Chapter 4

Ensemble Machines

4.1 Introduction

So far we have been looking at single neural networks, how they work and how they can be trained. Here we will combine many networks to form an *ensemble of networks* (or sometimes called committee machines).

We can divide ensembles into two major categories:

1. Static structures: Here the responses of several predictors (e.g. ANN) are combined in way that is **not** influenced by the inputs. This is why we call it static. We are going to look at two approaches here:
   - *Ensemble averaging*: Outputs are linearly combined to form an ensemble response.
   - *Boosting*: Responses from predictors are combined to achieve as high accuracy as possible, by focus the training on difficult cases.
2. Dynamic structures: Here the input signal is used in the mechanism that integrates the outputs of the individual predictors into the final output. Two well-known approaches are: *Mixture of exports* and *Hierarchical mixture of experts*.

We will only look at static structures in this course and we will begin with ensemble averaging.

4.2 Ensemble averaging

When we have trained a network on a dataset $\mathcal{D}$, we have a weight vector $\boldsymbol{\omega}_{\mathcal{D}}$ that represents our model. However, there may be random elements in our training, *e.g.*, initialization of

weights, minibatch selection in stochastic gradient descent, and node blocking in dropout regularization. Therefore, it may be that training two networks ("1" and "2") with identical network topology, on the same dataset, results in different trained weight vectors $\boldsymbol{\omega}_{\mathcal{D}1} \neq \boldsymbol{\omega}_{\mathcal{D}2}$.

It is therefore possible to create many different models m using the same dataset $\mathcal{D}$, and we may combine them into an *ensemble* of networks, and treat the average of all model outputs as an ensemble output:

$$y_{\text{ens}}(\mathbf{x}) = \frac{1}{M}\sum_m y_m(\mathbf{x}), \tag{4.1}$$

where there are M members in the ensemble, and we have suppressed the dataset-dependence in the notation.

When we discussed generalization, we were interested in how well outputs agreed with the target expectation value, and we studied the squared error $(y(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2$. Hiding the $\mathbf{x}$-dependence for a while, we then see that each member's generalization ability could be measured by $(y_m - \langle d\rangle)^2$ and the mean generalization ability is

$$\begin{aligned}\frac{1}{M}\sum_m (y_m - \langle d\rangle)^2 &= \frac{1}{M}\sum_m (y_m - y_{\text{ens}} + y_{\text{ens}} - \langle d\rangle)^2 = \\ &= \ldots = \frac{1}{M}\sum_m (y_m - y_{\text{ens}})^2 + (y_{\text{ens}} - \langle d\rangle)^2,\end{aligned} \tag{4.2}$$

where the final results comes from the fact that $\sum_m y_m = M y_{\text{ens}}$, and that neither y_{ens} nor $\langle d\rangle$ depends on m.

Re-introducing the $\mathbf{x}$-dependence, we can formulate the following important decomposition of the ensemble error

$$(y_{\text{ens}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 = \tag{4.3}$$

$$= \frac{1}{M}\sum_m (y_m(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 - \frac{1}{M}\sum_m (y_m(\mathbf{x}) - y_{\text{ens}}(\mathbf{x}))^2. \tag{4.4}$$

In words: The error made by the ensemble is the mean error made by the individual networks **minus** the (approximate) variance of the ensemble members. The last term is often called the "diversity term". To conclude: For the ensemble to work well we want a set of *accurate* ensemble members that *disagree* as much as possible.

Since the relation holds for all $\mathbf{x}$ and datasets $\mathcal{D}$, it will remain after we integrate over $\mathbf{x}$ with an input distribution $p(\mathbf{x})$, and after we take and expectation value $E[\cdot]$ with respect to datasets $\mathcal{D}$.

Actually, it is quite common that the ensemble performs better than its best member! One reason is that some members may overestimate $\langle d\rangle$, while others underestimate it, so that y_{ens}, which is the mean of all y_m, is closer to $\langle d\rangle$ than any y_m. Another reason is that overall

performance involves integration over inputs. Unless the same model m is best for all inputs $\mathbf{x}$, it can easily happen that

$$\min_m \int (y_m(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 p(\mathbf{x}) d\mathbf{x} \; > \; \int (y_{\text{ens}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2 p(\mathbf{x}) d\mathbf{x}.$$

With this in mind, we can actively introduce extra randomness in our training, to create more diverse (but still reasonably accurate) ensemble members!

4.3 Generating an ensemble

So how do we create a good ensemble? We can split the question into two:

1. How should we train the individual ensemble members?
2. How should we weight the different members together to form the ensemble output?

We start with the first question. Here we should remember what we just discovered, namely that we want accurate and diverse networks. Here are a few approaches:

1. Different initial conditions: Just train L networks with different (random) initial values of the weights. This may lead to finding different local minima of the error function, which may produce slightly different networks.

