STAT3007/7007 Turorial Paper: Ensemble methods and Dropout

Henrik Olaussen (s4818395)

Abstract

This tutorial paper examines the concept of ensemble methods, in particular dropout, boosting and bagging, exploring their relation to each other. The main focus of this tutorial paper is to explain and compare bagging and dropout due to their close relationship. While both bagging and dropout aim to reduce the model variance, bagging is less suitable for neural networks. Training many independent models until convergence, as required by bagging, results in high computational costs, particularly when applied to neural networks. Dropout is therefore introduced as an alternative when working with neural networks. Due to parameter sharing among the generated sub-networks and the weight scaling inference rule, the computational cost of dropout is significantly lower compared to bagging. Additionally, a coding example is provided to demonstrate the implementation of a neural network class with dropout, and to illustrate the results of training a model with and without dropout.

1 Introduction

In an ensemble method, we train multiple base models and then combine them into a single powerful model that generalizes better to new data than each individual model. This can be done by e.g. averaging all the model predictions or through a majority vote. Two well-known ensemble methods that this tutorial paper will examine are boosting and bagging. These methods aim to obtain better predictions by reducing either the variance or bias in the model, respectively. Additionally, the idea of training multiple models independently is often infeasible for neural networks, as training is computationally expensive. Dropout is a method that, in a way, extends the idea of bagging to make it possible to train ensembles of many deep neural networks. This tutorial paper will give an introduction to bagging and boosting, and then examine how dropout works compared to bagging.

2 Bootstrap aggregating

As mentioned, bootstrap aggregating (bagging) aims to reduce the variance in the model. High variance often results in overfitting, which is the concept of fitting the model too closely to the training data. This makes the model fit the noise and fluctuations in the data, and not the underlying relationship between the predictors and the response. Consequently, the model will not obtain good predictions on new, unseen data. However, by training several high-variance and low-biased models and then averaging the predictions, the final, combined model has both low variance and bias. Let us see how this works.

2.1 Training and testing

The idea of bagging is to average the results obtained from training different models on bootstrap samples of the training data. Bootstrap is a method where you sample from your original data, but with replacement. As a result, we get different versions of the training data, where each training example might occur several times in each sample. After bootstrapping B samples, we can train B models independently, one on each bootstrap-sample. Each of these models will then obtain a prediction. As mentioned, bagging obtains a single prediction by averaging these B predictions. Let us see how this reduces the overall variance, but without increasing the bias notably.

2.2 Influence on the total variance and bias

First of all, it is important to note that all the individual predictions are dependent and identically distributed as they are all sampled from the same training data (through bootstrapping). Let us therefore assume that we have an identically distributed and dependent sample of random variables $\hat{y}_1, .., \hat{y}_B$, that represent the prediction of model i. Secondly, we have that $E[\hat{y}_i] = \mu$ and $\text{Var}[\hat{y}_i] = \sigma^2$, where $i = 1, .., B$. The expected value of the averaged sample is:

$$E\left[\frac{1}{B}\sum_{i=1}^{B}\hat{y}_i\right] = \frac{1}{B}\sum_{i=1}^{B}E\left[\hat{y}_i\right] = \mu \tag{1}$$

Since the mean of the ensemble average is the same as the mean for each single model's prediction, the bias given by

$$E\left[\frac{1}{B}\sum_{i=1}^{B}\hat{y}_i\right] - y$$

where y is the actual value that we want to predict, is the same as the bias from each individual model.

Furthermore, the variance of the averaged sample is given by the following expression (derivation in Appendix), where ρ is the positive pairwise correlation between the variables $\hat{y}_i$:

$$\text{Var}\left[\frac{1}{B}\sum_{i=1}^{B}\hat{y}_i\right] = \frac{1-\rho}{B}\sigma^2 + \rho\sigma^2 \tag{2}$$

Equation (2) represents the variance in the final prediction obtained from bagging. The expression tells us that if $\rho < 1$ is small and if we increase B, we can make the variance arbitrarily small. Experience has shown that ρ is often small enough to give a significant decrease in variance. Consequently, by averaging the correlated predictions in bagging, we can reduce the variance.

3 Boosting

Having explored an ensemble method focused on variance reduction, we will now take a look at another technique known as boosting, which aims for bias reduction. In boosting, the idea is to combine several high-biased base models into making a single prediction. Even if each model has a large bias, they might still capture some of the input-output relationship.

During training, each model is trained successively, where every new model aims to correct the errors of the previous one. The way this works is that each new model place greater emphasise (weights) on samples that gave poorer results in the previous model compared to other data points. The weights assigned to each training example can be determined through various algorithms, such as AdaBoost or Gradient Boosting. This tutorial paper will not look any closer at these two algorithms. Furthermore, the final prediction comes from a majority vote or a weighted average among all the predictions.

4 Dropout

Deep neural networks can possibly learn complex features. However, applying deep neural networks to data with simple input-output relationships easily overfits the data. As mentioned earlier, we can avoid overfitting by reducing the variance in the model. Bagging was one way

to do this. However, training large neural networks are often computationally expensive in terms of memory and runtime. Training an ensemble of neural networks, like in bagging, could therefore be extremely costly. Dropout is a relatively computationally inexpensive method that resembles bagging.

