Measurements of $H \to b\bar{b}$ decays and VH production

Thomas Charman

Supervisor: Dr. Jonathan Hays

Queen Mary University of London

Submitted in partial fulfillment of the requirements of the Degree of Doctor of Philosophy February 10, 2021.

Signature:

Date:

Details of collaboration and publications:

Contents

List of Figures

List of Tables

Chapter 1

Machine Learning

The next chapter is somewhat of a diversion from the physics discussed so far. It will focus on techniques in machine learning which are often referred to in High Energy Physics as a Multi-variate Analysis (MVA). The reason for this diversion is that these techniques have become very widespread in the field, they are used in several of the reconstruction and selection algorithms that are used to obtain the events on which the analysis is performed. Furthermore an MVA is used to obtain the distribution that acts as the final discriminant for the analysis, and machine learning techniques are also used to model the backgrounds. Given how widespread the use of these techniques is in the analysis it makes sense to describe them before diving into the details.

The two main algorithms used are Boosted Decision Trees (BDTs) and Neural Networks (NNs) which will be described in sections 1.1 and 1.2 respectively. These algorithms are used in many places outside High Energy Physics and so rather than referring to individual pieces of data that enter into the algorithm as an event, in this chapter they will be referred to as an example. This terminology comes from the fact that in general these algorithms must be shown a large number of examples before they are suitable "trained" for their purpose, and that in general those examples could be data that represent anything. Both of these algorithms can be operated in classification or regression modes. The main difference between these modes is that classification mode provides a score for each of a given number of classes which can be interpreted as a probability that a given example belongs to the given class, whereas regression outputs a single number per example whose interpretation depends on the problem.

Describe training/testing set here.

1.1 Boosted Decision Trees

We will start by discussing a decision tree for a classification problem. Decision trees have a structure as in figure 1.1, which shows a tree dividing examples into two classes, red and blue.

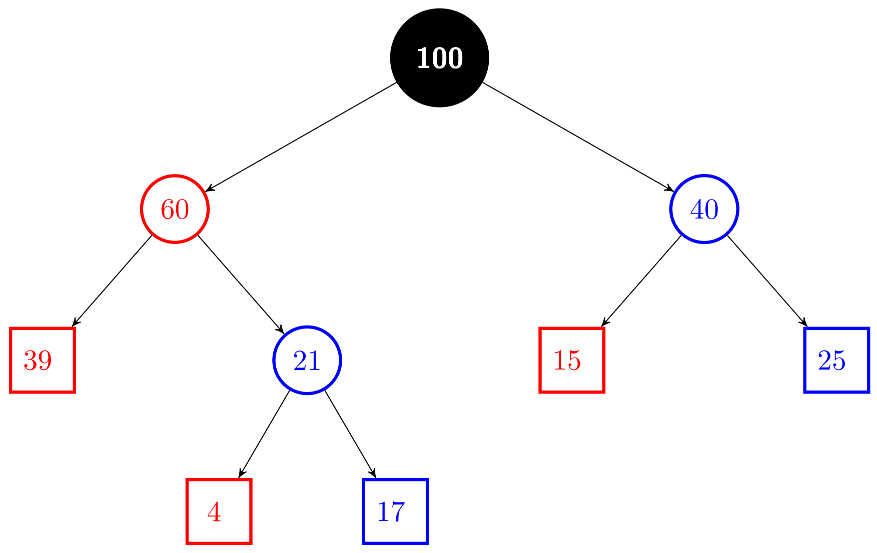

Figure 1.1: The structure of a decision tree set up for a classification problem. This diagram serves as a summary of where 100 examples end up after being passed through the tree. The number in each node corresponds to the number of examples that pass through that node.

Each circular node in the tree represents a cut on one of a number of variables provided as input to the algorithm. The tree is read top to bottom with each node being followed by two edges branching left and right that represent the path taken by examples which pass or fail the cut respectively. Square nodes represent that the termination criteria have been reached and that events in these nodes have been classified according to the colour of the node. The variable chosen at each node is optimised in order to maximise a criteria related to the separation of classes. For a problem containing two classes a common separation criteria is the Gini index,

$$G = p(1 - p), \tag{1.1}$$

where p is the purity of a chosen class that one wants to maximise.