   + : easy to implement.

   - : networks are not diverse enough.

2. Bagging: Bagging creates diverse networks by training on "slightly" different training data sets. Each network in the bagging ensemble is trained on a bootstrap sample of the original training data set. This means that we resample the original training data set **with replacement**. The size of the bootstrap sample is the same as the original one. q

   + : simply good!

   - : not enough diversity for very large data sets.

3. K-fold cross splitting: In this approach we divide the training data set into K parts of (approximately) equal size (compare K-fold cross validation). Ensemble member k is trained on the training set you obtain when leaving out part k. The procedure is illustrated in figure 4.1. The whole procedure can be repeated N times with different random split into K parts. As a result we have an ensemble of size $N \cdot K$.

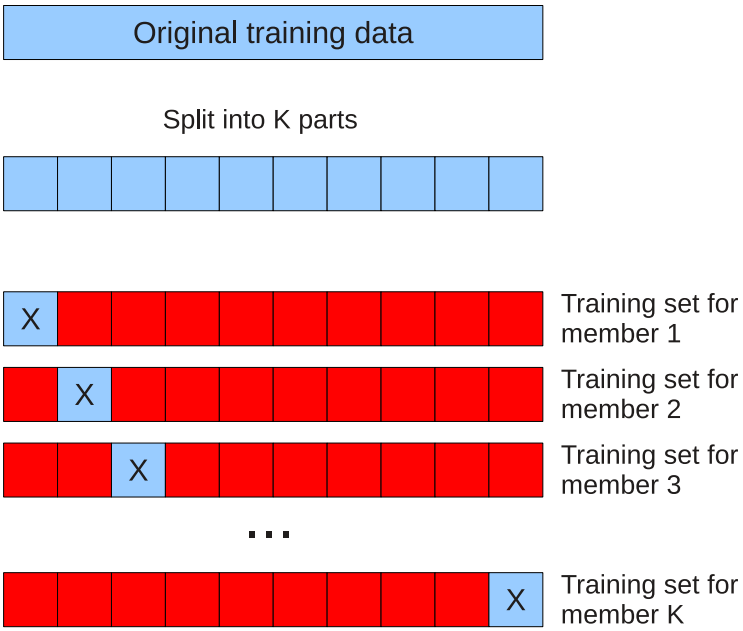

Figure 4.1: The K-fold splitting method for training ensemble members. For each of K splits you remove one part of the data, thereby having K training datasets that are similar, but not identical. If K is small the dataset becomes more different and the diversity should increase.

+ : simply good!, faster than bagging since we train each member on fewer data.

- : again for really big dataset the diversity is not enough.

4. K-fold mini-batches: This is the same approach as K-fold cross-splitting, but here we only train on the k:th part in each K-fold cross split. Looking at figure 4.1 this approach would correspond to train on each blue square with the 'X' and disreagard the rest. Again, the whole procedure can be repeated N times with different random split into K parts. As a result we have an ensemble of size $N \cdot K$.

   + : will generate diverse members and is suitable for large datasets.

   - : will possibly have too large bias for smaller datasets.

4.4 Optimal ensemble weighting

The second question regarding of how to build the ensemble is the way we weight them together. The, by far most, common approach is to simply take the mean of all members i,

i.e.

$$y_{\text{ens}}(\mathbf{x}) = \frac{1}{M}\sum_{i}^{M} y_i(\mathbf{x})$$

The advantage of this approach is that it does not contain any parameters, avoiding possible overtraining. Let's now briefly look into the task of actually do optimal weighting of ensemble members i. We start with,

$$y_{\text{ens}}(\mathbf{x}) = \sum_{i}^{M} \alpha_i y_i(\mathbf{x})$$

Introducing errors

$$\epsilon_i(\mathbf{x}) = y_i(\mathbf{x}) - \langle d|\mathbf{x}\rangle,$$

we get

$$y_{\text{ens}}(\mathbf{x}) = \langle d|\mathbf{x}\rangle + \sum_{i}^{L} \alpha_i \epsilon_i(\mathbf{x})$$

If we let D_{ens} be the expectation value many datasets of the ensemble error, we get

$$D_{\text{ens}} = \mathrm{E}\left[(y_{\text{ens}}(\mathbf{x}) - \langle d|\mathbf{x}\rangle)^2\right] = \mathrm{E}\left[(\sum_i \alpha_i\epsilon_i)(\sum_j \alpha_j\epsilon_j)\right] = \tag{4.5}$$

$$= \mathrm{E}\left[\sum_i\sum_j \alpha_i\alpha_j\epsilon_i\epsilon_j\right] = \sum_i\sum_j \alpha_i\alpha_j \mathrm{E}\left[\epsilon_i(\mathbf{x})\epsilon_j(\mathbf{x})\right] \tag{4.6}$$

