden units, and take some away $\Rightarrow$ ***Pruning algorithms***

Many algorithms exist for each approach – here we shall just consider a couple of popular simple examples for each.

Constructive Algorithms

Training a series of networks with various numbers of hidden units and picking the one with the best generalization is very wasteful in terms of having to start the training from scratch for each network. A better approach would be to keep adding further hidden units to an existing network and only train the new portions of it.



For example, one can add hidden units in the sequence shown, with each additional unit trained to deal with the remaining incorrect training patterns. Another popular approach is the ***Cascade Correlation*** neural network, which has a pool of potential extra hidden units at each stage, with one chosen according to correlation with the residual error.

Pruning Algorithms

We will see later how adding a regularization term into the gradient descent cost function can be used to encourage the decay of unnecessary weights and effectively prune out unwanted connections.

One can also remove connections explicitly. If the gradient descent (e.g., sum squared) error function is $E(\{w_{ij}\})$, then we can define the ***saliency*** of each weight w_{kl} as

$$s_{kl} = E(\{w_{ij} : w_{kl} = 0\}) - E(\{w_{ij}\})$$

i.e. how much the error increases when that weight is removed from the network. We can then define an iterative procedure whereby we train the network, compute all the saliencies, remove the weight(s) with the lowest saliency, and repeat the process until even the lowest saliencies are "large", or until the error on some validation set starts rising again. This can be computationally expensive, but we can compute derivatives to estimate the saliencies and avoid multiple runs to determine the $E(\{w_{ij} : w_{kl} = 0\})$.

Optimal Brain Damage

We have already noted that we can expand the error/cost function as a Taylor series

$$E(\{w_{ij} + \Delta w_{ij}\}) = E(\{w_{ij}\}) + \sum_{i,j} \frac{\partial E(\{w_{ij}\})}{\partial w_{ij}} \Delta w_{ij} + \tfrac{1}{2} \sum_{i,j} \sum_{k,l} \frac{\partial^2 E(\{w_{ij}\})}{\partial w_{ij} \partial w_{kl}} \Delta w_{ij} \Delta w_{kl} + O(\Delta w^3)$$

in which the second order derivative is known as the ***Hessian*** matrix

$$H_{ijkl} = \frac{\partial^2 E(\{w_{ij}\})}{\partial w_{ij} \partial w_{kl}}$$

If we assume individual weights are small, and use the fact that the first derivatives are zero for a fully trained network, then we can set $\Delta w_{ij} = - w_{ij}$ and estimate the saliency as

$$s_{ij} = \tfrac{1}{2} H_{ijij} w_{ij} w_{ij}$$

Iteratively training and pruning the network weights using the lowest values of this quantity is known as ***optimal brain damage***. A related, more efficient, approach is known as ***optimal brain surgeon***.

Stopping Training Early

Neural networks are often set up with more than enough free parameters for over-fitting to occur, and so other procedures have to be employed to prevent it.

For the iterative gradient descent based network training procedures we have considered (such as batch back-propagation and conjugate gradients), the training set error will naturally decrease with increasing numbers of epochs of training.

The error on the unseen validation and testing data sets, however, will start off decreasing as the under-fitting is reduced, but then it will eventually begin to increase again as over-fitting occurs.

The natural solution to get the best generalization, i.e. the lowest error on the test set, is to use the procedure of ***early stopping***. One simply trains the network on the training set until the error on the validation set starts rising again, and then stops. That is the point at which we expect the generalization error to start rising as well.

Practical Aspects of Stopping Early

One potential problem with the idea of stopping early is that the validation error may go up and down numerous times during training. Lower learning rates can help smooth the learning, but the safest approach is generally to train to convergence (or at least until the validation error appears unlikely to reach a new minimum), saving the weights at each epoch, and use the weights at the epoch or epochs with the lowest validation error.

It can be shown that there is an approximate relationship between stopping early and a particular form of regularization. That relationship indicates that this approach will work best if the training starts off with very small random initial weights.

There are also the more general practical problems concerning how to best split the available training data into distinct training and validation data sets. For example: What fraction of the patterns should be in the validation set? Should the data really be split randomly, or by some systematic algorithm? As so often, these issues are very problem dependent and there are no simple general answers.