A decision tree by itself is able to separate examples into a number of classes, however a single tree is prone to over–training. Over–training is the term used to describe the phenomena where a classifier learns from statistical fluctuations in the data rather learning a generalised separation boundary between classes. The reason that decision trees are susceptible to this is that if two variables yield a similar separation criteria then a fluctuation in the training data may lead to the choice of one variable over another for a particular node, this choice will lead to a very different tree structure than if the fluctuation were not present.

In order to mitigate the over–training tendencies of decision trees they are often used in an ensemble algorithm such as bagging [**?**] or [**?**]. Here only boosting will be discussed. Ensembles of decision trees are often referred to as random forests. Boosting works by training a sequence of trees, weighting misclassified events from a tree so that they have more influence over the structure of the next tree in the sequence. The final classification of any given example is a weighted average over all trees, this can be weighted by the overall accuracy of each tree, but in general can take any weighted average.

Gradient boosting here.

1.2 Neural Networks

Neural networks have a structure as in figure 1.4.

This section outlines some of the mathematical formalism surrounding NNs that act as classifiers. These kinds of NNs can be used to solve binary classification or multi-class problems, where the probability that each data candidate belongs to any of the classes is mutually exclusive. Many other types of NN exist, however their details will not be discussed in this work. In simple terms a NN is comprised of connected layers of nodes as in figure **??**. Nodes fall into three types input, output and hidden, with layers being homogeneous with regard to node type and therefore inheriting their name. The input layer is comprised of one input node per dimension of a single entry in the dataset that one wishes to pass into the NN for classification. Each output node is a predictive unit corresponding to one of K classes and it is the goal of the intermediate hidden layers (full of hidden units) to provide a relationship between the inputs and outputs such that the predictive unit with the highest value corresponds to the correct class for the given data. The output layer should therefore be comprised of K nodes. In order to delve further into the details, a good mathematical representation of the hidden layers is required. One such representation is given by Bishop in Pattern Recognition and Machine Learning [1]. In this section the nomenclature used by Bishop will be outlined and then adapted for our specific use.

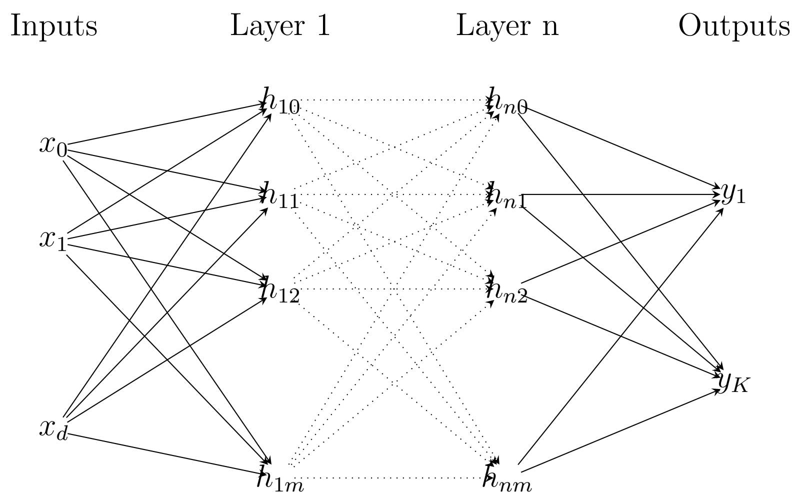

Figure 1.2: A more complex neural network containing an input layer of d nodes corresponding to data of dimensionality d, n hidden layers of m hidden units each h_{ij} (where i indexes hidden layer and j indexes a particular unit) and an output layer of K predictive units y_k.