4.1 Training with dropout

Training with dropout is different from training an ensemble model in bagging. The main idea of dropout is to randomly remove non-output units (neurons) (with a probability p_i defined by the user) from a base network, to create a sub-network. An example of a sub-network obtained from randomly removing some units is shown in Figure (1). For each optimization step, we randomly remove a new set of hidden units. The weights related to the hidden units not removed are the only ones that are updated during this specific iteration. A hidden unit can be removed from the network for a given iteration by multiplying its output by 0. When training with dropout, stochastic gradient descent is often used as the optimisation algorithm.

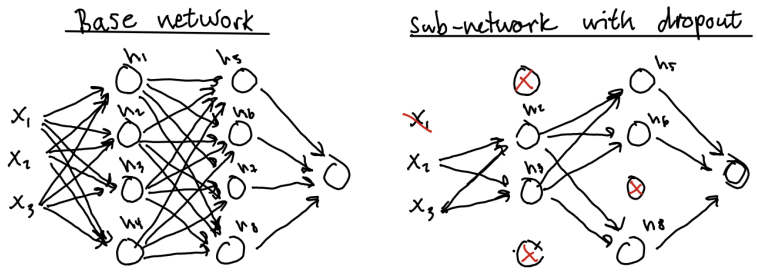

Figure 1: Example of a sub-network from using dropout

4.2 Dropout as an ensemble method

All the sub-networks created by randomly removing the hidden units can be seen as an ensemble of neural networks. Usually, by following the procedure of dropout, we train exponentially many neural networks. If we on the other hand followed the bagging-algorithm, all models would be trained until convergence, independently of each other. This would be infeasible for exponentially many neural networks. In dropout, the sub-networks inherits the parameters of the previous sub-networks. The sharing of parameters across the sub-networks is the reasons why dropout can represent such a large number of models in an ensemble, and still be a relatively computationally inexpensive method.

4.3 At prediction time

Furthermore, as previously mentioned, assessing each sub-network individually and then averaging their predictions to derive a final prediction, similar to bagging, would be infeasible. However, in the paper of Hinton et al., they propose an approximation to taking the average over all the predictions. Following their proposal, a forward pass through the entire network is made at prediction time, without randomly removing hidden units. However, the network's weights are now modified. This modification involves multiplying each trained parameter going out of a unit i with the probability q_i of not removing that specific (given by $q_i = 1 - p_i$).

As a result, the expected value of the output of unit i is roughly the same during testing and training. In Goodfellow et al., this is called the weight scaling inference rule. Nevertheless, it is important to highlight that there are no solid theoretical justification for the accuracy of this approximation. However, Hinton et al. argues that empirical evidence has showed its effectiveness in practical applications.

4.4 Regularization

Lastly, note that because dropout reduces the variance, it is a regularization method. Because dropout randomly removes some units in each iteration, the network becomes less reliant on specific neurons. For example, suppose we implement a neural network that detects certain objects in pictures. By removing a hidden unit i, then the feature tied with that unit disappears. For example, suppose we remove the feature representing a door when detecting a house. This implies that the network must learn how to detect the house without the need of the door-feature. Consequently, networks trained with dropout have a tendency to generalize better, and lower the risk of overfitting the training data.

4.5 Limitations of Dropout

Firstly, dropout effectively reduces the variance in a neural network. By doing so, the model complexity of the neural network reduces, meaning that the model can learn less complex features. To retain the model complexity, it is necessary to increase the size of the neural network when implementing dropout. In Goodfellow et al., they argue that the validation set error is typically much lower when using dropout, but that this implies that we require both larger models and an increased number of iterations during training. If we are dealing with very large systems, then obtaining good results with dropout could be computationally expensive. The benefit of regularization with dropout might therefore be outweighed by the computational cost.

Secondly, Goodfellow et al., claims that dropout is less effective when we only have a few training examples available.

To sum up, the need for larger models and the fact that dropout is less effective when only a few examples are available, are two drawbacks of using dropout.

4.6 Coding Example: Dropout

In this example we will take a look at how to apply dropout to a neural network using pytorch, and how the training error compares to the same network without dropout. The code is included as a supplementary file. I have implemented a simple CNN in Pytorch for classification of handwritten digits in the MNIST dataset. The CNN consists of two CNN layers followed by a fully connected layer and an output layer. The 'nn.Dropout()' method has to be placed after the dense layer we want to apply dropout to. If we had a deeper dense neural network, we could have applied dropout to several layers, with a different probability of randomly removing the neurons associated with each layer.

In our case, the probability of randomly removing a hidden layer is 0.4, the learning rate is 0.1 and the number of epochs is 15. Cross entropy loss is used as loss function. The CNN class is shown in Figure 2.