Regularization

The general technique of ***regularization*** encourages smoother network mappings by adding a penalty term Ω to the standard (e.g., sum squared error) cost function

$$E_{reg} = E_{sse} + \lambda\Omega$$

where the regularization parameter λ controls the trade-off between reducing the error E_{sse} and increasing the smoothing. This modifies the gradient descent weight updates so

$$\Delta w_{kl}^{(m)} = -\eta \frac{\partial E_{sse}(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}} - \eta\lambda \frac{\partial \Omega(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}}$$

For example, for the case of soft weight sharing we can use the regularization function

$$\Omega = -\sum_{k,l,m} \ln\left(\sum_{j=1}^{M} \frac{\alpha_j}{\sqrt{2\pi\sigma_j^2}} \exp\left\{ -\frac{(w_{kl}^{(m)} - \mu_j)^2}{2\sigma_j^2} \right\} \right)$$

in which the $3M$ parameters α_j, μ_j, σ_j are optimised along with the weights.

Weight Decay

One of the simplest forms of regularization has a regularization function which is just the sum of the squares of the network weights (not including the thresholds):

$$\Omega = \frac{1}{2} \sum_{k,l,m} (w_{kl}^{(m)})^2$$

In conventional curve fitting this regularizer is known as ***ridge regression***. It is clear why it is called ***weight decay*** when we compute the extra term in the weight updates:

$$-\eta\lambda \frac{\partial \Omega(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}} = -\eta\lambda w_{kl}^{(m)}$$

In each epoch the weights decay in proportion to their size, i.e. exponentially.

Empirically, this leads to significant improvements in generalization. This is because producing over-fitted mappings requires high curvature and hence large weights. Weight decay keeps the weights small and hence the mappings are smooth.

Adding Noise / Jittering

Adding ***noise*** or ***jitter*** to the inputs during training is also found empirically to improve network generalization. This is because the noise will "smear out" each data point and make it difficult for the network to fit the individual data points precisely, and consequently reduce over-fitting.

Actually, if the added noise is ξ with variance λ, the error function can be written:

$$E_{sse}(\xi_i) = \tfrac{1}{2}\sum_p \sum_{\xi} \sum_j \left(y_j^p - net_j(x_i^p + \xi_i)\right)^2$$

and after some tricky mathematics one can show that

$$\lambda\Omega = E_{sse}(\xi_i) - E_{sse}(0) = \tfrac{1}{2}\lambda \sum_p \sum_j \left(\frac{\partial net_j(x_k^p)}{\partial x_i^p}\right)^2$$

which is a standard ***Tikhonov regularizer*** minimising curvature. The gradient descent weight updates can then be performed with an extended back-propagation algorithm.

Which is the Best Approach ?

Achieving optimal balance between the statistical bias and variance requires optimizing the effective complexity of the model, which for neural networks means optimizing either the effective number of adaptable/free parameters or the amount of training.

The obvious ways to optimise the complexity are by adjusting the number of weights or the number of epochs of training. The other approaches are all equivalent to some form of regularization, which involves adding a penalty term to the standard gradient descent error/cost function. The degree of regularization, and hence the effective complexity of the model, is then controlled by adjusting the associated regularization parameter λ.

In practice, the optimal number of weights, or amount of training, or regularization level, is found using a validation data set or cross-validation. The approaches studied all work well, and which one chooses ultimately depends on which is most convenient for the given problem. Unfortunately, there is no overall best approach. However, early stopping does have the big advantage that only one network needs to be trained.

Overview and Reading

1. We began by recalling the aim of good generalization.
2. Then the ideas of validation and cross-validation were introduced as convenient methods for estimating generalization using only the available training data.
3. We then studied the principal approaches for improving generalization: optimizing the number of weights, weight sharing, stopping training early, regularization, weight decay, and adding noise to the inputs.
4. We concluded that there was no generally optimal approach.

Reading

1. Bishop: Sections 9.2, 9.3, 9.4, 9.5, 9.8
2. Haykin-2009: Sections 4.11, 4.13, 4.14
3. Gurney: Section 6.10