The building blocks of the NN, called activations, resemble Fisher discriminants [2] and take the form

$$a_j = \sum_{i=1}^{d} w_{ji} x_i + w_{j0} \tag{1.2}$$

where the w_{ji} terms are known as weights and the w_{j0} as biases. Collectively the weights and biases shall be referred to as adaptive parameters, due to the fact that (as shall be made clear later) they are to be adapted by a training algorithm. In order to take the activations and turn them into something closer to a perceptron [3] they must be passed through an activation function denoted $\mathcal{H}$,

$$h_j = \mathcal{H}(a_j) \tag{1.3}$$

becoming what are known as hidden units. These activation functions may be non-linear and have the restriction that they must be differentiable, crucially this is different from the perceptron which uses a non-differentiable step function. Considering a network with only a single hidden layer as in figure **??**, there are further steps required to get from the hidden units to the predictions y_k. The complete network function as an argument of a vector of data points $\vec{x}$ and a matrix of adaptive parameters $\vec{w}$ should return predictions given by

$$y_k(\vec{x}, \vec{w}) = \mathcal{O}\left(\sum_{j=1}^{m} w_{kj}^{(2)} \mathcal{H}\left(\sum_{i=1}^{d} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right). \tag{1.4}$$

where now, as in Bishop's representation, the superscript number in brackets labels the layer to which adaptive parameters belong, not counting the input layer (as it does not contain them). The network function (1.4) is comprised by performing the same construction on the hidden unit (1.3) as was performed on the original data point x_i in (1.2), with a special type of activation function an output function $\mathcal{O}$. Notably the hidden units must be summed over in the same way that data points are, however instead of summing over the dimensions of the data, in order to reproduce the network in figure **??** we sum up to m the desired number of hidden units, which is referred to as the "size" of the hidden layer. A common choice of output function for binary classification problems is the logistic sigmoid function

$$\mathcal{O}(z) = \frac{1}{1 + exp(-z)} \tag{1.5}$$

where in each of these z merely denotes the argument of the output function. For multi-class problems where $K > 2$ a generalisation of the logistic sigmoid, the softmax function

$$\mathcal{O}(z)_k = p(k|\vec{x}) = \frac{exp(z_k)}{\sum_{i=1}^{k} exp(z_i)} \tag{1.6}$$

gives the probability of being in class k given the data $\vec{x}$ and where i is summed all classes. The index k appears on the output function as it must be calculated for each $k \in K$ the total number of classes. It is at this stage that the true meaning of the predictive units is solidified, each should give a number between zero and one that represents the probability of data belonging to the corresponding class, and logically all predictive units should sum to one. For the sake of compactness we shall, as Bishop does, re-write (1.4) by introducing $x_0 = 1$ in order to absorb the biases into the sums, yielding

$$y_k(\vec{x}, \vec{w}) = \mathcal{O}\left(\sum_{j=0}^{m} w_{kj}^{(2)} \mathcal{H}\left(\sum_{i=0}^{d} w_{ji}^{(1)} x_i\right)\right). \tag{1.7}$$

Now it is our goal to generalise this network function to one not only of an arbitrary number of hidden layers but also such that each hidden layer can be of arbitrary size and have an arbitrary activation function. For this purpose, networks will now be described in terms of the number of hidden layers, instead of the number of layers that contain adaptive parameters (hidden plus output) as before. We may start by writing a function for a network of two hidden layers

$$y_k(\vec{x}, \vec{w}) = \mathcal{O}\left(\sum_{j_2=0}^{m_2} w_{kj_2} \mathcal{H}_2\left(\sum_{j_1=0}^{m_1} w_{j_2 j_1} \mathcal{H}_1\left(\sum_{i=0}^{d} w_{j_1 i} x_i\right)\right)\right). \tag{1.8}$$

In order to arive at the two layer function (1.8) the same process was used as for the original network function (1.4). Now there are two activation functions and hidden layer sizes denoted m. Clearly adding a layer to the network simply involves repeated application of this process and picking up the required number of additional parameters. A function for a network of n hidden layers may therefore be written as

$$y_k(\vec{x},\vec{w}) = \mathcal{O}\left(\sum_{j_n=0}^{m_n} w_{kj_n}\mathcal{H}_n\left(\dots\mathcal{H}_2\left(\sum_{j_1=0}^{m_1} w_{j_2j_1}\mathcal{H}_1\left(\sum_{i=0}^{d} w_{j_1i}x_i\right)\right)\dots\right)\right) \tag{1.9}$$