Now define the error correlation matrix $\mathbf{C}$ with its matrix elements C_{ij} as

$$C_{ij} = \mathrm{E}\left[\epsilon_i(\mathbf{x})\epsilon_j(\mathbf{x})\right]$$

This means that the error can be written as $D_{\text{ens}} = \sum_i \sum_j \alpha_i\alpha_j C_{ij}$. We can now consider the optimization problem of finding α_i such that $\sum_i \sum_j \alpha_i\alpha_j C_{ij}$ is minimized for fixed C_{ij}. To avoid the trivial solution $\alpha_i = 0, \forall i$, we require

$$\sum_i \alpha_i = 1$$

This optimization problem can be solved using Lagrange multipliers (see exercise 4.1). The solution is given by

$$\alpha_i = \frac{\sum_j (\mathbf{C}^{-1})_{ij}}{\sum_k \sum_j (\mathbf{C}^{-1})_{kj}}$$

Since we don't now C_{ij} exactly we can approximate it using a validation data set

$$C_{ij} \approx \frac{1}{N} \sum_{n=1}^{N} \left(y_i(\mathbf{x}_n) - d_n\right) \left(y_j(\mathbf{x}_n) - d_n\right)$$

A problem with this approach is that we may get solutions with large positive and negative weighting factors α_i. To overcome this, we can introduce yet another constraint on the α's:

$$\sum_i \alpha_i = 1 \qquad \text{and} \qquad \alpha_i > 0 \; \forall i$$

The drawback is now that the optimization problem is more difficult. However optimal weighting has the disadvantage of having parameters that should be trained using a new dataset different from the training dataset to avoid overfitting. If it fair to say that the simple mean ensemble is the most common one.

4.5 Boosting

Boosting is a general idea that can be applied to many machine learning algorithms that do supervised learning. It is biased toward "weak" learners in order to boost them to become "strong" learners. We will look at the well-known AdaBoost (Adaptive Boosting) (meta)-algorithm, which can be used to improve the performance of other learning algorithms. It can be regarded as an ensemble learning algorithm since it based on the predictions of many individual members. It constructs the ensemble by putting more and more emphasis on certain data points.

Below is a description of the AdaBoost algorithm, but we need a few definitions before we start. We focus on a binary classification problem, and we let our output be binary. We could for example use a threshold function for the output, or turn a continuous output into a binary variable using a threshold.

- **learner:** A learner t will provide class labels $y_t(\mathbf{x})$ for a given data point $\mathbf{x}$.
- **Training sample:** We have the usual training data set of inputs and labels $\{\mathbf{x}_n, d_n\}_{n=1,\ldots N}$
- **Weight distribution over training data:** Let P_t be a set of weights for the input data, where $P_{tn} > 0$. These weights will be used focus the training on data points with large weights and less focus on data points with small weights.
- **Number of iterations:** Let T be the number of iterations of AdaBoost. This means that we will train T learners. T will hence be the size of the ensemble.

AdaBoost

1. Let P_1 be uniform, i.e.
$$P_{1n} = \frac{1}{N} \quad \forall n$$
2. Do the following for $t = 1, 2, ..., T$
   (a) Call the learning algorithm for your learner using P_t. For a neural network, we could introduce P_{tn} as weights in the error function, $E(\boldsymbol{\omega}) \propto \sum_n P_{tn} E_n(\boldsymbol{\omega})$. Then, the trained weights $\boldsymbol{\omega}_t$ will depend on P_t and the outputs $y_t(\mathbf{x})$ will also depend on P_t.
   (b) Calculate the error of $y_t(\mathbf{x})$ according to:
$$\epsilon_t = \frac{\sum_{n:y_{tn} \neq d_n} P_{tn}}{\sum_n P_{tn}}$$
   In words, $1 - \epsilon_t$ is a "weighted accuracy" of the learner, with P_{tn} as weights.
   (c) Set
$$\alpha_t = \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$
   (d) Update the weights P_{tn}
$$P_{t+1,n} = \begin{cases} P_{tn}, & y_{tn} = d_n \\ P_{tn}\frac{1-\epsilon_t}{\epsilon_t}, & y_{tn} \neq d_n \end{cases}$$
3. The output from the final classifier is then given by
$$y_{\text{AdaBoost}}(\mathbf{x}) = \arg\max_d \sum_{t:y_t(\mathbf{x})=d} \alpha_t$$
   So the output of $y_{\text{AdaBoost}}(\mathbf{x})$ is the label d that maximizes the sum of "weights" α_t of the learners that predicted that label.

AdaBoost is not the most popular boosting algorithm. Today *Gradient Boosting* and *Extreme Gradient Boosting (XGBoost)* are more used, where the latter is the current state-of-the-art boosting method. It is unfortunately beyond the scope of this course to give a more detailed description of them.