As discussed, dropout works different during training and testing. We distinguish between training and testing by activating our model into training and testing mode using 'model.train()' and 'model.eval()', respectively. Otherwise, the training and testing procedure is similar to what

```python
class CNN_dropout(nn.Module):
    def __init__(self):
        super(CNN_dropout, self).__init__()

        #convolutional layer
        self.cnn_layers = nn.Sequential(
            nn.Conv2d(in_channels=1,out_channels=4,kernel_size=3, stride = 1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2)
            )

        #dense_layer
        self.dense_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features = 4 * 13 * 13, out_features = 50),
            nn.Dropout(p=0.4), #Place after the dense layer we want to apply dropout to
            nn.Linear(in_features = 50, out_features = 10)
            )

    def forward(self, x):
        x = self.cnn_layers(x)
        x = self.dense_layers(x)
        out = torch.softmax(x, dim=1)
        return out
```

Figure 2: CNN class with dropout

we have seen for other pytorch implementations of neural networks. The training performance of the model with dropout is compared to the base network (without dropout) in Figure 3.

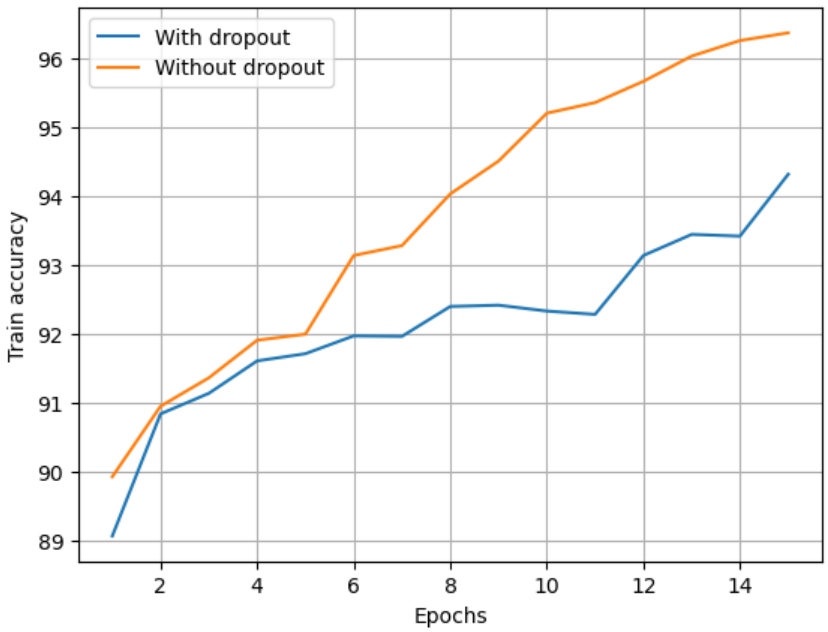

Figure 3: Training accuracy with and without dropout

As we can see from Figure 2, the training accuracy with dropout is always less than the training accuracy without dropout. Like we have discussed in this tutorial paper, this makes sense as dropout aims to avoid overfitting the training data. As mentioned, using dropout might also require us to implement a larger model to offset the reduction in model complexity. However, the difference is not too significant. An alternative set of hyperparameters, or a potentially deeper dense neural network, could provide clearer distinctions between models with and without dropout.

Appendix A.

4.7 Derivation of Equation (2)

Note that all samples have equal variance s.t. $\text{Cov}(x_i, x_j) = \text{Corr}(x_i, x_j)\left(\text{Var}[x_i]\text{Var}[x_j]\right)^{\frac{1}{2}} = \rho\sigma^2$.

$$\text{Var}\left[\frac{1}{B}\sum_{i=1}^{B} y_i\right] = \frac{1}{B^2}\text{Var}\left[\sum_{i=1}^{B} y_i\right] = \frac{1}{B^2}\left(\sum_{i=1}^{n}\text{Var}\left[y_i\right] + \sum_{i\neq j}\text{Cov}(y_i, y_j)\right)$$

$$= \frac{1}{B^2}\left(B\sigma^2 + B(B-1)\rho\sigma^2\right) = \frac{\sigma^2}{B} + \frac{(B-1)\rho\sigma^2}{B} = \frac{1-\rho}{B}\sigma^2 + \rho\sigma^2$$

References

[1] A. Lindholm, N. Wahlström, F. Lindsten, T. B. Schön. *Machine Learning: A First Course for Engineers and Scientists*. Cambridge University Press, 2022.

[2] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov. *Improving neural networks by preventing co-adaptation of feature detectors* . Department of Computer Science, University of Toronto, 2012.

[3] I. Goodfellow, Y. Bengio, A Courville. *Deep Learning*. MIT Press Academic, 2017.

[4] T. Hastie, R. Tibshirani, J Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[5] Brijesh Soni. *Understanding Boosting in Machine Learning: A Comprehensive Guide - Medium.com*. Available online: `https://medium.com/@brijesh_soni/understanding-boosting-in-machine-learning-a-comprehensive-guide-bdeaa1167a6`, 2023. [Accessed: 05.05.2024]

I give consent for this to be used as a teaching resource.