where there are n different versions of the activation function and hidden layer size. Our new hidden units obey the following notation

$$h_{nj} = \mathcal{H}_n(a_j) \tag{1.10}$$

eliminating the need for the superscript labelling of layer. The new labelling identifies hidden layer by the left-hand index of the hidden unit or the right-hand hidden layer of the weights and biases. In equation (1.9) the hidden units are shown in their expanded form and distinguished by the fact that their j indices take on the subscript of n corresponding to the layer they belong to. Figure **??** depicts a network of n hidden layers, as described in equation (1.9) except that every hidden layer is depicted with the same size m.

There are now a great deal of parameters that we have to keep track of. It is therefore important to distinguish between adaptive parameters and the parameters which we must pick by hand, known as hyper-parameters. A relationship can be written between the number of hidden layers n, and the number of hyper-parameters, as follows

$$\#\text{ of hyper-parameters} = 2n + 1. \tag{1.11}$$

As previously stated there are simply n lots of activation functions and n hidden layers to determine a size for. Often it is sufficient to set all activation functions to the same function, and in this work all hidden layers share the same size for simplicity. The adaptive parameters are far more numerous, and will be optimised by means of a training algorithm. In TensorFlow the variable object is choice for

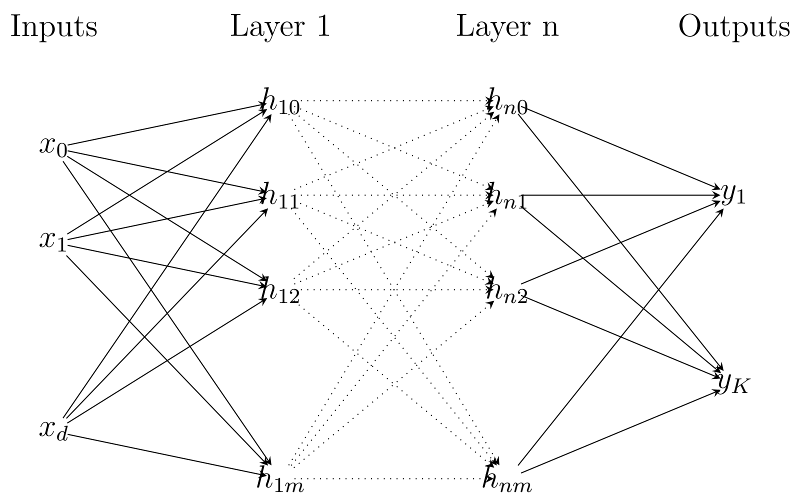

Figure 1.3: A more complex neural network containing an input layer of d nodes corresponding to data of dimensionality d, n hidden layers of m hidden units each h_{ij} (where i indexes hidden layer and j indexes a particular unit) and an output layer of K predictive units y_k.

implementing the adaptive parameters as training algorithms in the software will update them by default. The adaptive parameters are initialised randomly from a Gaussian distribution resulting in poor predictive power to begin with. In order to improve this some figure of merit, known as a loss function, must be used in order for the training algorithm to measure quantitatively the performance of the NN. We also must provide the algorithm with a dataset to train on, complete with a set of targets or labels that, for the training set, reveal the correct classification for each entry. Due to the this requirement, methods such as these are referred to as supervised learning methods. A natural loss function one may use to describe the error of the model given a current set of adaptive parameters is sum-of-squares error

$$E(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(\vec{x}_n, \vec{w}) - t_n)^2 \tag{1.12}$$

where t_n are the targets for the given data entries $\vec{x}_n$. Minimising this function with some algorithm does work in practice, however it has been shown that, using

$$E(\vec{w}) = -\sum_{n=1}^{N}\Bigg(t_n \ln(y_n) + (1 - t_n)\ln(1 - y_n)\Bigg) \tag{1.13}$$

known as cross-entropy, is faster and more generalised [4] (further discussion on generalisation in section **??**). The particular training algorithm that will be used to update parameters in this report is known as adaptive moment estimation or ADAM [5]. ADAM is a variant of the gradient descent algorithm, which is widely used and has spawned many other variants [6]. The reasons for picking between ADAM and vanilla gradient descent are given in chapter **??**.

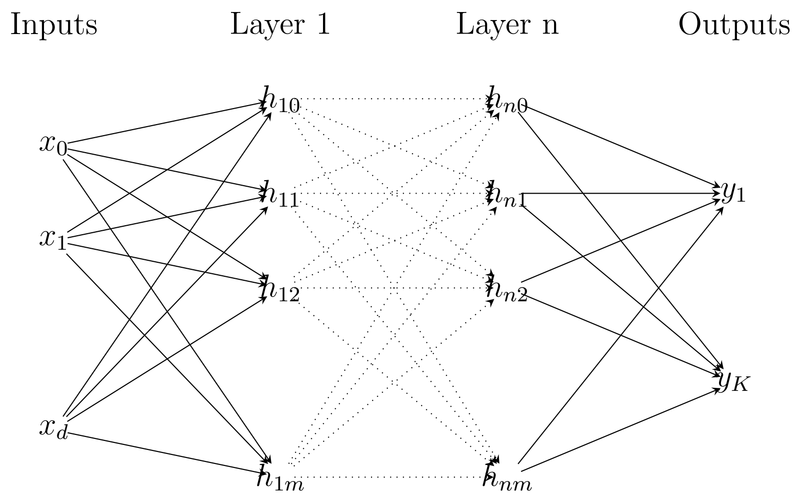

Figure 1.4: A more complex neural network containing an input layer of d nodes corresponding to data of dimensionality d, n hidden layers of m hidden units each h_{ij} (where i indexes hidden layer and j indexes a particular unit) and an output layer of K predictive units y_k.

1.3 Parametrised Neural Networks

Parametrised neural networks take extra inputs equal to the number of relevant parameters, as seen in figure 1.5.

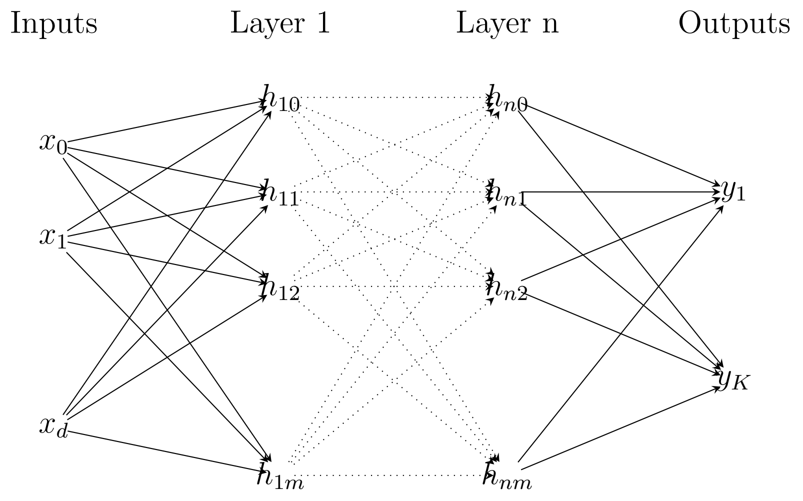

Figure 1.5: A more complex neural network containing an input layer of d nodes corresponding to data of dimensionality d, n hidden layers of m hidden units each h_{ij} (where i indexes hidden layer and j indexes a particular unit) and an output layer of K predictive units y_k.

Bibliography

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

[2] R. A. FISHER. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936. ISSN 2050-1439. doi: 10.1111/j.1469-1809.1936.tb02137.x. URL http://dx.doi.org/10.1111/j.1469-1809.1936.tb02137.x.

[3] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[4] John Platt Patrice Y. Simard, Dave Steinkraus. Best practices for convolutional neural networks applied to visual document analysis. Institute of Electrical and Electronics Engineers, Inc., August 2003. URL https://www.microsoft.com/en-us/research/publication/best-practices-for-convolutional-neural-networks-applied-to-visual-document-analysis/.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

[6] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.